UČEBNÍ TEXTY VYSOKÝCH ŠKOL

Vysoké učení technické v Brně Fakulta elektrotechnická

Doc. RNDr. Milan Češka, CSc.
Doc. Ing. Tomáš Hruška, CSc.
Ing. Miroslav Beneš

PŘEKLADAČE

Tento učební text je určen studentům oboru Informatika a výpočetní technika pro úvodní kurs Základy překladačů a rozšiřující kurs Výstavba překladačů. Cílem obou kursů je podat základní informace o struktuře překladačů klasických programovacích jazyků a metodách jejich konstrukce. Předpokladem pro jejich úspěšné zvládnutí jsou znalosti teorie formálních jazyků, programovacích technik a strojově orientovaných jazyků.

Postup výkladu v textu byl zvolen takový, aby odpovídal logické struktuře překladače i za cenu toho, že se témata, která jsou náplní obou kursů, navzájem prolínají. Zvolené uspořádání vede k mnohem kompaktnějšímu textu bez mnoha rušivých odkazů, což zřejmě ocení všichni, kteří budou tento učební text používat při řešení větších projektů nebo při přípravě ke státní zkoušce.

Na zpracování jednotlivých tématických celků se autoři podíleli takto:

Doc. RNDr. Milan Češka, CSc.	- kapitoly 9, 10
Doc. Ing. Tomáš Hruška, CSc.	- články 3.3-3.5, kapitola 11
Ing. Miroslav Beneš	- ostatní části

Obsah

Kapitola 1

Základní pojmy

V této kapitole se budeme zabývat obecným popisem činnosti a struktury překladače, jeho komunikace s okolím a některými podpůrnými prostředky používanými při výstavbě překladačů.

1.1 Úvod

Překladač je obvykle program, který čte *zdrojový program* (source program) a převádí ho do ekvivalentního *cílového programu* (object program). Zdrojový program je napsaný ve *zdrojovém jazyce*, cílový program je v *cílovém jazyce*. Důležitou částí tohoto procesu překladu jsou *diagnostické zprávy*, kterými překladač informuje uživatele například o přítomnosti chyb ve zdrojovém programu. Techniky překladačů se používají i pro realizaci počítačových architektur specializovaných na vyšší programovací jazyky (Modula, Lisp, Prolog). V tomto učebním textu ale budeme pojem překladač používat pouze pro program. Typickým zdrojovými jazyky budou programovací jazyky jako Modula-2, Pascal nebo C; typickým cílovým jazykem pro nás bude strojový kód nebo jazyk asembleru nějakého počítače.

1.1.1 Vývoj technik strojového překladu

První počítače byly velmi jednoduché, například počítač Mark 1 z roku 1948 měl pouze sedm instrukcí a 32 slov hlavní paměti. Pro takový počítač postačovalo vkládání programů pomocí posloupností binárních číslic. S příchodem složitějších počítačů se rozšiřovaly také instrukční soubory a koncem 40. let bylo poukázáno na to, že převod mnemonických názvů instrukcí do binárního kódu může být proveden pomocí počítače. Programy, které to prováděly, se nazývaly *asemblery* a příslušný mnemotechnický kód *jazyk asembleru*.

Další krok spočíval v zavedení *autokódů*, které umožňovaly reprezentovat jednou instrukcí několik strojových operací. Programy zajišťující jejich překlad se již nazývaly *překladače*, nebo také *kompilátory*. Jedním z používaných autokódů byl například MAT pro počítač Minsk-22, jehož mnemotechnické kódy byly odvozené z českých názvů operací.

Pojem *překladač* se používá od začátku 50. let, kdy se začaly vyvíjet uživatelsky orientované programovací jazyky vyšší úrovně, podstatně méně závislé na strojovém kódu konkrétního počítače. V tu dobu však ještě vládla všeobecná skepse nad použitelností "automatického programování," jak se tehdy programování ve vyšších jazycích nazývalo. První jazyky tohoto typu (např. FORTRAN) a autokódy, ze kterých se vyvinuly, však byly silně poznamenány tehdy existujícími instrukčními soubory počítačů. Například FORTRAN IV umožňoval práci pouze

s trojrozměrnými poli, neboť jeho první implementace byla provedena na počítači IBM 709, který měl pouze tři indexové registry. Dokonce i jazyk C, který se objevil uprostřed 70. let, má některé konstrukce (např. operátor inkrementace ++) zavedené díky dostupnosti ekvivalentních instrukcí původního cílového počítače PDP-11.

Algol 60, navržený ve skutečnosti již v roce 1958, přinesl další nový přístup. Byl navržen s ohledem na řešení konkrétních problémů a potlačoval otázky týkající se možností překladu na konkrétních počítačích. Umožňoval například užití lokálních proměnných a rekurzivních volání procedur. Již se nezabýval tím, jak provést překlad na počítači s jediným společným adresovým prostorem a jedinou instrukcí skoku do podprogramu. Tento přístup je v moderních programovacích jazycích běžný. Jazyky jako Pascal, Modula-2 a Ada byly navrženy nezávisle na jakékoliv konkrétní architektuře.

Moderní jazyky vysoké úrovně svým obvykle stručným zápisem umožňují zvýšit produktivitu práce programátora, poskytují různá sémantická omezení (např. typovou kontrolu), kterými se dají redukovat logické chyby v programech, a zjednodušují ladění programů. Další velmi významnou vlastností současných programovacích jazyků je možnost vytváření strojově nezávislých programů, které se dají přenášet i mezi principiálně různými architekturami počítačů. Jejich nevýhodou je rychlost překladu (typicky 2–10 krát nižší než u ručně psaných programů v jazyce asembleru) a velikost, jak překladače, tak přeloženého kódu. Tyto nevýhody jsou však redukovány s rozvojem moderních počítačových architektur. V oblasti návrhu a implementace jazyků se nyní často dostáváme do zcela opačné situace, než jaká byla na počátku vývoje jazyků, kdy jsou navrhovány procesory již s ohledem na překlad konkrétních jazyků (existují například specializované procesory pro Lisp, Pascal nebo Modulu).

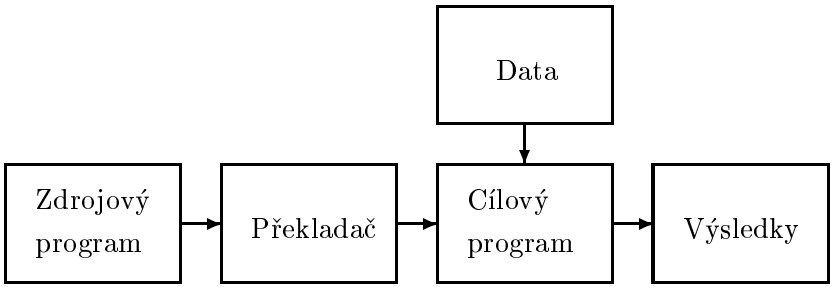
Teorie překladu a formálních jazyků dnes umožňuje běžně používané jazyky překládat bez obtíží. Pro automatickou výstavbu překladačů je k dispozici mnoho specializovaných prostředků. Zatímco na vývoj prvního překladače jazyka FORTRAN bylo třeba 18 "člověkoroků," nyní je vytvoření jednoduchého překladače jazyka Pascal zvládnutelné i pro studenta vysoké školy.

1.1.2 Přístupy ke strojovému překladu

Máme-li program napsaný v některém vyšším programovacím jazyce, existuje několik možných přístupů k jeho spuštění. Buď můžeme program převést do ekvivalentního programu ve strojovém kódu počítače — překladače tohoto typu se označují názvem *kompilátory* nebo *kompilační překladače*, nebo můžeme napsat program, který bude interpretovat příkazy zdrojového jazyka tak, jak jsou napsané, a přímo provádět odpovídající akce. Programy realizující druhý přístup se nazývají *interprety* nebo *interpretační překladače*. Obrázky 1.1 a 1.2 představují schémata činnosti obou typů překladačů.

Výhodou kompilace je, že analýza zdrojového programu a jeho překlad se provádějí jen jednou, i když může jít o časově dosti náročný proces. Dále již spouštíme pouze ekvivalentní program ve strojovém kódu, který je výsledkem překladu. Nevýhodou je někdy dosti obtížné hledání chyb ve zdrojovém programu, pokud máme pouze informace o místu chyby vyjádřené v pojmech strojového jazyka (adresy, výpisy obrazu paměti). Moderní překladače však často vytvářejí zároveň s cílovým kódem i pomocné datové struktury, které umožňují provádět ladění programu přímo na úrovni zdrojového jazyka — provádět program po jednotlivých příkazech, vypisovat hodnoty proměnných nebo posloupnost volání funkcí a hodnot jejich parametrů.

Kód generovaný kompilačním překladačem nemusí být obecně ekvivalentní se strojovým kódem nějakého konkrétního počítače. Obecně můžeme cílové kódy podle jejich vztahu k určitému procesoru a operačnímu systému rozdělit takto:



Obr. 1.1: Kompilační překladač

- *Čistý strojový kód.* Jedná se o strojový kód konkrétního počítače bez předpokladu existence určitého operačního systému nebo knihoven. Čistý strojový kód obsahuje pouze instrukce z instrukčního souboru počítače, pro který jsou překládané programy určeny. Tento přístup je velmi řídký, občas se používá pro jazyky určené k vytváření systémových programů (např. jader operačních systémů, které pracují autonomně bez další programové podpory).

- *Rozšířený strojový kód.* Tento typ zahrnuje kromě instrukcí daných architekturou procesoru také podprogramy operačního systému a podpůrné knihovní podprogramy (např. pro matematické funkce). Rozšířený strojový kód se dá považovat za kód *virtuálního počítače*, tvořeného kombinací konkrétního technického a programového vybavení nějakého počítače. Poměr obou složek se může u konkrétních implementací lišit, například překladač jazyka FORTRAN obvykle využívá knihoven pouze pro vstupy a výstupy a pro matematické funkce, zatímco velká část moderních překladačů pracuje s operacemi pro bitová pole, volací posloupnosti procedur a funkcí nebo pro dynamické přidělování paměti.

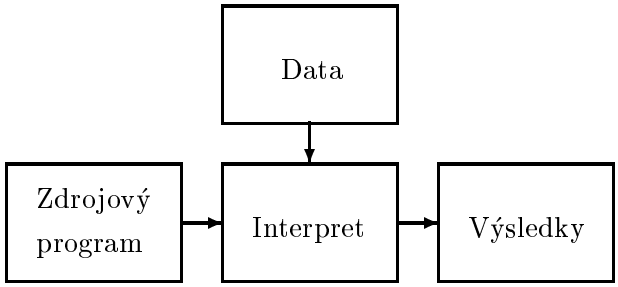
- *Virtuální strojový kód.* Nejobecnější forma strojového kódu obsahuje pouze virtuální instrukce, které nejsou závislé na žádné konkrétní architektuře nebo konkrétním operačním systému. Tato forma umožňuje vytvářet *přenositelné překladače*; při přenosu stačí pouze napsat interpret virtuálního kódu. Příkladem takového překladače je Wirthův Pascal P, jehož výstupem je tzv. *P-kód* pro virtuální zásobníkový počítač. Velice rychlá přenositelnost takového překladače možná byla jedním z důvodů velké popularity Pascalu.

Další vlastností cílového kódu, který podstatně ovlivňuje složitost návrhu kompilačního překladače, je jeho formát. Pro cílový kód se nejčastěji používá jeden z následujících formátů:

- *Symbolický formát.* Cílový program v symbolickém formátu má obvykle tvar zdrojového souboru v jazyce asembleru. V tomto případě je značně ulehčena práce překladače, neboť se nemusí zabývat například řešením dopředných odkazů v programu nebo přidělováním adres pro data. Tento přístup je častý pod operačním systémem Unix a je vhodný zejména tam, kde chceme překladač využívat k vytváření programů pro jiný počítač, než na kterém překladač běží (tzv. *křížový překladač*). I přes uvedené výhody se však nedoporučuje, protože se tak silně zpomaluje překlad (je třeba provést konverzi vnitřních datových struktur na text a ten musí zase asembler znovu analyzovat). Pro účely kontroly vygenerovaného kódu je však vhodné, když překladač dovede kód vypsat v symbolickém tvaru.

- *Relokatibilní binární formát.* Tento formát obsahuje cílový kód v binárním tvaru, ovšem bez vyřešených odkazů na externí symboly a pouze s adresami, počítanými relativně od začátku nějakého stanoveného úseku. Takový tvar je typický pro výstup z asembleru, takže se ušetří jeden krok následného zpracování cílového kódu. Symbolický a relokatibilní binární formát umožňují modulární překlad, odkazy na moduly překládané z jiných jazyků a využívání podpůrných knihoven podprogramů. K tomu však vyžadují dodatečné zpracování spojovacím programem.

- *Absolutní binární formát (Load-and-Go).* Program v absolutním binárním tvaru je překladačem ihned po překladu spuštěn. Tím se obejde pomalá fáze sestavování spustitelného programu za cenu omezené dostupnosti vazeb na externí knihovny. Navíc je pro každé spuštění programu nutný jeho opětovný překlad. Tento přístup je výhodný pro studentské programy a pro ladění, kdy se předpokládá častější překlad než spouštění programů.



Obr. 1.2: Interpretační překladač

Interpretace je mnohem pomalejší než kompilace, neboť je třeba analyzovat zdrojový příkaz pokaždé, když na něj program narazí. Pro poměr mezi rychlostí interpretovaného a kompilovaného programu se uvádějí hodnoty mezi 10:1 až 100:1, v závislosti na konkrétním jazyce. Interprety bývají také náročné na paměťový prostor, neboť i při běhu programu musí být stále k dispozici celý překladač.

Interprety však mají i své výhody oproti kompilačním překladačům. Při výskytu chyby máme vždy přesné informace o jejím výskytu a můžeme poměrně rychle odhalit její příčinu. Tento přístup je tedy vhodný zvláště při ladění programů. Interprety umožňují modifikaci textu programu i během jeho činnosti, což se využívá často u jazyků jako je Prolog nebo LISP. U jazyků, které nemají blokovou strukturu (např. BASIC, APL), se může změnit některý příkaz, aniž by se musel znovu překládat zbytek programu. Interprety se dále používají tam, kde se mohou typy objektů dynamicky měnit v průběhu provádění programu — typickým příkladem je jazyk Smalltalk-80. Jejich zpracování je pro kompilační překladače značně obtížné. Interpretační překladače bývají značně strojově nezávislé, neboť negenerují strojový kód. Pro přenos na jiný počítač obvykle postačí interpret znovu zkompilovat.

Uvedené dva přístupy jsou však extrémní, mnoho překladačů využívá spíše jejich kombinace. Některé interpretační překladače například nejdříve převedou zdrojový program do nějakého vnitřního tvaru (v nejjednodušším případě alespoň nahradí klíčová slova jejich binárními kódy) a ten potom interpretují. Výsledné řešení je kompromisem mezi časově náročným překladem kompilovaného a pomalým během interpretovaného programu.

Výběr vhodného přístupu, zda kompilovat nebo interpretovat, závisí obvykle na povaze jazyka a prostředí, ve kterém se používá. Pro časově náročné matematické výpočty se používají kompilační překladače, naopak pro účely výuky jazyků nebo na malých mikropočítačích se dává přednost interpretaci (typickými příklady takových jazyků jsou BASIC, LOGO nebo Smalltalk-80). Pro jazyk LISP se často používá zároveň obou přístupů, neboť jeho kompilace je časově značně náročná a kompilovaný program nemá dostatečné prostředky pro ošetření chyb. Interpretační překlad se také běžně užívá u různých příkazových jazyků, kdy se očekává okamžité provedení příkazu — příkladem mohou být dotazovací databázové jazyky jako SQL nebo jazyky řídicích programů, umožňující spouštění programů a komunikaci s operačním systémem počítače, např. `sh` nebo `csh` v systému Unix.

Tento učební text je orientován převážně na kompilační překladače, i když mnoho uvedených algoritmů je možné použít také při psaní interpretu. V obou případech bývá stejná analýza zdrojového kódu a často bývají podobné i metody hledání nejefektivnějšího kódu pro interpretaci s metodami generování cílového kódu.

1.1.3 Další použití překladačů

Techniky překladačů se samozřejmě nejčastěji používají pro překladače programovacích jazyků. Mají však mnohem širší využití i v jiných oblastech. Mnoho podpůrných programových prostředků, které manipulují se zdrojovým programem, provádí rovněž jistý druh analýzy. Tyto prostředky zahrnují například:

- *Strukturované editory.* Strukturovaný editor má jako vstup posloupnost příkazů pro vybudování zdrojového programu. Struktorovaný editor neprovádí pouze funkce pro vytváření a modifikaci textu jako běžný textový editor, ale analyzuje navíc text programu a vkládá do něj vhodnou hierarchickou strukturu. Strukturovaný editor tedy může plnit ještě další úkoly, které jsou užitečné při přípravě programu. Může například kontrolovat, zda je vstup správně syntakticky zapsán, může automaticky doplňovat klíčová slova (např. když uživatel napíše **while**, doplní editor odpovídající **do** a připomene uživateli, že mezi nimi musí být logický výraz) nebo může přecházet z klíčového slova **begin** nebo levé závorky na odpovídající **end** nebo pravou závorku. Navíc výstup takového editoru je často podobný výstupu analytické části překladače.

- *Formátovací programy.* Formátovací program (pretty printer) analyzuje program a tiskne ho takovým způsobem, aby byla zřetelná jeho struktura. Například poznámky mohou být vytištěny jiným typem písma a příkazy mohou být odsazeny v závislosti na úrovni jejich zanoření v hierarchické struktuře příkazů.

- *Programy pro sazbu textů.* Programy pro sazbu textů umožňují kombinovat text knihy, článku nebo dopisu s příkazy, které zajišťují členění na odstavce, kapitoly, změnu typu a velikosti písma, vytváření obsahu nebo indexu, speciální sazbu matematických textů nebo dokonce sazbu not nebo šachových partií. Typickým zástupcem této třídy programů jsou TEX a LATEX [12, 13], kterými byl připraven tento učební text.

Zdrojovým jazykem překladače nemusí být vždy nějaký programovací jazyk. Může se jednat také o některý přirozený jazyk (např. angličtinu), speciální jazyk popisující strukturu křemíkového integrovaného obvodu nebo strukturu grafických informací, které se mají zobrazit na tiskárně. Cílovým kódem takového překladače pak může být třeba jiný přirozený jazyk, maska

integrovaného obvodu nebo posloupnost příkazů pro ovladač laserové tiskárny. Programovacím jazykem tohoto typu je například PostScript [2], který se používá pro vytváření grafiky, nebo Metafont [11], kterým se definují tvary znaků používaných při sazbě textů připravených programem TEX. Tyto jazyky mají i prostředky pro vytváření cyklů, podmíněných příkazů nebo pro definování vlastních procedur nebo funkcí.

V dalších kapitolách se budeme věnovat výhradně klasickým překladačům, opět s tím, že uvedené techniky jsou použitelné i v jiných oblastech, zejména techniky analýzy zdrojového textu.

1.2 Struktura překladače

Překladač musí provádět dvě základní činnosti: analyzovat zdrojový program a vytvářet k němu odpovídající cílový program. Analýza spočívá v rozkladu zdrojového programu na jeho základní součásti, na základě kterých se během syntézy vybudují moduly cílového programu. Obě části překladače, analytická i syntetická, využívají ke své činnosti společné tabulky.

Analýza zdrojového programu při překladu probíhá na následujících třech úrovních:

- *Lexikální (lineární) analýza.* Zdrojový program vstupuje do procesu překladu jako posloupnost znaků. Tato posloupnost se čte lineárně zleva doprava a sestavují se z ní lexikální symboly (tokens) jako konstanty, identifikátory, klíčová slova nebo operátory.
- *Syntaktická (hierarchická) analýza.* Z posloupnosti lexikálních symbolů se vytvářejí hierarchicky zanořené struktury, které mají jako celek svůj vlastní význam, např. výrazy, příkazy, deklarace nebo program.
- *Sémantická analýza.* Během sémantické analýzy se provádějí některé kontroly, zajišťující správnost programu z hlediska vazeb, které nelze provádět v rámci syntaktické analýzy (např. kontrola deklarací, typová kontrola apod.).

Uvedené členění na úrovně analýzy vychází z toho, že běžné programovací jazyky jsou z hlediska Chomského klasifikace typu 1, tj. kontextové. Pro přímou analýzu kontextových jazyků dosud nebyly vyvinuty — na rozdíl od jazyků bezkontextových — dostatečně efektivní prostředky. Proto se na každé z těchto úrovní používají speciální metody specifikace i implementace, které využívají vlastností jazyků příslušných typů, tj. lineární analýza se provádí prostředky pro analýzu regulárních jazyků a hierarchická analýza prostředky pro analýzu bezkontextových jazyků. Pro sémantickou analýzu se obvykle využívá některá modifikace atributových gramatik, větší část sémantické analýzy však bývá implementována přímo prostředky jazyka, jímž je realizován překladač.

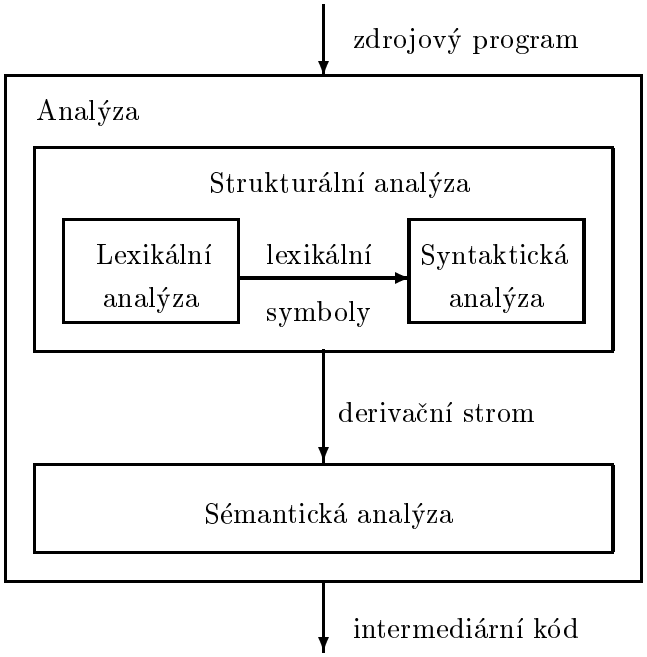
1.2.1 Lexikální analýza

Fáze lexikální analýzy (lexical analysis, scanning) čte znaky zdrojového programu a sestavuje je do posloupnosti *lexikálních symbolů*, v níž každý symbol představuje logicky související posloupnost znaků jako identifikátor nebo operátor obdobný :=. Posloupnost znaků tvořících symbol se nazývá *lexém* (lexeme).

Po lexikální analýze znaků např. v tomto přiřazovacím příkazu

```
pozice := počátek + rychlost * 60
```
(1.1)

by se vytvořily následující lexikální jednotky:



Obr. 1.3: Struktura analytické části překladače

1. identifikátor `pozice`
2. symbol přiřazení `:=`
3. identifikátor `počátek`
4. operátor `+`
5. identifikátor `rychlost`
6. operátor `*`
7. číslo `60`

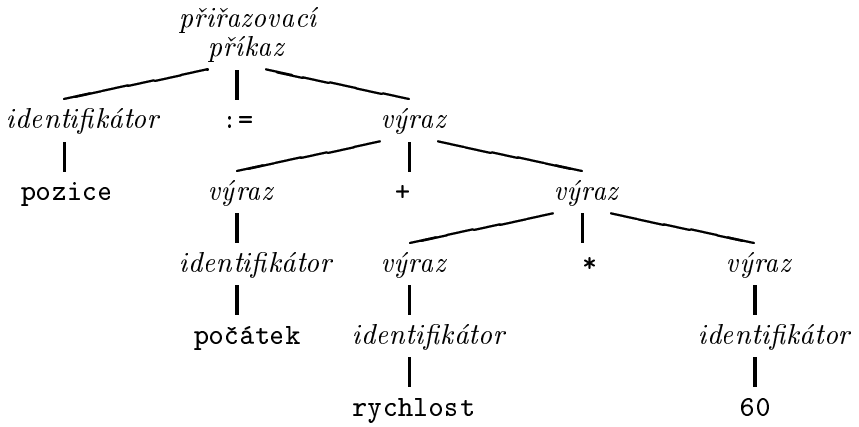
Symboly, které zahrnují celou třídu lexikálních jednotek (identifikátor, číslo, řetězec), jsou reprezentovány obvykle jako dvojice <druh symbolu, hodnota>, přičemž druhá část dvojice může být pro některé symboly prázdná. Výstupem lexikálního analyzátoru pro příkaz (1.1) by tedy mohla být posloupnost

```
<id,pozice> <:=> <id,počátek> <+> <id,rychlost> <*> <num,60>
```

Mezery, konce řádků a poznámky oddělující lexikální symboly se obvykle během lexikální analýzy vypouštějí.

1.2.2 Syntaktický analyzátor

Syntaktická analýza (parsing, syntax analysis) spočívá v sestavování lexikálních jednotek ze zdrojového programu do gramatických frází, které překladač používá pro syntézu výstupu. Gramatické fráze zdrojového programu se obvykle reprezentují *derivačním stromem* obdobným stromu na obr. 1.4.



Obr. 1.4: Derivační strom pro výraz `pozice:=počátek+rychlost*60`

Ve výrazu `počátek+rychlost*60` je fráze `rychlost*60` logickou jednotkou, neboť podle běžných matematických konvencí pro aritmetické výrazy se násobení provádí před sčítáním. Vzhledem k tomu, že za výrazem `počátek+rychlost` následuje `*`, nevytváří tento výraz v situaci na obr. 1.4 frázi.

Hierarchická struktura programu se obvykle vyjadřuje pomocí rekurzivních pravidel, zapsaných ve formě bezkontextové gramatiky. Například pro definici části výrazu můžeme mít následující pravidla:

```
výraz  ->  identifikátor                    (1)
výraz  ->  číslo                            (2)
výraz  ->  výraz + výraz                    (3)
výraz  ->  výraz * výraz                    (4)
výraz  ->  ( výraz )                       (5)
```

Pravidla (1) a (2) jsou (nerekurzivní) základní pravidla, zatímco (3)–(5) definují výraz pomocí operátorů aplikovaných na jiné výrazy. Podle pravidla (1) jsou tedy `počátek` a `rychlost` výrazy. Podle pravidla (2) je `60` výraz, zatímco z pravidla (4) můžeme nejprve odvodit, že `rychlost*60` je výraz a konečně z pravidla (5) také `počátek+rychlost*60` je výraz.

Podobným způsobem jsou definovány příkazy jazyka, jako např.:

```
příkaz  ->  identifikátor := výraz
příkaz  ->  while ( výraz ) do příkaz                    (1.2)
příkaz  ->  if ( výraz ) then příkaz
```

Dělení na lexikální a syntaktickou analýzu je dosti volné. Obvykle vybíráme takové rozdělení, které zjednodušuje činnost analýzy. Jedním z faktorů, které přitom uvažujeme, je to, zda jsou konstrukce zdrojového jazyka regulární nebo ne. Lexikální jednotky lze obvykle popsat jako

regulární množiny, zatímco konstrukce vytvořené z lexikálních jednotek již vyžadují obecnější přístupy.

Typickou regulární konstrukcí jsou identifikátory, popsané obvykle jako posloupnosti písmen a číslic začínající písmenem. Běžně rozpoznáváme identifikátory jednoduchým prohlížením vstupního textu, v němž očekáváme znak, který není písmeno ani číslice, a potom seskupíme všechna písmena a číslice nalezené až do tohoto místa do lexikální jednotky pro identifikátor. Znaky takto shromážděné zaznamenáme do tabulky (tabulky symbolů) a odstraníme je ze vstupu tak, aby mohlo pokračovat zpracování dalšího symbolu.

Tento způsob lineárního prohledávání na druhé straně není dostatečný pro analýzu výrazů nebo příkazů. Nemůžeme například jednoduše kontrolovat dvojice závorek nebo klíčových slov `begin` a `end` v příkazech bez zavedení jakéhosi druhu hierarchické struktury na vstupu.

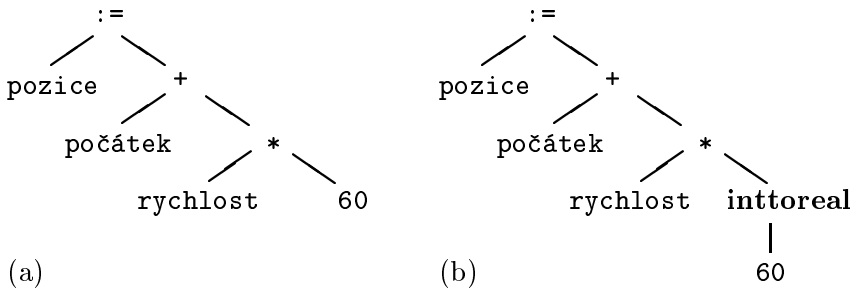
Derivační strom na obr. 1.4 popisuje syntaktickou strukturu vstupu, ale obsahuje informace, které nejsou důležité pro další průběh překladu. Mnohem běžnější vnitřní reprezentaci této syntaktické struktury dává *syntaktický strom* na obr. 1.5(a). Syntaktický strom je zhuštěnou reprezentací derivačního stromu; operátory v něm vystupují jako vnitřní uzly a operandy těchto operátorů jsou následníky jejich příslušných uzlů.

1.2.3 Sémantická analýza

Fáze sémantické analýzy zpracovává především informace, které jsou uvedeny v *deklaracích*, ukládá je do vnitřních datových struktur a na jejich základě provádí sémantickou kontrolu příkazů a výrazů v programu. K identifikaci operátorů a operandů těchto výrazů a příkazů využívá hierarchickou strukturu, určenou ve fázi syntaktické analýzy.

Důležitou složkou sémantické analýzy je *typová kontrola*. Kompilátor zde kontroluje, zda všechny operátory mají operandy povolené specifikací zdrojového jazyka. Mnoho definic programovacích jazyků například vyžaduje, aby kompilátor hlásil chybu, kdykoliv je reálné číslo použito jako index pole. Specifikace jazyka však může dovolit některé implicitní transformace operandů, například při aplikaci binárního aritmetického operátoru na celočíselný a reálný operand. V tomto případě může kompilátor požadovat konverzi celého čísla na reálné.

Jsou-li např. všechny proměnné v našem ukázkovém příkazu reálné, je třeba provést konverzi celočíselné konstanty `60` na reálnou, jak znázorňuje obr. 1.5(b). V tomto případě je rovněž možné typovou konverzi provést přímo a konstantu `60` nahradit hodnotou `60.0`.



Obr. 1.5: Syntaktický strom

1.2.4 Generování mezikódu

Po ukončení syntaktické a sémantické analýzy generují některé překladače explicitní *intermediární reprezentaci* zdrojového programu (*mezikód*). Intermediární reprezentaci můžeme považovat za program pro nějaký abstraktní počítač. Tato reprezentace by měla mít dvě důležité vlastnosti: měla by být jednoduchá pro vytváření a jednoduchá pro překlad do tvaru cílového programu.

Intermediární kód slouží obvykle jako podklad pro optimalizaci a generování cílového kódu. Může však být také konečným produktem překladu v interpretačním překladači, který vygenerovaný mezikód přímo provádí.

Intermediární reprezentace mohou mít různé formy. Například tříadresový kód se podobá jazyku symbolických instrukcí pro počítač, jehož každé místo v paměti může sloužit jako registr. Tříadresový kód se skládá z posloupnosti instrukcí s nejvýše třemi operandy. Zdrojový program z (1.1) by mohl v tříadresovém kódu vypadat následovně:

```
temp1  := inttoreal(60)
temp2  := rychlost * temp1                    (1.3)
temp3  := počátek + temp2
pozice := temp3
```

Intermediárními reprezentacemi, které se využívají v překladačích, se budeme zabývat v kapitole 8. Obecně tyto reprezentace musejí dělat více než jen výpočty výrazů; musejí si například poradit s řídicími konstrukcemi a voláním procedur.

1.2.5 Optimalizace kódu

Fáze optimalizace kódu se pokouší vylepšit intermediární kód tak, aby jeho výsledkem byl rychlejší nebo kratší strojový kód. Pojem "optimalizace" se nechápe doslovně jako nalezení nejlepší varianty, některé optimalizační algoritmy mohou ve zcela speciálních případech vést dokonce ke zhoršení vlastností původního kódu.

Některé optimalizace jsou triviální. Například přirozený algoritmus generuje intermediární kód (1.3) pomocí jedné instrukce pro každý operátor ve stromové reprezentaci po sémantické analýze, i když existuje lepší způsob provedení těchže výpočtů pomocí celkem dvou instrukcí:

```
temp1  := rychlost * 60.0
pozice := počátek + temp1                     (1.4)
```

Uvedenou optimalizaci lze bez problémů v překladači realizovat. Překladač totiž může zjistit, že konverzi hodnoty `60` z celočíselného na reálný tvar lze provést jednou provždy v čase překladu, takže operaci `inttoreal` je možné vypustit. Dále hodnota `temp3` se používá pouze jednou pro přenesení její hodnoty do proměnné `pozice`. Můžeme tedy bez obav použít místo `temp3` přímo proměnnou `pozice`, takže není potřebný poslední příkaz v (1.3) a dostaneme kód (1.4).

V množství různých prováděných optimalizací se jednotlivé překladače od sebe značně liší. Překladače, tzv. "optimalizující," které provádějí většinu optimalizací, stráví podstatnou část doby překladu právě v této fázi. Existují však i jednoduché optimalizace, které podstatně zlepší dobu běhu přeloženého programu bez velkého zpomalení překladu.

1.2.6 Generování cílového kódu

Poslední fází překladače je generování cílového kódu, což je obvykle přemístitelný strojový kód nebo program v jazyce asembleru. Všem proměnným použitým v programu se přidělí místo

v paměti. Potom se instrukce mezikódu překládají do posloupnosti strojových instrukcí, které provádějí stejnou činnost. Kritickým problémem je přiřazení proměnných do registrů.

Překlad kódu (1.4) může například s použitím registrů 1 a 2 vypadat takto:

```
MOVF rychlost, R2
MULF #60.0, R2
MOVF počátek, R1
ADDF R2, R1
MOVF R1, pozice
```

První operand každé instrukci je zdrojový, druhý operand cílový. Písmeno `F` ve všech instrukcích znamená, že pracujeme s hodnotami v pohyblivé řádové čárce. Uvedený kód přesune obsah adresy `rychlost` (obešli jsme zatím důležitý problém přidělení paměti identifikátorům zdrojového programu) do registru 2, potom ho vynásobí reálnou konstantou `60.0`. Znak `#` znamená, že se má hodnota `60.0` zpracovat jako konstanta. Třetí instrukce přesouvá hodnotu `počátek` do registru 1 a přičítá k němu hodnotu vypočtenou dříve v registru 2. Na konec se přesune hodnota z registru 1 na adresu `pozice`, takže tento kód implementuje přiřazení z obr. 1.4.

1.2.7 Tabulka symbolů

Základní funkcí tabulky symbolů je zaznamenávání identifikátorů použitých ve zdrojovém programu a shromažďování informací a různých atributech každého identifikátoru. Tyto atributy mohou poskytovat informaci o paměti přidělené např. proměnné, její typu, rozsah platnosti a v případě jmen procedur takové věci jako počet a typy argumentů, způsob předávání každého argumentu (např. odkazem) a typ vrácené hodnoty, pokud nějaká existuje.

Tabulka symbolů (symbol table) je datová struktura obsahující pro každý identifikátor jeden záznam s jeho atributy. Tato datová struktura umožňuje rychlé vyhledání záznamu pro konkrétní identifikátor a rychlé ukládání nebo vybírání příslušných dat ze záznamu. Tabulkami symbolů se budeme zabývat v kapitole 5.

Rozpozná-li lexikální analyzátor ve zdrojovém programu identifikátor, může ho rovnou uložit do tabulky symbolů. Během lexikální analýzy však normálně nemůžeme všechny atributy identifikátoru určit. Například v pascalovské deklaraci

```
var pozice, počátek, rychlost : real;
```

není typ `real` znám v okamžiku, kdy lexikální analyzátor vidí identifikátory `pozice`, `počátek` a `rychlost`.

Informace o identifikátorech ukládají do tabulky symbolů zbývající fáze, které je také různým způsobem využívají. Během sémantické analýzy a generování intermediárního kódu například potřebujeme znát typy proměnných a funkcí, abychom mohli zkontrolovat jejich správné použití ve zdrojovém programu a generovat pro ně správné operace. Generátor kódu typicky ukládá a používá podrobné informace o paměti přidělené jednotlivým objektům v programu.

1.2.8 Diagnostika a protokol o průběhu překladu

Velkou část chyb zpracovávají fáze syntaktické a sémantické analýzy. Lexikální fáze odhaluje chyby v případě, že znaky na vstupu netvoří žádný symbol jazyka. Chyby, kdy posloupnost symbolů porušuje strukturní pravidla (syntaxi) jazyka, se detekují během syntaktické analýzy. Během sémantické analýzy se překladač pokouší nalézt konstrukce, které mají sice syntaktickou strukturu odpovídající bezkontextové gramatice jazyka, avšak porušují kontextová omezení

(např. nedeklarované proměnné) nebo sémantická pravidla jazyka, např. pokud se pokoušíme sečíst v Pascalu dva identifikátory, z nichž jeden je jménem pole a druhý jménem procedury. Zpracováním chyb v jednotlivých fázích se budeme zabývat podrobněji vždy v příslušné kapitole.

Při výskytu chyby ve zdrojovém textu (případně chyby způsobené vnějšími okolnostmi, jako např. neúspěšný zápis do pracovního souboru v důsledku zaplnění disku) musí překladač nějakým způsobem reagovat. Možné reakce překladače můžeme obecně shrnout do následujícího seznamu:

I. Nepřijatelné reakce

   (a) Nesprávné reakce (bez ohlášení chyby)

      - Překladač *zhavaruje* nebo *cyklí*.
      - Překladač *pokračuje*, ale generuje nesprávný cílový program.

   (b) Správné reakce (ale nepoužitelné)

      - Překladač nahlásí první chybu a *zastaví se*.

II. Přijatelné reakce

   (a) Možné reakce

      - Překladač nahlásí chybu a *zotaví se*, pokračuje v hledání dalších možných chyb.
      - Překladač nahlásí a *odstraní chybu*, pokračuje v generování správného cílového kódu.

   (b) Nemožné reakce (se současnými metodami)

      - Překladač nahlásí a *opraví chybu*, pokračuje v generování programu odpovídajícího přesně záměrům programátora.

Nejproblematičtější je případ, kdy překladač na chybu nezareaguje a vytvoří cílový kód. Taková chyba se může projevit až po delší době a může způsobit i vážnou ztrátu dat. Přeložený program může mít neočekávané chování, které není vysvětlitelné na základě jeho zdrojového kódu. Ukončení překladu po první nalezené chybě značně prodlužuje proces ladění programu nutností neustále opakovaných překladů. Tento typ reakce je snad ještě možný v integrovaných vývojových prostředích, kdy se oprava a nové spuštění programu provede velmi jednoduše, ale obecně lze říci, že minimální přijatelnou reakcí překladače na chybu je zotavení. Algoritmy, které umožňují odstranění chyb (modifikací zdrojového textu nebo vnitřního tvaru programu) jsou časově náročné a tedy nevhodné pro interaktivní prostředí. Navíc umožňují spuštění nesprávně modifikovaného programu s možnými důsledky jako při neohlášení chyby.

1.3 Organizace překladu

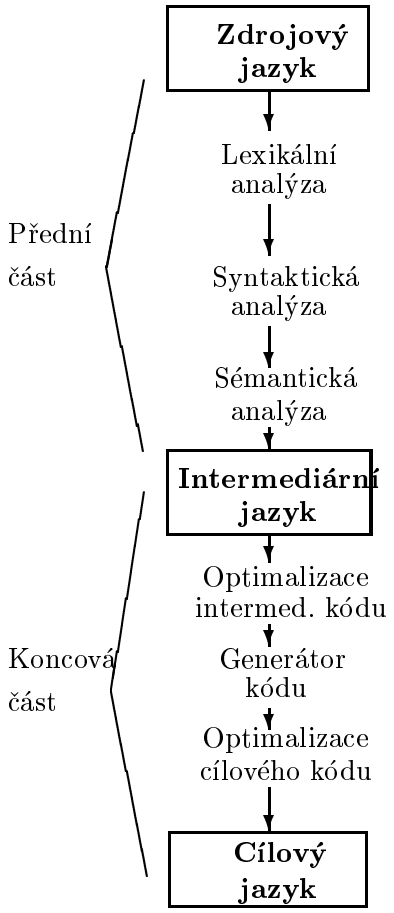
1.3.1 Fáze překladu

Obecné schéma překladače z hlediska jeho členění na fáze je uvedeno na obr. 1.6. Toto členění odpovídá logické struktuře překladače, která však nemusí přímo odpovídat skutečné implementaci.

Jednotlivé fáze se často rozdělují na *přední část* (front end) a *koncovou část* (back end). Přední část se skládá z těch fází nebo jejich částí, které závisejí převážně na zdrojovém jazyku a jsou dosti nezávislé na cílovém počítači. Obvykle zahrnuje lexikální a syntaktickou analýzu,

vytváření tabulky symbolů, sémantickou analýzu a generování intermediárního kódu. V přední části překladače lze provést rovněž jistou část optimalizace kódu. Obsahuje také obsluhu chyb, které vznikají během analýzy.

Koncová část zahrnuje ty části překladače, které již závisejí na cílovém počítači, a obecně nezávisí na zdrojovém jazyku, ale na intermediárním kódu. V koncové části překladače nalezneme prvky fáze optimalizace kódu a generování kódu společně s nutnými operacemi pro obsluhu chyb a operace s tabulkou symbolů. Dělení na přední a koncovou část obvykle koresponduje s dělením na analytickou a syntetickou část překladu, i když přední část také provádí syntézu intermediárního kódu a koncová část zase tento kód analyzuje.



Obr. 1.6: Fáze překladače

Při přenosu překladače na jiný cílový počítač se při dobře provedeném návrhu pouze převezme přední část, ke které se připojí nově vytvořená koncová část. Je-li koncová část vhodně navržena, nemusí dokonce být nutné ji příliš měnit. V rozsáhlejších návrhových systémech s více jazyky se někdy také snažíme překládat několik různých programovacích jazyků do téhož intermediárního jazyka a použít pro různé přední části jedinou koncovou část. Vzhledem k tomu, že ale mezi koncepcemi různých jazyků existují určité rozdíly, má tento postup jen omezené možnosti. Uvedený postup zvolila např. firma JPI ve své řadě překladačů TopSpeed (C, C++, Pascal, Modula-2).

1.3.2 Průchody

Několik fází překladu se obvykle implementuje do jediného *průchodu* (pass) skládajícího se ze čtení vstupního souboru a zápisu výstupního souboru. V praxi existuje mnoho variací ve způsobu rozdělení fází překladače do průchodů, které závisejí především na následujících okolnostech:

- Vlastnosti zdrojového a cílového jazyka.
- Velikost dostupné paměti pro překlad.
- Rychlost a velikost překladače.
- Rychlost a velikost cílového programu.
- Požadované informace a prostředky pro ladění.
- Požadované techniky detekce chyb a zotavení.
- Rozsah projektu — velikost programátorského týmu, časové možnosti.

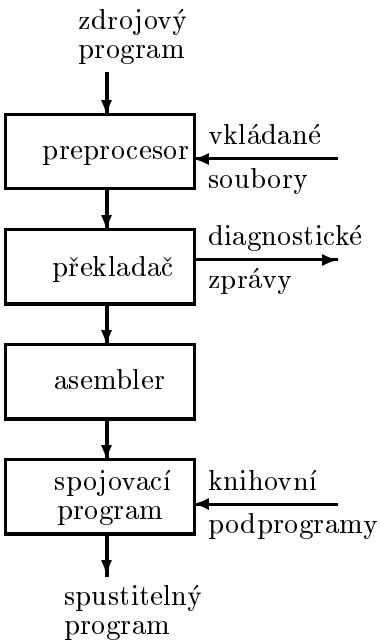
Překladače určené především pro výuku jsou obvykle jednoprůchodové. Neprovádějí mnoho optimalizací, neboť se předpokládá častější spouštění překladače než samotného přeloženého programu. Větší důraz se u nich klade na zpracování chyb a možnosti ladění. Naopak v překladačích používaných pro vytváření uživatelských aplikací je důležitá důkladná optimalizace, která se obvykle provádí ve více průchodech. Některé jazyky dokonce není možné překládat v jednom průchodu z prinicipiálních důvodů, neboť například umožňují volat procedury dříve, než jsou známy typy jejich parametrů.

Činnost fází, které vytvářejí jeden průchod, se často navzájem překrývá. Například lexikální, syntaktická a sémantická analýza mohou vytvářet jediný průchod. Posloupnost symbolů po lexikální analýze pak můžeme překládat přímo do intermediárního kódu. Syntaktický analyzátor můžeme při podrobnějším pohledu brát jako řídicí prvek. Pokouší se odkrýt gramatickou strukturu symbolů, které vidí; symboly získává tehdy, když je potřebuje, voláním lexikálního analyzátoru. Po rozpoznání gramatické struktury syntaktický analyzátor volá generátor intermediárního kódu, aby provedl sémantickou analýzu a vygeneroval část kódu. Náš pohled na návrh překladače bude směřovat právě k tomuto způsobu organizace.

1.4 Příbuzné programy

K překladači mohou být navíc nutné pro vytvoření proveditelného programu i některé další pomocné programy (viz obr. 1.7). Typický proces zpracování zdrojového programu v sobě může zahrnovat spuštění preprocesoru, který zpracuje makrodefinice, příkazy pro podmíněný překlad nebo příkazy pro vložení textu z jiného souboru do zdrojového programu. Po překladu vznikne cílový kód, který může mít buď tvar přemístitelného binárního modulu, nebo v některých jednodušších překladačích může být výstupem program, který je třeba dále zpracovat asemblerem. Přeložené moduly musí dále zpracovat spojovací program, který k nim připojí knihovní podprogramy a obvykle i část kódu, která zajišťuje různé pomocné činnosti v době běhu programu (tzv. run-time systém). Výsledkem činnosti spojovacího programu je již spustitelný program.

Některé rozsáhlejší vývojové systémy obsahují kromě uvedených základních prostředků ještě různé podpůrné programy, zajišťující například tyto činnosti:



Obr. 1.7: Postup při vytváření spustitelného programu

- Ladění programu na symbolické nebo strojové úrovni.
- Zkoumání uschovaného obsahu paměti po havárii programu.
- Zpětný překlad cílového programu do zdrojového tvaru.
- Formátování programu pro tisk.
- Tisk seznamu křížových referencí.
- Generování statistik o činnosti programu (profilování) — např. počet volání každé procedury, využití operační paměti, času procesoru apod.
- Archivace vývojových verzí programu.
- Údržba aktuální verze programu — automatické spouštění překladu změněných programových modulů a budování spustitelného programu (programy typu `make`).
- Údržba knihoven podprogramů.
- Specializované editory.

Při návrhu překladače je třeba mít použití těchto prostředků na paměti tak, aby jich mohl uživatel co nejvíce využívat. Překladač například musí zajistit generování dostatečných informací pro symbolické ladění programu (jména a umístění proměnných a procedur, odkazy na začátky zdrojových řádků apod.) nebo musí do generovaného programu vkládat volání speciálních služeb pro vyhodnocování statistik o činnosti programu.

1.5 Automatizace výstavby překladačů

V rámci teorie a praktických aplikací byla vyvinuta řada programových nástrojů, které usnadňují implementaci překladačů. Jejich spektrum zahrnuje jednoduché generátory (konstruktory) lexikálních a syntaktických analyzátorů, ale i komplexní systémy nazývané *generátory překladačů* (compiler-generators), *kompilátory kompilátorů* (compiler-compilers) nebo *systémy pro psaná překladačů* (translator-writing systems). Tyto systémy na základě specifikace zdrojového jazyka a cílového počítače generují překladač pro daný jazyk. Vstupní specifikace může zahrnovat

- popis lexikální a syntaktické struktury zdrojového jazyka,
- popis, co se má generovat pro každou konstrukci zdrojového jazyka,
- popis počítače, pro který má být generován kód.

V mnoha případech jsou tyto specifikace v podstatě souborem programů, které generátor kompilátorů vhodně "spojí." Některé generátory však umožňují, aby části specifikací měly neprocedurální charakter, tj. aby například namísto sytaktického analyzátoru mohl tvůrce zadat pouze bezkontextovou gramatiku a generátor sám převede tuto gramatiku na program realizující syntaktickou analýzu zdrojového jazyka. Všechny tyto systémy však mají určitá omezení. Problém spočívá v kompromisu mezi množstvím práce, které dělá generátor kompilátoru automaticky, a pružností celého systému. Ilustrujme tento problém na příkladě lexikálního analyzátoru.

Většina systémů pro psaní překladačů dodává ve skutečnosti tentýž podprogram lexikální analýzy pro generovaný kompilátor, lišící se pouze v seznamu klíčových slov specifikovaných uživatelem. Pro většinu případů je toto řešení vyhovující, problém však nastane v případě nestandardní lexikální jednotky, např. identifikátoru, který může kromě číslic a písmen obsahovat i jiné znaky. I když existuje obecnější přístup k automatické konstrukci tohoto analyzátoru (reprezentovaný například generátorem `lex`, kterému se budeme věnovat podrobněji v článku 2.6), větší pružnost systému vyžaduje podrobnější specifikaci a tudíž i více práce.

K základním možnostem existujících generátorů překladačů patří:

- generátor lexikálního analyzátoru,
- generátor syntaktického analyzátoru a
- prostředky pro generování kódu.

Principy činnosti a výstavby obou generátorů analyzátorů jsou založeny na teorii formálních jazyků a gramatik. Podstatnou výhodou použití těchto generátorů je zvýšení spolehlivosti překladače. Mechanicky generované části překladače jsou daleko méně zdrojem chyb než části programované ručně.

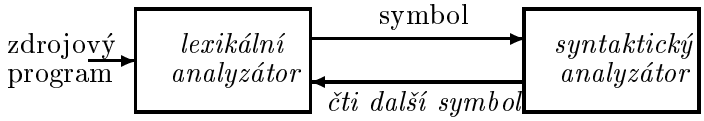
Jako prostředků usnadňujících generování kódu se v těchto systémech obvykle používá vyššího programovacího jazyka. Slouží ke specifikaci generování jak intermediárního kódu, tak i symbolických instrukcí nebo strojového jazyka. Ve tvaru např. sémantických podprogramů jsou pak tyto specifikace volány automaticky generovaným syntaktickým analyzátorem na vhodných místech. Mnoho systémů pro psaní překladačů používá také mechanismu pro zpracování rozhodovacích tabulek, které vybírají generovaný cílový kód. Tyto tabulky jsou spolu s jejich interpretem generovány na základě popisu vlastností cílového jazyka a tvoří součást výsledného kompilátoru.

Kapitola 2

Lexikální analýza

2.1 Činnost lexikálního analyzátoru

Lexikální analyzátor je první fází překladače. Jeho hlavním úkolem je číst znaky ze vstupu a na svůj výstup dávat symboly, které dále používá syntaktický analyzátor. Tato interakce, schematicky shrnutá na obr. 2.1, se běžně implementuje tak, že lexikální analyzátor vytvoříme jako podprogram nebo koprogram syntaktického analyzátoru. Po přijetí příkazu "dej další symbol" od syntaktického analyzátoru čte lexikální analyzátor vstupní znaky až do té doby, než může identifikovat další symbol.



Obr. 2.1: Interakce lexikálního a syntaktického analyzátoru

Vzhledem k tomu, že lexikální analyzátor je tou částí překladače, která čte zdrojový text, může na uživatelském rozhraní provádět i další úkoly. Jedním takovým úkolem je odstraňování poznámek a odsazovačů (mezer, tabelátorů a konců řádků) ze zdrojového programu. Dalším úkolem je udržování konzistence chybových hlášení překladače a zdrojového textu. Lexikální analyzátor může například sledovat počet načtených znaků konce řádku a umožnit ke každému chybovému hlášení připojení čísla příslušného řádku s chybou. V některých překladačích je lexikální analyzátor pověřen prováděním opisu zdrojového programu s vyznačenými chybovými hlášeními. Pokud zdrojový jazyk obsahuje některé funkce makroprocesoru, potom tyto funkce mohou být implementovány během lexikální analýzy.

Pro rozdělení analytické fáze překladu na lexikální analýzu a syntaktickou analýzu existuje několik důvodů.

1. Zřejmě nejpodstatnějším důvodem je jednodušší návrh překladače. Oddělení lexikální a syntaktické analýzy často umožňuje jednu nebo obě fáze zjednodušit. Například syntaktický analyzátor zahrnující i konvence pro poznámky a mezery je podstatně složitější než analyzátor, který předpokládá, že poznámky a mezery už byly odstraněny lexikálním analyzátorem.

SYMBOL	PŘÍKLADY LEXÉMŮ	NEFORMÁLNÍ POPIS VZORU
const	`const`	`const`
if	`if`	`if`
relation	`<, <=, =, <>, >, >=`	`<` nebo `<=` nebo `=` nebo `<>` nebo `>=` nebo `>`
id	`pi, count, D2`	písmeno následované písmeny a číslicemi
num	`3.1416, 0, 6.02E23`	libovolná číselná konstanta
literal	`"core dumped"`	libovolné znaky v uvozovkách kromě uvozovek

Obr. 2.2: Příklady symbolů

2. Zlepší se efektivita překladače. Oddělený lexikální analyzátor umožňuje použít specializované a potenciálně mnohem efektivnější algoritmy. Čtením zdrojového programu a jeho rozdělováním do symbolů se ztrácí mnoho času. Specializované techniky práce s vyrovnávací pamětí při čtení vstupních znaků mohou podstatně zvýšit výkonnost překladače.

3. Zvýší se přenositelnost překladače. Zvláštnosti vstupní abecedy a jiné anomálie konkrétních vstupních zařízení se mohou omezovat pouze na lexikální analyzátor. Například jazyk C umožňuje použití speciálních tříznakových kombinací pro znaky, které nebývají dostupné na některých klávesnicích ('??(' pro '[', '??<' pro '{' apod.).

Pro podporu automatizace vytváření oddělených lexikálních a syntaktických analyzátorů byly vytvořeny specializované prostředky. S programem Lex se seznámíme v této kapitole, programu Yacc bude věnována část kapitoly následující.

2.2 Základní pojmy

2.2.1 Symboly, vzory, lexémy

Když hovoříme o lexikální analýze, používáme výrazů *symbol*, *vzor* a *lexém* se specifickým významem. Příklady jejich použití ukazuje obrázek 2.2. Obecně existuje množina vstupních řetězců, pro které se na výstup dává týž symbol. Tato množina je popsána pravidlem zvaným vzor symbolu. Lexém je posloupnost znaků zdrojového programu, která odpovídá vzoru pro konkrétní symbol. Například v příkazu jazyka Pascal

```
const pi = 3.1416;
```

je podřetězec *pi* lexémem pro symbol *identifikátor*.

Symboly považujeme za terminální symboly gramatiky zdrojového jazyka. Lexémy odpovídající vzorům pro symboly představují řetězce znaků zdrojového programu, které můžeme považovat za jedinou lexikální jednotku.

V mnoha programovacích jazycích se za symboly považují následující konstrukce: klíčová slova, operátory, identifikátory, konstanty, řetězce (ve smyslu literálů) a interpunkční symboly jako závorky, čárky a středníky. Ve výše uvedeném příkladu se při výskytu posloupnosti znaků `pi` ve zdrojovém programu vrátí syntaktickému analyzátoru symbol reprezentující identifikátor. Vracení symbolů se často implementuje jako vracení celých čísel, která jsou symbolům přidělena (případně hodnot výčtového typu, pokud to implementační jazyk umožňuje).

Vzor je pravidlo popisující množinu lexémů, které mohou představovat ve zdrojovém programu konkrétní symbol. Vzor pro symbol **const** na obr. 2.2 je právě jediný řetězec `const`,

jímž je klíčové slovo označeno. Vzor pro symbol **relation** je množina relačních operátorů jazyka Pascal. Pro přesný popis mnohem složitějších symbolů jako je **id** (pro identifikátor) a **num** (pro číslo) budeme používat regulárních výrazů.

Některé jazykové konvence mají dopad na složitost lexikální analýzy. Jazyky jako Fortran vyžadují, aby určité konstrukce byly na pevné pozici ve vstupním řádku. Umístění lexému může být tedy důležité při určování správnosti zdrojového programu. Trend tvorby moderních programovacích jazyků směřuje ke vstupu ve volném formátu, který umožňuje umístění konstrukcí kdekoliv na vstupním řádku, takže tento aspekt lexikální analýzy se stává stále méně důležitým.

Zpracování mezer se značně jazyk od jazyka liší. V některých jazycích jako je Fortran, Basic nebo Algol 68 nejsou mezery v příkazech programu významné, až na mezery uvnitř literálových řetězců. Mohou být doplněny pro zvýšení čitelnosti programu. Konvence týkající se mezer mohou značně komplikovat úkol identifikace symbolů.

Populárním příkladem, který dokumentuje potenciální obtíže při rozpoznávání symbolů, je příkaz `DO` ve Fortranu. V příkazu

```
DO 5 I = 1.25
```

až do okamžiku, než uvidíme desetinnou tečku, nemůžeme poznat, že `DO` není klíčové slovo, ale část identifikátoru `DO5I`. Na druhé straně v příkazu

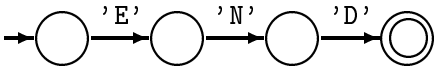
```
DO 5 I = 1,25
```

máme sedm symbolů, které odpovídají klíčovému slovu `DO`, návěští příkazu `5`, identifikátoru `I`, operátoru `=`, konstantě `1`, čárce a konstantě `25`. Zde si nemůžeme být až do výskytu čárky jistí, zda je `DO` klíčové slovo.

V mnoha jazycích jsou některé řetězce rezervovány, tj. jejich význam je předdefinován a nemůže být uživatelem změněn. Nejsou-li klíčová slova rezervována, musí klíčové slovo od uživatelem definovaného identifikátoru rozlišit lexikální analyzátor. V jazyce PL/I nejsou klíčová slova rezervovaná; pravidla pro rozlišení klíčových slov od identifikátorů jsou tedy značně komplikovaná, jak ukazuje následující příkaz PL/I:

```
IF THEN THEN THEN =  ELSE; ELSE ELSE = THEN;
```

Pro analýzu klíčových slov můžeme použít v podstatě dvou přístupů. Můžeme je definovat jako samostatné symboly se svou vlastní strukturou, např. klíčové slovo `END` jako řetězec



nebo můžeme pro klíčová slova použít stejného vzoru jako pro identifikátory a teprve po rozpoznání identifikátoru otestovat na základě tabulky klíčových slov, zda se jedná skutečně o identifikátor nebo o klíčové slovo a podle toho vrátit příslušný kód symbolu. Druhý přístup je výhodnější z hlediska složitosti automatu a pro většinu moderních jazyků zřejmě nemá smysl používat přístup první.

2.2.2 Atributy symbolů

Odpovídá-li vzoru více jak jeden lexém, musí lexikální analyzátor následujícím fázím překladače poskytnout informaci o tom, který konkrétní lexém byl rozpoznán. Například řetězcům `0` a `1` odpovídá vzor pro **num**, avšak pro generátor kódu je podstatné znát, o který řetězec se skutečně jedná.

Lexikální analyzátor shromažďuje informace o symbolech v atributech symbolů. Symboly mají vliv na rozhodování syntaktického analyzátoru; atributy ovlivňují překlad symbolů. V praxi má symbol často pouze jeden atribut — ukazatel na položku tabulky symbolů, která obsahuje informace o symbolu. Pro účely diagnostiky nás může zajímat jak lexém identifikátoru, tak i číslo řádku, na kterém se poprvé objevil. Obě tyto informace mohou být rovněž uloženy v položce tabulky symbolů pro identifikátor.

Příklad 2.1. Symboly a k nim příslušné hodnoty atributů pro příkaz jazyka Fortran

```
E = M * C ** 2
```

jsou uvedeny dále jako posloupnost dvojic:

```
<id, ukazatel na položku tabulky symbolů pro E>
<assign_op,>
<id, ukazatel na položku tabulky symbolů pro M>
<mult_op,>
<id, ukazatel na položku tabulky symbolů pro C>
<exp_op,>
<num, celočíselná hodnota 2>
```

Povšimněte si, že některé dvojice nemusejí obsahovat hodnotu atributu; první složka je dostatečná pro identifikaci lexému. V tomto malém příkladu dostal symbol **num** atribut s celočíselnou hodnotou. Překladač také může uložit řetězec znaků, který tvoří číslo, do tabulky symbolů a jako atribut symbolu **num** ponechat ukazatel na položku tabulky. ■

2.3 Vstup zdrojového textu

Na začátku této kapitoly jsme uvedli, že jedním z úkolů lexikálního analyzátoru je čtení znaků ze vstupního (zdrojového) souboru. Čtení znaků můžeme realizovat v nejjednodušším případě např. voláním standardní funkce `getchar()` jazyka C nebo procedury `read()` jazyka Pascal. Obecně se však jedná o podstatně složitější problém, a to z následujících důvodů:

- čtení po jednotlivých znacích může být značně neefektivní ve srovnání se čtením po řádcích nebo po velkých blocích textu, např. může představovat volání funkce jádra operačního systému se všemi kontrolami, které k tomu příslušejí. Analyzátor tedy musí zajistit nějakou správu vyrovnávacích pamětí, ze kterých se budou dále jednotlivé znaky odebírat. Ani prostředky vyrovnávaného vstupu dat, které poskytují standardní knihovny jazyka C, nejsou z hlediska efektivity dostatečné; v mnoha implementacích se čtená data kopírují až třikrát před tím, než je obdrží uživatelský program (z disku do vyrovnávací paměti operačního systému, dále do vyrovnávací paměti, která je částí struktury `FILE`, a nakonec do řetězce, který obsahuje lexém).

- při čtení zdrojového textu se může provádět jeho opis do výstupní tiskové sestavy, přičemž tento opis se může dále doplňovat o informace získané při překladu (úroveň zanoření závorkových struktur, adresy instrukcí apod.). I tehdy, když se opis celého zdrojového textu neprovádí, musí lexikální analyzátor udržovat pro účely hlášení chyb alespoň informaci o čísle zdrojového řádku, případně text aktuálního řádku a současnou pozici).

- v případě, že jazyk umožňuje vkládání částí zdrojového textu z jiných souborů, podmíněný překlad nebo práci s makrodefinicemi (např. jazyk C), je třeba tuto činnost, která může

být značně složitá, provést buď jako samostatný průchod před lexikální analýzou, nebo se musí provést zároveň s činností lexikálního analyzátoru, a to právě během čtení znaků.

- během analýzy často potřebujeme provést návrat ve vstupním souboru; v případě, že nám implementační jazyk návrat neumožňuje nebo jsou-li možnosti navracení omezené (např. funkce `ungetc()` jazyka C umožňuje vrátit pouze jediný znak), je třeba tuto akci provádět ve vlastní režii.

Čtení zdrojového textu je vhodné implementovat jako samostatný programový modul komunikující s lexikálním analyzátorem přes určité rozhraní. Oddělením činností spojených se čtením zdrojového textu můžeme dosáhnout větší přenositelnosti překladače, neboť většina systémově závislých operací se soustřeďuje právě do vstupního modulu.

Příklad 2.2. Následující program je velmi jednoduchým příkladem implementace vstupního modulu. Definuje funkci `getch()`, která poskytuje následující znak ve vstupním souboru, a funkci `ungetch()` pro návrat o znak zpět. Dále jsou k dispozici proměnné obsahující číslo současného zdrojového řádku, text tohoto řádku a ukazatel na znak, který bude zpracován jako následující. Tyto informace lze dále využít pro hlášení chyb.

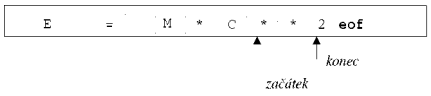
```
int  line = 0;            /* číslo zdrojového řádku   */
char source[ 256 ];       /* zdrojový řádek           */
char *gchptr = source;    /* ukazatel současné pozice */

int getch( void )     /* čtení jednoho znaku */
{
   char ch;
   if( *gchptr == '\0' ) {    /* jsme za koncem řádku     */
      gchptr = gets( source );
      if( gchptr == NULL )     /* konec zdrojového souboru */
         return( EOF );
      line++;
   }
   return (ch = *gchptr++) ? ch : '\n';
}

void ungetch( void )    /* návrat o znak zpět */
{
   if( gchptr != source ) {  /* nejsme na začátku řádku  */
      gchptr--;
}
```

■

V některých programovacích jazycích je často třeba, aby měl lexikální analyzátor možnost si prohlédnout několik znaků za lexémem ještě před tím, než může spolehlivě ohlásit, o který symbol se jedná. Podprogramy `getch` a `ungetch` z příkladu 2.2 například umožňovaly přečíst znaky nejvýše do konce řádku a pak je zase vrátit zpět. Vzhledem k tomu, že neustálým přesouváním znaků může docházet ke značným časovým ztrátám, používají se specializované techniky

pracující s vyrovnávacími pamětmi (v příkladu 2.2 jsme měli vyrovnávací paměť na jeden zdrojový řádek). Tyto techniky jsou obvykle značně závislé na vlastnostech konkrétního operačního systému, proto pouze naznačíme jednu z možností.

Pro vstup zdrojového textu můžeme využít vyrovnávací paměti rozdělené na dvě části o velikosti N znaků (viz obr. 2.3). Typická hodnota N je daná velikostí diskového bloku, např. 1024 nebo 4096 slabik. Do každé poloviny načteme N znaků textu, a to vždy jedním voláním operace čtení pro celý blok, ne pro jednotlivé znaky. Zbývá-li na vstupu méně než N znaků, uloží se do vyrovnávací paměti za poslední načtený znak speciální znak **eof**.



Obr. 2.3: Rozdělená vstupní vyrovnávací paměť

Pro přístup do vyrovnávací paměti budeme udržovat dva ukazatele Na počátku budou oba ukazatele ukazovat na tentýž znak; během analýzy bude jeden ukazatel označovat pozici prvního znaku lexému a druhý se bude přesunovat tak dlouho, až se nalezne konec lexému. Řetězec znaků mezi oběma ukazateli potom představuje současný lexém; po jeho zpracování se oba ukazatelé přesunou za konec lexému a činnost se opakuje.

Jestliže se ukazatel konce lexému má přesunout do pravé poloviny vyrovnávací paměti, naplní se pravá polovina dalšími N znaky. Má-li se ukazatel přesunout za pravý konec vyrovnávací paměti, naplní se levá polovina dalšími N znaky a ukazatel se přesune cyklicky na začátek vyrovnávací paměti.

Toto schéma umožňuje jen omezenou délku pohledu vpřed ve vstupním textu — omezení je dáno velikostí vyrovnávací paměti. Pokud však délka prohledávaného řetězce nepřekročí velikost vyrovnávací paměti, je vždy zajištěno, že se můžeme vrátit na začátek lexému. To je výhodné například tehdy, jestliže pro rozpoznání určité konstrukce potřebujeme znát širší kontext, v němž je tato konstrukce uvedena. V praxi se mohou používat některé další modifikace, které dále zvyšují efektivitu čtení zdrojového textu.

2.4 Specifikace a rozpoznávání symbolů

Při implementaci lexikálního analyzátoru vždy vycházíme z více či méně formálního popisu struktury jednotlivých lexikálních jednotek. Tento popis může být v jednom z následujících tvarů:

1. slovní popis,
2. regulární nebo lineární gramatika,
3. graf přechodů konečného automatu,
4. regulární výraz, resp. regulární definice.

Všechny tyto možnosti se v praxi vyskytují a až na případně možnou nejednoznačnost slovního popisu jsou rovnocenné. V dalších dvou odstavcích se budeme zabývat posledními dvěma variantami. Regulární nebo obecně lineární gramatiky lze snadno převést na konečný automat, podobně jako slovní popis struktury jazyka.

2.4.1 Regulární výrazy

Regulární výrazy jsou důležitou notací pro specifikaci vzorů symbolů. Každý vzor odpovídá množině řetězců, takže regulární výraz slouží vlastně jako pojmenování množiny řetězců. Článek 2.6 tuto notaci rozšiřuje na jazyk pro specifikaci lexikálních analyzátorů.

Regulární množiny byly formálně definovány v [16]. Pro naše účely si definici rozšíříme o některé velmi často se vyskytující konstrukce. Regulární výrazy nad abecedou Σ a jazyky jimi označované budeme definovat následujícím způsobem:

1. ϵ je regulární výraz označující $\{\epsilon\}$, tj. množinu obsahující prázdný řetězec.

2. Je-li a symbol v Σ, potom `a` je regulární výraz označující $\{a\}$, tj. množinu obsahující řetězec a. Ačkoliv pro tři různé významy používáme stejný zápis, je ve skutečnosti regulární výraz `a` odlišný od řetězce a nebo od symbolu a. Z kontextu bude vždy zřejmé, zda hovoříme o regulárním výrazu, řetězci nebo symbolu.

3. Jsou-li $a, b, c, \ldots$ symboly v Σ, potom `[abc...]` je regulární výraz označující jazyk $\{a, b, c, \ldots\}$. Tvoří-li symboly posloupnost, lze je zapsat jako interval, např. `[a-z]`.

4. Předpokládejme, že r a s jsou regulární výrazy, které označují jazyky $L(r)$ a $L(s)$. Potom

   a) `(`r`)|(`s`)` je regulární výraz označující $L(r) \cup L(s)$,

   b) `(`r`)(`s`)` je regulární výraz označující $L(r)L(s)$,

   c) `(`r`)*` je regulární výraz označující $(L(r))^*$,

   d) `(`r`)+` je regulární výraz označující $(L(r))^+$,

   e) `(`r`)?` je regulární výraz označující $L(r) \cup \{\epsilon\}$,

   f) `(`r`)` je regulární výraz označující $L(r)$. (Toto pravidlo říká, že kolem regulárního výrazu můžeme podle potřeby napsat dvojici závorek.)

Příklad 2.3. Jazyk tvořený řetězci nul a jedniček s lichou paritou (tj. s lichým počtem jedniček) můžeme popsat regulárním výrazem

```
0*1(0*10*1)*0*
```

■

Regulární výrazy mohou popisovat pouze relativně jednoduché konstrukce. Některé jazyky nelze regulárními výrazy popsat, například v následujících situacích:

- Regulární výrazy nelze použít k popisu vyvážených nebo vnořených konstrukcí. Například množina všech řetězců s vyváženými závorkami se nedá regulárním výrazem popsat, stejně jako zanořené poznámky v jazyce Modula-2. Na druhé straně lze takové množiny popsat bezkontextovou gramatikou.

- Regulárními výrazy nelze popsat opakované řetězce. Množina

$$\{wcw | w \text{ je řetězec symbolů } a \text{ a } b\}$$

  se nedá popsat regulárním výrazem ani bezkontextovou gramatikou.

- Regulární výrazy lze použít pouze k popisu pevného počtu opakování nebo nespecifikovaného počtu opakování dané konstrukce. Nelze porovnat dvě libovolná čísla, zda jsou stejná. Nemůžeme tedy pomocí regulárních výrazů popsat hollerithovské řetězce tvaru nHa1a2...an z prvních verzí jazyka Fortran, neboť počet znaků následujících za H musí odpovídat desítkovému číslu před H.

Vzhledem k tomu, že většina lexikálních konstrukcí běžných programovacích jazyků patří do třídy regulárních jazyků, je použití regulárních výrazů typické právě pro tuto oblast, neboť jejich analýza je podstatně jednodušší než analýza jazyků bezkontextových nebo kontextových.

2.4.2 Regulární definice

Pro účely zápisu bychom chtěli regulární výrazy pojmenovat a jejich jména použít v jiných regulárních výrazech, jako by to byly symboly. Je-li Σ abeceda základních symbolů, potom regulární definice je posloupnost definicí ve tvaru

$$\begin{aligned} d_1 &\longrightarrow r_1 \\ d_2 &\longrightarrow r_2 \\ &\dots \\ d_n &\longrightarrow r_n \end{aligned}$$

kde d_i jsou navzájem odlišná jména a r_i jsou regulární výrazy nad abecedou $\Sigma \cup \{d_1, d_2, \dots, d_{i-1}\}$, tj. z množiny základních symbolů a dříve definovaných jmen. Omezením r_i pouze na symboly množiny Σ a dříve definovaná jména můžeme vytvořit regulární výraz nad Σ pro každé r_i opakovaným nahrazováním jmen regulárních výrazů výrazy, které označují. Je-li r_i použito v d_j pro nějaké $j \geq i$, potom by mohlo být r_i definováno rekurzivně a tento proces nahrazování by se nezastavil.

Pro odlišení jmen od symbolů budeme psát jména v regulárních definicích kurzívou.

Příklad 2.4. Identifikátory jazyka Pascal můžeme popsat následující regulární definicí:

$$\begin{aligned} \mathit{letter} &\longrightarrow [\texttt{A} - \texttt{Za} - \texttt{z}] \\ \mathit{digit} &\longrightarrow [\texttt{0} - \texttt{9}] \\ \mathit{id} &\longrightarrow \mathit{letter}(\mathit{letter} | \mathit{digit})^* \end{aligned}$$

■

Příklad 2.5. Čísla bez znaménka v Pascalu jsou řetězce jako 5280, 39.37, 6.336E4 nebo 1.894E-4. Následující regulární definice je přesnou specifikací této třídy řetězců:

$$\begin{aligned} \textit{digits} &\longrightarrow [\texttt{0}-\texttt{9}]^+ \\ \textit{optional\_fraction} &\longrightarrow (\texttt{.}\ \textit{digits})? \\ \textit{optional\_exponent} &\longrightarrow (\texttt{E}\ (+|-)?\ \textit{digits})? \\ \textit{num} &\longrightarrow \textit{digits optional\_fraction optional\_exponent} \end{aligned}$$

Tato definice říká, že *optional_fraction* je buď desetinná tečka následovaná jednou nebo více číslicemi, nebo chybí (je to prázdný řetězec). *Optional_exponent*, pokud nechybí, je `E` následované volitelným `+` nebo `-` a jednou nebo více číslicemi. Povšimněte si, že za tečkou musí být alespoň jedna číslice, takže *num* neodpovídá řetězci `1.`, ale odpovídá řetězci `1.0`. ■

Regulární výrazy a regulární definice tvoří základní prostředek pro specifikaci lexikální struktury jazyka v systémech pro podporu návrhu překladačů, konkrétně v tzv. konstruktorech lexikálních analyzátorů. Na základě regulárních definic tyto konstruktory obvykle vytvoří odpovídající deterministický konečný automat reprezentovaný buď tabulkou přechodů a jejím interpretem nebo přímo programem realizujícím lexikální analýzu.

2.4.3 Konečné automaty

Dalším prostředkem, který lze využít jak pro specifikaci, tak i pro implementaci lexikálních analyzátorů, jsou konečné automaty. Pro naše potřeby vyjdeme z definice rozšířeného konečného automatu (viz [16]) jako pětice

$$(Q, \Sigma, f, q_0, F),$$

kde Q je konečná množina vnitřních stavů, Σ (neprázdná) vstupní abeceda, f přechodová funkce $f : Q \times (\Sigma \cup \{\epsilon\}) -> 2^Q$, $q_0 \in Q$ počáteční stav a $F \subset Q$ množina koncových stavů.

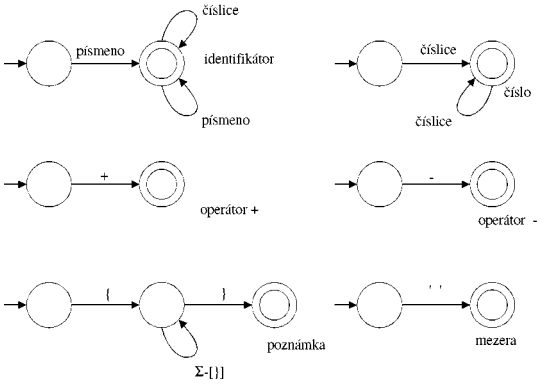
Pro všechny symboly jazyka můžeme sestrojit samostatné (obecně nedeterministické) automaty; všechny částečné automaty pak můžeme spojit do jediného automatu tak, že vytvoříme nový počáteční stav a pomocí ϵ-přechodů jej propojíme s počátečními stavy jednotlivých výchozích automatů. Takto získaný automat pak převedeme na deterministický, např. algoritmem uvedeným v [17]. Dostaneme výsledný deterministický konečný automat, který pak můžeme implementovat některou z dále uvedených metod.

Příklad 2.6. Jazyk obsahující identifikátory, celá čísla bez znaménka, operátory '+' a '–', poznámky a mezery můžeme popsat částečnými automaty podle obr. 2.4. Výsledný automat, který získáme jejich spojením, je uveden na obr. 2.5.

2.5 Implementace lexikálního analyzátoru

Výběr konkrétní metody implementace je závislý spíše na tom, zda máme k dispozici a chceme použít nějaký konstruktor (v tom případě bude zřejmě nejvýhodnější popis regulárními výrazy) nebo zda budeme analyzátor psát přímo v některém programovacím jazyku; dalším kritériem (někdy i rozhodujícím) mohou být i požadavky na efektivitu lexikálního analyzátoru, neboť lexikální analyzátor zpracovává zdrojový program znak po znaku a často tedy přímo určuje rychlost celého překladu. Pro vlastní implementaci můžeme použít jednu z následujících metod:

1. přímá implementace s využitím všech prostředků, které poskytuje implementační jazyk,



Obr. 2.4: Grafy částečných konečných automatů

2. implementace konečného automatu nebo

3. vytvoření analyzátoru konstruktorem.

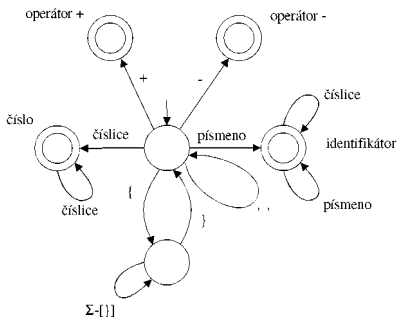
Z hlediska jednoduchosti je nejjednodušší použití konstruktoru a nejnáročnější přímá implementace, ovšem z hlediska efektivity překladu je pořadí obvykle přesně opačné. Z tohoto důvodu se často pro vývojové verze překladače použije konstruktoru, avšak pro definitivní překladač se lexikální analyzátor implementuje přímo.

V následujících odstavcích si předvedeme první dvě techniky na příkladu jazyka z obr. 2.5. Tento jazyk obsahuje identifikátory, celočíselné konstanty, operátory + a −, mezery a poznámky tvořené posloupností znaků uzavřených ve složených závorkách.

2.5.1 Přímá implementace

Přímá implementace lexikálního analyzátoru vychází z požadavků na maximální efektivitu jeho činnosti; využívá všech vhodných prostředků implementačního programovacího jazyka.

Příklad 2.7. Pro jazyk definovaný na obr. 2.5 ukážeme jednu z možných implementací lexikálního analyzátoru. Analyzátor bude představován funkcí `yylex` bez parametrů, která po každém zavolání vrátí kód následujícího symbolu. V případě identifikátoru ponechá v proměnné `yytext` příslušný lexém a v proměnné `yyleng` jeho délku ve znacích, pro číselnou konstantu po-



Obr. 2.5: Specifikace lexikálního analyzátoru pomocí DKA

nechá v proměnné `yyival` její binární hodnotu. (Použité názvy s výjimkou `yyival` jsou převzaty z pojmenování zavedeného v konstruktoru lex).

```
# include <stdio.h>
# include <ctype.h>
# define IDENT 256
# define NUM   257
char yytext[ 256 ];            /* lexém pro identifikátor */
int  yyleng;                   /* délka identifikátoru    */
int  yyival;                   /* hodnota čísla           */
int yylex(void)
{
     int ch;                   /* přečtený znak */
  START:
     while( (ch = getchar()) == ' ' );    /* vypuštění mezer */
     if ( isalpha(ch) ) {      /* zpracování identifikátoru */
        yyleng = 0;
        do {
           yytext[ yyleng++ ] = ch;
        } while ( isalnum(ch = getchar()) );
        yytext[ yyleng ] = '\0';
        ungetc( ch, stdin );   /* vrácení posledního znaku */
        return( IDENT );
     }
      else if ( isdigit(ch) ) {  /* zpracování čísla */
```

```
            yyival = 0;
            do {
               yyival = 10 * yyival + (ch - '0');
            } while ( isdigit(ch = getchar()) );
            ungetc( ch, stdin );   /* vrácení posledního znaku */
            return( NUM );
         }
          else if ( ch == '{' ) {    /* zpracování poznámky */
            while( (ch = getchar()) != '}' && ch != EOF );
            if ( ch == EOF ) {
               yyerror( "Neukončená poznámka" );
               return( EOF );
            }
            goto START;             /* pokračujeme dalším symbolem */
         }
          else                       /* ostatní znaky */
            return( ch );
   }
```

Kódování symbolů je zvoleno tak, aby jednoznakové symboly mohly být reprezentovány přímo kódem odpovídajícího znaku. Složené symboly pak mají přiděleny kódy počínaje hodnotou 256. Analyzátor předává informaci o konci zdrojového souboru rovněž jako symbol — jeho kód (`EOF`) je převzat ze standardního záhlaví `<stdio.h>` jazyka C. Je-li na vstupu zjištěn znak, kterým nezačíná žádný z definovaných symbolů, je analyzátorem jeho kód vrácen a chyba není hlášena — ohlásí se až při syntaktické analýze (neboť vrácený kód nemůže odpovídat kódu žádného z očekávaných symbolů na vstupu). Rovněž by bylo možné zjistit, zda se jedná o znak '+' nebo '−' a v případě, že tomu tak není, nahlásit chybu (např. "Neplatný znak"), znak vynechat a pokračovat v analýze. ■

2.5.2 Implementace lexikálního analyzátoru jako automatu se stavovým řízením

V případě, že je lexikální struktura jazyka popsána konečným automatem, můžeme implementovat přímo činnost tohoto automatu, a to buď pomocí tabulky přechodové funkce automatu nebo přímo přepisem automatu do programu. Konstruktory lexikálních analyzátorů používají především první variantu, neboť jak formát tabulky, tak i příslušný interpretační program mohou být standardizovány.

Příklad 2.8. Ukážeme implementaci lexikálního analyzátoru pro jazyk z obr. 2.5 do programu v jazyce C formou automatu.

```
# include <stdio.h>
# include <ctype.h>

# define IDENT 256
# define NUM   257

char yytext[ 256 ];             /* lexém pro identifikátor */
int  yyleng;                    /* délka identifikátoru    */
```

```
int  yyival;                    /* hodnota čísla           */

static int state;               /* současný stav automatu */

int next( int newst )           /* přechod do nového stavu */
{
    state = newst;
    return getchar();
}

int yylex( void )
{
     int ch = next(0);          /* současný znak na vstupu */
     for ( ;; )
        switch (state) {
           case 0:        /* stav 0 - startovací stav */
              if ( ch == ' ' )
                 ch = next(0);
              else if ( isalpha(ch) ) {
                 yyleng = 1;
                 yytext[ yyleng ] = ch;
                 ch = next( 1 );
              }
              else if ( isdigit(ch) ) {
                 yyival = ch - '0';
                 ch = next( 2 );
              }
              else if ( ch == '{' )
                 ch = next( 3 );
              else {
                 return ( ch );
           case 1:        /* stav 1 - analýza identifikátoru */
              if( isalnum(ch) ) {
                 yytext[ yyleng++ ] = ch;
                 ch = next( 1 );
              }
              else {
                 yytext[ yyleng ] = '\0';
                 ungetc( ch, stdin );
                 return( IDENT );
              }
           case 2:        /* stav 2 - analýza čísla */
              if( isdigit(ch) ) {
                 yyival = 10 * yyival + (ch - '0');
                 ch = next( 2 );
              }
              else {
```

```
                ungetc( ch, stdin );
                return( NUM );
             }
          case 3:        /* stav 3 - analýza poznámky */
             if( ch == EOF ) {
                yyerror( "Neukončená poznámka" );
                return( EOF );
             }
             else if( ch == '}' )
                ch = next( 0 );
             else
                ch = next( 3 );
       }
}
```

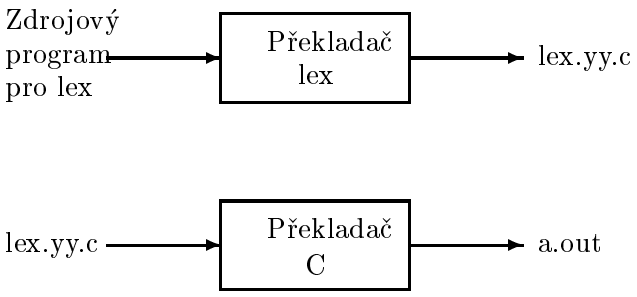
Volání funkce `next()` reprezentuje jednu hranu grafu přechodů; tato funkce nastaví nový stav automatu a přečte další znak ze vstupu. Povšimněte si, že uvedená implementace není zcela přesná, neboť zde nejsou realizovány koncové stavy reprezentující operátory '+' a '−'; v tomto smyslu se vlastně jedná o částečně optimalizovaný automat. I přesto je tato implementace mnohem méně efektivní než ta, která byla uvedena v předchozím odstavci. Je to způsobeno zejména neustálým rozhodováním o současném stavu a přechody, které nemění stav — např. ve stavu 0 při mezeře.

2.6 Lex — generátor lexikálních analyzátorů

2.6.1 Činnost programu `lex`

Pro vytváření lexikálních analyzátorů na základě speciálního zápisu založeného na regulárních výrazech bylo vytvořeno mnoho prostředků. S použitím regulárních výrazů a automatů pro specifikaci symbolů jsme se již seznámili. Nyní si uvedeme příklad prostředku, který by byl schopen vygenerovat lexikální analyzátor pouze na základě specifikace jazyka, konkrétně prostředek zvaný `lex`, který se široce využívá pro specifikaci lexikálních analyzátorů pro řadu jazyků. Budeme jej nazývat překladač `lex` a jeho vstupní specifikaci jazyk `lex`. Diskuse kolem tohoto jazyka nám umožní ukázat, jak lze specifikaci vzorů pomocí regulárních výrazů kombinovat s akcemi, tj. např. s vytvářením položek tabulky symbolů. Překladač `lex` byl implementován pod operačním systémem Unix (dále budeme popisovat právě tuto verzi), dnes je však dostupný i pod jinými operačními systémy, dokonce i v různých zdokonalených variantách.

`Lex` se obecně používá způsobem, který je znázorněn na obr. 2.6. Nejprve připravíme specifikaci lexikálního analyzátoru vytvořením zdrojového textu (např. v souboru `lex.l`) v jazyku `lex`. Potom soubor `lex.l` zpracujeme programem `lex` a tím vytvoříme program v C pod názvem `lex.yy.c`. Program `lex.yy.c` se skládá z tabulkové reprezentace grafu přechodů vytvořeného na základě regulárních výrazů obsažených v `lex.l` zároveň se standardními podprogramy, které tyto tabulky používají pro rozpoznávání symbolů. Akce spojené s regulárními výrazy v `lex.l` jsou reprezentovány úseky kódu v C; překladač `lex` je okopíruje přímo do souboru `lex.yy.c`. Konečně se soubor `lex.yy.c` zpracuje překladačem cc jazyka C, který vytvoří modul lexikálního analyzátoru a případně jej i sestaví s ostatními moduly do cílového programu – překladače.



Obr. 2.6: Vytvoření lexikálního analyzátoru programem `lex`

2.6.2 Struktura zdrojového textu

Program v jazyku `lex` se skládá ze tří částí, které jsou odděleny dvěma znaky `%` na začátku samostatného řádku:

```
deklarace
%%
překladová pravidla
%%
pomocné procedury
```

Oddíl deklarací obsahuje deklarace proměnných, pojmenovaných konstant a regulárních definicí. Deklarace, které se mají okopírovat do výstupního textu, musejí být uzavřeny do závorek `%{` a `%}`. Uvedené deklarace budou globální pro všechny funkce obsažené ve vygenerovaném programu. Regulární definice jsou příkazy ve tvaru

```
jméno   výraz
```

kde jméno je označení uvedeného regulárního výrazu, které může být v dalších výrazech použito ve tvaru {`jméno`}. Poznamenejme, že tyto definice jsou implementovány jako makra, takže případné chyby v jejich zápisu se projeví až při rozvoji v překladových pravidlech.

Druhý oddíl obsahuje vlastní definici lexikální struktury jazyka a činnosti analyzátoru formou překladových pravidel. Překladová pravidla pro `lex` jsou příkazy ve tvaru

p_1 $\quad$ $action_1$
p_2 $\quad$ $action_2$
...
p_n $\quad$ $action_n$

kde p_i jsou regulární výrazy a $action_i$ jsou části programu popisující činnost lexikálního analyzátoru po rozpoznání lexému odpovídajícího vzoru p_i (jediný příkaz jazyka C nebo blok příkazů ve složených závorkách). V jazyku `lex` se akce zapisují jako příkazy jazyka C, obecně by však zde mohl být libovolný jiný implementační jazyk. Regulární výrazy jsou v pravidle zapsány bezprostředně od začátku řádku a bez mezer, od akce jsou odděleny alespoň jednou mezerou nebo tabulátorem.

Třetí oddíl obsahuje libovolné pomocné procedury potřebné pro akce; překladač `lex` pouze zkopíruje veškerý text ze třetího oddílu do výstupního souboru. Tyto procedury se mohou také překládat samostatně a potom spojit s lexikálním analyzátorem.

VÝRAZ	POPIS	PŘÍKLAD
`const`	const	`const`
c	libovolný znak c, jež není operátorem	`a`
$\backslash c$	libovolný znak c	`\*`
"s"	řetězec s libovolných znaků	`"**"`
.	jakýkoliv znak kromě konce řádku	`a.*b`
^	začátek řádku	`^abc`
$	konec řádku	`abc$`
$[s]$	libovolný znak z množiny s	`[abc]`
$[\hat{}s]$	libovolný znak, který není v množině s	`[^abc]`
$r*$	0 nebo více r	`a*`
$r+$	1 nebo více r	`a+`
$r?$	0 nebo jeden r	`a?`
$r\{m,n\}$	m až n výskytů r	`a{1,5}`
r_1r_2	r_1 následovaný r_2	`ab`
$r_1\|r_2$	r_1 nebo r_2	`a|b`
(r)	r	`(a|b)`
r_1/r_2	r_1, pokud za ním následuje r_2	`abc/123`

Obr. 2.7: Regulární výrazy jazyka `lex`

2.6.3 Zápis regulárních výrazů

Jazyk `lex` umožňuje podstatně komplikovanější zápis regulárních výrazů, jak ukazuje tabulka na obr. 2.7. Symbol c v tabulce označuje jeden znak, r regulární výraz a s řetězec znaků. Zápis $\backslash c$, resp. "s" se používá tehdy, jestliže potřebujeme uvést některý ze speciálních znaků jazyka `lex` v jeho původním významu; jedná se o znaky `\ " . ^ $ [ ] * + ? { } |` a `/`. Zápis se zpětným lomítkem rovněž umožňuje zadat speciální řídicí znaky písmenem (např. `\t`, `\n`) nebo osmičkovým kódem (`\011`, `\015`).

2.6.4 Komunikace s okolím

Lexikální analyzátor vytvořený programem `lex` spolupracuje se syntaktickým analyzátorem následujícím způsobem. Po vyvolání funkce `yylex()` ze syntaktického analyzátoru začne lexikální analyzátor číst zbývající vstup po znacích až do okamžiku, kdy najde nejdelší prefix vstupního textu odpovídající jednomu z regulárních výrazů p_i. Potom provede akci $action_i$. Obvykle $action_i$ provede na konci příkaz `return( symbol )`, kterým vrátí řízení syntaktickému analyzátoru a zároveň předá kód přečteného symbolu. Pokud akce nekončí příkazem návratu, pokračuje lexikální analyzátor ve vyhledávání dalších symbolů až po dosažení akce, která způsobí návrat do syntaktického analyzátoru, nebo do nalezení konce vstupního souboru. Opakované vyhledávání lexémů až do explicitního návratu umožňuje lexikálnímu analyzátoru výhodně zpracovávat mezery a poznámky.

Lexikální analyzátor vrací syntaktickému analyzátoru jedinou hodnotu — kód symbolu. Pro předání hodnoty atributu s informacemi o lexému jsou k dispozici další proměnné:

```
int   yylineno;  /* číslo současného vstupního řádku  */
char  yytext[ ]; /* text naposledy přečteného lexému  */
int   yyleng;    /* délka naposledy přečteného lexému */
```

Navíc jsou zpřístupněny další proměnné, které umožňují měnit přiřazení vstupního a výstupního souboru pro lexikální analyzátor

```
FILE *yyin;      /* vstupní soubor  - implicitně stdin  */
FILE *yyout;     /* výstupní soubor - implicitně stdout */
```

Pro čtení jednoho znaku ze vstupního souboru a zápis jednoho znaku na výstup jsou k dispozici makra `input()` a `output()`, která je možno podle potřeby předefinovat. Výstupní soubor má význam tehdy, jestliže používáme `lex` pro vytvoření tzv. filtru, tj. programu, který čte vstupní soubor, provádí v něm určité transformace a transformovaný text zapisuje na výstup. Analyzátor vytvořený programem `lex` v případě, že část vstupního textu nelze přiřadit žádné z uvedených regulárních definic, tento text opíše do výstupního souboru `yyout`. Na to je třeba naopak pamatovat při návrhu skutečného lexikálního analyzátoru, kdy musí být pokryty skutečně všechny možné posloupnosti znaků na vstupu regulárními definicemi. Jinak by se např. v případě chybně zapsaného symbolu mohly na standardním výstupu objevit neočekávané texty.

Příklad 2.9. Poslední příklad této kapitoly ukazuje zápis lexikálního analyzátoru jazyka z obr. 2.5 prostředky konstruktoru `lex`.

```
%{
# include <stdlib.h>          /* pro funkci atoi() */
# define IDENT 256
# define NUM   257
int yyival;                   /* atribut symbolu NUM */
%}
                              /* regulární výrazy */
delim       [ \t\n]
ws          {delim}+|\{[↑}]*\}
letter      [A-Za-z]
digit       [0-9]
id          {letter}({letter}|{digit})*
number      {digit}+

%%
{ws}        {/* žádná akce a bez návratu */}
{id}        return(IDENT);
{number}    {yyival = atoi( yytext ); return(NUM);}
.           return(yytext[0]);
```

Obr. 2.8: Program v jazyce `lex`

První výraz v části překladových pravidel udává, že po rozpoznání `ws`, tj. maximální posloupnosti mezer, tabelátorů, konců řádků a poznámek, se neprovede žádná akce a tedy že se bude pokračovat čtením dalšího symbolu. V pravidle pro `id` obsahuje příslušná akce pouze návrat z lexikální analýzy a předání kódu symbolu `IDENT` jako návratové hodnoty. Pravidlo pro `number` nejprve převede textovou reprezentaci čísla z proměnné `yytext` na binární hodnotu standardní funkcí `atoi()` a vrátí kód symbolu `NUM`. Poznamenejme, že proměnná `yyival`, do níž se hodnota čísla ukládá, není definována programem `lex` a její definice tedy musí být uvedena v části deklarací. ■

Příklad 2.10. V příkladu 2.9 jsme si předvedli zápis lexikálního analyzátoru, u něhož jsme předpokládali opakované volání, vždy pro získání jediného vstupního symbolu. Poněkud jiným způsobem se v jazyku `lex` vytvářejí filtry, které — jak jsme již uvedli — pouze transformují vstupní text a zapisují jej na výstup. Celá činnost filtru se tedy může provést v rámci jediného volání funkce `yylex()`. Následující program bude představovat filtr, který ze vstupního souboru vypustí všechny nadbytečné mezery a tabulátory.

```
%%
[ \t]+     { output(' '); }
```

Tato specifikace ze vstupního souboru vybírá pouze posloupnosti mezer a tabulátorů, které zkracuje na jedinou mezeru. Všechny ostatní znaky se přenášejí beze změny na výstup. ■

2.7 Zotavení po chybě v lexikální analýze

Přímo v lexikální analýze se rozpoznává pouze málo chyb, neboť lexikální analyzátor má na zdrojový program příliš omezený pohled. Pokud se ve zdrojovém programu v jazyce C objeví poprvé řetězec `fi` v kontextu

```
fi ( a == f(x) ) ...
```

nemůže lexikální analyzátor říci, zda `fi` je chybně napsané klíčové slovo `if` nebo nedeklarovaný identifikátor funkce. Vzhledem k tomu, že `fi` je platný identifikátor, musí lexikální analyzátor vrátit symbol pro identifikátor a nechat zpracování chyby na některé další fázi překladače.

Předpokládejme však, že se naskytla situace, ve které není lexikální analyzátor schopen pracovat, neboť žádný ze vzorů pro symboly neodpovídá prefixu zbývajícího vstupu. Snad nejsnadnější strategií zotavení je metoda, kdy ze zbývajícího vstupu vypouštíme znaky tak dlouho, až se lexikálnímu analyzátoru podaří rozpoznat další správně vytvořený symbol. Další možností je, že lexikální analyzátor, aniž nahlásí chybu, vrátí kód speciálního terminálního symbolu, který není obsažený v gramatice jazyka, a nechá hlášení chyby a zotavení na syntaktický analyzátor. Obě metody se v praxi běžně používají a jsou obvykle dostatečně účinné.

Jiné možné činnosti při zotavení z chyby jsou:

- vypuštění přebývajícího znaku,
- vložení chybějícího znaku,
- náhrada nesprávného znaku správným,
- vzájemná výměna dvou sousedních znaků.

Podobnými chybovými transformacemi se můžeme pokoušet opravit chybu. Nejjednodušší takovou strategií je zjišťování, zda se nedá použitím právě jedné transformace převést zbývající vstup na platný lexém. Tato strategie předpokládá, že většina lexikálních chyb je výsledkem jediné chybové transformace (např. překlepu při pořizování zdrojového textu); takový předpoklad obvykle (ale ne vždy) odpovídá praxi.

Jedním ze způsobů nalezení chyb v programu je výpočet minimálního počtu chybových transformací požadovaných pro převod chybného programu na syntakticky správný program. Říkáme, že chybný program obsahuje k chyb, pokud nejkratší posloupnost chybových transformací, která jej zobrazuje na nějaký platný program, má délku k. Oprava chyb pomocí minimální vzdálenosti

je vhodný teoretický nástroj, avšak v praxi se obecně nepoužívá pro její velmi náročnou implementaci. Několik experimentálních překladačů však používalo kritéria minimální vzdálenosti pro lokální opravy.

Kapitola 3

Syntaktická analýza

3.1 Činnost syntaktického analyzátoru

Během syntaktické analýzy se překladač snaží zjistit, zda zdrojový text tvoří větu odpovídající gramatice překládaného jazyka. K tomu využívá posloupnost lexikálních symbolů získanou jako výsledek lexikální analýzy. Pokud text obsahuje nějaké chyby, překladač je nahlásí a obvykle provede určité zotavení tak, aby i při výskytu chyb mohl pokračovat dále v činnosti a odhalit případné další chyby.

Při implementaci překladače se obvykle používá jednoho ze dvou základních přístupů — překladu *shora dolů* nebo *zdola nahoru*. Tyto názvy odpovídají postupu při vytváření derivačního stromu; při překladu shora dolů vycházíme ze startovacího symbolu gramatiky a snažíme se postupnou expanzí nonterminálních symbolů dospět až k terminálním symbolům odpovídajícím posloupnosti lexikálních symbolů na vstupu, při překladu zdola nahoru se naopak snažíme posloupnost terminálních symbolů ze vstupu redukovat až na startovací nonterminál. Uvedeným dvěma přístupům odpovídají také dvě základní třídy gramatik, konkrétně LL a LR gramatiky, které popisují určité dostatečně velké podmnožiny bezkontextových jazyků. Často se pro analyzátory implementované ručně využívá LL gramatik; analyzátory větší třídy LR jazyků se obvykle vytvářejí automatizovanými prostředky.

V praktické implementaci syntaktického analyzátoru obvykle požadujeme více než jenom informaci o syntaktické správnosti zdrojového programu. Výstupem analyzátoru bude určitá reprezentace zdrojového textu, která bude obsahovat pouze informace podstatné pro další průběh překladu. Touto reprezentací může být například derivační strom nebo obecně určitá posloupnost akcí, které vytvářejí vnitřní reprezentaci struktury zdrojového programu a uchovávají informace o sémantice těchto struktur (atributy — jména identifikátorů, hodnoty literálů apod.). Uchované informace pak využívá sémantická analýza pro vyhodnocení těch závislostí, které nelze popsat prostředky bezkontextových gramatik.

3.2 Syntaktická analýza shora dolů

V této části se budeme zabývat základními principy překladu shora dolů a implementací odpovídajícího syntaktického analyzátoru, nazývaného často prediktivní syntaktický analyzátor.

Překlad shora dolů můžeme popsat buď jako proces hledání levé derivace vstupního řetězce, nebo jako proces vytváření derivačního stromu počínaje jeho kořenem. Tento proces může být realizován obecně metodou "pokusu a omylu", kdy se snažíme v určitém bodě překladu aplikovat

postupně jednotlivá pravidla gramatiky a v případě, že je aplikace určitého pravidla neúspěšná, provedeme návrat do bodu, ze kterého lze pokračovat dále volbou jiné varianty. Tento rekurzivní postup se nazývá *syntaktická analýza s návraty*; je značně neefektivní a pro účely překladu programovacích jazyků nevhodný. Naštěstí většina běžných konstrukcí programovacích jazyků je taková, že umožňují přímočarou analýzu bez návratů.

3.2.1 Množiny FIRST a FOLLOW

Konstrukce prediktivního analyzátoru je založena na dvou funkcích spojených s gramatikou G. Tyto funkce, $FIRST$ a $FOLLOW$, umožňují definovat řídicí tabulku pro deterministický zásobníkový automat. Množiny symbolů získané funkcí $FIRST$ i $FOLLOW$ lze také použít jako synchronizační množiny pro zotavení.

Je-li α řetězec symbolů gramatiky, potom $FIRST(\alpha)$ je množina terminálních symbolů, jimiž mohou začínat řetězce derivované z α. Pokud $\alpha \stackrel{\star}{\Rightarrow} \epsilon$, je ϵ rovněž ve $FIRST(\alpha)$.

Množinu $FOLLOW(A)$ pro nonterminál A definujeme jako množinu všech terminálních symbolů a, které se mohou vyskytovat bezprostředně vpravo od A v nějaké větné formě, tj. množina takových terminálních symbolů, pro něž existuje derivace ve tvaru $S \stackrel{\star}{\Rightarrow} \alpha A a \beta$ pro nějaké α a β. Povšimněte si, že během derivace mohou mezi A a a být nějaké symboly. Pokud je tomu tak, pak tyto symboly derivují prázdný řetězec ϵ a vymizí. Může-li být A nejpravějším symbolem v nějaké větné formě, je ve $FOLLOW(A)$ rovněž symbol \$, který představuje konec vstupního řetězce.

Množiny $FIRST(X)$ pro všechny symboly X gramatiky vypočteme aplikací následujících pravidel opakovaně tak dlouho, až nelze do žádné množiny $FIRST$ přidat další terminální symbol nebo ϵ.

1. Je-li X terminální symbol, potom $FIRST(X)$ je rovno $\{X\}$.
2. Je-li $X \to \epsilon$ pravidlo, potom přidáme do $FIRST(X)$ symbol ϵ.
3. Je-li X nonterminál a $X \to Y_1 Y_2 \cdots Y_k$ pravidlo, potom přidáme do $FIRST(X)$ symbol a, jestliže pro nějaké i je $a \in FIRST(Y_i)$ a ϵ je ve všech množinách $FIRST(Y_1)$, ..., $FIRST(Y_{i-1})$, tj. jestliže $Y_1 \cdots Y_{i-1} \stackrel{\star}{\Rightarrow} \epsilon$. Je-li ϵ ve $FIRST(Y_j)$ pro všechna $j = 1, 2, \ldots, k$, potom přidáme ϵ do $FIRST(X)$. Například všechny terminální symboly z $FIRST(Y_1)$ jsou určitě ve $FIRST(X)$. Pokud Y_1 nederivuje ϵ, nepřidáme do $FIRST(X)$ již nic dalšího, ale jestliže $Y_1 \stackrel{\star}{\Rightarrow} \epsilon$, přidáme $FIRST(Y_2)$ atd.

Nyní můžeme vypočítat $FIRST(X)$ pro libovolný řetězec $X_1 X_2 \cdots X_n$ následujícím postupem. Přidáme do $FIRST(X_1 X_2 \cdots X_n)$ všechny symboly z $FIRST(X_1)$ různé od ϵ. Pokud je ϵ ve $FIRST(X_1)$, přidáme rovněž symboly z $FIRST(X_2)$, je-li ϵ ve $FIRST(X_1)$ i $FIRST(X_2)$, přidáme symboly z $FIRST(X_3)$ atd. Konečně přidáme do $FIRST(X_1 X_2 \cdots X_n)$ symbol ϵ, pokud všechny množiny $FIRST(X_i)$ obsahují ϵ.

Výpočet množin $FOLLOW(A)$ pro všechny nonterminály A provedeme aplikací následujících pravidel opakovanou tak dlouho, až nelze do žádné množiny $FOLLOW$ přidat další symbol.

1. Do $FOLLOW(S)$, kde S je startovací symbol gramatiky, vložíme symbol \$ označující konec vstupního řetězce.
2. Máme-li pravidlo $A \to \alpha B \beta$, potom vše z množiny $FIRST(\beta)$ kromě ϵ se umístí do $FOLLOW(B)$.

3. Máme-li pravidlo $A \to \alpha B$ nebo $A \to \alpha B \beta$ kde $FIRST(\beta)$ obsahuje ϵ (tj. $\beta \stackrel{\star}{\Rightarrow} \epsilon$), potom prvky z množiny $FOLLOW(A)$ jsou obsaženy zároveň v množině $FOLLOW(B)$.

Příklad 3.1. Uvažujme gramatiku

$$\begin{array}{lrcl}
(1) & E & \to & TE' \\
(2) & E' & \to & +TE' \\
(3) & & | & \epsilon \\
(4) & T & \to & FT' \\
(5) & T' & \to & *FT' \\
(6) & & | & \epsilon \\
(7) & F & \to & (E) \\
(8) & & | & \mathbf{id}
\end{array}$$

Potom

$$\begin{array}{l}
FIRST(E) = FIRST(T) = FIRST(F) = \{(, \mathbf{id}\} \\
FIRST(E') = \{+, \epsilon\} \\
FIRST(T') = \{*, \epsilon\} \\
FOLLOW(E) = FOLLOW(E') = \{), \$\} \\
FOLLOW(F) = \{+, *,), \$\}
\end{array}$$

Například **id** a levá závorka se přidaly do $FIRST(F)$ na základě pravidla (3) z definice $FIRST$ v obou případech s $i = 1$, neboť $FIRST(\mathbf{id}) = \{\mathbf{id}\}$ a $FIRST('(') = \{(\}$ podle pravidla (1). Potom podle pravidla (3) s $i = 1$ z pravidla $T \to FT'$ plyne, že **id** a levá závorka jsou rovněž ve $FIRST(T)$. Dále je například podle pravidla (2) symbol ϵ prvkem $FIRST(E')$.

Výpočet množin $FOLLOW$ zahájíme vložením \$ do $FOLLOW(E)$ podle pravidla (1). Podle (2) s pravidlem $F \to (E)$ je ve $FOLLOW(E)$ také pravá závorka. Aplikace (3) na pravidlo $E \to TE'$ vede k tomu, že \$ a pravá závorka jsou ve $FOLLOW(E')$. Vzhledem k tomu, že $E' \stackrel{\star}{\Rightarrow} \epsilon$, jsou také ve $FOLLOW(T)$. Jako poslední příklad aplikace pravidel pro $FOLLOW$ uvažujme případ $T \to TE'$ v pravidle (2), podle něhož všechno z $FIRST(E')$ s výjimkou ϵ se musí umístit do $FOLLOW(T)$. To, že \$ je ve $FOLLOW(T)$, jsme již zjistili. ■

3.2.2 Konstrukce rozkladových tabulek

Syntaktický analyzátor pracující metodou shora dolů můžeme popsat jako zásobníkový automat tvořený vstupní páskou, zásobníkem, výstupní páskou a rozkladovou tabulkou. Automat čte symboly ze vstupní pásky a na výstupní pásku zapisuje čísla aplikovaných pravidel gramatiky — *levý rozklad* vstupní věty. Konfigurace tohoto automatu je dána trojicí

$$(x, X\alpha, \pi),$$

kde x je nepřečtená část vstupního řetězce, $X\alpha$ obsah zásobníku (se symbolem X na vrcholu) a π je obsah výstupní pásky. Automat začíná pracovat v počáteční konfiguraci

$$(w, S\#, \epsilon),$$

kde w je vstupní řetězec, S startovací nonterminál a # speciální zásobníkový symbol označující dno zásobníku. Pokud automat přijme vstupní řetězec w, dostane se do koncové konfigurace

$$(\epsilon, \#, \pi),$$

kde π je levý rozklad.

Rozkladová tabulka reprezentuje zobrazení

$$M : (\Sigma \cup N \cup \{\#\} \times (\Sigma \cup \{\$\}) \to \{\textbf{expand } 1, \textbf{expand } 2, \ldots, \textbf{expand } n, \textbf{pop}, \textbf{accept}, \textbf{error}\}$$

kde význam jednotlivých akcí je následující:

- **expand** i Je-li $p_i : A \to \alpha$ i-té pravidlo gramatiky, na vrcholu zásobníku je nonterminál A, na vstupu symbol a a $M[A, a] =$ **expand** i, provede automat přechod

  $$(ax, A\beta, \pi) \vdash (ax, \alpha\beta, \pi\, i)$$

  tj. nonterminál A se na vrcholu zásobníku nahradí pravou stranou α pravidla p_i a na výstup se dá číslo použitého pravidla i.

- **pop** Je-li na vstupu i na vrcholu zásobníku týž terminální symbol a, provede automat přechod

  $$(ax, a\beta, \pi) \vdash (x, \beta, \pi)$$

  tj. symbol a se odstraní z vrcholu zásobníku i ze vstupu.

- **accept** Akce **accept** představuje přijetí vstupního řetězce v koncové konfiguraci automatu, přičemž výstupní řetězec obsahuje úplný levý rozklad vstupní věty.

- **error** Akce **error** nastane tehdy, jestliže vstupní řetězec není prvkem jazyka, takže automat nemůže dále pokračovat v činnosti.

Příklad 3.2. Rozkladová tabulka deterministického zásobníkového automatu pro gramatiku

$$\begin{array}{lrcl} (1) & S & \to & aAS \\ (2) & S & \to & b \\ (3) & A & \to & a \\ (4) & A & \to & bSA \end{array}$$

bude mít následující tvar (akce **expand** i je zapsána jako **e**i, akce **accept** jako **acc** a prázdná políčka představují akci **error**):

ZÁSOBNÍK	VSTUPNÍ SYMBOL		
	a	b	\$
S	**e**1	**e**2	
A	**e**3	**e**4	
a	**pop**		
b		**pop**	
#			**acc**

Pro vstupní řetězec *abbab* potom můžeme vytvořit následující posloupnost přechodů automatu:

$$(abbab\$, S\#, \epsilon) \overset{e1}{\vdash} (abbab\$, aAS\#, 1) \overset{pop}{\vdash} (bbab\$, AS\#, 1) \overset{e4}{\vdash} (bbab\$, bSAS\#, 14) \overset{pop}{\vdash}$$
$$(bab, SAS\#, 14) \overset{e2}{\vdash} (bab\$, bAS\#, 142) \overset{pop}{\vdash} (ab\$, AS\#, 142) \overset{e3}{\vdash} (ab\$, aS\#, 1423) \overset{pop}{\vdash}$$
$$(b\$, S\#, 1423) \overset{e2}{\vdash} (b\$, b\#, 14232) \overset{pop}{\vdash} (\$, \#, 14232),$$

která nám dá rozklad věty *abbab* ve tvaru 14232. ■

Pro konstrukci rozkladové tabulky deterministického zásobníkového automatu ke gramatice G můžeme využít algoritmu 3.1. Je založen na následující myšlence. Předpokládejme, že $A \to \alpha$ je pravidlo a že a je ve $FIRST(\alpha)$. Potom, je-li současným vstupním symbolem a, provede analyzátor expanzi A na α. Jediná komplikace nastane, pokud $\alpha = \epsilon$ nebo $\alpha \stackrel{*}{\Rightarrow} \epsilon$. V tom případě musíme opět expandovat A na α, je-li současný vstupní symbol ve $FOLLOW(A)$ nebo byl-li dosažen konec vstupního řetězce (symbol \$) a \$ je ve $FOLLOW(A)$. Akce **pop** se bude provádět tehdy, je-li na vrcholu zásobníku i na vstupu týž terminální symbol a akce **accept** nastane v situaci, kdy bude vstupní řetězec vyčerpán (na vstupu bude ukončovací symbol \$) a zásobník vyprázdněn (na vrcholu bude symbol #).

Algoritmus 3.1. (Konstrukce rozkladové tabulky prediktivního analyzátoru)
Vstup. Gramatika G.
Výstup. Rozkladová tabulka M.
Metoda.

1. Pro všechna pravidla p_i tvaru $A \to \alpha$ proveď kroky 2 a 3.
2. Pro všechny terminální symboly a ve $FIRST(\alpha)$ přidej **expand** i do $M[A, a]$.
3. Je-li ϵ ve $FIRST(\alpha)$, přidej **expand** i do $M[A, b]$ pro všechny terminální symboly b z množiny $FOLLOW(A)$. Pokud ϵ je ve $FIRST(\alpha)$ a \$ ve $FOLLOW(A)$, přidej **expand** i do $M[A, \$]$.
4. Pro všechny terminální symboly a přidej **pop** do $M[a, a]$.
5. Nastav $M[\#, \$]$ na **pop**.
6. Všechny nedefinované položky v M nastav na **error**.

Příklad 3.3. Použijme algoritmus 3.1 na gramatiku z příkladu 3.1. Vzhledem k tomu, že $FIRST(TE') = FIRST(T) = \{(, \mathbf{id}\}$, budou položky $M[E, (]$ a $M[E, \mathbf{id}]$ obsahovat **expand** 1.

Pravidlo $E' \to +TE'$ vede k tomu, že $M[E', +]$ bude obsahovat **expand** 2. Pravidlo $E' \to \epsilon$ vede dále k tomu, že $M[E',)]$ a $M[E', \$]$ budou obsahovat **expand** 3, neboť $FOLLOW(E') = \{), \$\}$.

Celá rozkladová tabulka vytvořená algoritmem 3.1 je na obr. 3.1.

3.2.3 LL(1) gramatiky

Algoritmus 3.1 lze aplikovat na libovolnou gramatiku G a získat tak rozkladovou tabulku M. Pro některé gramatiky se však může stát, že v některých položkách rozkladové tabulky budeme mít více konfliktních akcí. Například je-li gramatika G zleva rekurzivní nebo nejednoznačná, bude tabulka M obsahovat alespoň jednu násobně definovanou položku.

Gramatika, jejíž rozkladová tabulka neobsahuje násobně definované položky, se nazývá *LL(1) gramatika*. První "L" v názvu znamená, že se vstupní text prohlíží zleva doprava, druhé "L" představuje vytváření levého rozkladu a "1" vyjadřuje počet symbolů ve vstupním textu, které potřebujeme znát při rozhodování o průběhu analýzy. Lze ukázat, že algoritmus 3.1 pro všechny LL(1) gramatiky G vede k rozkladové tabulce deterministického zásobníkového automatu, který přijímá právě jazyk $L(G)$.

ZÁSOBNÍK	VSTUPNÍ SYMBOL					
	id	+	*	(	)	$
E	**e**1			**e**1		
E'		**e**2			**e**3	**e**3
T	**e**4			**e**4		
T'		**e**6	**e**5		**e**6	**e**6
F	**e**8			**e**7		
id	**pop**					
+		**pop**				
*			**pop**			
(				**pop**		
)					**pop**	
$						**acc**

Obr. 3.1: Rozkladová tabulka prediktivního analyzátoru

Z definice LL(1) gramatiky (viz [16]) vyplývá několik vlastností, které umožňují rozhodnout, zda daná gramatika je či není typu LL(1). Následující dvě vlastnosti musí každá LL(1) gramatika nutně splňovat:

Nechť $A \to \alpha_1|\alpha_2|\ldots|\alpha_n$ jsou všechna A-pravidla gramatiky G. Potom:

- *Vlastnost FF.* Množiny $FIRST$ všech pravých stran musejí být po dvojicích disjunktní, tj.
$$FIRST(\alpha_i) \cap FIRST(\alpha_j) = \emptyset \text{ pro } i \neq j$$
- *Vlastnost FFL.* Je-li dále pro nějaké i $\alpha_i \stackrel{\star}{\Rightarrow} \epsilon$, musí být $FOLLOW(A)$ po dvojicích disjunktní s množinami $FIRST$ zbývajících pravých stran, tj.
$$FIRST(\alpha_j) \cap FOLLOW(A) = \emptyset \text{ pro } i \neq j$$

Z uvedených pravidel například vyplývá, že LL(1) gramatika nemůže obsahovat levou rekurzi, neboť by pro některý nonterminál A takový, že $A \stackrel{+}{\Rightarrow} A\alpha, \alpha \in (N \cup \Sigma)^*$, byla porušena podmínka FF. Například je-li v gramatice přímo pravidlo $A \to A\alpha|\beta$, potom $FIRST(\beta) \subseteq FIRST(A\alpha)$.

3.2.4 Transformace na LL(1) gramatiku

V mnoha případech není výchozí gramatika, pro kterou chceme vytvořit syntaktický analyzátor, typu LL(1). To znamená, že v ní existují pravidla, která porušují některou z podmínek FF nebo FFL. Transformaci takové gramatiky na typ LL(1) nám mohou umožnit následující postupy (podrobnější popis uveden v [16]):

- *Odstranění levé rekurze* Jak již bylo uvedeno výše, gramatika, která obsahuje levou rekurzi, nemůže být typu LL(1). Obecně můžeme zleva rekurzivní pravidlo zapsat jako
$$A \to A\alpha_1 \mid A\alpha_2 \mid \cdots \mid A\alpha_n \mid \beta_1 \mid \cdots \mid \beta_m$$

kde řetězce β_i nezačínají nonterminálem A. Takové pravidlo můžeme přepsat zavedením nového nonterminálu A' jako

$$A \to \beta_1 A' \mid \beta_2 A' \mid \cdots \mid \beta_m A'$$

$$A' \to \alpha_1 A' \mid \alpha_2 A' \mid \cdots \mid \alpha_n A' \mid \epsilon$$

- *Faktorizace pravidel* Začíná-li několik pravých stran A-pravidla týmž řetězcem terminálních symbolů, tj. má-li pravidlo tvar

$$A \to \beta\alpha_1 \mid \beta\alpha_2 \mid \cdots \mid \beta\alpha_n,$$

můžeme provést jejich "vytknutí" opět zavedením nového nonterminálu A' s pravidly

$$A \to \beta A'$$

$$A' \to \alpha_1 \mid \alpha_2 \mid \cdots \mid \alpha_n$$

Tato úprava, stejně jako předchozí, však nemusí zaručit, že nepřinese další konflikty. Budou-li například některé z řetězců α_i neprázdný průnik množin $FIRST$, dojde opět k porušení podmínky FF v nonterminálu A'.

- *Eliminace pravidel* Některým konfliktům se můžeme vyhnout tak, že za některé nonterminály dosadíme jejich pravé strany a tím odstraníme z gramatiky pravidla, která způsobovala konflikt.

- *Redukce množiny FOLLOW* Je-li pro některý nonterminál porušena podmínka FFL, můžeme přidat nový nonterminál, který vede ke zmenšení počtu prvků konfliktní množiny $FOLLOW$ a případně i k disjunktnosti této množiny $FOLLOW$ s množinami $FIRST$ zbývajících pravých stran pravidel konfliktního nonterminálu (příklad viz [16], str. 103).

Uvedené transformace nemusí obecně vést k cíli, a to i v případě, že k transformované gramatice LL(1) gramatika existuje.

3.2.5 Analýza rekurzivním sestupem

Jednou z implementací syntaktické analýzy shora dolů je analýza rekurzivním sestupem. Tato metoda spočívá v zápisu samostatných procedur pro analýzu každého nonterminálního symbolu gramatiky. Překlad programu se pak spustí voláním procedury odpovídající startovacímu nonterminálu.

Máme-li pro nonterminál A jediné pravidlo ve tvaru $A \to X_1\ X_2\ \ldots\ X_n$, bude tělo příslušné procedury obsahovat posloupnost akcí provádějících postupně analýzu symbolů X_1 až X_n. Je-li symbol X_i nonterminálním symbolem gramatiky, bude odpovídající akcí volání podprogramu pro analýzu symbolu X_i, je-li X_i terminální symbol, zavoláme podprogram `expect(`X_i`)`. Tento podprogram zjistí, zda je na vstupu požadovaný symbol a v případě, že ano, přečte další vstupní symbol; v opačném případě nahlásí syntaktickou chybu. Příklad implementace procedury `expect` v jazyce Pascal je na obr. 3.2. Předpokládáme, že lexikální analyzátor je reprezentován procedurou `lex`, která při každém zavolání naplní globální proměnnou `sym` typu `symbol` následujícím vstupním symbolem.

```
procedure expect(s: symbol);
begin
    if sym = s then
        lex
    else
        error
end;
```

Obr. 3.2: Implementace procedury `expect`

Například pro analýzu nonterminálu A s jediným pravidlem $A \to xBy$ bude implementace procedury následující (předpokládáme, že terminálním symbolům x a y odpovídají konstanty `SYM_X` a `SYM_Y`):

```
procedure A;
begin
   expect(SYM_X);
   B;
   expectSYM_Y)
end;
```

V případě, že nonterminál A je definován více A-pravidly gramatiky, např. pokud gramatika obsahuje A-pravidla $A \to \alpha_1 \mid \alpha_2 \mid \ldots \mid \alpha_k$, musíme nejprve na základě následujícího vstupního symbolu vybrat vhodnou pravou stranu. Pro každou variantu α_i budeme mít úsek ve tvaru

```
if sym in Φ(A, α_i) then begin
    /* implementace analýzy řetězce α_i */
end
```

kde funkce $\Phi(A, \alpha_i)$ je definována jako

$$\Phi(A, \alpha) = \begin{cases} FIRST(\alpha), & \epsilon \notin FIRST(\alpha) \\ FOLLOW(A) \cup (FIRST(\alpha) \setminus \{\epsilon\}), & \epsilon \in FIRST(\alpha) \end{cases}$$

Tato funkce definuje množinu symbolů, které se mohou vyskytovat na vstupu v okamžiku expanze nonterminálu A na řetězec α. Pokud tento řetězec vždy obsahuje alespoň jeden symbol, je touto množinou $FIRST(\alpha)$. Může-li však expandovaný řetězec být prázdný, je třeba očekávat na vstupu i ty symboly, které jsou součástí množiny $FOLLOW(A)$ nonterminálu na levé straně pravidla. Je-li na vstupu symbol, který nepatří do žádné z množin $\Phi(A, \alpha_i)$, jde o syntaktickou chybu.

Vzhledem k tomu, že výběr pravé strany musí být u analyzátoru bez návratů jednoznačný, musí být množiny symbolů definované funkcí $\Phi(A, \alpha_i)$ pro jednotlivé pravé strany α_i disjunktní. Toto tvrzení ale není nic jiného, než vyjádření podmínek FF a FFL pro LL(1) gramatiku.

Příklad 3.4. Mějme dánu gramatiku pro aritmetický výraz s operátory `+` a `*`, závorkami a celočíselnými konstantami:

$$\begin{aligned} E &\to T\ E1 \\ E1 &\to +\ T\ E1 \mid \epsilon \\ T &\to F\ T1 \end{aligned}$$

$$\begin{aligned} T1 &\rightarrow * \; F \; T1 \mid \epsilon \\ F &\rightarrow (\; E \;) \mid \mathbf{id} \end{aligned}$$

Pro nonterminál $E1$ můžeme vypočítat následující množiny:

$$\begin{aligned} FIRST(+ \; T \; E1) &= \{+\}, \\ FIRST(\epsilon) &= \{\epsilon\}, \\ FOLLOW(E1) &= \{), \$\}, \\ \Phi(E1, + \; T \; E1) &= \{+\}, \\ \Phi(E1, \epsilon) &= \{), \$\}, \end{aligned}$$

takže jej můžeme implementovat procedurou

```
procedure E1;
begin
    if sym in [ADDSYM] then begin
        expect(ADDSYM);
        T;
        E1
        end
    else if sym in [RPRSYM, EOFSYM] then begin
        /* prázdná pravá strana */
        end
    else
        error
end;
```

Typ `symbol` je v tomto případě reprezentován výčtem konstant `ADDSYM` (operátor +), `MULSYM` (operátor $*$), `LPRSYM` (levá závorka), `RPRSYM` (pravá závorka), `IDSYM` (identifikátor) a `EOFSYM` (konec vstupního textu \$).

Je zřejmé, že uvedené řešení lze implementovat mnohem efektivněji, pokud provedeme následující optimalizace:

- Test, zda je symbol obsažen v jednoprvkové množině, lze nahradit přímo testem na rovnost.
- V případě, že pravá strana pravidla začíná terminálním symbolem, není třeba volat proceduru `expect`, neboť máme již při výběru pravé strany zaručen kladný výsledek testu na obr. 3.2. Můžeme tedy rovnou volat lexikální analyzátor.
- Je-li pravá strana pravidla prázdná (tj. je-li tvořena pouze symbolem ϵ), je možné ji implementovat vždy jako poslední a obrátit příslušný test.

Po naznačených optimalizacích dostaneme konečnou verzi procedury analyzující nonterminál $E1$:

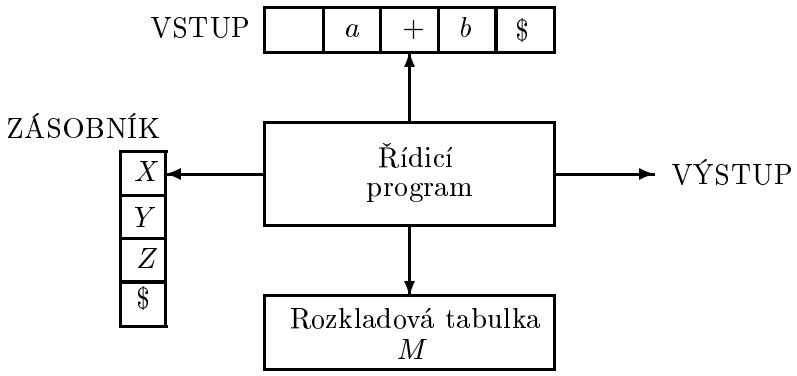
```
procedure E1;
begin
    if sym = ADDSYM then begin
        lex;
        T;
```

```
            E1
            end
        else if not (sym in [RPRSYM, EOFSYM]) then
            error;
    end;
```

Podobným způsobem můžeme implementovat i zbývající nonterminály gramatiky. ■

3.2.6 Nerekurzivní prediktivní analýza

Implementace syntaktického analyzátoru z předchozího článku využívala pro uchování informací o rozpracované části věty implicitního zásobníku, který používá hostitelský překladač (tj. v našem případě překladač Pascalu) pro realizaci volání rekurzivních podprogramů. Je však také možné vytvořit prediktivní syntaktický analyzátor, který používá svůj vlastní zásobník. Struktura takového analyzátoru je na obr. 3.3.



Obr. 3.3: Model nerekurzivního prediktivního syntaktického analyzátoru

Tento typ analyzátoru, nazývaný *syntaktický analyzátor řízený tabulkou*, je tvořen vstupní pamětí, zásobníkem, rozkladovou tabulkou a výstupem. Vstupní paměť obsahuje analyzovaný řetězec zakončený speciálním symbolem \$, který označuje konec vstupního řetězce. Zásobník obsahuje posloupnost symbolů gramatiky; dno zásobníku je indikováno opět speciálním symbolem #. Rozkladová tabulka je dvojrozměrné pole $M[A, a]$, kde A je nonterminál a a je terminální symbol nebo symbol \$.

Samostatnou částí analyzátoru je řídicí program, který opakovaně prohlíží symbol X na vrcholu zásobníku a současný vstupní symbol a, na základě nichž se rozhoduje o své další činnosti. Algoritmus rozhodování je následující:

- Je-li $X = \#$ a $a = \$$, vyčerpali jsme vstupní řetězec i zásobník; analyzátor se zastaví a ohlásí úspěšné ukončení.
- Je-li $X = a \neq \$$, odstraníme symbol X z vrcholu zásobníku a přesuneme se na následující vstupní symbol.
- Je-li X nonterminální symbol, provedeme jeho expanzi na některou z odpovídajících pravých stran pravidel gramatiky. Pokud položka rozkladové tabulky $M[X, a]$ obsahuje X-pravidlo gramatiky, nahradíme symbol X na vrcholu zásobníku pravou stranou tohoto

pravidla a na výstup předáme číslo použitého pravidla. Pokud je však $M[X, a] =$ **error**, jde o syntaktickou chybu, kterou musí analyzátor nahlásit a provést zotavení.

- V ostatních případech jde opět o syntaktickou chybu.

Tento algoritmus můžeme vyjádřit programem na obr. 3.4. Proměnná *top* obsahuje index vrcholu zásobníku symbolů *stack*, funkce *pop*() odstraní vrchol zásobníku a funkce *push*() uloží na zásobník řetězec symbolů. Funkce *error*() provádí hlášení syntaktických chyb a případné zotavení, funkce *lex*() představuje lexikální analyzátor, který při každém zavolání vrátí jeden symbol ze vstupu.

```
top := 0;
push(#S);
a := lex();
repeat
    X := stack[top];
    if X je terminální symbol nebo $ then
        if X = a then begin
            pop();
            a := lex();
        end
        else error()
    else      /* X je nonterminál */
        if M[X, a] = X → Y1Y2 ··· Yk then begin
            pop();
            push(YkYk−1 ··· Y1);
            vypiš číslo použitého pravidla
        end
        else error()
until X = #      /* zásobník je prázdný */
```

Obr. 3.4: Řídicí program prediktivního analyzátoru

3.2.7 Zotavení po chybě při analýze shora dolů

K důležitým úkolům syntaktického analyzátoru patří také diagnostická činnost. Aby v rámci jednoho průchodu zdrojovým programem kompilátor odhalil co nejvíce chyb, je třeba implementovat prostředky, které dovolí, aby syntaktický analyzátor pokračoval v kontrole správnosti programu i po výskytu syntaktické chyby. Problém zotavení ze syntaktické chyby není obecně jednoduchý. Běžně používané metody vycházejí z následujícího obecného postupu:

1. Po odhalení syntaktické chyby se ve vstupním řetězci hledá místo (*bod zotavení*, od kterého může analýza pokračovat v činnosti, přičemž se vynechá určitá část textu. Bod zotavení je obvykle dán nalezením symbolu z množiny tzv. *klíčů*.

2. Syntaktický analyzátor provede synchronizaci podle pozice nalezeného klíče v gramatice a pokračuje dále v činnosti.

Množina klíčů musí být definována tak, aby obsahovala pokud možno pouze ty symboly, jejichž výskyt v gramatice je jednoznačný. Tím lze zajistit vyšší spolehlivost synchronizace analyzátoru při zotavování. Například v gramatice jazyka Pascal je klíčové slovo **else** použito jednoznačně, na rozdíl od identifikátoru nebo klíčového slova **end** (konec složeného příkazu, příkazu **case**, resp. záznamu). Je-li však množina klíčů příliš omezená, roste délka neanalyzovaného textu, který se vynechává při vyhledávání klíče ve vstupní větě.

Pro zotavení na základě množiny klíčů se používajím například tyto metody:

- *Nerekurzivní metoda s pevnou množinou klíčů.* Tato metoda vychází z předem vypočtené množiny klíčů. Ke každému klíči je k dispozici informace o tom, kterou syntaktickou konstrukci ukončuje. Například klíč `')'` může ukončovat výrazy a klíč `';'` příkazy. Vyskytne-li se pak chyba během analýzy výrazu a při zotavení se najde pravá závorka, odstraní se ze zásobníku všechno, co souviselo s rozpracovaným výrazem a pokračuje se v analýze tak, jako by byl výraz analyzován správně.

- *Rekurzivní metoda s pevnou množinou klíčů.* Předchozí metoda se dá vylepšit ještě tím, že se určí rovněž množiny klíčů, kterými začínají jisté syntaktické konstrukce. Je-li během vyhledávání bodu zotavení nalezen některý z těchto klíčů, spustí se analýza vnořené konstrukce a po jejím ukončení se pokračuje v zotavení. Tím je možné omezit rozsah neanalyzovaného textu a mohou být tedy odhaleny další chyby v zanořených konstrukcích.

- *Metoda s dynamicky budovanou množinou klíčů.* Při této metodě se množina klíčů vytváří vždy na základě okamžitého kontextu; například při analýze příkazů v těle pascalovského cyklu **repeat** bude klíčem symbol **until**, zatímco při analýze výrazu v indexu bude klíčem pravá závorka. Jednou z metod této skupiny je Hartmannova metoda, která jako množiny klíčů využívá sjednocení množin $FOLLOW$ rozpracovaných nonterminálů. Její implementací se budeme dále zabývat podrobněji.

Hartmannovo schéma zotavení

Každému syntakticky správně vytvořenému programu analyzovanému syntaktickým analyzátorem přísluší derivační strom. Při analýze metodou rekurzivního sestupu je derivační strom budován postupným vyvoláváním procedur odpovídajících jednotlivým nonterminálům gramatiky a jejich prováděním. Výskyt syntaktické chyby představuje z hlediska syntaktického analyzátoru situaci, kdy v jistém stadiu rozpracování derivačního stromu není možné v budování tohoto stromu pokračovat. Začlenění prostředků pro zotavení po chybě Hartmannovou metodou předpokládá, že analyzátor při výskytu chyby

- ukončí vytváření derivačního podstromu obsahujícího chybu (neurčujeme zatím, kterého podstromu; v nejhorším případě dojde k ukončení vytváření celého stromu a tím i analýzy) s tím, že tento podstrom je nadále uvažován jako správně vytvořený

- přeskočí všechny symboly na vstupu mezi chybou a koncem fráze odpovídající uzavřenému derivačnímu podstromu.

Snahou dobrého zotavování je uzavřít po chybě co nejtěsnější podstrom obklopující chybu (podstrom, jehož kořen je co nejvíce vzdálen od vrcholu derivačního stromu). Čím těsnější podstrom je uzavřen, tím méně symbolů je třeba přeskočit. Přeskakované symboly nejsou analyzovány; mohou být zdrojem dalších syntaktických chyb a pokud je symbolů přeskočeno příliš mnoho, nelze v jedné analýze odhalit všechny chyby.

V každém okamžiku analýzy je vytvářen derivační podstrom pro jistý počet nonterminálů, přičemž tyto podstromy jsou do sebe vnořeny. Přiřaďme každému rozpracovanému derivačnímu podstromu množinu symbolů nazvanou $CONTEXT(A)$, která je sjednocením množin $FOLLOW(A_i)$ všech nonterminálů, jež mají v okamžiku expanze nonterminálu A rozpracovaný derivační podstrom, včetně množiny $FOLLOW(A)$. Vznikne-li v průběhu vytváření derivačního podstromu pro nonterminál A chyba, musí proběhnout zotavení.

Množina $CONTEXT(A)$ je dynamicky budovanou množinou klíčů, které využíváme při hledání bodu zotavení. Přeskočení symbolů na vstupu mezi chybou a koncem fráze odpovídající jistému podstromu, je realizováno přeskočením všech symbolů na vstupu, které nejsou v množině $CONTEXT(A)$. Protože všechny symboly z množiny $CONTEXT(A)$ jsou zároveň prvky jedné nebo více množin $FOLLOW$ pro jednotlivé vnořené derivační podstromy, je zajištěno, že bude přeskočen nejmenší možný počet symbolů ze vstupu a nalezen nejbližší možný bod zotavení v daném kontextu. Zároveň je třeba postupně uzavřít analýzu všech nonterminálů počínaje od nejvnořenějšího, v jejichž množinách $FOLLOW$ není obsažen nastavený vstupní symbol. Posledním nonterminálem, jehož analýza se uzavře, je nonterminál, v jehož množině $FOLLOW$ bod zotavení je.

Během analýzy metodou rekurzivního sestupu může dojít k detekci syntaktické chyby ve dvou situacích:

- je-li na vstupu jiný terminální symbol než se očekává, nebo
- nelze-li při expanzi nonterminálu vybrat na základě současného vstupního symbolu žádnou pravou stranu pravidla (vstupní symbol není prvkem $\Phi(A, \alpha_i)$ pro žádné i).

První případ odpovídá situaci, kdy se chyba hlásí z procedury `expect`, druhý případ nastává bezprostředně při vstupu do procedury analyzující konkrétní nonterminál. Je-li k dispozici množina klíčů $CONTEXT$ (budeme ji nazývat také *kontextová množina*, neboť definuje kontext, v němž analýza probíhá), můžeme upravit proceduru `expect` tak, aby při chybě provedla zároveň i synchronizaci, jak ukazuje obr. 3.5.

Test na začátku analýzy nonterminálu zároveň se zotavením může provést procedura `check(s, context)`, která jako první parametr obdrží sjednocení množin $\Phi(A, \alpha_i)$ pro všechny pravé strany α_i nonterminálu A. Není-li současný vstupní symbol v této množině, nahlásí se chyba a provede se zotavení pomocí kontextové množiny. Při hledání bodu zotavení se ještě připouští, aby se na vstupu ještě objevil symbol z množiny očekávaných symbolů s, což umožňuje efektivní zotavení v situaci, kdy je na vstupu nějaký symbol navíc.

Vlastní postup při začlenění zotavení do analýzy rekurzivním sestupem je pak následující:

- procedury pro analýzu nonterminálů budou jako vstupní parametr předávaný hodnotou dostávat aktuální kontextovou množinu, tj. deklarace procedur budou mít tvar

```
procedure A(context: symbols);
```

- při volání procedury pro analýzu nonterminálu nebo procedury `expect` se vždy vypočte nová kontextová množina
- před volbou varianty v nonterminálu se zavolá procedura `check(`$\bigcup_i \Phi(A, \alpha_i)$`, context)`, která zjistí, zda současný vstupní symbol odpovídá některé z pravých stran pro nonterminál A.

```
type symbols = set of symbol;

procedure expect(s:symbol; context: symbols);
begin
   if sym = s then
      lex
   else begin
      error;
      while not (sym in context) do lex
   end
end;

procedure check(s, c: symbols);
begin
   if not (sym in s) then begin
      error;
      while not (sym in c+s) do lex
   end
end;
```

Obr. 3.5: Implementace pomocných procedur pro zotavení

Výpočet kontextové množiny symbolu X_i na pravé straně pravidla $A \to X_1 X_2 \ldots X_i X_{i+1} \ldots X_k$ spočívá v rozšíření současné kontextové množiny $CONTEXT(A)$ o symboly, které se stanou klíči pro analyzovaný terminální nebo nonterminální symbol. Možné jsou například tyto přístupy:

1. Kontextovou množinu nonterminálu X_i vždy rozšíříme o prvky množiny $FOLLOW(X_i)$, tj.
$$CONTEXT(X_i) = CONTEXT(A) \cup FOLLOW(X_i), X_i \in N$$
zatímco kontextovou množinu terminálních symbolů (tj. argument procedury *expect*) ponecháme původní, tj.
$$CONTEXT(X_i) = CONTEXT(A), X_i \in \Sigma$$

2. Kontextovou množinu symbolu X_i (terminálního i nonterminálního) vždy rozšíříme o symboly, jimiž může začínat zbývající část řetězce na pravé straně pravidla, tj.
$$CONTEXT(X_i) = CONTEXT(A) \cup (FIRST(X_{i+1} \ldots X_k) \setminus \{\epsilon\})$$

3. Kontextovou množinu symbolu X_i rozšíříme o symboly ležící ve $FIRST$ všech následujících symbolů v pravidle, tj.
$$CONTEXT(X_i) = CONTEXT(A) \cup (\bigcup_{j=i+1}^{k} FIRST(X_j) \setminus \{\epsilon\})$$

První varianta je nejjednodušší, ovšem vyžaduje výpočet množin $FOLLOW$ a vede obecně k přeskočení zbytku rozpracovaného pravidla při chybě uvnitř některého ze symbolů na pravé

straně. Další dvě varianty se liší mohutností kontextové množiny, přičemž nejvýhodnější řešení je zřejmě kombinací všech tří přístupů, kdy do kontextové množiny nebudeme přidávat ty symboly, které jsou nejednoznačné (tj. takové, které se ve zdrojovém textu mohou vyskytovat v různých významech). Na výběru kontextových množin podstatně závisí kvalita zotavení, která se projevuje nejen počtem odhalených skutečných chyb, ale (v opačném smyslu) i počtem hlášených zavlečených chyb

Příklad 3.5. Uvažujme následující gramatiku pro deklarace proměnných s inicializací:

$$\begin{aligned} S &\rightarrow \textbf{var id } L \textbf{ = num} \\ L &\rightarrow \textbf{, id } L \mid \epsilon \end{aligned}$$

Použijeme-li posledního přístupu k výpočtu kontextových množin, můžeme syntaktickou analýzu se zotavením implementovat následujícími procedurami (symboly **var**, **id**, **num**, čárka, rovnítko a $ jsou pojmenovány po řadě `VARSYM`, `IDSYM`, `NUMSYM`, `COMSYM`, `EQSYM` a `EOFSYM`):

```
procedure S(c: symbols);
begin
   expect(VARSYM, c + [IDSYM, COMSYM, EQSYM, NUMSYM]);
   expect(IDSYM,  c + [COMSYM, EQSYM, NUMSYM]);
   L(c + [EQSYM, NUMSYM]);
   expect(EQSYM,  c + [NUMSYM]);
   expect(NUMSYM, c)
end;
procedure L(c: symbols);
begin
   check([COMSYM, EQSYM], c);
   if sym = COMSYM then begin
      lex;
      expect(IDSYM, c + [COMSYM]);
      L(c)
   end
end;
```

Poznamenejme, že v situaci, kdy některý nonterminál A může generovat prázdný řetězec, je podle definice funkce Φ součástí prvního parametru funkce `check` také množina $FOLLOW(A)$. Vzhledem k tomu, jak se vytváří kontextová množina, můžeme množinu $FOLLOW(A)$ nahradit obecnější množinou $CONTEXT(A)$, která v dané situaci lépe reprezentuje množinu přípustných symbolů, které mohou v konkrétní situaci za nonterminálem A následovat. Například při analýze výrazů reprezentovaných nonterminálem E je v množině $FOLLOW(E)$ vždy obsažena pravá závorka jako důsledek pravidla $E \rightarrow (E)$. Pokud však nebyla ještě otevřena žádná levá závorka, není pravá závorka vlastně platným klíčem a neměla by být ani součástí kontextové množiny. Například v proceduře pro nonterminál L se může funkce `check` volat jako

```
check(c + [COMSYM], c)
```

■

3.3 Syntaktická analýza zdola nahoru

V této kapitole uvedeme obecný způsob syntaktické analýzy zdola nahoru, známý jako syntaktická analýza typu *přesun — redukce*. Budeme diskutovat jeden ze způsobů této analýzy nazvaný *syntaktická analýza LR*, který bude využit jako teoretický základ konstruktorů pro automatické vytváření syntaktických analyzátorů.

Syntaktická analýza přesun-redukce se pokouší budovat derivační strom ze vstupního řetězce od listů směrem ke kořenu. Na tento postup můžeme pohlížet rovněž jako na "redukci" vstupní věty w na startovací symbol gramatiky. Při každém z redukčních kroků je nahrazen jistý podřetězec, který se shoduje s pravou stranou přepisovacího pravidla, symbolem na levé straně pravidla.

Příklad 3.6. Uvažujme následující gramatiku

$$\begin{aligned} S &\to aABe \\ A &\to Abc \mid b \\ B &\to d \end{aligned}$$

Posloupnost *abbcde* může být redukována na S v následujících krocích:

$$\begin{aligned} &abbcde \\ &aAbcde \\ &aAde \\ &aABe \\ &S \end{aligned}$$

Prohledáváme řetězec *abbcde* tak, že vyhledáváme podřetězce shodné s pravou stranou některého z pravidel. Na počátku vyhovují b a d. Zvolíme b, které leží více vlevo a nahradíme jej za A, tj. levou stranu přepisovacího pravidla $A \to b$; tím obdržíme řetězec *aAbcde*. Nyní jsou k dispozici podřetězce *Abc*, b a d shodné s pravými stranami pravidel. Ačkoliv b je nejlevější řetězec shodný s pravou stranou přepisovacího pravidla, nahradíme řetězec *Abc* za A, levou stranu pravidla $A \to Abc$. Obdržíme tím *aAde*. Potom nahradíme d za B, levou stranu pravidla $B \to d$, čímž obdržíme *aABe*. To je pravá strana pravidla $S \to aABe$, kterou můžeme nahradit za S. Posloupností čtyř redukcí jsme redukovali *abbcde* na S. Tyto redukce brané v opačném pořadí jsou vlastně posloupností větných forem v pravé derivaci

$$S \Rightarrow aABe \Rightarrow aAde \Rightarrow aAbcde \Rightarrow abbcde$$

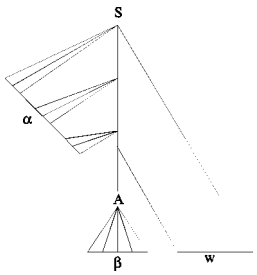
■

3.3.1 Pracovní fráze

Pracovní fráze řetězce je podřetězec, který je shodný s pravou stranou přepisovacího pravidla a jehož redukce na levostranný nonterminál pravidla reprezentuje jeden zpětný krok pravé derivace. V mnoha případech není nejlevější podřetězec β, který je shodný s pravou stranou přepisovacího pravidla $A \to \beta$ pracovní frází, protože redukce podle pravidla $A \to \beta$ vytvoří řetězec,

který není redukovatelný na startovací symbol. V příkladu 3.6, pokud bychom nahradili b za A ve druhém řetězci $aAbcde$, obdrželi bychom řetězec $aAAcde$, který není následně redukovatelný na S. Budeme proto pracovní frázi definovat precizněji.

Formálně je *pracovní fráze* pravé větné formy γ přepisovací pravidlo $A \to \beta$ a pozice v γ, kde řetězec β může být nahrazen za A tak, že náhradou vznikne předchozí pravá větná forma pravé derivace řetězce γ. Tedy, pokud $S \overset{*}{\Rightarrow} \alpha Aw \Rightarrow \alpha\beta w$, potom $A \to \beta$ v pozici následující za α je pracovní frází $\alpha\beta w$. Řetězec w napravo od pracovní fráze obsahuje pouze terminální symboly. Je-li gramatika jednoznačná, pak každá pravá větná forma gramatiky má právě jednu pracovní frázi. Ve shora uvedeném příkladu 3.6 je $abbcde$ je pravou větnou formou, jejíž pracovní frází je $A \to b$ na pozici 2. Obdobně $aAbcde$ je pravou větnou formou, jejíž pracovní frází je $A \to Abc$ na pozici 2. Často lze též říci "podřetězec β je pracovní frází řetězce $\alpha\beta w$," pokud u pozice β a pravidla $A \to \beta$ nemůže dojít k nejednoznačnosti.



Obr. 3.6: Pracovní fráze $A \to \beta$ v derivačním stromu pro $\alpha\beta w$

Obrázek 3.6 znázorňuje pracovní frázi $A \to \beta$ v derivačním stromu pravé větné formy $\alpha\beta w$. Pracovní fráze reprezentuje nejlevější úplný podstrom sestávající z uzlů a všech jejich potomků. Na obr. 3.6 je A ten vnitřní uzel, který je nejvíce nalevo a nejhlouběji a obsahuje v podstromu všechny své potomky. Redukování β na A v $\alpha\beta w$ můžeme nazývat "redukování pracovních frází" a znamená odstraňování potomků uzlu A z derivačního stromu.

Příklad 3.7. Uvažujme následující gramatiku

$$\begin{array}{lll} (1) & E & \to & E + E \\ (2) & E & \to & E * E \\ (3) & E & \to & (E) \\ (4) & E & \to & \mathbf{id} \end{array}$$

a pravou derivaci

$$\begin{array}{rcl} E & \Rightarrow & \underline{E + E} \\ & \Rightarrow & E + \underline{E * E} \\ & \Rightarrow & E + E * \underline{\mathbf{id}_3} \end{array}$$

$$\begin{aligned} &\Rightarrow\ E\ +\ \underline{\mathbf{id}_2}\ *\ \mathbf{id}_3 \\ &\Rightarrow\ \underline{\mathbf{id}_1}\ +\ \mathbf{id}_2\ *\ \mathbf{id}_3 \end{aligned}$$

Indexy byly použity pro pozdější rozlišení, pracovní fráze pravých větných forem jsou podtrženy. Například $\mathbf{id}_1$ je pracovní fráze pravé větné formy $\mathbf{id}_1 + \mathbf{id}_2 * \mathbf{id}_3$, protože **id** je pravou stranou pravidla $E \to \mathbf{id}$ a nahrazením $\mathbf{id}_1$ za E vytvoří předchozí pravou větnou formu $E + \mathbf{id}_2 * \mathbf{id}_3$. Poznamenejme, že řetězec nacházející se napravo od pracovní fráze obsahuje pouze terminální symboly.

Protože zadaná gramatika je víceznačná, existuje i jiná pravá derivace stejného řetězce:

$$\begin{aligned} E\ &\Rightarrow\ \underline{E\ *\ E} \\ &\Rightarrow\ E\ *\ \underline{\mathbf{id}_3} \\ &\Rightarrow\ \underline{E\ +\ E}\ *\ \mathbf{id}_3 \\ &\Rightarrow\ E\ +\ \underline{\mathbf{id}_2}\ *\ \mathbf{id}_3 \\ &\Rightarrow\ \underline{\mathbf{id}_1}\ +\ \mathbf{id}_2\ *\ \mathbf{id}_3 \end{aligned}$$

Uvažujme pravou větnou formu $E + E * \mathbf{id}_3$. V této derivaci je $E + E$ pracovní frází větné formy $E + E * \mathbf{id}_3$, zatímco v předchozí derivaci byl pracovní frází řetězec $\mathbf{id}_3$.

3.3.2 Redukování pracovních frází

Pravou derivaci v opačném pořadí můžeme obdržet "redukováním pracovních frází." Tato činnost vychází z počátečního řetězce terminálních symbolů w, který chceme analyzovat. Pokud je w věta generovaná zvolenou gramatikou, potom $w = \gamma_n$, kde γ_n je n-tá pravá větná forma, některé doposud neznámé pravé derivace

$$S = \gamma_0 \Rightarrow \gamma_1 \Rightarrow \gamma_2 \Rightarrow \ldots \Rightarrow \gamma_{n-1} \Rightarrow \gamma_n = w$$

Abychom mohli rekonstruovat tuto derivaci v opačném pořadí, budeme lokalizovat pracovní frázi β_n v γ_n a nahradíme β_n za levou stranu přepisovacího pravidla $A_n \to \beta_n$. Tím obdržíme $(n-1)$. pravou větnou formu γ_{n-1}. Poznamenejme, že doposud nevíme, jakým způsobem pracovní frázi vyhledat. Zmíníme se o tom později.

Proces náhrady opakujeme. Tedy nalezneme pracovní frázi β_{n-1} ve γ_{n-1} a redukujeme pracovní frázi na γ_{n-2}. Opakováním tohoto procesu můžeme nakonec získat pravou větnou formu sestávající pouze ze startovacího symbolu S. Tím proces syntaktické analýzy zdola nahoru úspěšně končí. Zpětně čtená posloupnost pravidel použitá v redukcích je pravou derivací vstupního řetězce.

Příklad 3.8. Uvažujme gramatiku z příkladu 3.7 a vstupní řetězec $\mathbf{id}_1 + \mathbf{id}_2 * \mathbf{id}_3$. Posloupnost redukcí ukázaná na obr. 3.7 redukuje $\mathbf{id}_1 + \mathbf{id}_2 * \mathbf{id}_3$ na startovací symbol E. Čtenář se může přesvědčit, že posloupnost pravých větných forem v tomto příkladu je právě opačně zapsanou posloupností pravé derivace z příkladu 3.7. ■

pravá větná forma	pracovní fráze	redukční pravidla
$\mathbf{id}_1 + \mathbf{id}_2 * \mathbf{id}_3$	$\mathbf{id}_1$	$E \to \mathbf{id}$
$E + \mathbf{id}_2 * \mathbf{id}_3$	$\mathbf{id}_2$	$E \to \mathbf{id}$
$E + E * \mathbf{id}_3$	$\mathbf{id}_3$	$E \to \mathbf{id}$
$E + E * E$	$E * E$	$E \to E * E$
$E + E$	$E + E$	$E \to E + E$
E		

Obr. 3.7: Redukce prováděné analyzátorem typu přesun-redukce

3.3.3 Implementace analýzy typu přesun-redukce zásobníkem

Existují dva problémy, které musí být řešeny, chceme-li provádět redukování pracovních frází. Prvním problémem je nalezení podřetězce, který má být v pravé větné fromě redukován a druhým je volba přepisovacího pravidla pro redukci v případě, že existuje více, než jedno pravidlo se stejným řetězcem na pravé straně. Dříve, než budeme řešit tyto problémy, specifikujme nejprve datové struktury, které budeme používat v syntaktickém analyzátoru typu přesun-redukce.

Obvyklým způsobem implementace takového analyzátoru je užití zásobníku pro uložení symbolů gramatiky a vstupní vyrovnávací paměti pro uložení analyzovaného řetězce w. Pro označení dna zásobníku a také pro označení pravého konce vstupního řetězce budeme užívat symbolu \$. Na začátku je zásobník prázdný a na vstupu je řetězec w:

ZÁSOBNÍK	VSTUP
\$	w\$

Analyzátor pracuje tak, že přesouvá žádný, jeden nebo více symbolů na vrchol zásobníku tak dlouho, dokud není na vrcholu zásobníku pracovní fráze β. Potom analyzátor redukuje β na levou stranu odpovídajícího přepisovacího pravidla. Analyzátor opakuje tento cyklus tak dlouho, dokud nenalezne chybu nebo dokud zásobník neobsahuje startovací symbol a zásobník není prázdný.

ZÁSOBNÍK	VSTUP
\$$S$	\$

Po dosažení této konfigurace se analyzátor zastaví a ohlásí úspěšné ukončení analýzy.

Příklad 3.9. Projděme nyní krok po kroku akce analyzátoru typu přesun-redukce, které by mohli být provedeny analyzátorem při analýze vstupní věty $\mathbf{id}_1 + \mathbf{id}_2 * \mathbf{id}_3$ podle gramatiky z příkladu 3.7. Posloupnost akcí je uvedena na obr. 3.8. Protože gramatika má dvě pravé derivace odpovídající danému vstupnímu řetězci, existuje také jiná posloupnost akcí, kterou by mohl analyzátor provést.

I když základními operacemi analyzátoru jsou přesun a redukce, existují ve skutečnosti čtyři možné akce, které analyzátor může provádět: (1) *přesun*, (2) *redukce*, (3) *přijetí* a (4) *chyba*.

- Během akce *přesun* je přesunut další vstupní symbol na vrchol zásobníku.
- Během akce *redukce* má analyzátor informace o pravém konci pracovní fráze na vrcholu zásobníku. Musí nalézt levý konec pracovní fráze uvnitř zásobníku a rozhodnout, kterým nonterminálním symbolem nahradí pracovní frázi.
- Během akce *přijetí* ohlásí analyzátor úspěšné ukončení analýzy.

	ZÁSOBNÍK	VSTUP	AKCE
(1)	$\$$	$\mathbf{id}_1 + \mathbf{id}_2 * \mathbf{id}_3\$$	přesun
(2)	$\$\mathbf{id}_1$	$+ \mathbf{id}_2 * \mathbf{id}_3\$$	redukce podle $E \to \mathbf{id}$
(3)	$\$E$	$+ \mathbf{id}_2 * \mathbf{id}_3\$$	přesun
(4)	$\$E +$	$\mathbf{id}_2 * \mathbf{id}_3\$$	přesun
(5)	$\$E + \mathbf{id}_2$	$* \mathbf{id}_3\$$	redukce podle $E \to \mathbf{id}$
(6)	$\$E + E$	$* \mathbf{id}_3\$$	přesun
(7)	$\$E + E *$	$\mathbf{id}_3\$$	přesun
(8)	$\$E + E * \mathbf{id}_3$	$\$$	redukce podle $E \to \mathbf{id}$
(9)	$\$E + E * E$	$\$$	redukce podle $E \to E * E$
(10)	$\$E + E$	$\$$	redukce podle $E \to E + E$
(11)	$\$E$	$\$$	přijetí

Obr. 3.8: Analýza věty $\mathbf{id}_1 + \mathbf{id}_2 * \mathbf{id}_3$

- Během akce *chyba* analyzátor odhalí výskyt syntaktické chyby a vyvolá podprogram pro obsluhu chyby.

Zajímavým faktem, který zdůvodňuje užití zásobníku při analýze typu přesun-redukce, je to, že pracovní fráze (pokud existuje), se vždy vyskytuje na vrcholu zásobníku a nikdy uvnitř. Tento fakt se ozřejmí, uvažujme-li možné tvary dvou úspěšných kroků pravé derivace. Tyto dva případy mohou být ve tvaru:

$$(1) \qquad S \overset{*}{\Rightarrow} \alpha Az \Rightarrow \alpha\beta Byz \Rightarrow \alpha\beta\gamma yz$$
$$(2) \qquad S \overset{*}{\Rightarrow} \alpha BxAz \Rightarrow \alpha Bxyz \Rightarrow \alpha\gamma xyz$$

V případě (1) je A nahrazeno za βBy a potom nejpravější nonterminál B na pravé straně je nahrazen za γ. V případě (2) je opět A nahrazeno nejdříve, avšak tentokrát obsahuje pravá strana y pouze terminální symboly. Další nejpravější nonterminál B bude někde nalevo od y.

Uvažujme nyní případ (1) v opačném pořadí a následující konfigurace analyzátoru:

ZÁSOBNÍK	VSTUP
$\$\alpha\beta\gamma$	$yz\$$

Analyzátor nyní redukuje pracovní frázi γ na B, čímž dosáhne konfigurace

ZÁSOBNÍK	VSTUP
$\$\alpha\beta B$	$yz\$$

Protože B je nejpravější nonterminál v $\alpha\beta Byz$, pravý konec pracovní fráze $\alpha\beta Byz$ se nesmí nacházet uvnitř zásobníku. Analyzátor proto přesune řetězec y na vrchol zásobníku, aby získal konfiguraci

ZÁSOBNÍK	VSTUP
$\$\alpha\beta By$	$z\$$

ve které βBy je pracovní fráze redukovatelná na A.

Ve druhém případě v konfiguraci

ZÁSOBNÍK	VSTUP
$\$\alpha\gamma$	$xyz\$$

je pracovní fráze γ na vrcholu zásobníku. Po redukci pracovní fráze γ na B přesune analyzátor řetězec xy na vrchol zásobníku, aby získal další pracovní frázi y:

ZÁSOBNÍK	VSTUP
$\$\alpha Bxy$	$z\$$

Nyní analyzátor redukuje y na A.

V obou případech po vykonání redukce má analyzátor za úkol přesunout žádný, jeden nebo více symbolů tak, aby získal další pracovní frázi na vrcholu zásobníku. Nikdy nehledá pracovní frázi uvnitř zásobníku. Tento aspekt činnosti analyzátoru nám dává zásobník jako vhodnou implementační datovou strukturu analyzátoru. Nyní je třeba objasnit, jakým způsobem analyzátor volí akci, jež má být v následujícím kroku provedena.

3.3.4 Perspektivní prefixy

Množina prefixů pravé větné formy, které se mohou vyskytovat na vrcholu zásobníku analyzátoru typu přesun-redukce se nazývá množina *perspektivních prefixů*. Ekvivalentní definice perspektivního prefixu je následující: je to prefix pravé větné formy, který nesmí pokračovat za pravým koncem nejpravější fáze větné formy. Při použití této definice je vždy možné přidat na konec perspektivního prefixu terminální symboly abychom získali pravou větnou formu.

3.3.5 Konflikty během analýzy typu přesun-redukce

Existují bezkontexové gramatiky, které nelze použít pro analýzu typu přesun-redukce. Každý analyzátor typu přesun-redukce může pro takovou gramatiku dosáhnout stavu, ve kterém (i když je znám celý obsah zásobníku i celý zbytek vstupního řetězce) nelze rozhodnout, zdali se má přesouvat nebo redukovat (*konflikt přesun-redukce*). Dále může být nerozhodnutelné, kterou z několika možných redukcí provést (*konflikt redukce-redukce*). Uvedeme nyní několik příkladů syntaktických konstrukcí, které jsou součástí takových gramatik. Tyto gramatiky nejsou třídy LR(k), která je definována v kapitole 3.4.2; nazýváme je ne-LR gramatiky; k v LR(k) odkazuje na počet symbolů, které jsou prohlíženy v dosud nepřečtené části vstupního řetězce. Gramatiky užívané pro překlad jsou většinou typu LR(1), tj. je prohlížen pouze první symbol nepřečtené části vstupu.

Příklad 3.10. Víceznačná gramatika nikdy nemůže být LR. Uvažujme například gramatiku s neúplným i úplným podmíněným příkazem:

$$\begin{aligned} stmt \;\rightarrow\; & \textbf{if}\ expr\ \textbf{then}\ stmt \\ \mid\; & \textbf{if}\ expr\ \textbf{then}\ stmt\ \textbf{else}\ stmt \\ \mid\; & \textbf{other} \end{aligned}$$

Pokud je analyzátor v konfiguraci

ZÁSOBNÍK	VSTUP
... **if** *expr* **then** *stmt*	**else** ... $

nemůžeme říci, zdali **if** *expr* **then** *stmt* je pracovní frází bez ohledu na to, co leží v zásobníku pod tímto řetězcem. Jde tedy o konflikt typu přesun-redukce. V závislosti na tom, co následuje na vstupu za **else** by mohlo být správné redukovat **if** *expr* **then** *stmt* na *stmt* nebo by mohlo být správné přesunout **else** a pak hledat další *stmt* ke kompletaci alternativy **if** *expr* **then** *stmt* **else** *stmt*. Proto můžeme s určitostí říci, že gramatika není LR(1). Obecněji řečeno, žádná víceznačná gramatika není LR(k) pro žádné k. V následujících kapitolách si ukážeme, jakým způsobem lze konstruovat analyzátory typu přesun-redukce i pro některé víceznačné gramatiky.

Příklad 3.11. Předpokládejme, že máme lexikální analyzátor, který vrací symbol **id** pro všechny identifikátory, bez ohledu na použití. Předpokládejme také, že náš jazyk volá procedury udáním jejich jmen s parametry ohraničenými v závorkách. Pole jsou odkazována syntakticky totožně. Vzhledem k tomu, že překlad indexů v odkazu na pole a parametrů ve volání procedury je různý, chceme užít různá pravidla pro generování seznamu skutečných parametrů a indexů. Navržená gramatika by mohla mít (mimo jiné) následující pravidla:

(1)	*stmt*	$\rightarrow$	**id**(*parameter_list*)
(2)	*stmt*	$\rightarrow$	*expr* := *expr*
(3)	*parameter_list*	$\rightarrow$	*parameter_list*, *parameter*
(4)	*parameter_list*	$\rightarrow$	*parameter*
(5)	*parameter*	$\rightarrow$	**id**
(6)	*expr*	$\rightarrow$	**id**(*expr_list*)
(7)	*expr*	$\rightarrow$	**id**
(8)	*expr_list*	$\rightarrow$	*expr_list*, *expr*
(9)	*expr_list*	$\rightarrow$	*expr*

Příkaz začínající `A(I,J)` bude předán syntaktickému analyzátoru jako řetězec terminálních symbolů **id**(**id**, **id**). Po přesunu prvních tří symbolů na vrchol zásobníku by mohl být analyzátor v následující konfiguraci:

ZÁSOBNÍK	VSTUP
... **id**(**id**	, **id**) ...

Je jasné, že **id** na vrcholu zásobníku musí být redukováno, ale podle kterého pravidla? Pokud `A` je procedurou, pak je správná volba (5), je-li to pole, je správná volba (7). Zásobník však neobsahuje informaci pro tuto volbu. Měly by být užity informace z tabulky symbolů získané z deklarace identifikátorů.

Jedním z řešení je záměna symbolu **id** v pravidle (1) za **procid** a užití rafinovanějšího lexikálního analyzátoru, který rozpozná identifikátor, jenž je jménem procedury. Aby to mohl provést, musí takový lexikální analyzátor spolupracovat s tabulkou symbolů.

Provedeme-li tuto modifikaci, bude analyzátor při zpracování `A(I,J)` v konfiguraci

ZÁSOBNÍK	VSTUP
... **procid**(**id**	, **id**) ...

nebo v konfiguraci uvedené předtím. V prvním případě zvolíme redukci podle pravidla (5), ve druhém případě pak podle pravidla (7). ■

3.4 Analyzátory LR

V tomto článku uvedeme efektivní způsob syntaktické analýzy zdola nahoru, který může být užit pro analýzu široké třídy bezkontextových jazyků. Tato technika analýzy je nazvána analýza LR(k): "L" vzhledem k tomu, že vstupní řetězec se zpracovává zleva doprava, "R" protože se vytváří pravá derivace v opačném pořadí a k pro určení počtu symbolů z dosud nepřečtené části vstupního řetězce určených pro rozhodování. Je-li (k) vynecháno, předpokládá se, že je rovno 1. Analýza LR je atraktivní z mnoha důvodů:

- analyzátory LR lze vytvořit k rozpoznání téměř všech možných konstrukcí programovacích jazyků, které jsou syntakticky definovány bezkontextovými gramatikami,

- metoda analýzy LR je nejobecnější známá metoda typu přesun-redukce pracující bez návratů, lze pro ni implementovat efektivní syntaktický analyzátor,
- LR analyzátor může detekovat chybu tak rychle po jejím výskytu, jak je to možné při prohledávání vstupního řetězce zleva doprava.

Stinnou stránkou metody je značná náročnost manuální konstrukce LR analyzátoru pro gramatiky typických programovacích jazyků. Je nutné užít speciálního automatizačního prostředku - konstruktoru (generátoru) LR analyzátorů. Naštěstí je dostupných mnoho takových konstruktorů, přičemž my budeme diskutovat návrh a použití nejznámějšího z nich — programu yacc. Máme-li k dispozici takový konstruktor, je možné navrhovat pouze bezkontextovou gramatiku a konstruktor z ní automaticky vygeneruje syntaktický analyzátor. Pokud je gramatika víceznačná nebo jinak odporuje podmínkám LR, může konstruktor taková pravidla gramatiky odhalit a informovat o jejich výskytu tvůrce gramatiky.

Po diskusi operací LR analyzátoru uvedeme tři techniky pro konstrukci rozkladové tabulky LR analyzátoru. První metoda, pro jednoduché LR (zkráceně SLR) gramatiky, je nejsnadnější pro implementaci, ale pokrývá nejmenší třídu gramatik. Může selhat při vytváření rozkladových tabulek pro gramatiky, pro něž budou další metody úspěšné. Druhá metoda, pro obecné LR gramatiky, je nejobecnější, je však velmi náročná na čas i paměťový prostor. Třetí metoda, pro LALR gramatiky (look-ahead) je kompromisem prvními dvěma metodami. Lze ji použít pro většinu gramatik programovacích jazyků a efektivně implementovat. Budeme diskutovat rovněž některé techniky pro kompresi rozsahu LR rozkladových tabulek, jejichž vytváření popíšeme.

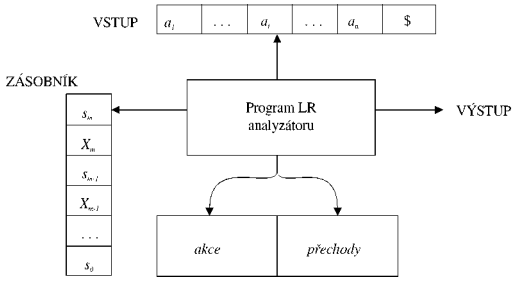
3.4.1 Algoritmus analýzy pro LR analyzátory

Schematický tvar LR analyzátoru je na obr. 3.9. Skládá se ze vstupu, výstupu, zásobníku, řídicího programu a rozkladové tabulky, která má dvě části (*akce* a *přechody*). Řídicí program je **stejný** pro všechny zde diskutované LR analyzátory, mění se pouze způsob konstrukce rozkladové tabulky. Řídicí program čte znaky ze vstupní vyrovnávací paměti jeden po druhém. Užívá zásobník k uložení řetězce ve tvaru $s_0X_1s_1X_2s_2 \ldots X_ms_m$, kde s_m je na vrcholu zásobníku. X_i označuje symbol gramatiky a s_i je symbol nazvaný *stav*. Každý stavový symbol sumarizuje informace obsažené v zásobníku pod ním a kombinace stavového symbolu na vrcholu zásobníku a aktivního vstupního symbolu se užívá pro indexování prvku rozkladové tabulky určujícího prováděnou akci. V konkrétní implementaci není nutné, aby se v zásobníku nacházely symboly gramatiky; my je však použijeme pro snažší vysvětlení chování LR analyzátoru.

Rozkladová tabulka sestává ze dvou částí, tabulky definující funkci *akce* a tabulky definující funkci *přechody*. Program řídicí LR analyzátor se chová následovně. Podle s_m, což je stavový symbol na vrcholu zásobníku, a a_i, což je aktuální vstupní symbol, vyhledá funkční hodnotu $akce[s_m, a_i]$. Funkční hodnotou je jedna ze čtyř následujících akcí:

- *přechod* s_i, kde s_i je stav,
- *redukce* podle přepisovacího pravidla $A \to \beta$,
- *přijetí* a
- *chyba*.

Funkce *přechody* dává pro parametry *stav* a *symbol gramatiky* jako funkční hodnotu *stav*. Uvidíme, že funkce *přechody* je část rozkladové tabulky vytvářená z gramatiky G užitím metody



Obr. 3.9: Model LR analyzátoru

SLR, kanonické LR metody nebo metody LALR. Jde o přechodovou funkci deterministického konečného automatu (DKA), který rozpoznává perspektivní prefixy gramatiky G. Připomeňme, že perspektivní prefix gramatiky G je takový prefix pravé větné formy, který se může vyskytnout na vrcholu zásobníku analyzátoru typu přesun-redukce, protože se nemůže rozšířit za nejpravější pracovní frázi. Počáteční stav shora uvedeného DKA je stavem, který je na počátku analýzy v zásobníku LR analyzátoru.

Konfigurace LR analyzátoru je dvojice, jejíž první složkou je obsah zásobníku a druhou složkou je dosud nepřečtená část vstupního řetězce:

$$(s_0 X_1 s_1 X_2 s_2 \ldots X_m s_m, a_i a_{i+1} \ldots a_n \$)$$

Tato konfigurace reprezentuje pravou větnou formu

$$X_1 X_2 \ldots X_m a_i a_{i+1} \ldots a_n$$

stejnou jako v obecně popsaném analyzátoru typu přesun-redukce. Pouze výskyt stavů v zásobníku je nový.

Akce analyzátoru, která bude následně provedena je určena stavem na vrcholu zásobníku s_m, a aktuálním vstupním symbolem a_i. Prováděná akce je funkční hodnotou $akce[s_m, a_i]$ získané z rozkladové tabulky. Konfigurace, v nichž se bude nacházet LR analyzátor po provedení každé z akcí, budou následující:

- Je-li $akce[s_m, a_i] = \textit{přesun}\ s$, potom analyzátor přejde do konfigurace

  $$(s_0 X_1 s_1 X_2 s_2 \ldots X_m s_m a_i s, a_{i+1} \ldots a_n \$)$$

  Analyzátor přesunul na vrchol zásobníku aktuální vstupní symbol a_i a potom stavový symbol s, který je určen funkční hodnotou $akce[s_m, a_i]$. Symbol a_{i+1} se stane aktuálním vstupním symbolem.

- Je-li $akce[s_m, a_i] = redukce\ A \to \beta$, potom analyzátor přejde do konfigurace

$$(s_0 X_1 s_1 X_2 s_2 \ldots X_{m-r} s_{m-r} A s, a_i a_{i+1} \ldots a_n \$)$$

kde $s = přechody[s_{m-r}, A]$ a r je délka pravé strany pravidla β. V našem případě analyzátor odstraní ze zásobníku $2r$ symbolů (r stavových symbolů a r symbolů gramatiky), čímž se zviditelní stav s_{m-r}. Analyzátor potom uloží na vrchol zásobníku symbol A, tj. levou stranu redukčního pravidla a s, tj. funkční hodnotu $přechody[s_{m-r}, A]$. Aktuální vstupní symbol není akcí redukce změněn. V analyzátorech LR, které budeme vytvářet, je posloupnost $X_{m-r+1} \ldots X_m$ symbolů gramatiky odstraňovaných ze zásobníku vždy rovna β, tj. pravé straně přepisovacího pravidla.

- Je-li $akce[s_m, a_i] = přijetí$, je analýza ukončena
- Je-li $akce[s_m, a_i] = chyba$, odhalil analyzátor syntaktickou chybu a volá podprogram pro ošetření chyby.

Nyní můžeme algoritmus analýzy LR shrnout. Všechny LR analyzátory, kterými se budeme zabývat, se chovají tímto způsobem; jediný rozdíl mezi nimi je v množství informací obsažených v rozkladové tabulce.

Algoritmus 3.2. (Algoritmus LR analýzy)

Vstup. Vstupní řetězec w a rozkladová tabulka LR s funkcemi *akce* a *přechody* pro gramatiku G.

Výstup. Je-li w v jazyce $L(G)$, analýza zdola nahoru pro w, jinak chybová indikace.

Metoda. Na počátku je v zásobníku stav s_0, kde s_0 je počáteční stav a ve vstupním zásobníku je $w\$$. Analyzátor provádí následující program, dokud nenarazí na akci *přijetí* nebo *chyba* (*ip* je zde ukazatel ve vstupní větě).

```
nastav ip na první symbol řetězce w;
loop
  nechť s je stav na vrcholu zásobníku a a je symbol zpřístupněný pomocí ip;
  if akce[s, a] = přesun s' then
    ulož a a potom s' na vrchol zásobníku;
    posuň ip tak, aby ukazovalo na následující symbol
  else if akce[s, a] = redukce A → β then
    odstraň 2 * |β| symbolů ze zásobníku;
    nechť je nyní na vrcholu zásobníku s';
    ulož A a potom přechody[s', A] na vrchol zásobníku;
    na výstup předej pravidlo A → β
  else if akce[s, a] = přijetí then
    return
  else
    chyba()
end
```

■

Příklad 3.12. Obrázek 3.10 obsahuje funkce akcí a přechodů ve formě LR rozkladové tabulky pro následující gramatiku. Gramatika generuje aritmetické výrazy s binárními operátory + a *:

$$
\begin{array}{lllll}
(1) & E & \to & E + T \\
(2) & E & \to & T \\
(3) & T & \to & T * F \\
(4) & T & \to & F \\
(5) & F & \to & (E) \\
(6) & F & \to & \mathbf{id}
\end{array}
$$

stav	*akce*						*přechody*		
	id	+	*	(	)	$	E	T	F
0	s5			s4			1	2	3
1		s6				acc			
2		r2	s7		r2	r2			
3		r4	r4		r4	r4			
4	s5			s4			8	2	3
5		r6	r6		r6	r6			
6	s5			s4				9	3
7	s5			s4					10
8		s6			s11				
9		r1	s7		r1	r1			
10		r3	r3		r3	r3			
11		r5	r5		r5	r5			

Obr. 3.10: Rozkladová tabulka LR(1)

Kódování akcí v rozkladové tabulce je následující:

- si znamená přesun a nový stav na vrcholu zásobníku i,
- rj znamená redukci podle přepisovacího pravidla s číslem j,
- acc znamená přijetí a
- volné políčko znamená chybu.

Poznamenejme, že hodnota *přechody*$[s, a]$ pro daný terminální symbol a se takto hledá v tabulce akcí, neboť je spojena s akcí přesunu vstupního symbolu a. Tabulka přechodů obsahuje *přechody*$[s, A]$ pouze pro nonterminální symboly A. Demonstrujme nyní v krátkosti, jak jsou z rozkladové tabulky vybírány akce pro vstupní větu $\mathbf{id} * \mathbf{id} + \mathbf{id}$.

Pro vstupní větu $\mathbf{id} * \mathbf{id} + \mathbf{id}$ je posloupnost stavů zásobníku a nepřečtených částí vstupního řetězce zobrazena na obr. 3.11. Např. na řádku (1) je LR analyzátor ve stavu 0 a aktuálním vstupním symbolem je **id**. Akcí v řádku 0 a sloupci **id** tabulky akcí je s5. To znamená akci přesunu a umístění stavu 5 na vrchol zásobníku. Na řádku (2) je uveden výsledek této akce: první symbol **id** a stavový symbol 5 budou umístěny na vrchol zásobníku a ze vstupu bude odstraněn symbol **id**.

Aktuálním vstupním symbolem se stane * a akcí ve stavu 5 při vstupním symbolu * je redukce podle pravidla $F \to \mathbf{id}$. Z vrcholu zásobníku jsou odstraněny dva symboly (jeden stavový symbol

	ZÁSOBNÍK	VSTUP	AKCE
(1)	0	**id** $*$ **id** $+$ **id** \$	přesun
(2)	0 **id** 5	$*$ **id** $+$ **id** \$	redukce podle $F \to$ **id**
(3)	0 F 3	$*$ **id** $+$ **id** \$	redukce podle $T \to F$
(4)	0 T 2	$*$ **id** $+$ **id** \$	přesun
(5)	0 T 2 $*$ 7	**id** $+$ **id** \$	přesun
(6)	0 T 2 $*$ 7 **id** 5	$+$ **id** \$	redukce podle $F \to$ **id**
(7)	0 T 2 $*$ 7 F 10	$+$ **id** \$	redukce podle $T \to T * F$
(8)	0 T 2	$+$ **id** \$	redukce podle $E \to T$
(9)	0 E 1	$+$ **id** \$	přesun
(10)	0 E 1 $+$ 6	**id** \$	přesun
(11)	0 E 1 $+$ 6 **id** 5	\$	redukce podle $F \to$ **id**
(12)	0 E 1 $+$ 6 F 3	\$	redukce podle $T \to F$
(13)	0 E 1 $+$ 6 T 9	\$	redukce podle $E \to E + T$
(14)	0 E 1	\$	přijetí

Obr. 3.11: Analýza věty **id** $*$ **id** $+$ **id**

5 a jeden symbol gramatiky **id**). Na vrchol zásobníku se dostane stav 0. Protože hodnotou funkce přechodů pro stav 0 a symbol F je 3, jsou na vrchol zásobníku umístěny symboly F a 3. Nyní jsme v konfiguraci, které odpovídá řádek (3). Obdobným způsobem můžeme rekonstruovat i další provedené akce. ■

3.4.2 LR gramatiky

Nyní přistoupíme k otázce, jak sestrojit LR rozkladovou tabulku pro danou gramatiku. Gramatiky, pro něž je to možné, nazveme *LR gramatiky*. Existují bezkontextové gramatiky, které nejsou LR, ale pro typické konstrukce programovacích jazyků je možné se takovým gramatikám vyhnout. Intuitivně je gramatika LR, pokud je analyzátor schopen rozeznat pracovní frázi, když se tato nachází na vrcholu zásobníku. LR analyzátor by neměl prohlížet celý zásobník, aby poznal, kdy se pracovní fráze nachází na vrcholu. Naopak, stavový symbol na vrcholu by měl obsahovat všechny potřebné informace. Význačným faktem je, že pokud je možné rozpoznat pracovní frázi pouze se znalostí symbolů v zásobníku, pak existuje konečný automat, který je schopen při čtení symbolů gramatiky a přesouvání na zásobník směrem ode dna k vrcholu rozpoznat pracovní frázi, pokud je tato fráze na vrcholu zásobníku. Funkce přechodů LR rozkladové tabulky je přechodovou funkcí právě takového konečného automatu. Pro automat není možné, aby zkoumal celý zásobník při každé akci. Stavový symbol na vrcholu zásobníku je stavem konečného automatu rozpoznávajícího pracovní fráze a je známo, že zásobník je plněn symboly gramatiky ode dna k vrcholu. Potom LR analyzátor může určit ze stavu na vrcholu zásobníku všechno, co je třeba znát o jeho obsahu.

Dalším zdrojem informací, který může LR analyzátor užít pro určení akce je následujících k symbolů vstupního řetězce. Pouze případy $k = 0$ a $k = 1$ jsou prakticky použitelné a my budeme uvažovat pouze LR analyzátory $k \leq 1$. Např. tabulka akcí z obr. 3.10 užívá pohled vpřed do vstupního řetězce délky jeden symbol. Gramatika, která může být analyzována LR analyzátorem při pohledu vpřed délky maximálně k vstupních symbolů se nazývá *LR(k) gramatika*.

Mezi LL a LR gramatikami jsou některé význačné rozdíly. Aby byla gramatika typu LR(k),

musíme být schopni rozpoznat pravou stranu přepisovacího pravidla, přičemž je nám známo vše, co lze z této pravé strany derivovat a navíc ještě k dalších vstupních symbolů. Tento požadavek je mnohem méně přísný, nežli pro LL(k) gramatiky, kde musíme být schopni rozpoznat pravidlo při znalosti prvních k symbolů derivovaných z jeho pravé strany. Z toho plyne, že LR gramatiky popisují mnohem více jazyků, nežli LL gramatiky.

3.4.3 Konstrukce rozkladových tabulek

Ukažme nyní, jakým způsobem vytvořit LR rozkladovou tabulku z LR gramatiky. Budeme diskutovat tři metody, které se liší v síle a složitosti implementace. První je nazvaná "jednoduché LR" nebo zkráceně SLR (simple LR). Je nejslabší z hlediska počtu gramatik, pro něž je konstrukce úspěšná. Rozkladovou tabulku vytvořenou touto metodou budeme nazývat *SLR rozkladovou tabulkou* a LR analyzátor užívající SLR rozkladovou tabulku *SLR analyzátorem*. Gramatiku, pro níž lze sestrojit SLR analyzátor budeme nazývat *SLR gramatikou*. Další dvě metody rozšiřují SLR metodu o informace pohledu vpřed (lookahead), takže SLR metoda je dobrou startovní pozicí pro studium LR analýzy.

LR(0) položkou (zkráceně pouze *položkou*) gramatiky G je přepisovací pravidlo gramatiky s tečkou označující pozici v pravé straně pravidla. Tak např. pravidlo $A \to XYZ$ generuje čtyři položky:

$$\begin{aligned} A &\to \cdot XYZ \\ A &\to X \cdot YZ \\ A &\to XY \cdot Z \\ A &\to XYZ \cdot \end{aligned}$$

Přepisovací pravidlo $A \to \epsilon$ generuje pouze jedinou položku $A \to \cdot$. Položka může být reprezentována dvojicí celých čísel, první z nich udává pořadové číslo přepisovacího pravidla a druhé pozici tečky. Méně formálně řečeno, položka indikuje, jaký díl pravidla jsme prohlédli v daném okamžiku procesu analýzy. Např. shora uvedená první položka indikuje, že hodláme na vstupu prohlížet řetězec derivovatelný z XYZ. Druhá položka indikuje, že jsme právě prohlédli na vstupu řetězec derivovatelný z X a že budeme zpracovávat řetězec derivovatelný z YZ.

Základní ideou při konstrukci SLR rozkladové tabulky je vytvoření deterministického konečného automatu rozpoznávajícího perspektivní prefixy. Položky sloučíme do množin, které budou tvořit stavy SLR analyzátoru. Položky mohou být chápány jako stavy nedeterministického konečného automatu rozpoznávajícího prespektivní prefixy a slučování položek do množin je klasickým převodem NKA na DKA.

Soubor LR(0) položek, který se nazývá *kanonický* soubor LR(0) položek je základem konstrukce SLR analyzátorů. Dříve, než popíšeme konstrukci kanonického souboru LR(0) položek, definujme pojem rozšířené gramatiky a dvě funkce *Closure* a *Goto*.

Je-li G gramatika se startovacím symbolem S, potom *rozšířená gramatika* G' pro G je G s novým startovacím symbolem S' a novým pravidlem $S' \to S$. Účelem nového "startovacího" přepisovacího pravidla je umožnit analyzátoru, aby mohl ukončit analýzu a ohlásit přijetí vstupního řetězce. Tj. přijetí je aktuální tehdy a jen tehdy, když analyzátor redukuje pravidlo $S' \to S$.

Operace *Closure* (uzávěr)

Je-li I množina položek gramatiky G, potom uzávěr $Closure(I)$ je množina položek vytvořená podle následujících dvou pravidel:

- Na počátku je každá položka z I také prvkem $Closure(I)$.
- Pokud $A \to \alpha \cdot B\beta$ (B je nonterminální symbol) je v $Closure(I)$ a $B \to \gamma$ je přepisovací pravidlo, potom přidej $B \to \cdot\gamma$ do $Closure(I)$, pokud v této množině dosud není. Toto pravidlo bude aplikováno tak dlouho, dokud jsou přidávány do $Closure(I)$ nové položky.

Příklad 3.13. Uvažujme následující rozšířenou gramatiku:

$$\begin{array}{lcl} E' & \to & E \\ E & \to & E \ + \ T \mid T \\ T & \to & T \ * \ F \mid F \\ F & \to & (E) \mid \mathbf{id} \end{array}$$

Pokud I je množina položek obsahující jedinou položku $\{[E' \to \cdot E]\}$, potom $Closure(I)$ obsahuje následující položky:

$$\begin{array}{lcl} E' & \to & \cdot E \\ E & \to & \cdot E \ + \ T \\ E & \to & \cdot T \\ T & \to & \cdot T \ * \ F \\ T & \to & \cdot F \\ F & \to & \cdot(E) \\ F & \to & \cdot\mathbf{id} \end{array}$$

Postup konstrukce je následující. Nejprve do množiny $Closure(I)$ uložena položka $E' \to \cdot E$ podle prvního pravidla. Protože se E nachází bezprostředně vpravo od tečky, potom podle druhého pravidla konstrukce přidáme všechna E-pravidla s tečkou na začátku, tj. $E \to \cdot E + T$ a $E \to \cdot T$. Nyní se T nachází bezprostředně vpravo od tečky, proto přidáme další položky $T \to \cdot T \ * \ F$ a $T \to \cdot F$. Nyní se F nachází bezprostředně vpravo od tečky, což způsobí přidání položek $F \to \cdot(E)$ a $F \to \cdot\mathbf{id}$. Žádné další položky již podle druhého pravidla konstrukce nepřidáváme (a ani žádné další neexistují). ■

Algoritmus výpočtu funkce *Closure* je následující:

function $Closure(I)$;
begin
  $J := I$;
  repeat
    for každou položku $A \to \alpha \cdot B\beta$
    a každé přepisovací pravidlo $B \to \gamma$ gramatiky G
    takové, že $B \to \cdot\gamma$ není dosud v J **do**
      přidej $B \to \cdot\gamma$ do J
  until nelze přidat další položku do J;
  return J;
end

Vhodným způsobem implementace funkce *Closure* je booleovské pole *added*, indexované nonterminály gramatiky G. Hodnota $added[B]$ je nastavena na **true**, pokud přidáváme položky $B \to \cdot\gamma$ pro některé B-pravidlo $B \to \gamma$.

Poznamenejme, že když je některé z B-pravidel přidáno do uzávěru I s tečkou na levém konci, potom jsou podobně přidány do uzávěru všechna B-pravidla. V některých případech není také nezbytné vyjmenovávat všechny položky $B \to \gamma$ přidávané do uzávěru. V takových případech postačuje seznam nonterminálů B. Z tohoto hlediska můžeme množinu položek rozdělit na dvě třídy:

- *Položky jádra*, které zahrnují počáteční položku $S' \to \cdot S$ a všechny položky, které nemají tečku na levém konci.
- *Položky mimo jádro*, které mají tečku na levém konci.

Každou množinu položek, kterou se budeme zabývat lze sestrojit uzávěrem položek jádra; položky přidávané uzávěrem nejsou nikdy součástí jádra. Z toho plyne možnost reprezentovat množinu položek s malou spotřebou paměti počítače — je možné vyloučit položky mimo jádro s tím, že mohou být kdykoliv vypočítány funkcí *Closure*.

Operace *Goto*

Druhou užitečnou funkcí je $Goto(I, X)$, kde I je množina položek a X je symbol gramatiky. $Goto(I, X)$ je definována jako uzávěr množiny položek $[A \to \alpha X \cdot \beta]$ takové, že $[A \to \alpha \cdot X\beta]$ je v I. Neformálně řečeno, pokud I je množinou položek, která je platná pro nějaký perspektivní prefix γ, potom $Goto(I, X)$ je množinou položek, která je platná pro perspektivní prefix γX.

Příklad 3.14. Pokud máme k dispozici množinu položek $\{[E' \to E\cdot], [E \to E \cdot + T]\}$, potom $Goto(I, +)$ obsahuje položky:

$$\begin{array}{lcl} E & \to & E \ + \ \cdot T \\ T & \to & \cdot T \ * \ F \\ T & \to & \cdot F \\ F & \to & \cdot(E) \\ F & \to & \cdot \mathbf{id} \end{array}$$

Hodnotu $Goto(I, +)$ vypočteme tak, že zkoumáme všechny položky v I, které obsahují $+$ bezprostředně vpravo od tečky. $[E' \to E\cdot]$ není takovou položkou, avšak $[E \to E \cdot + T]$ tuto vlastnost má. Přesuneme tečku přes $+$ a obdržíme $\{[E \to E \ + \ \cdot T]\}$. Nakonec vypočteme uzávěr této množiny. ■

Konstrukce množiny položek

Nyní již máme k dispozici všechny prostředky k formulaci algoritmu pro konstrukci C, kanonického souboru množin LR(0) položek rozšířené gramatiky G'. Algoritmus je následující:

procedure $Items(G')$;
begin
  $C := \{Closure(\{[S' \to \cdot S]\})\}$;
  repeat
    for každou položku I v C a každý symbol gramatiky X
      takový, že $Goto(I, X)$ není prázdné a není v C **do**

přidej $Goto(I, X)$ do C
until není přidána žádná nová položka do C
end

I_0 :
$E' \to \cdot E$
$E \to \cdot E + T$
$E \to \cdot T$
$T \to \cdot T * F$
$T \to \cdot F$
$F \to \cdot(E)$
$F \to \cdot \mathbf{id}$

I_1 :
$E' \to E\cdot$
$E \to E \cdot + T$

I_2 :
$E \to T\cdot$
$T \to T \cdot * F$

I_3 :
$T \to F\cdot$

I_4 :
$F \to (\cdot E)$
$E \to \cdot E + T$
$E \to \cdot T$
$T \to \cdot T * F$
$T \to \cdot F$
$F \to \cdot(E)$
$F \to \cdot \mathbf{id}$

I_5 :
$F \to \mathbf{id}\cdot$

I_6 :
$E \to E + \cdot T$
$T \to \cdot T * F$
$T \to \cdot F$
$F \to \cdot(E)$
$F \to \cdot \mathbf{id}$

I_7 :
$T \to T * \cdot F$
$F \to \cdot(E)$
$F \to \cdot \mathbf{id}$

I_8 :
$F \to (E\cdot)$
$E \to E \cdot + T$

I_9 :
$E \to E + T\cdot$
$T \to T \cdot * F$

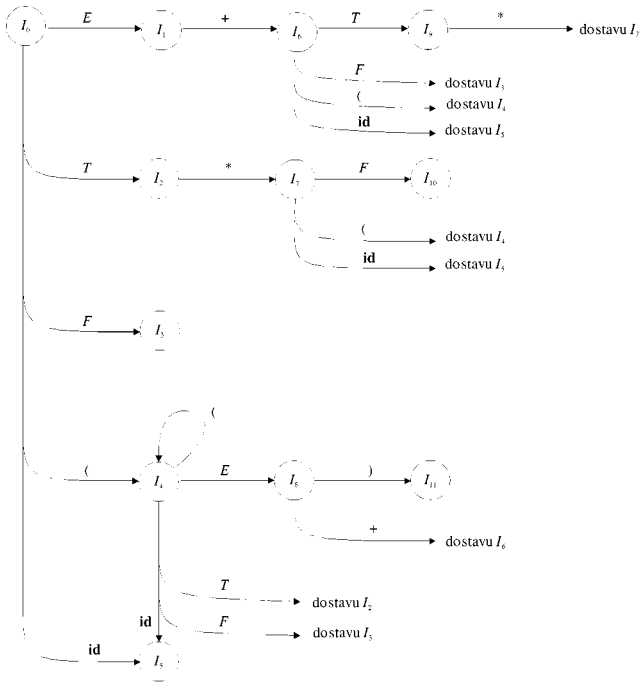
I_{10} :
$T \to T * F\cdot$

I_{11} :
$F \to (E)\cdot$

Obr. 3.12: Kanonická množina LR(0) položek

Příklad 3.15. Kanonický soubor množin LR(0) položek pro gramatiku z příkladu 3.13 uvedeme nyní. Funkce *Goto* pro tyto množiny položek je zobrazena ve formě diagramu přechodů deterministického konečného automatu D na obr. 3.13. ■

Je-li každý za stavů diagramu D na obr. 3.13 koncový a I_0 je počáteční stav, pak D vlastně rozpoznává perspektivní prefixy dané gramatiky. To platí obecně, tj. pro každou gramatiku funkce *Goto* kanonického souboru množin položek definuje deterministický konečný automat, který rozpoznává perspektivní prefixy gramatiky G. Lze pochopitelně také zobrazit nedeterministický konečný automat N, jehož stavy jsou položky samotné. V něm existuje přechod z položky $A \to \alpha \cdot X\beta$ do $A \to \alpha X \cdot \beta$ označený X a existuje přechod z $A \to \alpha \cdot B\beta$ do $B \to \cdot\gamma$ označený ϵ. Funkce $Closure(I)$ pro množinu stavů I automatu N je funkcí ϵ-uzávěr, známou z převodu rozšířeného nedeterministického automatu na deterministický. Funkce $Goto(I, X)$ určuje přechody z I přes symbol X v DKA vytvořeném z N obvyklým převodem na deterministický automat. Z tohoto pohledu je procedura *Items* jen jiným zápisem převodu nedeterministického konečného automatu na deterministický, tak jak jej známe.



Obr. 3.13: Diagram přechodů konečného automatu pro perspektivní prefixy

Platné položky

Říkáme, že položka $A \to \beta_1 \cdot \beta_2$ je *platná* pro perspektivní prefix $\alpha\beta_1$, pokud existuje derivace $S' \overset{*}{\Rightarrow} \alpha A w \Rightarrow \alpha\beta_1\beta_2 w$. Obecně tedy může být jedna položka platná pro více perspektivních prefixů. Fakt, že $A \to \beta_1 \cdot \beta_2$ je platná pro $\alpha\beta_1$ nám dává informace o tom, zda přesouvat nebo redukovat, nalezneme-li $\alpha\beta_1$ na vrcholu zásobníku. Pokud $\beta_2 \neq \epsilon$, potom není ještě celá pracovní fráze na vrcholu zásobníku a vhodnou operací je *přesun*. Pokud $\beta_2 = \epsilon$, potom je jasné, že $A \to \beta_1$ je pracovní frází a můžeme provést *redukci* podle tohoto pravidla. Ovšem, že dvě platné položky nám mohou specifikovat různé akce pro tentýž perspektivní prefix. Některé z těchto konfliktů můžeme řešit pohledem na další vstupní symbol. Nelze však tvrdit, že všechny konflikty toho typu lze řešit metodami LR analýzy a tudíž, že lze vždy vytvořit LR rozkladovou tabulku pro libovolnou gramatiku.

Můžeme jednoduše spočítat množinu platných položek pro každý perspektivní prefix, který se může vyskytnout na vrcholu zásobníku LR analyzátoru. Základním teorémem LR analýzy je, že množina platných položek pro perspektivní prefix γ je právě množinou položek dosažitelných ze startovacího stavu podél cesty ohodnocené γ v DKA vytvořeném z kanonického souboru množin položek s přechody danými funkcí *Goto*. Shrnuto, množina platných položek zahrnuje všechny užitečné informace, které mohou být získány ze zásobníku.

Příklad 3.16. Uvažujme gramatiku z příkladu 3.13. Množiny položek a funkce *Goto* jsou uvedeny v příkladu 3.15. Je jasné, že řetězec $E + T *$ je perspektivním prefixem této gramatiky. Automat z obr. 3.13 bude po přečtení řetězce $E + T *$ ve stavu I_7. Stav I_7 obsahuje položky

$$\begin{array}{lcl} T & \to & T * \cdot F \\ F & \to & \cdot(E) \\ F & \to & \cdot\mathbf{id} \end{array}$$

které jsou právě platnými položkami pro $E + T *$. Abychom to ozřejmili, uvažujme následující tři pravé derivace:

$$\begin{array}{lcl} E' & \Rightarrow & E + T \Rightarrow E + T * F \\ E' & \Rightarrow & E + T \Rightarrow E + T * F \Rightarrow E + T * (E) \\ E' & \Rightarrow & E + T \Rightarrow E + T * F \Rightarrow E + T * \mathbf{id} \end{array}$$

První derivace ukazuje platnost položky $T \to T * \cdot F$, druhá ukazuje platnost $F \to \cdot(E)$ a třetí platnost položky $F \to \cdot\mathbf{id}$ pro perspektivní prefix $E + T *$. Lze ukázat, že pro $E + T *$ neexistuje jiná platná položka. Důkaz ponecháváme čtenáři. ■

Konstrukce SLR rozkladové tabulky

Nyní ukážeme, jak lze odvodit funkce *akce* a *přechody* z deterministického konečného automatu, který rozpoznává perspektivní prefixy. Náš algoritmus může pro mnohé konstrukce programovacích jazyků selhat, neboť třída SLR gramatik je poměrně malá. Z výchozí gramatiky G provedeme rozšíření na G' a z G' vytvoříme C, kanonický soubor množin položek z G'. Ze souboru C vytvoříme funkce LR analyzátoru *akce* a *přechody* užitím následujícího algoritmu. Kromě C vyžaduje algoritmus znalost hodnoty funkce $FOLLOW(A)$ pro všechny nonterminály gramatiky.

Algoritmus 3.3. (Vytváření SLR rozkladové tabulky)
Vstup. Rozšířená gramatika G'.
Výstup. SLR rozkladová tabulka s funkcemi *akce* a *přechody* pro G.
Metoda.

1. Vytvoř $C = \{I_0, I_1, \ldots, I_n\}$, kanonický soubor množin LR(0) položek pro G'.

2. Stav i je vytvořen z I_i. Funkce analyzátoru pro stav i jsou vytvořeny následovně:

   - Pokud $[A \to \alpha \cdot a\beta]$ je v I_i a $Goto(I_i, a) = I_j$, potom hodnotou funkce $akce[i, a]$ je *přesun* j. a musí být terminální symbol.
   - Pokud je $[A \to \alpha\cdot]$ v I_i, potom hodnotou funkce $akce[i, a]$ je *redukce* podle pravidla $A \to \alpha$, pro všechna a ve $FOLLOW(A)$; nonterminál A nemůže být S'.
   - Pokud je $[S' \to S\cdot]$ v I_i, potom hodnotou funkce $akce[i, \$]$ je *přijetí*.

   Vznikne-li při generování konflikt, potom gramatika není SLR(1). V takovém případě nelze tímto algoritmem vytvořit rozkladovou tabulku.

3. Rozkladová tabulka pro funkci přechodů je vytvořena pro všechny nonterminály A podle předpisu: Pokud $Goto(I_i, A) = I_j$, potom $přechody[i, A] = j$.

4. Všechny ostatní hodnoty nedefinované v bodech (2) a (3) budou mít hodnotu *chyba*.

5. Počáteční stav analyzátoru bude ten, který obsahuje v množině položek $[S' \to \cdot S]$. ■

Rozkladová tabulka se skládá z funkcí akcí a přechodů vytvořených podle předchozího algoritmu a nazývá se *SLR(1) tabulka pro G*. LR analyzátor užívající SLR(1) tabulku pro G se nazývá *SLR(1) analyzátor* pro G a gramatika mající SLR(1) rozkladovou tabulku se nazývá *SLR(1) gramatika*. Často lze vynechat "(1)" za "SLR," protože se nebudeme zabývat analyzátory, které prohlížejí více než jeden symbol v dosud nezpracované části vstupního řetězce.

Příklad 3.17. Vytvořme nyní SLR rozkladovou tabulku pro gramatiku z příkladu 3.13. Kanonický soubor množin LR(0) položek pro tuto gramatiku byl uveden v příkladu 3.15. Nejdříve se budeme zabývat množinou I_0:

$$\begin{aligned} I_0: \quad & E' \to \cdot E \\ & E \to \cdot E + T \\ & E \to \cdot T \\ & T \to \cdot T * F \\ & T \to \cdot F \\ & F \to \cdot(E) \\ & F \to \cdot \mathbf{id} \end{aligned}$$

Položka $F \to \cdot(E)$ dává vzniknout funkční hodnotě $akce[0, (] = přesun\ 4$, položka $F \to \cdot\mathbf{id}$ generuje funkční hodnotu $akce[0, \mathbf{id}] = přesun\ 5$. Jiné položky z I_0 negenerují žádnou akci. Nyní uvažujme I_1:

$$\begin{aligned} I_1: \quad & E' \to E\cdot \\ & E \to E \cdot + T \end{aligned}$$

První položka generuje $akce[1, \$] = přijetí$, druhá položka generuje $akce[1, +] = přesun\ 6$. Dále uvažujme I_2:

$$\begin{array}{ll} I_2: & E \to T\cdot \\ & T \to T \cdot * F \end{array}$$

Protože $FOLLOW(E) = \{\$, +,)\}$ generuje první položka hodnotu $akce[2, \$] = akce[2, +] = akce[2,)] =$ *redukce* $E \to T$. Druhá položka generuje hodnotu $akce[2, *] =$ *přesun* 7. Obdobným způsobem lze obdržet hodnoty rozkladové tabulky i pro další množiny I_j. Výsledná rozkladová tabulka byla již uvedena na obr. 3.10. Na tomto obrázku jsou čísla pravidel v redukčních akcích stejná jako pořadí, ve kterém se vyskytují v původní gramatice v příkladu 3.12. Tedy $E \to E + T$ má číslo 1, $E \to T$ má číslo 2 atd. ■

Příklad 3.18. Žádná SLR(1) gramatika nesmí být víceznačná. Existují však i jednoznačné gramatiky, které nejsou SLR(1). Uvažujme nyní gramatiku s pravidly:

$$\begin{array}{lcl} S & \to & L = R \\ S & \to & R \\ L & \to & * R \\ L & \to & \mathbf{id} \\ R & \to & L \end{array}$$

Můžeme si představit, že L a R představují l-hodnotu a r-hodnotu a $*$ je operátor získání obsahu uloženého na adrese dané operandem. Kanonický soubor množin LR(0) položek pro tuto gramatiku je následující:

$$\begin{array}{lll} I_0: & S' \to \cdot S & \\ & S \to \cdot L = R & \\ & S \to \cdot R & \\ & L \to \cdot * R & \\ & L \to \cdot \mathbf{id} & \\ & R \to \cdot L & \\ & & \\ I_1: & S' \to S\cdot & \\ & & \\ I_2: & S \to L\cdot = R & \\ & R \to L\cdot & \\ & & \\ I_3: & S \to R\cdot & \\ & & \\ I_4: & L \to * \cdot R & \\ & L \to \cdot \mathbf{id} & \\ & R \to \cdot L & \\ & L \to \cdot * R & \end{array} \qquad \begin{array}{lll} I_5: & L \to \mathbf{id}\cdot & \\ & & \\ I_6: & S \to L = \cdot R & \\ & L \to \cdot * R & \\ & L \to \cdot \mathbf{id} & \\ & R \to \cdot L & \\ & & \\ I_7: & L \to * R\cdot & \\ & & \\ I_8: & R \to L\cdot & \\ & & \\ I_9: & S \to L = R\cdot & \end{array}$$

Uvažujme množinu položek I_2. První položka zde generuje hodnotu $akce[2, =] =$ *přesun* 6. Protože $FOLLOW(R)$ obsahuje $=$ (existuje derivace $S \Rightarrow L = R \Rightarrow * R = R$), druhá položka generuje hodnotu $akce[2, =] =$ *redukce* $R \to L$. $akce[2, =]$ je tedy vícenásobně definována. Protože jsou zde dvě různé hodnoty *přesun* a *redukce* pro $akce[2, =]$, má stav 2 konflikt *přesun/redukce* pro vstupní symbol $=$. ■

Gramatika z předchozího příkladu není víceznačná. Konflikt *přesun/redukce* se odvíjí z faktu, že SLR analyzátor není dostatečně mocný, aby si zapamatoval dostatečný levý kontext pro rozhodnutí, která akce se má provést při vstupu =, existuje-li řetězec redukovatelný na L. Kanonická LR metoda a LALR metoda, které budou diskutovány dále jsou úspěšné pro větší třídu gramatik, včetně této. Poznamenejme, že existují jednoznačné gramatiky, pro které vznikají LR rozkladové tabulky s konflikty u všech metod. Naštěstí tyto gramatiky lze obvykle pominout při vytváření překladačů programovacích jazyků.

Konstrukce kanonické LR rozkladové tabulky

Nyní uvedeme nejobecnější techniku pro vytváření LR rozkladové tabulky z gramatiky. Připomeňme, že v metodě SLR je ve stavu i volána redukce pro pravidlo $A \to \alpha$, pokud množina položek obsahuje $[A \to \alpha\cdot]$ a a je ve $FOLLOW(A)$. Avšak v některých situacích, nachází-li se stav i na vrcholu zásobníku, má perspektivní prefix $\beta\alpha$ tu vlastnost, že βA nesmí být následováno symbolem a v žádné pravé větné formě. Potom by redukce podle pravidla $A \to \alpha$ při vstupu a byla nesprávná.

Příklad 3.19. Vraťme se nyní ještě jednou k příkladu 3.18, kde ve stavu 2 jsme měli položku $R \to L\cdot$, která by mohla odpovídat naší shora uvedené položce $A \to \alpha\cdot$, přičemž znaménko $=$ z množiny $FOLLOW(R)$ pak odpovídá a. Analyzátor SLR volá ve stavu 2 redukci pro vstupní symbol $=$ (protože je zde konflikt, analyzátor volá ve stavu 2 i akci *přesun* pro položku $S \to L\cdot = R$). Nicméně v gramatice z příkladu 3.18 neexistuje pravá větná forma, která by začínala $R = \ldots$. Proto ve stavu 2, který je stavem odpovídajícím pouze perspektivnímu prefixu L, není možné ve skutečnosti volat redukci L na R. ■

Ve stavu je možné shromáždit více informací, které nám umožní vyloučit nesprávné redukce podle $A \to \alpha$. Rozdělením stavu v případech, kdy je to nezbytné je možné dosáhnout toho, že každý stav LR analyzátoru indikuje přesně, který vstupní symbol může následovat za pracovní frází α a pro který tudíž je možná redukce na A.

Dodatečné informace vložíme do stavů tak, že k položkám přidáme terminální symbol jako jejich druhou část. Obecný tvar položky budek potom $[A \to \alpha \cdot \beta, a]$, kde $A \to \alpha\beta$ je přepisovací pravidlo a a je terminální symbol nebo koncový symbol \$. Takový objekt nazveme *LR(1) položka.* 1 nám udává délku druhé části položky, kterou nazýváme *pohled vpřed* (lookahead) položky (pohledy vpřed délky větší než 1 jsou rovněž možné, my je však nebudeme uvažovat). Pohled vpřed nemá vliv v položkách tvaru $[A \to \alpha \cdot \beta, a]$, kde β není ϵ. Na druhé straně položka $[A \to \alpha\cdot, a]$ generuje redukci podle pravidla $A \to \alpha$ pouze v případě, že vstupní symbol je a. Množina všech takových a je jistě podmnožinou $FOLLOW(A)$, ale měla by být vlastní podmnožinou, jako v příkladu 3.19.

Formálně říkáme, že $[A \to \alpha\cdot\beta, a]$ je *platná* pro perspektivní prefix γ, pokud existuje derivace $S \stackrel{*}{\Rightarrow} \delta Aw \Rightarrow \delta\alpha\beta w$, kde

- $\gamma = \delta\alpha$
- buďto a je první symbol w nebo w je ϵ a a je \$.

Příklad 3.20. Uvažujme následující gramatiku

$$\begin{aligned} S &\to BB \\ B &\to aB \mid b \end{aligned}$$

Existuje pravá derivace $S \stackrel{*}{\Rightarrow} aaBab \Rightarrow aaaBab$. Vidíme, že položka $[B \to a \cdot B, a]$ je platná pro perspektivní prefix $\gamma = aaa$, pokud ztotožníme $\delta = aa$, $A = B$, $w = ab$, $\alpha = a$ a $\beta = B$ podle předchozí definice.

Existuje rovněž pravá derivace $S \stackrel{*}{\Rightarrow} BaB \Rightarrow BaaB$. Z této derivace vidíme, že položka $[B \to a \cdot B, \$]$ je platná pro perspektivní prefix Baa. ■

Metoda konstrukce souboru množin LR(1) položek je v zásadě stejná jako pro LR(0) položky. Je jen nutné modifikovat procedury *Closure* a *Goto*.

Abychom objasnili novou definici operace *Closure*, uvažujme položku ve tvaru $[A \to \alpha \cdot B\beta, a]$ v množině platných položek pro nějaký perspektivní prefix γ. Potom existuje pravá derivace $S \stackrel{*}{\Rightarrow} \delta Aax \Rightarrow \delta\alpha B\beta ax$, kde $\gamma = \delta\alpha$.

Dále předpokládejme, že βax derivuje terminální řetězec by. Potom pro každé pravidlo ve tvaru $B \to \eta$ pro nějaké η, existuje derivace $S \stackrel{*}{\Rightarrow} \gamma Bby \Rightarrow \gamma\eta by$. Tedy $[B \to \cdot\eta, b]$ je platná pro η. Poznamenejme, že b může být první terminál derivovaný z β, nebo je možné, že β derivuje ϵ v derivaci $S \stackrel{*}{\Rightarrow} by$ a proto b může být rovno a. Abychom shrnuli obě možnosti, říkáme, že b může být libovolný terminál z $FIRST(\beta ax)$.

Algoritmus 3.4. (Konstrukce množiny LR(1) položek)

Vstup. Rozšířená gramatika G'.

Výstup. Množina LR(1) položek, které jsou platné pro jeden nebo více perspektivních prefixů z G'.

Metoda. Užijeme procedur *Closure*, *Goto* a hlavní proceduru *Items* následujícího tvaru:

```
function Closure(I);
begin
  repeat
    for každou položku[A → α · Bβ, a]v I a
      každé přepisovací pravidlo B → γ gramatiky G'.
      Dále pro každý terminální symbol b z FIRST(βa) takový,
      že [B → ·γ, b] není dosud v I do
        přidej [B → ·γ, b] do I
  until nelze přidat další položku do I;
  return I;
end;

function Goto(I, X);
begin
  nechť J je množina položek [A → αX · β, a] takových,
  že [A → α · Xβ, a] je v I;
  return Closure(J)
end;

procedure Items(G');
begin
  C := {Closure({[S' → ·S, $]})};
  repeat
```

for každou položku I v C a každý symbol gramatiky X takový,
že $Goto(I, X)$ není prázdné a není v C **do**
přidej $Goto(I, X)$ do C
until není přidána žádná nová položka do C
end;

■

Příklad 3.21. Uvažujme následující rozšířenou gramatiku:

$$\begin{array}{rcl} S' & \to & S \\ S & \to & CC \\ C & \to & cC \mid d \end{array}$$

Začneme s výpočtem uzávěru $\{[S' \to \cdot S, \$]\}$. Abychom to mohli provést srovnáme položku $[S' \to \cdot S, \$]$ s položkou $[A \to \alpha \cdot B\beta, a]$ v proceduře *Closure*. V tom případě $A = S', \alpha = \epsilon, B = S, \beta = \epsilon$ a $a = \$$. Funkce *Closure* nám předepisuje přidat položku $[B \to \cdot\gamma, b]$ pro každé přepisovací pravidlo $B \to \gamma$ a terminální symbol b, z $FIRST(\beta a)$. V pojmech uvažované gramatiky je $B \to \gamma$ rovno $S \to CC$ a protože β je ϵ a a je \$, b může být pouze \$. Přidáme tedy položku $[S \to \cdot CC, \$]$.

Ve výpočtu uzávěru pokračujeme přidáním všech položek $[C \to \cdot\gamma, b]$ pro b ve $FIRST(C\$)$. Pokud se na $[S \to \cdot CC, \$]$ podíváme jako na $[A \to \alpha \cdot B\beta, a]$, potom $A = S', \alpha = \epsilon, B = C, \beta = C$ a $a = \$$. Protože z C nelze derivovat prázdný řetězec, $FIRST(C\$) = FIRST(C)$. Protože $FIRST(C)$ obsahuje terminální symboly c a d, přidáme k uzávěru položky $[C \to \cdot cC, c]$, $[C \to \cdot cC, d]$, $[C \to \cdot d, c]$ a $[C \to \cdot d, d]$. Nyní již žádná další položka neobsahuje nonterminální symbol bezprostředně vpravo od tečky, takže množina LR(1) položek je kompletní. Tím vznikne počáteční množina položek I_0:

$$\begin{array}{rl} I_0: & S' \to \cdot S, \$ \\ & S \to \cdot CC, \$ \\ & C \to \cdot cC, c/d \\ & C \to \cdot d, c/d \end{array}$$

Z důvodu větší přehlednosti jsme vynechali hranaté závorky a užili jsme zápis $[C \to \cdot cC, c/d]$ jako zkratku pro dvojici položek $[C \to \cdot cC, c]$, $[C \to \cdot cC, d]$.

Nyní budeme počítat $Goto(I_0, X)$ pro různá X. Pro $X = S$ musíme vypočítat uzávěr položky $[S' \to S\cdot, \$]$. Nepřidává se žádná další položka, neboť tečka je na pravém konci pravidla. Máme tedy množinu položek

$$\begin{array}{rl} I_1: & S' \to S\cdot, \$ \end{array}$$

Pro $X = C$ provedeme uzávěr $[S \to C \cdot C, \$]$. Přidáme C-pravidla s druhou položkou \$ a dále již nelze přidat žádná položka.

$$\begin{array}{rl} I_2: & S \to C \cdot C, \$ \\ & C \to \cdot cC, \$ \\ & C \to \cdot d, \$ \end{array}$$

Nechť $X = c$. Vypočteme tedy uzávěr $\{[C \to c\cdot C, c/d]\}$. Výsledek obdržíme přidáním C-pravidel ke druhé částí položky c/d. Potom

$$\begin{array}{rl} I_3: & C \to c \cdot C, c/d \\ & C \to \cdot cC, c/d \\ & C \to \cdot d, c/d \end{array}$$

Nakonec pro $X = d$ získáme množinu položek

$$I_4: \quad C \to d\cdot, c/d$$

Tímto výpočtem jsme ukončili výpočet *Goto* pro I_0. I_1 negeneruje žádné další množiny položek, avšak I_2 generuje přechody přes C, c a d. Přechod přes C nám dává

$$I_5: \quad S \to CC\cdot, \$$$

a není nutné počítat uzávěr. Pro c budeme počítat uzávěr $\{[C \to c \cdot C, \$]\}$ a obdržíme

$$\begin{aligned} I_6: \quad & C \to c \cdot C, \$ \\ & C \to \cdot cC, \$ \\ & C \to \cdot d, \$ \end{aligned}$$

Vidíme, že I_6 se liší od I_3 jen ve druhých částech položek. Obecným rysem mnohých množin LR(1) položek je to, že mají stejné první části položek a liší se pouze v druhých částech položek. Pokud vytvoříme kanonický soubor LR(0) položek pro tutéž gramatiku, každá z množin LR(0) položek odpovídá množině prvních částí položek jedné nebo více množin souboru LR(1). O této vlastnosti budeme mluvit více u metody LALR.

Budeme pokračovat výpočtem $Goto(I_2, d)$

$$I_7: \quad C \to d\cdot, \$$$

Nyní obrátíme pozornost k I_3. Přechody z I_3 přes c a d jsou I_3 a I_4 a $Goto(I_3, C)$ je

$$I_8: \quad C \to cC\cdot, c/d$$

I_4 a I_5 nemají žádný přechod. Přechody pro I_6 přes c a d jsou I_6 a I_7. Zbývá $Goto(I_6, C)$, která je rovna

$$I_9: \quad C \to cC\cdot, \$$$

Zbývající množiny položek nevytvářejí žádné přechody, čímž je algoritmus ukončen. Na obr. 3.14 jsou zobrazeny jednotlivé množiny položek spolu s jejich přechody. ■

Nyní již máme definovány prostředky proto, abychom mohli definovat algoritmus pro vytvoření rozkladové tabulky z kanonického souboru LR(1) položek. Funkce *akce* a *přechody* jsou v tabulce reprezentovány stejně jako u SLR analyzátoru. Jediným rozdílem jsou hodnoty položek.

Algoritmus 3.5. (Vytvoření kanonické LR rozkladové tabulky)
Vstup. Rozšířená gramatika G'.
Výstup. Kanonická LR rozkladová tabulka s funkcemi *akce* a *přechody* pro G'.
Metoda.

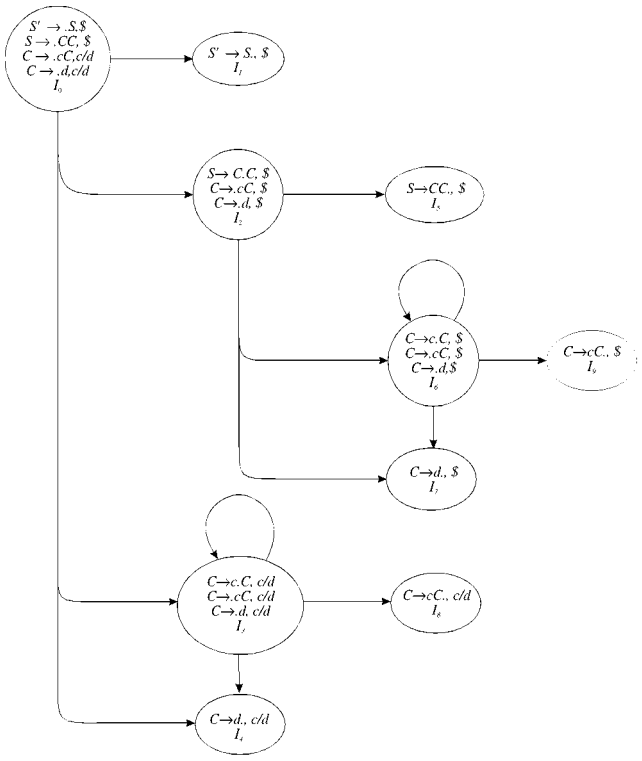
1. Vytvoř $C = \{I_0, I_1, \ldots, I_n\}$, kanonický soubor množin LR(1) položek pro G'.

2. Stav i je vytvořen z I_i. Funkce analyzátoru pro stav i jsou vytvořeny následovně:

   - Pokud $[A \to \alpha \cdot a\beta, b]$ je v I_i a $Goto(I_i, a) = I_j$, potom hodnotou funkce $akce[i, a]$ je *přesun* j. a musí být terminální symbol.
   - Pokud $[A \to \alpha\cdot, a]$ je v I_i, potom hodnotou funkce $akce[i, a]$ je *redukce* podle pravidla $A \to \alpha$; nonterminál A nemůže být S'.
   - Pokud $[S' \to S\cdot, \$]$ je v I_i, potom hodnotou funkce $akce[i, \$]$ je *přijetí*.

   Vznikne-li při generování konflikt, potom gramatika není LR(1). V takovém případě nelze tímto algoritmem vytvořit rozkladovou tabulku.

3. Rozkladová tabulka pro funkci přechodů je vytvořena pro všechny nonterminály A podle předpisu: Pokud $Goto(I_i, A) = I_j$, potom $přechody[i, A] = j$.



Obr. 3.14: Graf přechodů

4. Všechny ostatní hodnoty nedefinované v bodech (2) a (3) budou mít hodnotu *chyba*.

5. Počáteční stav analyzátoru bude ten, který obsahuje v množině položek $[S' \to \cdot S, \$]$. ■

Tabulka vytvořená podle algoritmu 3.5 se nazývá *kanonická* LR(1) rozkladová tabulka. LR analyzátor užívající tuto tabulku se nazývá kanonický LR(1) analyzátor. Pokud funkce akcí nemá násobně definované položky, potom daná gramatika se nazývá *LR(1) gramatika*. Podobně jako u SLR i zde často vynecháváme "(1)," pokud nemůže dojít k nejednoznačnostem.

Příklad 3.22. Kanonická rozkladová tabulka pro gramatiku z příkladu 3.21 je ukázána na obr. 3.15. Pravidla 1, 2 a 3 jsou $S \to CC$, $C \to cC$ a $C \to d$. ■

STAV	*akce*			*přechody*	
	c	d	\$	S	C
0	s3	s4		1	2
1			acc		
2	s6	s7			5
3	s3	s4			8
4	r3	r3			
5			r1		
6	s6	s7			9
7			r3		
8	r2	r2			
9			r2		

Obr. 3.15: Kanonická LR(1) rozkladová tabulka

Každá SLR(1) gramatika je také LR(1), avšak kanonický LR(1) analyzátor pro SLR(1) gramatiku může mít více stavů, nežli SLR analyzátor. Gramatika z předchozího příkladu je SLR a existuje pro ni SLR analyzátor se sedmi stavy.

Konstrukce LALR rozkladové tabulky

Jako poslední uvedeme nyní metodu konstrukce analyzátoru nazvanou technika LALR (*look-ahead LR* - LR s pohledem vpřed). Tato metoda je nejčastěji používána v praxi, protože výsledné rozkladové tabulky jsou značně menší, nežli kanonické LR tabulky, avšak většinu obecných konstrukcí programovacích jazyků lze vhodně vyjádřit LALR gramatikami. Stejné tvrzení platí i pro SLR gramatiky, avšak existuje mnohem více konstrukcí, které nejsou pomocí SLR analyzátoru zpracovatelné (např. gramatika z příkladu 3.18).

Pro srovnání uveďme, že SLR a LALR rozkladové tabulky mají stejný počet stavů a tento počet je pro jazyky typu Pascal několik stovek. Kanonická LR rozkladová tabulka by mohla mít pro jazyk stejné velikosti několik tisíc stavů. Je tedy mnohem jednodušší a mnohem ekonomičtější konstruovat rozkladové tabulky SLR a LALR a nikoliv rozkladové tabulky LR.

Pro účely úvodu do LALR uvažujme i nadále gramatiku z příkladu 3.21, jejíž množiny LR(1) položek byly ukázány na obr. 3.14. Vezměme dvojici podobných stavů I_4 a I_7. Oba tyto stavy mají pouze jednu položku s první částí rovnou $C \to d\cdot$. Ve stavu I_4 jsou pohledy vpřed c a d; ve stavu I_7 je jediný pohled vpřed \$.

Abychom snadněji odhalili rozdíl mezi rolemi stavů I_4 a I_7 v analyzátoru řekněme nejdříve, že gramatika generuje jazyk c^*dc^*d. Během čtení vstupu $cc \ldots cdcc \ldots cd$, analyzátor přesouvá první skupinu znaků c a jejich následné d na zásobník, přičemž po přečtení prvního d přejde do stavu 4. Potom je volána redukce podle pravidla $C \to d$, přičemž na vstupu je buďto vstupní symbol c nebo d. Požadavek c nebo d následující na vstupu má smysl, protože zde mají být symboly řetězce c^*d. Následuje-li na vstupu \$ za prvním d, analyzovali jsme řetězec ccd, který však není v jazyce generovaném gramatikou. Ve stavu 4 je v případě výskytu znaku \$ na vstupu hlášena chyba.

Po přečtení druhého d vstoupí analyzátor do stavu I_7. Zde musí být na vstupu znak \$ nebo nebyl analyzován řetězec generovaný gramatikou. Má tedy smysl, že ve stavu I_7 se provádí redukce podle pravidla $C \to d$ pouze v případě, že na vstupu se vyskytuje znak \$ a znaky c a d způsobí chybu.

Nyní nahraďme stavy I_4 a I_7 stavem I_{47}, sjednocením I_4 a I_7, sestávajícím ze tří položek $[C \to d\cdot, c/d/\$]$. Přechody přes d do stavů I_4 a I_7 vycházející z I_0, I_2, I_3 a I_6 nyní budou směřovat do I_{47}. Akce ve stavu 47 bude redukcí při jakémkoliv vstupu (pouze znaků z abecedy jazyka). Opravený analyzátor se chová v zásadě stejně jako originál, avšak provede redukci d na C i v případě, kdy by původní analyzátor hlásil chybu, například v případě vstupu ccd nebo $cdcdc$. Chyba bude nakonec odhalena, a to dříve, než bude proveden další přesun.

Obecněji řečeno, lze nalézt v množinách kanonického souboru LR(1) položek položky se stejným *jádrem*, tj. množinou prvních částí, a tyto položky se stejným jádrem lze sloučit do jediné množiny. Např. na obr. 3.14 tvoří I_4 a I_7 dvojici s jádrem $\{C \to d\cdot\}$. Obdobně I_3 a I_6 tvoří dvojici s jádrem $\{C \to c\dot{C}, C \to \cdot cC, C \to \cdot d\}$. Dále zde existuje dvojice I_8 a I_9 s jádrem $\{C \to cC\cdot\}$. Poznamenejme, že jádra jsou vlastně množiny LR(0) položek pro danou gramatiku a že LR(1) gramatika může generovat více než dvě množiny položek se stejným jádrem.

Protože $Goto(I, X)$ závisí pouze na jádře I, přechody sloučených množin mohou být rovněž sloučeny. Proto při slučování neexistuje problém vedoucí k přepočítání funkce $Goto$. Funkce akcí je modifikována tak, že odráží v daném stavu všechny nechybové akce sloučených stavů.

Předpokládejme, že máme LR(1) gramatiku, tj. gramatiku jejíž množiny LR(1) položek negenerují konflikty v LR(1) rozkladové tabulce. Pokud nahradíme všechny stavy mající stejné jádro jejich sjednocením, je možné, že výsledné sjednocení bude mít konflikt. Je to ale nepravděpodobné z následujícího důvodu: Předpokládejme, že ve sjednocení je konflikt při pohledu vpřed na a, protože zde existuje položka $[A \to \alpha\cdot, a]$ generující redukci podle pravidla $A \to \alpha$ a existuje zde i jiná položka $[B \to \beta \cdot a\gamma, b]$ generující přesun. Potom některá z množin, ze které bylo vytvořeno sjednocení obsahovala položku $[A \to \alpha\cdot, a]$ a protože jádra položek jsou stejná, musela obsahovat i položku $[B \to \beta \cdot a\gamma, c]$ pro nějaké c. Pak ale tento stav obsahuje tentýž konflikt přesun/redukce, jako stav sloučený a gramatika nebyla LR(1) jak jsme předpokládali. Z toho plyne, že sloučení stavů se stejnými jádry nikdy nevytvoří konflikt přesun/redukce, který se nebyl vyskytoval v původních stavech, protože akce přesunu jsou závislé pouze na jádrech a ne na pohledu vpřed.

Jak ale uvidíme v následujícím příkladu, je možné, že při sloučení vznikne konflikt redukce/redukce.

Příklad 3.23. Uvažujme následující gramatiku

$$\begin{aligned} S' &\to S \\ S &\to aAd \mid bBd \mid aBe \mid bAe \\ A &\to c \end{aligned}$$

$$B \rightarrow c$$

která generuje čtyři řetězce *acd*, *bcd*, *ace* a *bce*. Čtenář může dokázat, že jde o LR(1) gramatiku konstrukcí kanonické LR(1) rozkladové tabulky. Jakmile vytvoříme kanonický soubor položek, můžeme najít množinu položek $\{[A \rightarrow c\cdot, d], [B \rightarrow c\cdot, e]\}$ platných pro perspektivní prefix *ac* a množinu $\{[A \rightarrow c\cdot, e], [B \rightarrow c\cdot, d]\}$ platných pro *bc*. Žádná z těchto množin negeneruje konflikt a jejich jádra jsou stejná. Nicméně jejich sjednocení

$$\begin{aligned} A &\rightarrow c\cdot, d/e \\ B &\rightarrow c\cdot, d/e \end{aligned}$$

generuje konflikt redukce/redukce, protože redukce pro obě přepisovací pravidla $A \rightarrow c, B \rightarrow c$ jsou volány pro vstup jak d, tak e. ■

Pro konstrukci LALR rozkladové tabulky uvedeme dva algoritmy. Hlavní ideou prvního z nich je konstrukce LR(1) položek. Jestliže nevzniknou konflikty, následuje sloučení množin se společnými jádry. Rozkladová tabulka je potom vytvořena za sloučených množin položek. Tato metoda tedy konstruuje LALR rozkladovou tabulku stejným způsobem, jakým jsme LALR(1) gramatiky definovali. Avšak konstrukce celého kanonického souboru množin LR(1) položek vyžaduje velmi mnoho prostoru a času na to, aby byl tento postup prakticky použitelný. Pro pochopení principů konstrukce je však podstatný.

Algoritmus 3.6. (Konstrukce LALR rozkladové tabulky)
Vstup. Rozšířená gramatika G'.
Výstup. LALR rozkladová tabulka s funkcemi *akce* a *přechody* pro G'.
Metoda.

1. Vytvoř $C = \{I_0, I_1, \ldots, I_n\}$, kanonický soubor množin LR(1) položek pro G'.
2. Pro každé jádro vyskytující se v LR(1) položkách nalezni všechny množiny mající toto jádro a nahraď tyto množiny jejich sjednocením.
3. Nechť $C' = \{J_0, J_1, \ldots, J_n\}$ je výsledná množina LR(1) položek. Akce analyzátoru pro stav i jsou vytvářeny z J_i stejným způsobem jako v algoritmu 3.5. Pokud se vyskytne konflikt akcí analyzátoru, říkáme, že gramatika není LALR(1).
4. Tabulka funkce *přechody* je vytvářena následujícím způsobem. Je-li J sjednocení jedné nebo více množin LR(1) položek $J = I_1 \cup I_2 \cup \ldots \cup I_k$, potom jádra množin $Goto(I_1, X), Goto(I_2, X), \ldots Goto(I_k, X)$ jsou stejná, protože $I_1, I_2, \ldots, I_k$ mají stejná jádra. Nechť K je sjednocením množin položek, které mají stejné jádro $Goto(I_i, X)$. Potom $Goto(J, X) = K$. ■

Tabulka vytvořená algoritmem 3.6 se nazývá *LALR rozkladová tabulka* pro G. Nevyskytnou-li se při konstrukci žádné konflikty, říkáme, že jde o *LALR(1) gramatiku*. Soubor množin vytvořený v kroku (3) se nazývá *LALR(1) soubor položek*.

Příklad 3.24. I v tomto příkladu budeme uvažovat gramatiku z příkladu 3.21. Diagram přechodů této gramatiky byl ukázán na obr. 3.14. Jak jsme se již zmínili, jsou zde tři dvojice množin, které mohou být sjednoceny. I_3 a I_6 jsou nahrazeny sjednocením:

$$I_{36}: \quad C \to c \cdot C, c/d/\$$$
$$C \to \cdot cC, c/d/\$$$
$$C \to \cdot d, c/d/\$$$

I_4 a I_7 jsou nahrazeny sjednocením:

$$I_{47}: \quad C \to d\cdot, c/d/\$$$

a I_8 a I_9 jsou nahrazeny sjednocením:

$$I_{89}: \quad C \to cC\cdot, c/d/\$$$

Funkce akcí a přechodů LALR pro uvažované množiny položek jsou ukázány na obr. 3.16.

STAV	*akce*			*přechody*	
	c	d	\$	S	C
0	s36	s47		1	2
1			acc		
2	s36	s47			5
36	s36	s47			89
47	r3	r3	r3		
5			r1		
89	r2	r2	r2		

Obr. 3.16: LALR(1) rozkladová tabulka

Abychom ukázali, jakým způsobem jsou vypočteny hodnoty funkce přechodů, uvažujme $Goto(I_{36}, C)$. V původní množině LR(1) položek platilo $Goto(I_3, C) = I_8$ a I_8 je součástí I_{89}. Z toho plyne, že $Goto(I_{36}, C) = I_{89}$. Budeme-li uvažovat I_6 jako součást I_{36}, měli bychom obdržet stejné důsledky. Tedy $Goto(I_6, C) = I_9$ a I_9 je součástí I_{89}. Jiným příkladem je položka $Goto(I_2, c)$, která je aktuální po akci přesunu z I_2 přes c. V původní množině LR(1) položek je $Goto(I_2, c) = I_6$. Protože I_6 je nyní částí I_{36}, $Goto(I_2, c)$ získá hodnotu I_{36}. Tedy položkou z obr. 3.16 pro stav 2 a vstup c je $s36$, znamenající přesun a umístění stavu 36 na vrchol zásobníku. ■

Pokud zpracováváme řetězec jazyka c^*dc^*d jak LR analyzátorem z obr. 3.15 a LALR analyzátorem z obr. 3.16, oba procházejí stejnou posloupností přesunů a redukcí, i když jména stavů se mohou lišit, tj. pokud LR analyzátor dává na vrchol zásobníku I_3 nebo I_6, potom LALR analyzátor provádí totéž s I_{36}. Tento vztah platí obecně pro LALR gramatiky. LR a LALR analyzátor se budou navzájem napodobovat při zpracování správného vstupu.

Nicméně pokud bude zpracováván chybný vstup, může LALR analyzátor provést ještě několik redukcí navíc po tom, co by LR analyzátor již ohlásil chybu. Nikdy však LALR analyzátor neprovede navíc operaci přesunu po akci, na níž LR analyzátor ohlásil chybu. Uvažujme například vstupní řetězec ccd následovaný \$ a LR analyzátor z obr. 3.15. Obsah zásobníku bude následující

$$0\ c\ 3\ c\ 3\ d\ 4$$

a ve stavu 4 bude rozpoznána chyba, protože \$ následuje ve vstupním řetězci a pro tento symbol má stav 4 akci chyba. Naopak LALR analyzátor z obr. 3.16 bude provádět odpovídající akce a na zásobník umístí

$$0\ c\ 36\ c\ 36\ d\ 47$$

Stav 47 však pro vstupní symbol \$ obsahuje akci redukce podle pravidla $C \to d$. LALR analyzátor

změní obsah zásobníku na

$$0\ c\ 36\ c\ 36\ C\ 89$$

Akcí ve stavu 89 při vstupu \$ je redukce podle pravidla $C \to cC$. Zásobník se změní na

$$0\ c\ 36\ C\ 89$$

Zde je provedena obdobná redukce a stav zásobníku je

$$0\ C\ 2$$

Nakonec stav 2 obsahuje akci chyba pro vstup \$ a je ohlášena chyba. ■

Efektivní konstrukce LALR rozkladové tabulky

Existuje několik modifikací, které se snaží upravit algoritmus 3.6 tak, aby se zabránilo vytváření úplného souboru množin LR(1) položek v procesu vytváření LALR(1) rozkladové tabulky. Z prvního pohledu je jasné, že můžeme reprezentovat množinu položek I pouze jejím jádrem, tj. těmi položkami, které jsou buď inicializačními $[S' \to \cdot S, \$]$ nebo mají tečku někde jinde, nežli na začátku pravé strany.

Za druhé můžeme vypočítat akce analyzátoru generované I přímo z jádra samotného. Libovolná položka generující redukci podle $A \to \alpha$ se musí vyskytovat v jádře mimo případy, kdy $\alpha = \epsilon$. Redukce $A \to \epsilon$ je generována pro vstup a tehdy a jen tehdy, jestliže existuje v jádře položka $[B \to \gamma \cdot C\delta, b]$ taková, že $C \overset{*}{\Rightarrow} A\eta$ pro nějaké η a a je ve $FIRST(\eta\delta b)$. Množina nonterminálů A takových, že $C \overset{*}{\Rightarrow} A\eta$ může být pro každý nonterminál C vypočtena předem.

Akce přesunu generované množinou I mohou být vypočteny z jádra I následujícím způsobem. Symbol a na vstupu přesouváme na zásobník, existuje-li položka jádra $[B \to \gamma \cdot C\delta, b]$, kde $C \overset{*}{\Rightarrow} ax$ je derivace, ve které poslední krok nepoužívá ϵ-pravidlo. Taková množina pro a může být předem vypočtena pro každé C.

Nyní uvedeme jakým způsobem lze generovat z jádra pro I funkci přechodů. Pokud je v jádře I položka $[B \to \gamma \cdot X\delta, b]$, potom $[B \to \gamma X \cdot \delta, b]$ je v jádře $Goto(I, X)$. V jádře $Goto(I, X)$ je také položka $[A \to X \cdot \beta, a]$, pokud v jádře I existuje položka $[B \to \gamma \cdot C\delta, b]$ a $C \overset{*}{\Rightarrow} A\eta$ pro nějaké η. Pokud předem vypočteme pro každou dvojici nonterminálů C a A, zda $C \overset{*}{\Rightarrow} A\eta$ pro nějaké η, potom je výpočet množin položek pouze z jader jen o málo méně efektivní, nežli výpočet téhož s úplnými množinami položek.

Při výpočtu LALR(1) položek pro rozšířenou gramatiku G' začínáme s jádrem $S' \to \cdot S$ počáteční množiny položek I_0. Potom vypočítáme jádra cílů přechodů z I_0 shora uvedeným způsobem. Pokračujeme výpočtem přechodů pro každé nové jádro, dokud nejsou vytvořena jádra všech množin LR(0) položek.

Příklad 3.25. Uvažujme následující rozšířenou gramatiku

$$\begin{aligned} S' &\to S \\ S &\to L = R \mid R \\ L &\to * R \mid \mathbf{id} \\ R &\to L \end{aligned}$$

Jádra množin LR(0) položek této gramatiky jsou následující:

I_0 : $S' \to \cdot S$
I_1 : $S' \to S\cdot$
I_2 : $S \to L\cdot = R$
$R \to L\cdot$
I_3 : $S \to R\cdot$
I_4 : $L \to * \cdot R$
I_5 : $L \to \mathbf{id}\cdot$
I_6 : $S \to L = \cdot R$
I_7 : $L \to * R\cdot$
I_8 : $R \to L\cdot$
I_9 : $S \to L = R\cdot$

■

Nyní rozšíříme jádra připojením správných pohledů vpřed (druhých částí) pro každou LR(0) položku. Abychom viděli jakým způsobem se symboly rozmnožují z množiny položek I do $Goto(I, X)$ uvažujme LR(0) položku $B \to \gamma \cdot C\delta$ v jádře množiny I. Předpokládejme, že $C \stackrel{*}{\Rightarrow} A\eta$ pro nějaké η (např. pro $C = A$ a $\eta = \epsilon$) a $A \to X\beta$ je přepisovací pravidlo. Potom LR(0) položka $A \to X \cdot \beta$ je v $Goto(I, X)$.

Předpokládejme nyní, že nepočítáme LR(0) položky, ale LR(1) položky a $[B \to \gamma \cdot C\delta, b]$ je v množině I. Pro které hodnoty a bude potom $[A \to X \cdot \beta, a]$ v množině $Goto(I, X)$? Jistěže pokud některé a je ve $FIRST(\eta\delta)$, potom nám derivace $C \stackrel{*}{\Rightarrow} A\eta$ říká, že $[A \to X \cdot \beta, a]$ musí být v $Goto(I, X)$. V tomto případě je hodnota b irelevantní a my říkáme, že a jako pohled vpřed pro $A \to X \cdot \beta$ je generováno *samovolně*. Definicí je potom \$ generováno samovolně jako pohled vpřed pro položku $S' \to \cdot S$ v počáteční množině položek.

Existují zde však i jiné množiny pohledů vpřed pro položky $A \to X \cdot \beta$. Pokud $\eta\delta \stackrel{*}{\Rightarrow} \epsilon$, potom $[A \to X \cdot \beta, b]$ bude také v $Goto(I, X)$. V tomto případě říkáme, že pohled vpřed se *rozmnožil* z $B \to \gamma \cdot C\delta$ do $A \to X \cdot \beta$. V následujícím algoritmu uvedeme jednoduchou metodu pro určení, zda LR(1) položka v I generuje pohled vpřed v $Goto(I, X)$ samovolně, případně kdy se pohled vpřed rozmnožuje.

Algoritmus 3.7. (Určení pohledů vpřed)

Vstup. Jádro K množiny LR(0) položek I a symbol gramatiky X.

Výstup. Pohledy vpřed samovolně generované položkami v I pro položky jádra $Goto(I, X)$ a položky v I, ze kterých jsou pohledy vpřed rozmnožovány do položek v $Goto(I, X)$.

Metoda. Dále uvedený algoritmus užívá symbol # pro označení prázdného pohledu vpřed pro situaci, ve které se pohled vpřed rozmnožuje.

for každou položku $B \to \gamma \cdot \delta$ v K **do begin**
  $J' := Closure(\{[B \to \gamma \cdot \delta, \#]\})$;
  if $[A \to \alpha \cdot X\beta, a]$ je v J', kde a není # **then**
    pohled vpřed a je samovolně generován pro položku
    $A \to \alpha X \cdot \beta$ v $Goto(I, X)$;
  if $[A \to \alpha \cdot X\beta, \#]$ je v J' **then**
    pohled vpřed se rozmnožuje z $B \to \gamma \cdot \delta$ v I do
    $A \to \alpha X \cdot \beta$ v $Goto(I, X)$
end

■

Nyní uvažujme, jak nalezneme pohledy vpřed spojené s položkami v jádrech množin LR(0) položek. Za prvé je známo, že \$ je pohledem vpřed pro $S' \to \cdot S$ v počáteční množině LR(0) položek. Algoritmus 3.7 nám udává všechny pohledy vpřed generované samovolně. Po tom, co máme k dispozici seznam všech takových položek, musíme dovolit všem takovým pohledům vpřed, aby se rozmnožovaly tak dlouho, až žádné další rozmnožení není možné. Existuje více různých přístupů, přičemž všechny nějakým způsobem udržují informaci o "nových" pohledech vpřed, které se rozmnožily do položek, v nichž dosud nebyly. Následující algoritmus popisuje techniku rozmnožení pohledů vpřed do všech položek.

Algoritmus 3.8. (Efektivní výpočet jader souboru LALR(1) množin položek)
Vstup. Rozšířená gramatika G'.
Výstup. Jádra souboru LALR(1) množin položek pro G'.
Metoda.

1. Užitím shora uvedené metody vytvoř jádra množin LR(0) položek pro G.
2. Aplikací algoritmu 3.7 na jádro každé množiny I LR(0) položek a symboly gramatiky X urči, které pohledy vpřed jsou generovány samovolně pro položky v $Goto(I, X)$ a za kterých položek v I se pohledy vpřed rozmnožují do položek jádra $Goto(I, X)$.
3. Inicializuj tabulku, která obsahuje pro každou položku jádra v každé množině položek příslušné pohledy vpřed. Na počátku jsou přidruženy jen ty pohledy vpřed, které jsme v (2) generovali samovolně.
4. Proveď opakované průchody přes všechny položky ve všech množinách. Při průchodu položkou i vyhledej ty položky jádra, do nichž i rozmnožuje svůj pohled vpřed užitím informace získané v bodu (2). Stávající množina pohledů vpřed pro položku I je přidána k již dříve přidružené množině všech položek, kam i rozmnožuje své pohledy vpřed. V průchodech jádry položek množin položek pokračuj tak dlouho, dokud jsou rozmnožovány nějaké nové pohledy vpřed. ■

Příklad 3.26. Vytvořme jádra množin LALR(1) položek pro gramatiku z předchozího příkladu. Jádra jsou rovněž uvedena v předchozím příkladu. Když aplikujeme algoritmus 3.7 na jádro množiny položek I_0, vypočteme $Closure(\{[S' \to \cdot S, \#]\})$ a získáme následující položky

$$\begin{aligned}
S' &\to \cdot S, \# \\
S &\to \cdot L = R, \# \\
S &\to \cdot R, \# \\
L &\to \cdot * R, \#/= \\
L &\to \cdot \mathbf{id}, \#/= \\
R &\to \cdot L, \#
\end{aligned}$$

Dvě položky obsažené v tomto uzávěru způsobí generování pohledů vpřed samovolně. Položka $[L \to \cdot * R, =]$ způsobí, že pohled vpřed $=$ je generován samovolně pro položku $L \to * \cdot R$ v I_4 a položka $[L \to \cdot \mathbf{id}, =]$ způsobí, že pohled vpřed $=$ je generován samovolně pro položku $L \to \mathbf{id}\cdot$ v I_5.

ODKUD	KAM
$I_0 : S' \to \cdot S$	$I_1 : S' \to S\cdot$
	$I_2 : S \to L\cdot = R$
	$I_2 : R \to L\cdot$
	$I_3 : S \to R\cdot$
	$I_4 : L \to * \cdot R$
	$I_5 : L \to \mathbf{id}\cdot$
$I_2 : S \to L\cdot = R$	$I_6 : S \to L = \cdot R$
$I_4 : L \to * \cdot R$	$I_4 : L \to * \cdot R$
	$I_5 : L \to \mathbf{id}\cdot$
	$I_7 : L \to R\cdot$
	$I_8 : R \to L\cdot$
$I_6 : S \to L = \cdot R$	$I_4 : L \to * \cdot R$
	$I_5 : L \to * \cdot R$
	$I_8 : R \to L\cdot$
	$I_9 : S \to L = R\cdot$

Obr. 3.17: Rozmnožování pohledů vpřed

Vzory rozmnožování pro položky jader určené podle bodu (2) algoritmu 3.8 jsou shrnuty na obr. 3.17. Např. přechody z I_0 přes symboly S, L, R, $*$ a **id** jsou I_1, I_2, I_3, I_4 a I_5. Pro I_0 počítáme uzávěr jediné položky $[S' \to \cdot S, \#]$. Z výsledku plyne, že $S' \to \cdot S$ rozmnožuje svůj pohled vpřed do všech jader množin položek od I_1 do I_5.

MN	POLOŽKA	POHLED VPŘED			
		START	KROK 1	KROK 2	KROK3
I_0 :	$S' \to \cdot S$	\$	\$	\$	\$
I_1 :	$S' \to S\cdot$		\$	\$	\$
I_2 :	$S \to L\cdot = R$		\$	\$	\$
I_2 :	$R \to L\cdot$		\$	\$	\$
I_3 :	$S \to R\cdot$		\$	\$	\$
I_4 :	$L \to * \cdot R$	$=$	$= /\$$	$= /\$$	$= /\$$
I_5 :	$L \to \mathbf{id}\cdot$	$=$	$= /\$$	$= /\$$	$= /\$$
I_6 :	$S \to L = \cdot R$			\$	\$
I_7 :	$L \to *R\cdot$		$=$	$= /\$$	$= /\$$
I_8 :	$R \to L\cdot$		$=$	$= /\$$	$= /\$$
I_9 :	$S \to L = R\cdot$				\$

Obr. 3.18: Výpočet pohledů vpřed

Na obr. 3.18 jsou demonstrovány kroky (3) a (4) algoritmu 3.8. Sloupec označený INIT ukazuje samovolně generované pohledy vpřed pro každou položku jádra. Při prvním průchodu je pohled vpřed \$ rozmnožen z $S' \to S$ v I_0 do šesti položek vyjmenovaných na obr. 3.17. Pohled vpřed $=$ se rozmnožuje z položky $L \to * \cdot R$ v I_4 do položky $L \to *R\cdot$ v I_7 a $R \to L\cdot$ v I_8. Rozmnožuje se také do sebe sama a do $L \to \mathbf{id}\cdot$ v I_5, ale zde se tento pohled vpřed již vyskytuje.

Ve druhém průchodu se rozmnožuje nový pohled vpřed \$ do následníků množin I_2 a I_4 a ve třetím průchodu do následníka množiny I_6. Ve čtvrtém průchodu nejsou rozmnožovány žádné nové pohledy vpřed, takže konečné množiny pohledů vpřed jsou dány třetím průchodem a jsou v pravém sloupci obr. 3.18.

Poznamenejme, že konflikt přesun/redukce, který se pro tuto gramatiku vyskytl v příkladu 3.14 při použití metody SLR se při použití metody LALR nevyskytne. Důvod je ten, že s položkou $R \to L\cdot$ v množině I_2 je spojen pouze pohled vpřed \$, takže zde nenastane konflikt s akcí přesun přes symbol = generovanou položkou $S \to L\cdot = R$ v množině I_2. ■

3.4.4 Komprese LR rozkladových tabulek

Typická gramatika programovacího jazyka s 50 až 100 terminály a 100 přepisovacími pravidly může mít LALR rozkladovou tabulku o velikosti několika set stavů. Funkce akcí může obsahovat okolo 20000 položek a každá z nich vyžaduje nejméně 8 bitů pro zakódování. Proto jistě může být zajímavý jiný způsob reprezentace tabulek nežli dvourozměrné pole. Budeme nyní krátce uvažovat několik technik, které mohou být použity pro kompresi části akcí i přechodů u LR rozkladových tabulek.

První užitečnou technikou pro zhuštění pole akcí je dáno pozorováním, že mnoho řádků v této tabulce je stejných. Např. na obr. 3.15 mají stavy 0 a 3 stejné položky tabulky akcí a stavy 2 a 6 také. Lze tedy ušetřit značný prostor za malou časovou ztrátu tím, že vytvoříme pole s ukazatelem pro každý stav ve dvourozměrném poli akcí. Ukazatele na stavy se stejnými akcemi ukazují na tytéž řádky. Abychom získali informaci z takového pole přiřadíme každému terminálu číslo od nuly do počtu terminálů - 1 a užijeme toto číslo jako relativní adresu od adresu, na niž ukazuje příslušný ukazatel.

Další ušetření paměti lze získat za cenu mírně pomalejšího analyzátoru (obecně uvažováno je to rozumná cena, protože analyzátor typu LR většinou spotřebovává pouze malou část celkového času překladu). Jde o vytvoření seznamu akcí pro každý stav. Seznam se skládá z dvojic (terminální symbol, akce). Nejfrekventovanější akce pro každý stav může být umístěna na konec seznamu a na místě terminálu může mít poznámku "cokoliv." To znamená, že pokud by aktuální vstupní symbol nebyl nalezen během hledání v seznamu, použijeme tuto akci bez ohledu na to, co je na vstupu. Dále mohou být zcela bezpečně chybové položky nahrazeny za redukce, aby se zvětšila uniformita řádků. Chyby budou v takovém případě detekovány dále před prvním přesunem.

Příklad 3.27. Uvažujme rozkladovou tabulku z obr. 3.10. Nejprve poznamenejme, že akce pro stavy 0, 4, 6 a 7 jsou shodné. Můžeme je reprezentovat seznamem:

SYMBOL	AKCE
id	$s5$
(	$s4$
cokoliv	chyba

Stav 1 má podobný seznam:

+	$s6$
\$	přijato
cokoliv	chyba

Ve stavu 2 můžeme nahradit chybové položky za $r2$, čímž se redukce podle druhého přepisovacího pravidla bude provádět při všech vstupech kromě $*$. Potom seznam pro stav 2 bude:

$*$	$s7$
cokoliv	$r2$

Stav 3 obsahuje pouze chyby a položky $r4$. Můžeme nahradit ty první druhými, takže seznam pro stav 3 obsahuje pouze dvojice (cokoliv, $r4$). Stavy 5, 10 a 11 mohou být obslouženy obdobně. Seznam pro stav 8 bude:

$+$	$s6$
)	$s11$
cokoliv	chyba

a pro stav 9:

$*$	$s7$
cokoliv	$r1$

■

Tabulku přechodů je možné rovněž zakódovat seznamem, avšak v tomto případě je efektivnější vytvářet dvojice pro každý nonterminál A. Každá dvojice v seznamu bude mít tvar *(aktuální_stav, další_stav)*, vyjadřující hodnotu

$$přechody[aktuální\_stav, A] = další\_stav$$

Tato technika je užitečná, protože je zde tendence výskytu více stavů ve sloupci tabulky přechodů. Důvod je ten, že přechod přes nonterminál A se může vyskytovat pouze do stavu, v němž v některé z položek leží nonterminál bezprostředně vlevo od tečky. Nemůže však existovat množina s nonterminály X a Y bezprostředně vlevo od tečky taková, že $X \neq Y$. Z toho plyne, že se každý stav může vyskytnout nejvýše v jednom sloupci.

Příklad 3.28. Uvažujme opět obr. 3.10. Sloupec pro F má položku 10 pro stav 7 a všechny ostatní jsou buďto 3 nebo chyba. Můžeme nahradit chybu za 3 a tím vytvoříme pro sloupec F seznam:

aktuální_stav	*další_stav*
7	10
cokoliv	3

Podobně získáme seznam pro sloupec T:

6	9
cokoliv	2

Pro sloupec E můžeme za implicitní hodnotu zvolit 1 nebo 8, protože zde jsou nezbytné v každém případě dvě položky seznamu. Např. můžeme vytvořit následující seznam sloupce E:

4	8
cokoliv	1

Pokud čtenář sečte počet všech položek v tomto příkladu a v předchozí verzi tabulky a přičte počet ukazatelů ze stavů na seznamy akcí a z nonterminálů na seznamy přechodů, zjistí, že ušetřený prostor není příliš velký vzhledem k matici z obr. 3.10. Nesmíme se však dát ovlivnit takto malým příkladem. Pro prakticky použitelné gramatiky je prostor spotřebovaný pro reprezentaci seznamem typicky menší než 10% prostoru pro maticovou reprezentaci.

3.4.5 Užití víceznačných gramatik

Existuje věta o tom, že všechny víceznačné gramatiky nejsou LR, a tedy nejsou v žádné ze tříd diskutovaných v předchozí kapitole. Jak uvidíme v této kapitole, existuje však jistý typ víceznačných gramatik, který je užitečný pro specifikaci a implementaci jazyků. Pro jazykové konstrukce, jako jsou např. výrazy, poskytují víceznačné gramatiky kratší a přirozenější specifikací, nežli ekvivalentní jednoznačné gramatiky. Jiným místem použití víceznačných gramatik je izolace společných syntaktických konstrukcí v případě speciální optimalizace.

Zdůrazněme předem, že ačkoliv gramatiky, které budeme užívat jsou víceznačné, ve všech případech přidáme ke gramatice pravidla, která dovolí vytvořit ke každé větě jediný derivační strom. Tímto způsobem zůstává pak celá specifikace jazyka jednoznačná. Rovněž zdůrazněme, že víceznačné konstrukce budou použity šetrně a v přísně řízené podobě, jinak nelze garantovat, jaký jazyk je analyzátorem rozpoznáván.

Užití priority a asociativity k řešení konfliktů v tabulce akcí

Uvažujme výrazy v programovacích jazycích. Následující gramatika pro aritmetický výraz s operátory + a $*$

$$E \rightarrow E + E \mid E * E \mid (E) \mid \mathbf{id} \tag{3.1}$$

je víceznačná, protože nespecifikuje asociativitu a prioritu operátorů + a $*$. Jednoznačná gramatika

$$\begin{aligned} E &\rightarrow E + E \mid T \\ T &\rightarrow T * F \mid F \\ F &\rightarrow (E) \mid \mathbf{id} \end{aligned} \tag{3.2}$$

generuje stejný jazyk, avšak $*$ má přednost před + a oba operátory jsou asociativní zleva. Existují dva důvody, proč bychom mohli chtít použít gramatiku (3.1) namísto gramatiky (3.2). Jak uvidíme, můžeme jednoduše měnit asociativitu a prioritu operátorů bez zásahů do pravidel gramatiky (3.1, resp. počtu stavů výsledného analyzátoru. Za druhé analyzátor (3.2) stráví podstatnou část času redukcemi podle pravidel $E \rightarrow T$ a $T \rightarrow F$, jejichž jedinou funkcí je vyjadřovat prioritu a asociativitu. Analyzátor pro (3.2) nebude ztrácet čas těmito redukcemi, neboť tyto tzv. *jednoduché* redukce neprovádí. Množina LR(0) položek pro gramatiku (3.1) rozšířenou o $E' \rightarrow E$ je následující:

$$
\begin{array}{lll}
I_0: & E' \to \cdot E & \\
& E \to \cdot E + E & \\
& E \to \cdot E * E & \\
& E \to \cdot(E) & \\
& E \to \cdot \mathbf{id} &
\end{array}
\qquad
\begin{array}{lll}
I_5: & E \to E * \cdot E & \\
& E \to \cdot E + E & \\
& E \to \cdot E * E & \\
& E \to \cdot(E) & \\
& E \to \cdot \mathbf{id} &
\end{array}
$$

$$
\begin{array}{lll}
I_1: & E' \to E\cdot & \\
& E \to E \cdot + E & \\
& E \to E \cdot * E &
\end{array}
\qquad
\begin{array}{lll}
I_6: & E \to (E\cdot) & \\
& E \to E \cdot + E & \\
& E \to E \cdot * E &
\end{array}
$$

$$
\begin{array}{lll}
I_2: & E \to (\cdot E) & \\
& E \to \cdot E + E & \\
& E \to \cdot E * E & \\
& E \to \cdot(E) & \\
& E \to \cdot \mathbf{id} &
\end{array}
\qquad
\begin{array}{lll}
I_7: & E \to E + E\cdot & \\
& E \to E \cdot + E & \\
& E \to E \cdot * E &
\end{array}
$$

$$
\begin{array}{lll}
I_3: & E \to \mathbf{id}\cdot &
\end{array}
\qquad
\begin{array}{lll}
I_8: & E \to E * E\cdot & \\
& E \to E \cdot + E & \\
& E \to E \cdot * E &
\end{array}
$$

$$
\begin{array}{lll}
I_4: & E \to E + \cdot E & \\
& E \to \cdot E + E & \\
& E \to \cdot E * E & \\
& E \to \cdot(E) & \\
& E \to \cdot \mathbf{id} &
\end{array}
\qquad
\begin{array}{lll}
I_9: & E \to (E)\cdot &
\end{array}
$$

Protože gramatika (3.1) je víceznačná, vzniknou při vytváření LR rozkladové tabulky z této množiny položek konflikty. Tyto konflikty generují stavy odpovídající množinám I_7 a I_8. Předpokládejme, že užíváme SLR metodu konstrukce rozkladové tabulky. Konflikt generovaný množinou I_7 mezi redukcí $E \to E+E$ a přesuny + a $*$ není řešitelný, protože + i $*$ jsou ve $FOLLOW(E)$. Obě akce by se měli provést jak při vstupu +, tak při vstupu $*$. Podobný konflikt je generován množinou I_8 mezi redukcí $E \to E * E$ a přesuny při vstupu + resp. $*$. Tento konflikt generují všechny LR metody konstrukce rozkladové tabulky.

Uvedené problémy můžeme vyřešit dodatečnými informacemi o precedenci a asociativitě operátorů + a $*$. Uvažujme vstupní řetězec $\mathbf{id} + \mathbf{id} * \mathbf{id}$, který způsobí, že analyzátor založený na předchozím kanonickém souboru LR(0) položek přejde po zpracování $\mathbf{id} + \mathbf{id}$ do stavu 7; přesněji řečeno přejde do konfigurace

ZÁSOBNÍK	VSTUP
0 E 1 + 4 E 7	$*$ **id** \$

Předpokládejme, že $*$ má větší prioritu než +. Víme, že analyzátor by měl přesunout na zásobník $*$ a připravit redukci $*$ spolu s **id**, které jej obklopují. Naopak, pokud by operace + měla vyšší prioritu než $*$, víme, že analyzátor by měl redukovat podle pravidla $E+E$ na E. Tímto způsobem relativní priorita + následovaného $*$ jednoznačně určuje rozřešení konfliktu mezi redukcí podle $E \to E + E$ a přesunem symbolu $*$ ve stavu 7.

Pokud by vstupní řetězec byl $\mathbf{id} + \mathbf{id} + \mathbf{id}$ namísto předchozího řetězce, analyzátor by opět dosáhl konfigurace, ve které zásobník obsahuje 0 E 1 + 4 E 7 a je přečten řetězec $\mathbf{id} + \mathbf{id}$. Při vstupu + nastává ve stavu 7 opět konflikt přesun/redukce. I zde lze informací o asociativitě operátoru + konflikt vyřešit. Je-li + asociativní zleva, korektní akcí je redukce podle pravidla $E \to E + E$. V tomto případě **id** obklopující první + tvoří pracovní frázi.

Předpokládáme-li tedy, že + je asociativní zleva, ve stavu 7 při vstupu + by měla být provedena redukce podle pravidla $E \to E + E$ a předpokládáme-li, že $*$ má přednost před +, potom ve stavu 7 při vstupu $*$ by měl být proveden přesun. Podobně předpokládáme-li, že $*$ je asociativní zleva a má přednost před +, můžeme říci, že ve stavu 8, který se vyskytne na vrcholu zásobníku pouze jsou-li tři nejvyšší symboly v zásobníku $E * E$, by se měla provést akce redukce podle $E \to E * E$ jak při vstupu +, tak při vstupu $*$. V případě vstupu + je to z důvodu, že $*$ má přednost před +, zatímco v případě vstupu $*$ je to z důvodu levé asociativity operátoru $*$.

	akce						*přechody*
STAV	**id**	+	$*$	(	)	$	E
0	s3			s2			1
1		s4	s5			acc	
2	s3			s2			6
3		r4	r4		r4	r4	
4	s3			s2			7
5	s3			s2			8
6		s4	s5		s9		
7		r1	s5		r1	r1	
8		r2	r2		r2	r2	
9		r3	r3		r3	r3	

Obr. 3.19: Rozkladová tabulka pro gramatiku (3.1)

Postupujeme-li tímto způsobem, obdržíme LR rozkladovou tabulku uvedenou na obr. 3.19. Pravidla 1–4 jsou $E \to E + E$, $E \to E * E$, $E \to (E)$, respektive $E \to$ **id**. Je zajímavé, že stejnou rozkladovou tabulku bychom vytvořili vynecháním redukcí podle jednoduchých pravidel $E \to T$ a $T \to F$ z SLR tabulky pro gramatiku (3.2) ukázanou na obr. 3.10. Víceznačné gramatiky podobné (3.1) mohou být zpracovány podobným způsobem i pro kanonickou LR a LALR analýzu.

Víceznačnost else

Uvažujme následující gramatiku podmíněných příkazů:

$$\begin{aligned} stmt &\to \textbf{if } expr \textbf{ then } stmt \textbf{ else } stmt \\ &\mid \textbf{if } expr \textbf{ then } stmt \\ &\mid \textbf{other} \end{aligned}$$

Tato gramatika je víceznačná, jelikož v ní nelze rozřešit známý problém nejednoznačného **else**. Abychom diskusi zjednodušili, zavedeme abstrakci shora uvedené gramatiky, kde i představuje **if** $expr$ **then**, e představuje **else** a a představuje všechny jiné příkazy. V rozšíření bude mít tato gramatika tvar:

$$\begin{aligned} S' &\to S \\ S &\to iSeS \mid iS \mid a \end{aligned} \tag{3.3}$$

Množiny LR(0) položek pro tuto gramatiku jsou následující:

$$
\begin{array}{lll}
I_0: & S' \to \cdot S & I_3: \quad S \to a\cdot \\
& S \to \cdot iSeS & \\
& S \to \cdot iS & I_4: \quad S \to iS \cdot eS \\
& S \to \cdot a & \qquad\quad S \to iS\cdot \\
\\
I_1: & S' \to S\cdot & I_5: \quad S \to iSe \cdot S \\
& & \qquad\quad S \to \cdot iSeS \\
I_2: & S \to i \cdot SeS & \qquad\quad S \to \cdot iS \\
& S \to i \cdot S & \qquad\quad S \to \cdot a \\
& S \to \cdot iSeS & \\
& S \to \cdot iS & I_6: \quad S \to iSeS\cdot \\
& S \to \cdot a &
\end{array}
$$

Víceznačnost této gramatiky dává vzniknout konfliktu přesun/redukce ve stavu I_4. V tomto stavu položka $S \to iS \cdot eS$ generuje přesun e a protože $FOLLOW(S) = \{e, \$\}$, druhá položka $S \to iS\cdot$ generuje redukci podle pravidla při vstupu e.

Přeloženo zpět do terminologie **if** ... **then**... **else**, je-li na zásobníku

if *expr* **then** *stmt*

a **else** je následujícím vstupním symbolem, máme přesunout **else** na zásobník (tj. přesun e) nebo redukovat **if** *expr* **then** *stmt* na *stmt* (tj. redukce podle $S \to iS$)? Odpovědí je, že máme přesunout **else**, protože je "spojeno" s nejbližším předchozím **then**. V terminologii gramatiky (3.3) může e na vstupu nahrazující **else** vždy tvořit část pravé strany začínající iS na vrcholu zásobníku. Jestliže to co následuje za e ne vstupu nemůže být analyzováno jako S kompletující pravou stranu $iSeS$, potom lze ukázat, že není možný žádný rozklad této věty.

Navrhli jsme, že konflikt přesun/redukce ve stavu I_4 by měl být vyřešen dáním přednosti přesunu e. SLR rozkladová tabulka vytvořená ze souboru LR(0) položek naší gramatiky, která užívá navržené řešení konfliktu ve stavu I_4 a vstupu e je ukázána na obr. 3.20. Přepisovací pravidla 1–3 jsou $S \to iSeS$, $S \to iS$ a $S \to a$.

	akce				*přechody*
STAV	*i*	*e*	*a*	$	*S*
0	s2		s3		1
1				acc	
2	s2		s3		4
3		r3		r3	
4		s5		s5	
5	s2		s3		6
6		r1		r1	

Obr. 3.20: Rozkladová tabulka LR pro gramatiku nejednoznačného **else**

Např. pro vstupní vstupní řetězec *iiaea* analyzátor provede akce ukázané na obr. 3.21 odpovídající korektnímu řešení nejednoznačného **else**. Na řádku (5) stav 4 vybere akci *přesun* při vstupu e, zatímco na řádku (9) stav 4 vybere akci *redukce* podle pravidla $S \to iS$ při $ na vstupu.

	ZÁSOBNÍK	VSTUP
(1)	0	$iiaea\$$
(2)	$0i2$	$iaea\$$
(3)	$0i2i2$	$aea\$$
(4)	$0i2i2a3$	$ea\$$
(5)	$0i2i2S4$	$ea\$$
(6)	$0i2i2S4e5$	$a\$$
(7)	$0i2i2S4e5a3$	$\$$
(8)	$0i2i2S4e5S6$	$\$$
(9)	$0i2S4$	$\$$
(10)	$0S1$	$\$$

Obr. 3.21: Akce analyzátoru při vstupu *iiaea*

Víceznačnost plynoucí z pravidel pro speciální případy

Náš poslední příklad ukazuje užitečnost víceznačných gramatik v případě, kdy přidáme dodatečné přepisovací pravidlo pro zachycení speciálního případu obecnější syntaktické konstrukce. Přidáním takové konstrukce je často generován konflikt. Tento konflikt lze často řešit další metodou řešení konfliktů z víceznačnosti. Zavedením přepisovacích pravidel pro speciální případy nám potom umožňuje používat v těchto specifických případech specifických sémantických akcí.

Přepisovacích pravidel pro speciální případy bylo použito Kernighanem a Cherrym v jejich preprocesoru pro sazbu rovnic nazvaného EQN. V něm je syntaxe matematických výrazů popsána gramatikou, která užívá operátoru spodního indexu **sub** a operátoru horního indexu **sup**. Část gramatiky pro tyto operátory je označena (3.4). Složené závorky jsou v tomto preprocesoru použity pro ohraničení složeného výrazu a c je použito jako symbol reprezentující libovolný textový řetězec.

$$\begin{array}{lrcl} (1) & E & \rightarrow & E \ \textbf{sub} \ E \ sup \ E \\ (2) & E & \rightarrow & E \ \textbf{sub} \ E \\ (3) & E & \rightarrow & E \ \textbf{sup} \ E \\ (4) & E & \rightarrow & \{ \ E \ \} \\ (5) & E & \rightarrow & c \end{array} \tag{3.4}$$

Gramatika (3.4) je víceznačná z mnoha důvodů. Gramatika nespecifikuje asociativitu a prioritu operátorů **sub** a **sup**. Pokud bychom řešili víceznačnost přidáním asociativity a priority operátorů **sub** a **sup**, např. přiřazením stejné priority a pravé asociativity, gramatika bude stále víceznačná. Je to způsobeno tím, že přepisovací pravidlo (1) je speciálním případem výrazu generovaného pravidly (2) a (3), jmenovitě jde o výraz E **sub** E **sup** E. Důvodem pro použití výrazu tohoto speciálního tvaru je to, že výraz a **sub** i **sup** 2 by měl být vysázen tak, že horní index 2 a dolní index i jsou pod sebou a nikoli nejprve dolní a za ním horní index. Přidáním tohoto pravidla pro speciální případy je EQN schopen generovat speciální případ výstupu.

Abychom si ukázali, jakým způsobem je tento případ víceznačnosti zpracováván LR analyzátorem, vytvořme nyní SLR analyzátor pro gramatiku (3.4). Množina LR(0) položek je na obr. 3.22.

I_0 :
$E' \to \cdot E$
$E \to \cdot E$ **sub** E **sup** E
$E \to \cdot E$ **sub** E
$E \to \cdot E$ **sup** E
$E \to \cdot\{\ E\ \}$
$E \to \cdot c$

I_1 :
$E' \to E\cdot$
$E \to E \cdot$ **sub** E **sup** E
$E \to E \cdot$ **sub** E
$E \to E \cdot$ **sup** E

I_2 :
$E \to \{\cdot E\ \}$
$E \to \cdot E$ **sub** E **sup** E
$E \to \cdot E$ **sub** E
$E \to \cdot E$ **sup** E
$E \to \cdot\{\ E\ \}$
$E \to \cdot c$

I_3 :
$E \to c\cdot$

I_4 :
$E \to E \cdot$ **sub** E **sup** E
$E \to E \cdot$ **sub** E
$E \to \cdot E$ **sub** E **sup** E
$E \to \cdot E$ **sub** E
$E \to \cdot E$ **sup** E
$E \to \cdot\{\ E\ \}$
$E \to \cdot c$

I_5 :
$E \to E \cdot$ **sup** E
$E \to \cdot E$**sub** E **sup** E
$E \to \cdot E$**sub** E
$E \to \cdot E$**sup** E
$E \to \cdot\{\ E\ \}$
$E \to \cdot c$

I_6 :
$E \to E \cdot$ **sub** E **sup** E
$E \to E \cdot$ **sub** E
$E \to E \cdot$ **sup** E
$E \to \{\ E\cdot\}$

I_7 :
$E \to E \cdot$ **sub** E **sup** E
$E \to E$ **sub** $E \cdot$ **sup** E
$E \to E \cdot$ **sub** E
$E \to E$ **sub** $E\cdot$
$E \to E \cdot$ **sup** E

I_8 :
$E \to E \cdot$ **sub** E **sup** E
$E \to E \cdot$ **sub** E
$E \to E \cdot$ **sup** E
$E \to E$ **sup** $E\cdot$

I_9 :
$E \to \{\ E\ \}\cdot$

I_{10} :
$E \to E$ **sub** E **sup** $\cdot E$
$E \to E$ **sub** $\cdot E$
$E \to \cdot E$ **sub** E **sup** E
$E \to \cdot E$ **sub** E
$E \to \cdot E$ **sub** E
$E \to \cdot E$ **sup** E
$E \to \cdot\{\ E\ \}$
$E \to \cdot c$

I_{11} :
$E \to E \cdot$ **sub** E **sup** E
$E \to E$ **sub** E **sup** $E\cdot$
$E \to E \cdot$ **sub** E
$E \to E \cdot$ **sup** E
$E \to E$ **sup** $E\cdot$

Obr. 3.22: Množiny LR(0) položek pro gramatiku (3.4)

V tomto souboru tři z množin položek generují konflikt. I_7, I_8 a I_{11} generují konflikt přesun/redukce pro symboly **sub** a **sup**, protože nebyla specifikována ani přednost ani asociativita těchto operátorů. Tento konflikt lze řešit tím, že operátorům **sub** a **sup** se přiřadí stejná přednost a pravá asociativita. V tom případě bude vždy dávána přednost přesunu.

I_{11} generuje také konflikt redukce/redukce při vstupu } a \$ mezi pravidly $E \rightarrow E$ **sub** E **sup** E a $E \rightarrow E$ **sup** E. Stav I_{11} bude na vrcholu zásobníku, když jsme prohlédli vstupní řetězec, který byl redukován na E **sub** E **sup** E. Pokud vyřešíme konflikt redukce/redukce ve prospěch pravidla (1), můžeme zpracovávat rovnici ve tvaru E **sub** E **sup** E jako speciální případ. Užitím takového pravidla řešícího víceznačnost obdržíme SLR rozkladovou tabulku, která je uvedena na obr. 3.23.

	akce						*přechody*
STAV	**sub**	**sup**	{	}	*c*	\$	*E*
0			s2		s3		1
1	s4	s5				acc	
2			s2		s3		6
3	r5	r5		r5		r5	
4			s2		s3		7
5			s2		s3		8
6	s4	s5		s9			
7	s4	s10		r2		r2	
8	s4	s5		r3		r3	
9	r4	r4		r4		r4	
10			s2		s3		11
11	s4	s5		r1		r1	

Obr. 3.23: Rozkladová tabulka pro gramatiku (3.4)

Vytvoření jednoznačné gramatiky, která zohledňuje speciální případy syntaktických konstrukcí je velmi obtížné. Abychom si uvědomili, jak obtížné to je, je možné si vytvořit jednoznačnou gramatiku pro (3.4), která izoluje tvar E **sub** E **sup** E.

Zotavení z chyb při LR analýze

LR analyzátor nalezne chybu, pokud v tabulce akcí nalezne chybovou položku. Při hledání v tabulce přechodů nelze nikdy nalézt chybu. LR analyzátor ohlásí chybu tehdy, pokud neexistuje správné pokračování vstupního řetězce. Kanonický LR analyzátor před ohlášením chyby neprovede ani žádnou redukci. SLR a LALR analyzátory mohou provést před ohlášením chyby ještě jistý počet redukcí, avšak nikdy neprovedou přesun znaku způsobujícího chybu na zásobník.

U LR analyzátoru lze implementovat způsob zotavení po chybě následujícím postupem. Prohlížíme zásobník směrem od vrcholu ke dnu, dokud nenalezneme stav s s přechodem přes význačný nonterminál A. Potom je přeskočen žádný, jeden nebo několik vstupních symbolů, dokud není nalezen ve vstupním řetězci symbol a, který může ve správném řetězci následovat za A. Analyzátor umístí na vrchol zásobníku stav *přechody*$[s, A]$ a pokračuje v normální analýze. Mohla by nastat situace, že by pro daný nonterminál A existovala více než jedna možnost. Obvykle se jedná o nonterminál reprezentující nějakou význačnou část vstupního jazyka, např. výraz, příkaz nebo blok. Např. pokud A je nonterminálem *příkaz*, symbolem a může být středník

nebo **end**.

Tato metoda zotavení se pokouší izolovat frázi obsahující syntaktickou chybu. Analyzátor určí, že řetězec derivovatelný z A obsahuje chybu. Část vstupního řetězce byla již zpracována a výsledek tohoto částečného zpracování je na vrcholu zásobníku. Zbytek řetězce je stále na vstupu a analyzátor se pokuší o přeskočení tohoto zbytku. Hledá přitom symbol, který může legitimně následovat za A. Odstraněním symbolů z vrcholu zásobníku, přeskočením části vstupního řetězce a umístěním stavu *přechody*$[s, A]$ na vrchol předstírá analyzátor, že nalezl výskyt nonterminálu A a pokračuje v normální analýze.

Zotavení z chyb na úrovni fráze je implementováno pro zpracování každé chybové položky v LR rozkladové tabulce. Na základě zpracovávaného jazyka jsou určeny nonterminály A a symboly a. Potom lze vytvořit odpovídající proceduru analyzátoru.

Příklad 3.29. Uvažujme následující gramatiku pro výraz

$$E \rightarrow E + E \mid E * E \mid (E) \mid \mathbf{id}$$

Na obr. 3.24 je uvedena LR rozkladová tabulka pro tuto gramatiku z obr. 3.19 modifikovaná pro detekci chyb a zotavení po chybě. V každém stavu, kde se nachází redukce jsme nahradili chybové položky redukcí podle pravidla, které se v tomto stavu normálně redukuje. Tato změna má za následek pozdržení detekce chyby, dokud není provedena jedna nebo více redukcí, avšak chyba bude vždy odhalena dříve, než by se provedl přesun. Zbývající prázdné položky z obr. 3.19 byly nahrazeny voláním chybových podprogramů.

	akce						*přechody*
STAV	**id**	+	*	(	)	$	E
0	s3	e1	e1	s2	e2	e1	1
1	e3	s4	s5	e3	e2	acc	
2	s3	e1	e1	s2	e2	e1	6
3	r4	r4	r4	r4	r4	r4	
4	s3	e1	e1	s2	e2	e1	7
5	s3	e1	e1	s2	e2	e1	8
6	e3	s4	s5	e3	s9	e4	
7	r1	r1	s5	r1	r1	r1	
8	r2	r2	r2	r2	r2	r2	
9	r3	r3	r3	r3	r3	r3	

Obr. 3.24: Rozkladová tabulka LR se zpracováním chyb

Nyní popíšeme činnost jednotlivých chybových podprogramů.

- $e1$: /* Tento podprogram je volán ve stavech 0, 2, 4 a 5. Vždy se očekává začátek operandu, buďto **id** nebo levá závorka. Namísto toho je na vstupu operátor + nebo * nebo konec vstupního řetězce */
  uložit imaginární **id** na vrchol zásobníku a překrýt je stavem 3 (přechody stavů 0, 2, 4 a 5 při vstupu **id**).
  vypsat hlášení "chybějící operand"

- $e2$: /* Tento podprogram je volán ve stavech 0, 1, 2, 4 a 5 při nalezení pravé závorky */
  odstranit ze vstupu pravou závorku
  vypsat hlášení "nevyvážená pravá závorka"

- $e3$: /* Tento podprogram je volán ze stavů 1 nebo 6 když je očekáván operátor a je nalezen **id** nebo pravá závorka */
  uložit + na vrchol zásobníku a překrýt je stavem 4.
  vypsat hlášení "chybějící operátor"

- $e4$: /* Tento podprogram je volán ve stavu 6, je-li nalezen konec vstupního řetězce. Stav 6 očekává operátor nebo pravou závorku */
  uložit pravou závorku na vrchol zásobníku a překrýt ji stavem 9.
  vypsat hlášení "chybí pravá závorka"

Pokud bude na vstupu chybný řetězec **id** +), pak analyzátor projde posloupností konfigurací uvedených na obr. 3.25. ■

ZÁSOBNÍK	VSTUP	CHYBOVÁ HLÁŠENÍ A AKCE
0	**id**+)$	
0**id**3	+)$	
0E1	+)$	
0E1 + 4	)$	
0E1 + 4	$	`nevyvážené závorky`
		e2 odstraní pravou závorku
0E1 + 4**id**3	$	`chybějící operand`
		e1 vloží na vrchol zásobníku **id** 3
0E1 + 4E7	$	
0E1	$	

Obr. 3.25: Činnost LR analyzátoru se zotavením

3.5 Generátory syntaktických analyzátorů

V následující kapitole ukážeme, jakým způsobem lze užít generátorů syntaktických analyzátorů pro vytváření vstupních částí překladačů. Jako základ diskuse užijeme generátor Yacc pro LALR gramatiky, protože v předchozích částech jsme diskutovali mnohé široce použitelné koncepty použité v tomto generátoru. Yacc je zkratkou pro "yet another compiler compiler" (ještě jeden generátor překladačů). Odráží popularitu generátorů na začátku sedmdesátých let. V této době byla vytvořena první verze Yaccu S. C. Johnsonem. Yacc je dostupný jako příkaz operačního systému Unix a byl použit jako pomůcka pro implementaci stovek překladačů.

3.5.1 Generátor syntaktických analyzátorů Yacc

Překladač může být konstruován užitím Yaccu způsobem, který je zobrazen na obr. 3.26. Nejprve je třeba připravit soubor `translate.y` obsahující specifikace překladače v jazyce Yaccu. Příkaz Unixu

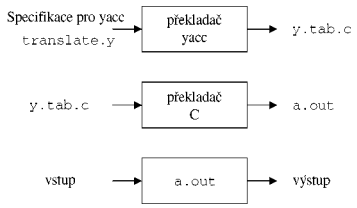
```
yacc translate.y
```

transformuje užitím LALR metody vyjádřené v Algoritmu 3.8 soubor `translate.y` na program v jazyce C, který je nazván `y.tab.c`. Program `y.tab.c` je reprezentací LALR analyzátoru

zapsaného v jazyce C společně s některými předem připravenými podprogramy uživatele. LALR rozkladová tabulka je zhuštěna metodou popsanou v sekci 3.4.4. Překladem programu `y.tab.c` společně s knihovnou `ly`, která obsahuje programy LR analýzy

```
cc y.tab.c -ly
```

obdržíme cílový program `a.out`, který vykonává překlad specifikovaný původním programem v jazyce Yaccu. Vyžaduje-li překladač další procedury, mohou být přeloženy spolu s `y.tab.c` nebo i sestaveny podobně jako jiné programy v jazyce C.



Obr. 3.26: Vytvoření překladače programem Yacc

Zdrojový program v jazyce Yacc má tři části:

deklarace
%%
překladová pravidla
%%
podporující funkce v jazyce C

Příklad 3.30. Abychom ilustrovali způsob, jak připravit zdrojový program v jazyce Yacc, uvažujme následující gramatiku aritmetického výrazu:

$$\begin{aligned} expr &\rightarrow expr + term \mid term \\ term &\rightarrow term * factor \mid factor \\ factor &\rightarrow (expr) \mid \textbf{digit} \end{aligned}$$

Symbol **digit** je jediná číslice v intervalu 0 až 9. Program v jazyce Yacc odvozený z této gramatiky pro stolní kalkulátor je následující:

```
%{
#include <ctype.h>
%}

%token DIGIT
```

```
%%
expr  :      expr '+' term   { printf("expr -> expr + term\n"); }
      |      term            { printf("expr -> term\n"); }
      ;
term  :      term '*' factor { printf("term -> term * factor\n"); }
      |      factor          { printf("term -> factor\n"); }
      ;
factor:      '(' expr ')'    { printf("factor -> ( expr )\n"); }
      |      DIGIT           { printf("factor -> digit\n"); }
      ;

%%
yylex() {
      int c;
      c = getchar();
      if (isdigit(c)) {
            yylval = c-'0';
            return DIGIT;
      }
      return c;
}
```

■

Deklarační část

V deklarační části programu v jazyce Yacc existují dvě volitelné části. V první takové části můžeme vložit obyčejné deklarace jazyka C ohraničené závorkami `%{` a `%}`. Sem umísťujeme deklarace pomocných proměnných užitých v překladových pravidlech nebo funkcích druhé a třetí části. V předchozím programu tato část obsahuje pouze příkaz include

```
#include <ctype.h>
```

který způsobí, že preprocesor jazyka C zahrne do programu soubor standardních záhlaví `<ctype.h>`, který obsahuje predikát `isdigit`.

Druhou částí deklarační části je deklarace vstupních symbolů gramatiky. V předchozím programu je to příkaz

```
%token DIGIT
```

který deklaruje, že vstupním symbolem je `DIGIT`. Vstupní symboly deklarované v této části mohou být užity ve druhé a třetí části specifikace v jazyce Yaccu.

Část překladových pravidel

Část programu v jazyce Yacc za první dvojicí `%%` obsahuje překladová pravidla. Každé pravidlo se skládá z přepisovacího pravidla a může obsahovat vztaženou sémantickou akci. Skupina

přepisovacích pravidel zapisovaná obvykle ve tvaru

$$\begin{array}{lcl} < \text{levá strana} > & \rightarrow & < \text{varianta } 1 > \\ & | & < \text{varianta } 2 > \\ & | & \ldots \\ & | & < \text{varianta } n > \end{array}$$

je zapisována v jazyce Yaccu jako

```
<levá strana> :  <varianta 1>      { sémantická akce 1 }
              |  <varianta 2>      { sémantická akce 2 }
              |  ...
              |  <varianta n>      { sémantická akce n }
              ;
```

Symbol uzavřený v apostrofech `'c'` je v pravidlech chápán jako terminální symbol `c`, řetězce písmen a číslic neuzavřené v apostrofech nedeklarované jako vstupní symboly jsou chápány jako nonterminální symboly. Alternativní pravé strany pravidel mohou být odděleny svislou čárkou. Středník ukončuje pravidlo sestávající z levé strany, alternativ a jejich sémantických akcí. První levá strana je chápána jako startovací nonterminální symbol. Sémantická akce v jazyce Yaccu je posloupnost příkazů jazyka C.

V původní specifikaci gramatiky jsme uvedli E-pravidlo

$$E \rightarrow E + T \mid T$$

Odpovídající zápis v jazyce Yacc má tvar

```
expr  :  expr '+' term
      |  term
      ;
```

Část podporujících funkcí v jazyce C

Třetí část specifikace pro Yacc sestává z podporujících funkcí v jazyce C. Povinně je třeba poskytnout lexikální analyzátor pojmenovaný `yylex()`. Ostatní funkce, jako např. zotavení po chybě nebo některé sémantické akce, mohou být přidány, jsou-li nezbytné.

Lexikální analyzátor `yylex()` vytváří dvojice sestávající ze symbolu a k němu vztažené hodnoty. Je-li vrácen symbol jako např. `DIGIT`, musí být deklarován v první části yaccovské specifikace. Hodnota atributu takového symbolu je předávána syntaktickému analyzátoru v proměnné Yaccu nazvané `yylval`.

Lexikální analyzátor z předchozího programu je velmi nedokonalý. Vstupní řetězec čte po jednom znaku pomocí funkce `getchar()`. Pokud je takovým znakem číslice, potom je hodnota takové číslice uložena do proměnné `yylval` a je vrácen symbol `DIGIT`. V jiných případech je vrácen jako lexikální symbol znak samotný.

Užití programu Yacc pro víceznačné gramatiky

Modifikujme nyní yaccovskou specifikaci tak, že místo jednotlivých číslic umožníme psát přímo celá čísla a rozšíříme množinu operátorů na aritmetické operátory $+$, $-$ (jak binární, tak unární), $*$ a $/$. Nejjednodušším způsobem specifikace této třídy výrazů je užití víceznačné gramatiky

$$E \rightarrow E + E \mid E - E \mid E * E \mid E/E \mid (E) \mid -E \mid \textbf{digit}$$

Odpovídající specifikace pro Yacc je následující

```
%{
#include <ctype.h>
#include <stdio.h>
%}

%term  NUMBER
%left '+' '-'
%left '*' '/'
%right UMINUS
%%
expr  : expr '+' expr         { printf("expr -> expr + expr\n"); }
      | expr '-' expr         { printf("expr -> expr - expr\n"); }
      | expr '*' expr         { printf("expr -> expr * expr\n"); }
      | expr '/' expr         { printf("expr -> expr / expr\n"); }
      | '(' expr ')'          { printf("expr -> ( expr )\n"); }
      | '-' expr %prec UMINUS { printf("expr -> - expr\n"); }
      | NUMBER                { printf("expr -> number\n"); }
      ;
%%
yylex() {
      int c;
      double val;
      while (( c=getchar()) == ' ');
      if ((c=='.') || (isdigit(c))) {
            ungetc (c, stdin);
            scanf("%lf",&val);
            return NUMBER;
      }
      return c;
}
```

Vzhledem k tomu, že gramatika v předchozí specifikaci je víceznačná, bude algortimus pro konstrukci rozkladové tabulky generovat konflikty. V takovém případě Yacc vydá o generovaných konfliktech několik zpráv. Výpis množin položek a konfliktních akcí syntaktického analyzátoru získáme, pokud je zadán programu Yacc parametr `-v`. Při zadání tohoto parametru je vytvářen další soubor `y.output`, který obsahuje jádra všech množin položek, popis všech konfliktů akcí syntaktického analyzátoru a čitelnou reprezentaci LR rozkladové tabulky ukazující, jakým způsobem byly konflikty řešeny. Kdykoliv Yacc podá zprávu, že byl nalezen konflikt akcí, je rozumné vytvořit a konzultovat soubor `y.output`, aby bylo zjištěno, proč byl generován konflikt a zda byl řešen správně.

Pokud není zadáno jinak, potom Yacc řeší konflikty akcí syntaktického analyzátoru následujícími dvěma způsoby:

- Konflikt typu redukce/redukce je řešen volbou pravidla, které je v yaccovské specifikaci uvedeno pozičně dříve. Tedy abychom dosáhli korektního řešení u gramatiky (3.3) jazyka pro sazbu textu, potom je nutné uvést ve specifikaci nejdříve pravidlo (1) a potom teprve pravidlo (3).

- Konflikt typu přesun/redukce je řešen tak, že je dána přednost přesunu. Toto pravidlo řeší např. konflikt v gramatice nejednoznačného **else** vhodným způsobem.

Protože tato pravidla nemusí vždy vyhovovat tvůrci překladače, poskytuje Yacc obecný mechanismus řešení konfliktů přesun/redukce. V deklarační části je totiž možné přiřadit terminálním symbolům prioritu i asociativitu. Deklarace

```
%left '+' '-'
```

určuje, že + a − mají stejnou prioritu a jsou asociativní zleva. Lze deklarovat i operátor asociativní zprava, např.

```
%right '^'
```

a lze také učinit operátor neasociativním binárním operátorem (tj. dva výskyty operátoru nemohou být kombinovány) zápisem

```
%nonassoc '<'
```

Terminální symboly mají dánu prioritu v tom pořadí, ve kterém se vyskytují v deklarační části. Nejdříve uvedené mají prioritu nejmenší. Symboly v téže deklaraci mají tutéž prioritu. Tak např. deklarace

```
%right UMINUS
```

v předchozím programu dává symbolu `UMINUS` přednost větší, nežli všem pěti předchozím terminálům.

Yacc řeší konflikt přesun/redukce přiřazením priority a asociativity každému pravidlu vyskytujícímu se v konfliktu, stejně tak jako každému terminálnímu symbolu vyskytujícímu se v konfliktu. Pokud je nutné volit mezi přesunem symbolu a a redukcí podle pravidla $A \to \alpha$, potom Yacc redukuje pokud priorita pravidla je větší než priorita a nebo pokud je priorita stejná a asociativita pravidla je `left`. V jiných případech je zvolena akce přesun.

Ve standardních případech je priorita pravidla stejná jako priorita nejpravějšího terminálního symbolu v pravidle. V mnoha případech je to rozumná volba. Např. pro daná pravidla

$$E \;\to\; E + E \mid E * E$$

bychom preferovali redukci podle $E \to E + E$ při pohledu vpřed +, protože + na pravé straně má stejnou prioritu jako pohled vpřed, ale je zleva asociativní. S pohledem vpřed bychom dávali přednost přesunu, protože pohled vpřed má větší prioritu, nežli + v pravidle.

V situacích, ve kterých nejpravější terminální symbol neposkytuje pravidlu vhodnou prioritu, můžeme vnutit pravidlu prioritu jinou pomocí zápisu

```
%prec <terminal>
```

Priorita a asociativita pravidla bude v tomto případě stejná jako u uvedeného terminálu, který je předem definován v deklarační části.

Yacc neoznamuje konflikty přesun/redukce, které jsou mechanismem priority a asociativity rozřešeny.

Terminálem může být i takový symbol (jako `UMINUS` v předchozím programu), který není nikdy vracen lexikálním analyzátorem, ale je deklarován jen proto, aby mohla být deklarována priorita pravidla. V předchozím programu deklarace

```
%right UMINUS
```

přiřazuje terminálnímu symbolu `UMINUS` prioritu, která je vyšší, než priorita symbolů $*$ a /. V překladovém pravidle potom sufix

```
%prec UMINUS
```

na konci pravidla

```
expr  :  '-' expr %prec UMINUS
```

způsobí, že priorita pravidla bude rovna prioritě unárního minus (tj. velmi vysoká).

Zotavení po chybě v Yaccu

Zotavení po chybě užívané v Yaccu vychází ze zadání ve formě chybových překladových pravidel. Uživatel nejprve zvolí "hlavní" nonterminály, se kterými bude svázáno zotavení po chybě. Typickou volbou je jistá podmnožina nonterminálů generujících výrazy, příkazy, bloky a procedury. Potom uživatel doplní do gramatiky pravidla ve tvaru $A \rightarrow$ **error** α, kde A je hlavní nonterminál a α je řetězec symbolů gramatiky, který může být i prázdný; **error** je rezervované slovo Yaccu. Z takové specifikace vygeneruje Yacc syntaktický analyzátor, který zachází s chybovými pravidly jako s normálními pravidly.

Nicméně, pokud analyzátor vygenerovaný Yaccem zjistí syntaktickou chybu, chová se ve stavech, které obsahují chybové pravidlo speciálním způsobem. Při výskytu chyby Yacc odstraňuje z vrcholu zásobníku stavy tak dlouho, dokud se na vrcholu zásobníku neobjeví stav, jehož množina položek obsahuje položku $A \rightarrow \cdot$**error** α. V takové situaci "přesune" analyzátor na vrchol zásobníku fiktivní symbol **error**, stejně jako kdyby se symbol **error** nacházel na vstupu.

Pokud je α rovno ϵ, potom se provádí bezprostředně redukce podle A a je vyvolána sémantická akce spojená s pravidlem $A \rightarrow$ **error** (což může být uživatelem definovaný podprogram zotavení po chybě). Analyzátor potom vynechává vstupní symboly tak dlouho, pokud nenalezne vstupní symbol, se kterým by mohl proces analýzy řádně pokračovat.

Pokud α není prázdné, Yacc přeskakuje vstupní řetězec tak dlouho, dokud nenalezne řetězec redukovatelný na α. Pokud α obsahuje jediný terminál, potom je ve vstupním řetězci hledán tento terminál a přenesen na vrchol zásobníku. V tomto okamžiku je na vrcholu **error** i α a může být provedena redukce na A a lze normálně pokračovat v analýze.

Například chybové pravidlo ve tvaru

```
stmt: error ';'
```

specifikuje analyzátoru, že při nalezení chyby má přeskočit vstupní řetězec až k dalšímu středníku. Sémantický podprogram pro chybové pravidlo by neměl manipulovat se vstupním řetězcem, ale mohl by generovat diagnostické hlášení a např. nastavit přepínač zákazu generování cílového kódu.

Příklad 3.31. Yaccovská specifikace uvedená v tomto příkladu ukazuje stolní kalkulátor, který zpracovává aritmetické výrazy zapsané vždy na samostatném řádku. Specifikace obsahuje chybové pravidlo

```
lines  :  error '\n'
```

Toto pravidlo způsobí, že stolní kalkulátor potlačí normální analýzu při nalezení chyby na vstupním řádku. Při nalezení chyby syntaktický analyzátor začne s odstraňováním symbolů ze zásobníku, dokud nenalezne stav s akcí přesunu pro symbol **error**. Tímto stavem je stav 0 (v tomto příkladu je to jediný takový stav), který obsahuje položku

```
lines:   . error '\n'
```

Stav 0 je vždy na dnu zásobníku. Analyzátor přesune symbol `error` na vrchol zásobníku a přeskočí ve vstupu část po nejbližší znak konce řádku. V tomto bodu přesune znak konce řádku na vrchol zásobníku a bude redukovat `error '\n'` na `lines`, přičemž vydá diagnostické hlášení "opakovat poslední řádek." Speciální podprogram Yaccu `yyerrok` nastaví analyzátor do stavu normální analýzy.

```
%{
#include <ctype.h>
#include <stdio.h>
%}

%token NUMBER
%left '+' '-'
%left '*' '/'
%right UMINUS

%%
lines :      lines expr '\n'
      |      lines '\n'
      |      /* epsilon */
      |      error '\n'   { yyerror("opakovat vstup:");
                            yyerrok; }
      ;
expr  :      expr '+' expr
      |      expr '-' expr
      |      expr '*' expr
      |      expr '/' expr
      |      '(' expr ')'
      |      '-' expr %prec UMINUS
      |      NUMBER
      ;
%%
```

■

Kapitola 4

Syntaxí řízený překlad

4.1 Základní pojmy teorie překladu

V tomto odstavci zavedeme některé základní pojmy teorie překladu, na které dále navážeme definicemi pojmů, které se přímo využívají při implementaci překladače.

Definice 4.1. Nechť Σ a Δ jsou abecedy. Abecedu Σ nazvěme *vstupní abecedou*, Δ *výstupní abecedou*. *Překladem* jazyka $L_1 \subset \Sigma^*$ do jazyka $L_2 \subset \Delta^*$ nazveme relaci TRAN : $L_1 \to L_2$. Je-li $[x, y] \in$ TRAN, pak řetězec y nazýváme *výstupem* pro řetězec x. ■

Typickým příkladem překladu je překlad infixového zápisu aritmetického výrazu na postfixový. Tento překlad je nekonečný (relace TRAN obsahuje nekonečně mnoho dvojic řetězců) a relace, jež ho definuje, je ve skutečnosti funkcí, neboť ke každému infixovému zápisu výrazu existuje právě jeden zápis postfixový. Problém konečné specifikace nekonečného překladu je analogický specifikaci nekonečného jazyka. Stejně jako tomu bylo u syntaktické analýzy, jsou i zde dva možné přístupy — prostřednictvím generativního systému (gramatiky) nebo prostřednictvím automatu.

Generativní systém, nazývaný *překladová párová gramatika*, je založený na dvou vzájemně spojených bezkontextových gramatikách. První z nich, tzv. *vstupní gramatika*, popisuje jazyk tvořený všemi větami zdrojového jazyka L_1; druhá, výstupní gramatika, popisuje jazyk $L_2 = \{y | [x, y] \in \text{TRAN}\}$ tvořený všemi výstupy pro řetězce jazyka L_1. Mechanismus zobecněné derivace umožňuje paralelní derivaci řetězce x ve vstupní gramatice a řetězce y ve výstupní gramatice.

Druhý přístup ke specifikaci překladu využívá pojmu *překladový automat*, který je získán rozšířením konečného nebo zásobníkového automatu o výstupní pásku a výstupní funkci, která předepisuje výstup automatu. Překlad definovaný překladovým automatem je množina dvojic řetězců $[x, y]$ takových, že automat přijme řetězec x a na výstup vyšle řetězec y. Teorií překladových automatů se tento učební text nebude zabývat, případné zájemce odkazujeme na [3].

Definice 4.2. *Překladová párová gramatika* je pětice

$$V = (N, \Sigma, \Delta, P, S)$$

kde N je konečná množina *neterminálních symbolů*, Σ konečná *vstupní abeceda*, Δ konečná *výstupní abeceda*, P množina *přepisovacích pravidel* a $S \in N$ *startovací* (neterminální) *symbol*.

Pravidla mají tvar $A \longrightarrow \alpha, \beta$, kde $A \in N$, $\alpha \in (N \cup \Sigma)^*$, $\beta \in (N \cup \Delta)^*$ a nonterminály v řetězci β jsou permutací nonterminálů z řetězce α. ■

Překladová párová gramatika představuje nejobecnější specifikaci překladu; z hlediska implementace přináší značné komplikace možnost záměny pořadí nonterminálů na pravé straně pravidel. V případě, že tuto možnost zakážeme, můžeme vstupní a výstupní část pravidla sloučit do jediného řetězce (za předpokladu, že jsou vstupní i výstupní abecedy disjunktní). Tím se dostáváme k pojmu *překladová gramatika*, který pro nás bude východiskem při dalším popisu činnosti překladače.

Definice 4.3. Nechť $V = (N, \Sigma, \Delta, P, S)$ je překladová párová gramatika, přičemž $N \cap \Delta = \emptyset$ a množina P obsahuje pouze pravidla tvaru

$$A \longrightarrow x_0 B_1 x_1 B_2 \ldots B_k x_k, y_0 B_1 y_1 B_2 \ldots B_k y_k \tag{4.1}$$

pro $x_i \in \Sigma^*$, $y_i \in \Delta^*$, $0 \leq i \leq k$. Pak *překladová gramatika* G_V příslušející gramatice V je pětice $G_V = (N, \Sigma, \Delta, P', S)$, kde množina P' obsahuje pouze pravidla ve tvaru

$$A \longrightarrow x_0 y_0 B_1 x_1 y_1 B_2 \ldots B_k x_k y_k$$

odvozená z původních pravidel ve tvaru (4.1). ■

Příklad 4.1. Uvažujme překladovou párovou gramatiku

$$V = (\{E, T, F\}, \{+, *, i, (,)\}, \{+, *, i\}, P, E)$$

s pravidly

$$\begin{array}{lcl} E & \rightarrow & E\ +\ T,\ E\ T\ + \\ E & \rightarrow & T,\ T \\ T & \rightarrow & T\ *\ F,\ T\ F\ * \\ T & \rightarrow & F,\ F \\ F & \rightarrow & (\ E\ ),\ E \\ F & \rightarrow & i,\ i \end{array}$$

Tato párová gramatika generuje překlad infixového aritmetického výrazu do postfixového výrazu, např.

$$\begin{array}{rcl} [E, E] & \Rightarrow & [E + T, ET+] \Rightarrow [T + T, TT+] \Rightarrow [F + T, FT+] \Rightarrow \\ & \Rightarrow & [i + T, iT+] \Rightarrow [i + T * F, iTF*+] \Rightarrow [i + F * F, iFF* =] \Rightarrow \\ & \Rightarrow & [i + i * F, iiF*+] \Rightarrow [i + i * i, iii*+] \end{array}$$

■

Příklad 4.2. Pravidla gramatiky z předchozího příkladu zachovávají pořadí odpovídajících si nonterminálů na pravých stranách. Po přejmenování symbolů výstupní abecedy tedy můžeme odvodit následující překladovou gramatiku:

$$G_V = (\{E, T, F\}, \{+, *, i, (,)\}, \{\texttt{ADD}, \texttt{MUL}, \texttt{ID}\}, P', E)$$

s pravidly

$$\begin{aligned} E &\rightarrow E + T \texttt{ ADD} \\ E &\rightarrow T \\ T &\rightarrow T * F \texttt{ MUL} \\ T &\rightarrow F \\ F &\rightarrow (E) \\ F &\rightarrow i \texttt{ ID} \end{aligned}$$

Tato gramatika umožňuje provést např. následující derivaci:

$$\begin{aligned} E &\Rightarrow E + T \texttt{ ADD} \Rightarrow T + T \texttt{ ADD} \Rightarrow F + T \texttt{ ADD} \Rightarrow i\ ID + T \texttt{ ADD} \Rightarrow \\ &\Rightarrow i \texttt{ ID} + T * F \texttt{ MUL ADD} \Rightarrow i \texttt{ ID} + F * F \texttt{ MUL ADD} \Rightarrow \\ &\Rightarrow i \texttt{ ID} + i \texttt{ ID} * F \texttt{ MUL ADD} \Rightarrow i \texttt{ ID} + i \texttt{ ID} * i \texttt{ ID MUL ADD} \end{aligned}$$

Vidíme, že ve větě "`i ID + i ID * i ID MUL ADD`" tvoří symboly vstupní abecedy jak jdou po sobě vstup a výstupní symboly odpovídají výstupu překladu, tj. dvojice `(i+i*i, ID ID ID MUL ADD)` je prvek překladu. ■

Z hlediska implementace mohou být symboly výstupní abecedy reprezentovány jako skutečné výstupní symboly (symboly `ID`, `ADD` a `MUL` z předchozích příkladů by např. mohly představovat instrukce zásobníkového mezikódu pro vyhodnocení aritmetického výrazu) nebo jako akce, např. pro symbol `ADD` volání procedury pro vygenerování instrukce sčítání nebo dokonce pro provedení součtu v případě interpretačního překladače. V dalších odstavcích se budeme zabývat rozšířením pojmu překladové gramatiky o atributy symbolů, přičemž konkrétní reprezentaci jednotlivých symbolů nebudeme v definicích uvažovat.

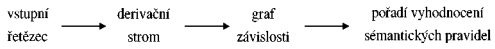
4.2 Atributovaný překlad

Prozatím jsme se zabývali pouze kontrolou, zda je věta, kterou překládáme, prvkem překládaného jazyka. V této kapitole přiřadíme k syntaktickým konstrukcím další informace — atributy, které se vyhodnocují na základě sémantických pravidel.

Pro připojení sémantických pravidel k pravidlům gramatiky existují dvě notace, syntaxí řízené definice a překladová schemata. Syntaxí řízené definice jsou specifikací překladu na vysoké úrovni abstrakce. Ukrývají mnoho implementačních detailů a osvobozují uživatele od nutnosti specifikovat explicitně pořadí, v jakém se bude překlad provádět. Překladová schemata určují pořadí vyhodnocování sémantických pravidel, takže umožňují ukázat i některé implementační detaily.

Obecně jak při překladu pomocí syntaxí řízených definic, tak i při použití překladových schemat rozkládáme vstupní posloupnost symbolů, budujeme derivační strom a potom procházíme stromem tak, abychom vyhodnotili sémantická pravidla v uzlech derivačního stromu (viz obr. 4.1). Vyhodnocením sémantických pravidel může být generování kódu, ukládání informací do tabulky symbolů, vydávání zpráv o chybách nebo provádění nějakých jiných činností. Výsledkem vyhodnocení sémantických pravidel je překlad posloupnosti vstupních symbolů.

Implementace nemusí být doslova shodná se schematem na obr. 4.1. Speciální případy syntaxí řízeného překladu lze implementovat v jednom průchodu s vyhodnocením sémantických



Obr. 4.1: Celkový pohled na syntaxí řízený překlad

pravidel během analýzy, bez explicitní konstrukce derivačního stromu nebo grafu ukazujícího závislosti mezi atributy. Vzhledem k tomu, že jednoprůchodová implementace je důležitá pro efektivitu překladače, je velká část této kapitoly věnována studiu takových případů. Jedna důležitá podtřída, zvaná *L-atributové definice*, zahrnuje téměř všechny překlady, které lze provádět bez explicitní konstrukce derivačního stromu.

4.2.1 Atributové překladové gramatiky

Definice 4.4. *Atributová překladová gramatika* (APG) je trojice

$$G_{AP} = (G_P, A, F),$$

kde $G_P = (N, \Sigma, \Delta, R, S)$ je překladová gramatika, A množina *atributů* a F množina *sémantických pravidel*. V případě, že je množina výstupních symbolů Δ prázdná, hovoříme pouze o *atributové gramatice* (AG).

Pro každý symbol $X \in N \cup \Sigma \cup \Delta$ jsou dány dvě (případně prázdné) disjunktní množiny — množina $I(X)$ *dědičných atributů* a množina $S(X)$ *syntetizovaných atributů*, přičemž pro $a \in I(S)$ jsou zadány počáteční hodnoty (dědičné atributy startovacího nonterminálu) a pro terminální symboly $X \in \Sigma$ je $I(X) = \emptyset$ (terminální symboly nemají dědičné atributy) a jejich syntetizované atributy jsou zadány.

Nechť r-té pravidlo gramatiky má tvar i$X_0 \longrightarrow X_1 X_2 \ldots X_{n_r}$, kde $X_0 \in N$, $X_i \in N \cup \Sigma \cup \Delta$ pro $1 \leq i \leq n_r$. Pak

a) pro každý symbol X_k, $1 \leq k \leq n_r$ na pravé straně pravidla r a jeho dědičný atribut $d \in I(X_k)$ je dáno sémantické pravidlo

$$d = f_r^{d,k}(a_1, a_2, \ldots, a_n)$$

kde a_i, $1 \leq i \leq n$ jsou atributy symbolů v témže pravidle r,

b) pro každý syntetizovaný atribut s symbolu X_0 na levé straně pravidla r je dáno sémantické pravidlo

$$s = f_r^{s,0}(a_1, a_2, \ldots, a_n)$$

kde a_i, $1 \leq i \leq n$ jsou atributy symbolů v témže pravidle r, a

c) pro každý syntetizovaný atribut výstupního symbolu $X_k \in \Delta$ v pravidle r je dáno sémantické pravidlo

$$s = f_r^{s,k}(a_1, a_2, \ldots, a_n)$$

kde a_i, $1 \leq i \leq n$ jsou pouze dědičné atributy symbolu $X_k \in \Delta$. ■

PRAVIDLA	SÉMANTICKÁ PRAVIDLA
$E_0 \longrightarrow E_1 + T$	$E_0.val = E_1.val + T.val$
$E \longrightarrow T$	$E.val = T.val$
$T_0 \longrightarrow T_1 * F$	$T_0.val = T_1.val * F.val$
$T \longrightarrow F$	$T.val = F.val$
$F \longrightarrow (E)$	$F.val = E.val$
$F \longrightarrow \mathbf{num}$	$F.val = \mathbf{num}.ival$

Obr. 4.2: Atributová gramatika pro aritmetický výraz

Sémantická pravidla realizujeme obvykle příkazy (funkcemi) vhodného vyššího programovacího jazyka (např. C nebo Pascal). Atributy pak chápeme jako proměnné či parametry jistého datového typu.

V dalším textu budeme atributy symbolů pojmenovávat kvalifikovanými jmény ve tvaru $X.a$, kde X je jméno symbolu a a jméno atributu. Sémantické funkce budeme psát vždy za pravidlo gramatiky, k němuž se vztahují. V případě, že se v jednom pravidle bude vyskytovat určitý symbol vícekrát, rozlišíme jednotlivé výskyty pomocí indexu.

Příklad 4.3. Atributová gramatika na obr. 4.2 popisuje aritmetický výraz tvořený celočíselnými konstantami, operátory `+`, `*` a závorkami. Nonterminály E, T a F mají celočíselný syntetizovaný atribut *val*, který udává hodnotu příslušných podvýrazů, syntetizovaný atribut *ival* terminálního symbolu **num** udává hodnotu celočíselné konstanty získanou z lexikální analýzy. Jednotlivá sémantická pravidla počítají hodnotu atributu *val* nonterminálu na levé straně z hodnot *val* symbolů na pravé straně pravidel gramatiky. ■

Vyhodnocením sémantického pravidla definujeme hodnoty atributů uzlů derivačního stromu pro vstupní řetězec. Derivační strom s hodnotami atributů v každém uzlu nazýváme ohodnocený derivační strom. Proces výpočtu hodnot atributů v uzlech nazýváme ohodnocením derivačního stromu.

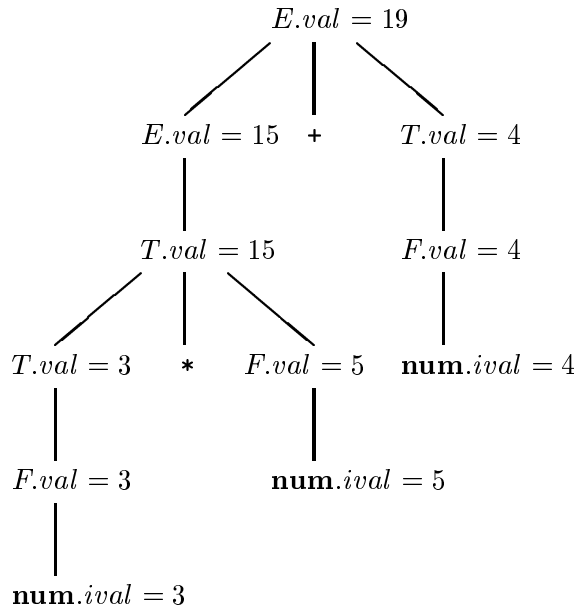
Příklad 4.4. Atributová gramatika z příkladu 4.3 vypočte hodnotu aritmetického výrazu s desítkovými čísly, závorkami a operátory `+` a `*`. Například pro výraz `3*5+4` vypočte hodnotu 19 jako hodnotu atributu $E.val$ startovacího nonterminálu E. Obr. 4.3 obsahuje ohodnocený derivační strom pro vstup `3*5+4`.

Abychom ukázali, jak se atributy vyhodnocují, uvažujme levý dolní vnitřní uzel, odpovídající použití pravidla $F \longrightarrow \mathbf{num}$. Odpovídající sémantické pravidlo $F.val := \mathbf{num}.ival$ přidělí atributu $F.val$ v tomto uzlu hodnotu 3, neboť hodnota $\mathbf{num}.ival$ následníka uzlu je 3. Podobně v předchůdci tohoto F-uzlu má atribut $T.val$ hodnotu 3. Nyní uvažujme uzel pro pravidlo $T \longrightarrow T * F$. Hodnota atributu $T.val$ v tomto uzlu je definována jako

PRAVIDLO	SÉMANTICKÉ PRAVIDLO
$T_0 \longrightarrow T_1 * F$	$T_0.val := T_1.val * F.val$

Pokud aplikujeme na tento uzel uvedené sémantické pravidlo, bude mít $T_1.val$ hodnotu 3 levého následníka a $F.val$ hodnotu 5 pravého následníka. $T_0.val$ tedy dostane v tomto uzlu hodnotu 15. Konečně pro startovací nonterminál E se podobným způsobem vypočte hodnota 19. ■

Sémantické funkce z definice atributové gramatiky nám z matematického hlediska umožňují pouze vyhodnocovat atributy a předávat je mezi jednotlivými symboly gramatiky bez možnosti



Obr. 4.3: Ohodnocený derivační strom pro `3*5+4`

využití vedlejších efektů (např. výstupní operace, práce s globálními proměnnými apod.). Pokud připustíme, aby sémantické funkce měly vedlejší efekty, hovoříme o *syntaxí řízené definici* (SDD).

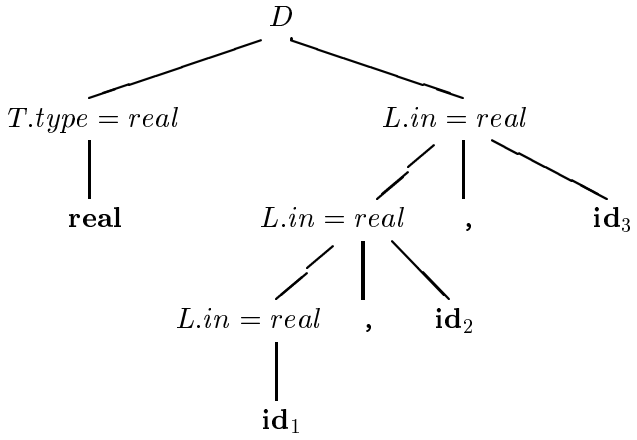
Příklad 4.5. Deklarace generovaná nonterminálem D v syntaxí řízené definici na obr. 4.4 se skládá z klíčového slova **int** nebo **real** následovaného seznamem identifikátorů. Nonterminál T má syntetizovaný atribut *type*, jehož hodnota je určena klíčovým slovem v deklaraci. Sémantické pravidlo $L.in := T.type$, svázané s pravidlem $D \longrightarrow T\ L$, nastavuje dědičný atribut $L.in$ na hodnotu typu v deklaraci. Pravidla přenášejí tento typ dolů derivačním stromem pomocí dědičného atributu $L.in$. Pravidla spojená s pravidlem gramatiky pro L volají proceduru *addtype*, která připojí typ k položce tabulky symbolů pro každý identifikátor (na položku ukazuje atribut *entry*).

Obr. 4.5 ukazuje ohodnocený derivační strom pro větu **real** $\mathbf{id}_1, \mathbf{id}_2, \mathbf{id}_3$. Hodnota $L.in$ ve třech L-uzlech udává typ identifikátorů $\mathbf{id}_1$, $\mathbf{id}_2$ a $\mathbf{id}_3$. Tyto hodnoty se určí výpočtem hodnoty atributu $T.type$ levého následníka kořene a pak výpočtem $L.in$ shora dolů ve třech L-uzlech pravého podstromu kořene. V každém L-uzlu také voláme proceduru *addtype*, která uloží do

PRAVIDLO	SÉMANTICKÉ PRAVIDLO
$D \longrightarrow T\ L$	$L.in := T.type$
$T \longrightarrow \mathbf{int}$	$T.type := integer$
$T \longrightarrow \mathbf{real}$	$T.type := real$
$L \longrightarrow L_1, \mathbf{id}$	$L_1.in := L.in$
	$addtype(\mathbf{id}.entry, L.in)$
$L \longrightarrow \mathbf{id}$	$addtype(\mathbf{id}.entry, L.in)$

Obr. 4.4: Syntaxí řízená definice s dědičným atributem $L.in$

tabulky symbolů informaci o tom, že identifikátor v pravém podstromu uzlu má typ *real*. ■



Obr. 4.5: Derivační strom s dědičnými atributy v uzlech L

4.2.2 Graf závislosti

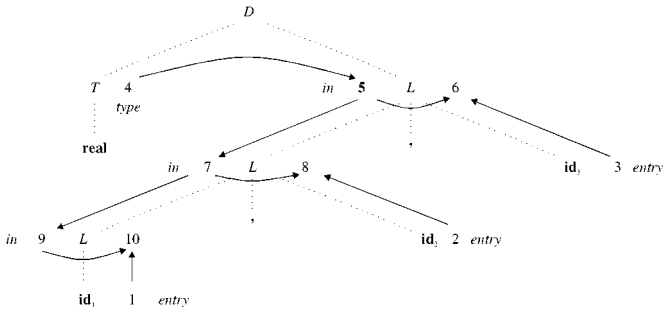
Sémantická pravidla udávají závislosti mezi atributy. Tyto závislosti reprezentujeme *grafem závislosti* (dependency graph), ze kterého pak můžeme odvodit pořadí vyhodnocení sémantických pravidel. Závisí-li atribut b uzlu derivačního stromu na atributu c, pak musí být sémantické pravidlo pro b vyhodnoceno po sémantickém pravidle definujícím c.

Ještě před tím, než začneme konstruovat graf závislosti k danému derivačnímu stromu, převedeme všechna sémantická pravidla do tvaru $b := f(c_1, c_2, \ldots, c_k)$ zavedením prázdného syntetizovaného atributu b pro všechna sémantická pravidla tvořená voláním procedury. Graf obsahuje ke každému atributu jeden uzel a hrany vedoucí z uzlu b do uzlu c, pokud atribut b závisí na atributu c. Graf závislosti lze konkrétně vytvořit následujícím způsobem:

for každý uzel n derivačního stromu **do**
  for každý atribut a symbolu gramatiky v n **do**
    vytvoř uzel grafu závislosti pro a;
for každý uzel n derivačního stromu **do**
  for každé sémantické pravidlo $b := f(c_1, c_2, \ldots, c_k)$
      spojené s pravidlem použitým v n **do**
    for $i := 1$ **to** k **do**
      vytvoř hranu z uzlu pro c_i do uzlu pro b;

Předpokládejme například, že $A.a := f(X.x, Y.y)$ je sémantické pravidlo k pravidlu gramatiky $A \longrightarrow X\ Y$. Toto pravidlo definuje syntetizovaný atribut $A.a$, jež závisí na atributech $X.x$ a $Y.y$. Je-li takové pravidlo použito v derivačním stromu, budeme mít v grafu závislosti tři uzly $A.a$, $X.x$ a $Y.y$ s hranou vedoucí z $A.a$ do $X.x$ ($A.a$ závisí na $X.x$) a hranou z $A.a$ do $Y.y$ ($A.a$ závisí také na $Y.y$).

Je-li s pravidlem $A \longrightarrow X\ Y$ spojeno sémantické pravidlo $X.i := g(A.a, Y.y)$, bude graf závislosti obsahovat hranu do $A.a$ z $X.i$ a také do $A.a$ z $Y.y$, neboť $X.i$ závisí jak na $A.a$, tak na $Y.y$.



Obr. 4.6: Graf závislosti

Příklad 4.6. Obr. 4.6 ukazuje graf závislosti pro derivační strom na obr. 4.5. Uzly v grafu závislosti jsou označeny čísly; tato čísla budeme používat dále. Pro $L.in$ zde máme hranu vedoucí do uzlu 5 z uzlu 4 pro $T.type$, neboť dědičný atribut $L.in$ závisí na atributu $T.type$ na základě sémantického pravidla $L_1.in := T.type$ pro pravidlo gramatiky $D \longrightarrow T\ L$. Dvě hrany vedoucí dolů do uzlů 7 a 9 vyplývají ze závislosti $L_1.in$ na $L.in$ podle sémantického pravidla $L_1.in := L.in$ pro pravidlo gramatiky $L \longrightarrow L_1, \mathbf{id}$. Sémantické pravidlo $addtype(\mathbf{id}.entry, L.in)$ spojené s L-pravidly vede k vytvoření prázdného atributu. Uzly 6, 8 a 10 byly vytvořeny právě pro tyto prázdné atributy. ■

4.2.3 Pořadí vyhodnocení pravidel

Definice 4.5. *Topologický sort* orientovaného acyklického grafu je libovolné uspořádání $m_1, m_2, \ldots, m_k$ uzlů grafu takové, že hrany vedou z uzlů uvedených dříve do uzlů uvedených později; to znamená, že je-li $m_i \longrightarrow m_j$ hrana z m_i do m_j, potom se m_i vyskytuje v tomto uspořádání před m_j. ■

Libovolný topologický sort grafu závislosti je použitelný jako pořadí, v němž se mají vyhodnocovat sémantická pravidla spojená s uzly derivačního stromu. V topologickém sortu jsou závislé atributy $c_1, c_2, \ldots, c_k$ v sémantickém pravidle $b := f(c_1, c_2, \ldots, c_k)$ k dispozici ještě před vyhodnocením f.

Překlad specifikovaný syntaxí řízenou definicí můžeme provést následujícím způsobem. Pro vytvoření derivačního stromu k zadanému vstupu použijeme výchozí gramatiku. Podle předchozího algoritmu vytvoříme graf závislosti. Z topologického sortu grafu závislosti získáme pořadí vyhodnocení sémantických pravidel a vyhodnocením sémantických pravidel v tomto pořadí získáme překlad vstupního řetězce.

Příklad 4.7. Hrany v grafu závislosti na obr. 4.6 vycházejí vždy z uzlu s nižším číslem do uzlu s vyšším číslem. Topologický sort grafu závislosti tedy získáme zapsáním uzlů v pořadí

podle jejich čísel. Na základě topologického sortu pak můžeme zapsat následující program. (Pro atribut svázaný s uzlem n grafu závislosti budeme používat označení a_n.)

$a_4 := real;$
$a_5 := a_4;$
$addtype(\mathbf{id}_3.entry, a_5);$
$a_7 := a_5;$
$addtype(\mathbf{id}_2.entry, a_7);$
$a_9 := a_7;$
$addtype(\mathbf{id}_1.entry, a_9);$

Vyhodnocením těchto sémantických pravidel vložíme do tabulky symbolů pro všechny deklarované identifikátory typ *real*. ■

Pro vyhodnocování sémantických pravidel bylo navrženo několik metod.

- *Metody derivačního stromu.* Tyto metody získávají pořadí vyhodnocení sémantických pravidel v čase překladu z topologického sortu grafu závislosti, vytvořeného z derivačního stromu pro každý vstupní text. Pořadí vyhodnocení tyto metody nenajdou pouze v případě, že graf závislosti pro uvažovaný derivační strom obsahuje cyklus.

- *Metody založené na pravidlech.* V době vytváření překladače se analyzují sémantická pravidla spojená s pravidly gramatiky ručně nebo specializovanými prostředky. Pro každé pravidlo gramatiky je pořadí, ve kterém se budou vyhodnocovat příslušné atributy, pevně určeno již při návrhu překladače.

- *Nezávislé metody.* Pořadí vyhodnocení se vybere bez ohledu na sémantická pravidla. Například probíhá-li překlad během syntaktické analýzy, je pořadí vyhodnocení dáno použitou metodou překladu, nezávisle na sémantických pravidlech. Nezávislé pořadí vyhodnocování omezuje třídu syntaxí řízených definic, jež mohou být implementovány.

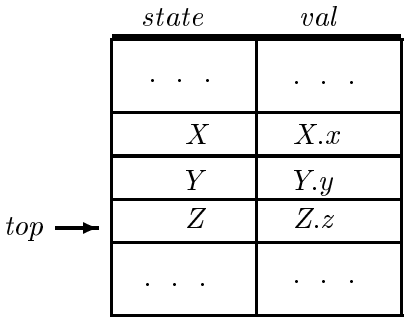
Metody založené na pravidlech a nezávislé metody nemusejí explicitně konstruovat během překladu graf závislosti, takže mohou být efektivnější s ohledem na dobu překladu i velikost požadované paměti.

4.3 Vyhodnocení S-atributových definic zdola nahoru

Vytvoření překladače pro obecnou syntaxí řízenou definici může být značně obtížný problém. Existují však dosti rozsáhlé třídy se speciálními vlastnostmi, pro které lze překladač implementovat jednoduše. Jednou z nich jsou S-atributové definice, tj. takové definice, které pracují pouze se syntetizovanými atributy.

Syntetizované atributy můžeme vyhodnocovat současně s analýzou zdrojového textu zdola nahoru. Atributy mohou být uloženy společně s ostatními informacemi, které používá analyzátor, na zásobníku. Při každé redukci podle nějakého pravidla se vypočtou atributy nonterminálního symbolu na levé straně pravidla a ty se uloží do zásobníku. Atributy symbolů na pravé straně jsou v okamžiku redukce umístěny na vrcholu zásobníku, takže jsou pro výpočet vždy k dispozici. Při vhodném návrhu překladového schematu je možné pracovat v omezené míře i s dědičnými atributy, jak dále ukážeme.

Syntaktický analyzátor pracující metodou zdola nahoru používá pro uchovávání informací o průběhu analýzy zásobník. Položky zásobníku můžeme rozšířit vždy o hodnotu atributu, jak



Obr. 4.7: Rozšířený zásobník syntaktického analyzátoru

ukazuje obr. 4.7. Každá položka odpovídá vždy jednomu symbolu v již zpracované části větné formy; tento symbol je uveden ve sloupci *state*. Ve sloupci *val* je pak uvedena hodnota atributu odpovídajícího symbolu z prvního sloupce. Jinou možnou implementací je použití dvou paralelních zásobníků, jednoho pro uchovávání informací o analýze a druhého pro atributy. Pokud může mít jeden symbol více atributů, můžeme je všechny umístit do jednoho záznamu a tento nový datový typ pak používat jako jediný (strukturovaný) atribut. Rovněž mají-li různé symboly různé typy atributů, můžeme všechny tyto typy sloučit do jediného pomocí záznamu s variantními složkami, unie nebo podobné konstrukce, kterou pro to poskytuje implementační jazyk.

Současný vrchol zásobníku je označen ukazatelem *top*. Předpokládáme, že se syntetizované atributy vyhodnocují právě před provedením redukce. Máme-li například s pravidlem $A \to XYZ$ svázáno sémantické pravidlo $A.a := f(X.x, Y.y, Z.z)$, je před redukcí atribut $Z.z$ uložen ve $val[top]$, atribut $y.y$ ve $val[top-1]$ a atribut $X.x$ ve $val[top-2]$. Pokud symbol nemá atribut, není odpovídající hodnota pole *val* definována. Po redukci se hodnota *top* sníží o 2, stav odpovídající A se uloží do $state[top]$ (tj. místo X) a vypočtená hodnota syntetizovaného atributu $A.a$ se uloží do $val[top]$.

Příklad 4.8. Uvažujme gramatiku z obr. 4.2 pro výpočet hodnoty aritmetického výrazu. Tato gramatika pracuje pouze se syntetizovanými atributy a může být tedy implementována přímo při překladu zdola nahoru. Opět předpokládáme, že lexikální analyzátor dodá hodnotu atributu **num**.*ival*; tuto hodnotu uložíme do zásobníku při provádění akce přesun. Obr. 4.8 uvádí možnou implementaci sémantických akcí s atributy uloženými v poli *val*.

PRAVIDLA	SÉMANTICKÁ AKCE
$E \to E_1 + T$	$val[ntop] := val[top-2] + val[top]$
$E \to T$	
$T \to T_1 * F$	$val[ntop] := val[top-2] * val[top]$
$T \to F$	
$F \to (\ E\ )$	$val[ntop] := val[top-1]$
$F \to$ **num**	

Obr. 4.8: Implementace aritmetického výrazu pomocí atributového zásobníku

Uvedené úseky kódu pro sémantické akce neřeší nastavování proměnných *top* a *ntop*. Provádí-

li se redukce podle pravidla s r hodnotami na pravé straně, nastaví se *ntop* na $top - r + 1$ a po provedení akce se *top* nastaví na *ntop*. Ještě vhodnější řešení je použít pro syntetizovaný atribut levé strany pravidla zvláštní proměnnou, která se pak přesune do zásobníku až po dokončení výpočtu. Obr. 4.9 ukazuje posloupnost akcí překladače při vstupu `2+3*5`.

VSTUP	state	val	POUŽITÉ PRAVIDLO
2+3*5	- -	- -	
+3*5	2	2	
+3*5	F	2	$F \to$ **num**
+3*5	T	2	$T \to F$
+3*5	E	2	$E \to T$
3*5	E +	3 -	
*5	E + 3	2 - 3	
*5	E + F	2 - 3	$F \to$ **num**
*5	E + T	2 - 3	$T \to F$
5	E + T *	2 - 3 -	
	E + T * 5	2 - 3 - 5	
	E + T * F	2 - 3 - 5	$F \to$ **num**
	E + T	2 - 15	$T \to T * F$
	E	17	$E \to E + T$

Obr. 4.9: Analýza a vyhodnocení výrazu `2+3*5`

Pro implementaci S-atributových definic je možné použít generátoru yacc. Implicitně yacc předpokládá, že všechny symboly jazyka mají jeden atribut typu `int`. Syntetizované atributy symbolů na pravé straně pravidla jsou dostupné pod symbolickým jménem `$`i, kde i je pořadové číslo symbolu počínaje 1. Atribut levostranného nonterminálu se ukládá do proměnné se symbolickým jménem `$$`. Nemá-li některé pravidlo uvedenu sémantickou akci, provede se implicitně akce { `$$ = $1;` }, která předá atribut prvního symbolu v pravidle jako atribut levé strany. Na obr. 4.10 je uvedeno totéž překladové schéma jako na obr. 4.8 zapsané pro generátor yacc.

```
%term NUM
%%
E  :  E '+' T   { $$ = $1 + $3; }
   |  T
   ;
T  :  T '*' F   { $$ = $1 * $3; }
   |  F
   ;
F  :  '(' E ')' { $$ = $2; }
   |  NUM
   ;
```

Obr. 4.10: Specifikace překladu se syntetizovanými atributy pro yacc

Pokud potřebujeme jako atribut použít jiného datového typu, například v případě našeho aritmetického výrazu typ `double`, stačí do úvodní části specifikace doplnit text

```
%{
#define YYSTYPE double
%}
```

V praktických situacích však obvykle nevystačíme s jediným typem atributu pro všechny symboly. Jak již bylo uvedeno dříve, můžeme typ atributu definovat jako unii několika různých typů. K tomu nabízí yacc své vlastní prostředky, které mnohem zjednoduší zápis sémantických akcí. V definiční části specifikace můžeme uvést deklaraci všech složek unie, například

```
%union
{  int ival;
   double rval;
}
```

kterou definujeme celočíselný atribut *ival* a reálný atribut *rval*. Dále musíme uvést pro každý terminální a nonterminální symbol s atributem jméno jeho atributu (jméno odpovídající složky unie). Toto jméno pak bude yacc vždy automaticky přidávat ke všem odkazům na atributy příslušných symbolů a zároveň bude kontrolovat, zda jsou definovány typy atributů, na které se v sémantických pravidlech odkazujeme. Definice typu atributu pro terminální symboly se uvádí v lomených závorkách bezprostředně za klíčovým slovem `%term` a platí pro všechny terminální symboly definované za tímto klíčovým slovem. Pro nonterminální symboly se používá obdobná syntaxe s klíčovým slovem `%type`.

Příklad 4.9. Rozšíříme gramatiku z příkladu 4.8 o reálné konstanty s tím, že výpočet hodnoty výrazu se bude celý provádět v pohyblivé čárce. K tomu budeme potřebovat atribut *ival* typu `int` pro celočíselné konstanty (symbol `INUM`) a atribut *rval* typu `double` pro reálné konstanty a nonterminály. Výsledná specifikace pro yacc je uvedena na obr. 4.11. ■

```
%union { int ival; double rval; }
%term <ival> INUM
%term <rval> RNUM
%type <rval> E T F
%%
E  :  E '+' T   { $$ = $1 + $3; }
   |  T
   ;
T  :  T '*' F   { $$ = $1 * $3; }
   |  F
   ;
F  :  '(' E ')' { $$ = $2; }
   |  INUM
   |  RNUM
   ;
```

Obr. 4.11: Specifikace překladu s atributy různých typů

4.4 L-atributové definice

Mnohem širší třídou syntaxí řízených definic jsou L-atributové definice, jejichž atributy se mohou vždy vypočítat během jednoho průchodu analyzátoru zdrojovým textem. Tato třída zahrnuje všechny syntaxí řízené definice založené na LL(1) gramatikách; po určitých úpravách je lze použít i při překladu zdola nahoru. Následující definice specifikuje vlastnosti L-atributových definic.

Definice 4.6. Syntaxí řízená definice je *L-atributová*, jestliže všechny dědičné atributy symbolů $X_j, 1 \leq j \leq n$ na pravé straně pravidla $A \to X_1 X_2 \cdots X_n$ závisejí pouze na

- atributech symbolů $X_1, X_2, \ldots, X_{j-1}$ vlevo od X_j v témže pravidle a
- dědičných atributech symbolu A na levé straně pravidla. ■

Poznamenejme, že každá S-atributová definice je zároveň L-atributová, neboť uvedená omezení se vztahují pouze na dědičné atributy.

Pro zápis L-atributových definic zavedeme pojem *překladové schéma* jako syntaxí řízenou definici, která umožňuje zápis sémantických akcí kdekoliv uvnitř pravé strany pravidla. Tyto sémantické akce budeme uzavírat do složených závorek a budeme předpokládat, že se provedou vždy před analýzou symbolů, které za nimi následují. Překladová schemata nám umožní definovat explicitně pořadí vyhodnocení sémantických akcí.

Příklad 4.10. Překlad výrazů s operátorem sčítání a celočíselnými konstantami můžeme popsat pomocí následujícího překladového schematu:

$$\begin{aligned} E &\to T\ R \\ R &\to +\ T\ \{print('+')\}\ R_1 \mid \epsilon \\ T &\to \mathbf{num}\ \{print(\mathbf{num}.val)\} \end{aligned}$$

■

Při návrhu překladových schémat musíme dbát na to, aby hodnota každého atributu byla dostupná v okamžiku, kdy se na ni odkazujeme. Obecně pokud máme dědičné i syntetizované atributy, je třeba dodržovat následující pravidla:

- Dědičný atribut symbolu na pravé straně pravidla se musí vypočítat akcí umístěnou před tímto symbolem.
- Akce se nesmí odkazovat na syntetizovaný atribut symbolu vpravo od ní.
- Syntetizovaný atribut symbolu na levé straně pravidla se může vypočítat až tehdy, jsou-li k dispozici hodnoty všech atributů, které používá. Výpočet tohoto atributu se obvykle umísťuje na konec pravé strany pravidla.

Následující překladové schéma nedodržuje první z uvedených tří podmínek:

$$\begin{aligned} S &\to A_1\ A_2\ \{A_1.in := 1; A_2.in := 2\} \\ A &\to a\ \ \{print(A.in)\} \end{aligned}$$

Dědičný atribut $A.in$ ve druhém pravidle totiž není v okamžiku pokusu o jeho tisk při analýze řetězce aa definován, pokud procházíme derivačním stromem do hloubky. Průchod začne uzlem

S a dále pokračuje v uzlech A_1 a A_2 ještě před tím, než se nastaví hodnoty $A_1.in$ a $A_2.in$. Umístíme-li akci definující hodnoty $A_1.in$ a $A_2.in$ mezi symboly A na pravé straně pravidla $A \to A_1 A_2$, bude $A.in$ již v okamžiku tisku definováno.

V případě, že máme L-atributovou syntaxí řízenou definici, lze z ní vždy vytvořit překladové schéma, jež splňuje uvedené tři požadavky.

4.5 Překlad shora dolů

V této části si ukážeme, jak lze implementovat L-atributové definice během prediktivní analýzy. Budeme pracovat spíše s překladovými schematy než se syntaxí řízenými definicemi, neboť ta nám umožňují vyjádřit explicitně pořadí akcí a výpočtu atributů. Ukážeme si také, jak se dá odstranit levá rekurze z překladového schematu se syntetizovanými atributy.

4.5.1 Odstranění levé rekurze z překladového schematu

Mnoho aritmetických operátorů je asociativních zleva, takže je přirozené pro jejich syntaxi použít zleva rekurzivní gramatiky. Následující postup umožňuje odstranit levou rekurzi z překladového schematu se syntetizovanými atributy. Předpokládejme, že máme následující překladové schéma:

$$\begin{array}{lcll} A & \to & A_1\ Y & \{A.a := g(A_1.a, Y.y)\} \\ A & \to & X & \{A.a := f(X.x)\} \end{array} \tag{4.2}$$

Všechny symboly mají syntetizované atributy pojmenované odpovídajícím písmenem malé abecedy, f a g jsou libovolné funkce.

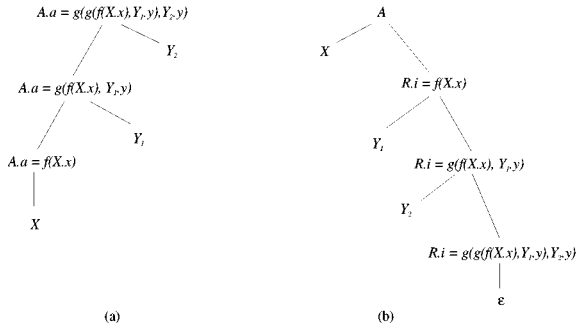
Algoritmem pro odstranění levé rekurze bychom z (4.2) dostali následující gramatiku:

$$\begin{array}{lcl} A & \to & X\ R \\ R & \to & Y\ R \mid \epsilon \end{array} \tag{4.3}$$

Uvažujeme-li sémantické akce, získáme transformované schéma:

$$\begin{array}{lcll} A & \to & X & \{R.i := f(X.x)\} \\ & & R & \{A.a := R.s\} \\ R & \to & Y & \{R_1.i := g(R.i, Y.y)\} \\ & & R_1 & \{R.s := R_1.s\} \\ R & \to & \epsilon & \{R.s := R.i\} \end{array} \tag{4.4}$$

Transformované schéma používá pro R atributy i a s. Aby bylo zřejmé, že výsledky (4.2) a (4.4) jsou shodné, uvažujme dva ohodnocené derivační stromy z obr. 4.12. Hodnota $A.a$ se na obr. 4.12(a) počítá podle (4.2). Obr. 4.12(b) obsahuje výpočet $R.i$ podle (4.4) při průchodu stromem směrem dolů. Hodnota $R.i$ se potom předává nahoru beze změny jako $R.s$ a nakonec se stane hodnotou atributu $A.a$ v kořeni ($R.s$ není na obr. 4.12(b) zakreslen).



Obr. 4.12: Dva způsoby výpočtu atributů

4.5.2 Implementace prediktivního syntaxí řízeného překladače

L-atributové definice, jak již bylo uvedeno, umožňují vyhodnocení atributů v jediném průchodu již během syntaktické analýzy. Pro jejich implementaci můžeme použít metodu rekurzivního sestupu, která byla popsána v předchozí kapitole; rozšíříme ji pouze o sémantické akce a atributy. Atributový zásobník bude v tomto případě implementován podobně jako syntaktický zásobník pomocí implicitního zásobníku implementačního jazyka.

Atributy nonterminálních symbolů můžeme při rekurzivním sestupu reprezentovat jako parametry příslušných procedur. Dědičné atributy při tomto přístupu budou představovat vstupní parametry procedury (tj. parametry předávané hodnotou), syntetizované atributy naopak budou výstupními parametry (tj. budou předávány odkazem). V některých speciálních případech můžeme téhož parametru předávaného odkazem použít zároveň pro dva atributy — jeden dědičný a jeden syntetizovaný.

Atributy terminálních symbolů se vytvářejí v lexikálním analyzátoru a předávají se obvykle v globálních proměnných. Obsah příslušné globální proměnné můžeme podle potřeby uschovat pro pozdější použití do lokální proměnné definované uvnitř procedury.

Sémantické akce můžeme zapsat přímo na odpovídající místa v proceduře. Je však třeba dbát na to, aby se definovaly hodnoty všech syntetizovaných atributů levostranného nonterminálu (tj. hodnoty všech parametrů předávaných odkazem) i v případě, že dojde k syntaktické chybě, ze které se překladač zotaví. Pro tyto účely lze často použít speciálních hodnot atributů, které mohou být dále identifikovány a se kterými lze dále pracovat jako s neznámou informací.

Příklad 4.11. Uvažujme následující překladové schéma pro vyhodnocení výrazů s aditivními operátory a celočíselnými konstantami.

$$
\begin{aligned}
E &\rightarrow T\ \{R.i := T.val\ \}\ R\ \{E.val := R.s\} \\
R &\rightarrow \textbf{addop}\ T\ \{
\end{aligned}
$$

$$
\begin{array}{rcl}
 & & \textbf{if } \textbf{addop}.op = add \textbf{ then} \\
 & & \quad R_1.i := R.i + T.val \\
 & & \textbf{else} \\
 & & \quad R_1.i := R.i - T.val\} \\
 & & R_1\ \{R.s := R_1.s\} \\
 & | & \epsilon\ \{R.s := R.i\} \\
T & \rightarrow & \textbf{num}\ \{T.val := \textbf{num}.ival\}
\end{array}
$$

Symbol **addop** má atribut *op* s hodnotou `'+'` nebo `'-'`; jeho hodnota bude uložena v globální proměnné *lexop*. Terminální symbol **num** představující celočíselnou konstantu má atribut *ival*, jehož hodnotu lexikální analyzátor uloží do globální proměnné *lexival*. Implementace tohoto překladového schematu je na obr. 4.13.

Nonterminál R má dědičný atribut i, jehož hodnotou je vždy levý operand součtu nebo rozdílu, a syntetizovaný atribut s představující mezivýsledek výpočtu hodnoty celého výrazu. Tyto dva atributy bychom mohli sloučit do jediného parametru procedury R a tím celou implementaci zjednodušit. Povšimněte si, že některé sémantické akce, spočívající pouze v přiřazení hodnoty atributu, nejsou zapsány explicitně jako přiřazovací příkazy — například akce $\{R.i := T.val\}$ v pravidle pro nonterminál E se realizuje předáním hodnoty proměnné `val1` jako argumentu procedury `R`. ■

4.6 Vyhodnocení dědičných atributů zdola nahoru

Pokud chceme implementovat L-atributovou definici během překladu zdola nahoru, narazíme na jeden zásadní rozdíl od přístupu shora dolů. Během analýzy zdola nahoru je pravidlo, podle kterého se bude redukovat, známo až v okamžiku redukce, tj. při dosažení jeho konce. To znamená, že všechny sémantické akce můžeme provádět až na konci pravidla. Přesto pomocí určitých transformací můžeme převést všechny L-atributové definice založené na LL(1) gramatikách do tvaru, který lze metodou zdola nahoru implementovat. Tyto transformace lze rovněž použít i na některé (ovšem ne všechny) definice založené na LR(1) gramatikách.

První užitečnou transformací je odstranění sémantických akcí, které jsou uvnitř pravidla. Tato transformace vkládá do původní gramatiky tzv. *marker*, nonterminální symbol generující prázdný řetězec ϵ. Každou sémantickou akci, která je uvnitř pravé strany pravidla, nahradíme novým markerem M a původní sémantickou akci přidáme na konec pravidla $M \rightarrow \epsilon$.

Příklad 4.12. Překladové schéma z příkladu 4.10 můžeme převést do tvaru

$$
\begin{array}{rcl}
E & \rightarrow & T\ R \\
R & \rightarrow & +\ M\ R_1 \mid \epsilon \\
T & \rightarrow & \textbf{num}\ \{print(\textbf{num}.val)\ \} \\
M & \rightarrow & \epsilon\ \{print('+')\}
\end{array}
$$

který je ekvivalentní původnímu, tj. obě gramatiky přijímají stejný jazyk a pro všechny vstupní řetězce se sémantické akce provedou vždy ve stejném pořadí. Sémantické akce jsou nyní na koncích pravidel, takže je můžeme provést bezprostředně před redukcí. ■

```
procedure E(var val: integer);
var val1: integer;
begin
    T(val1);
    R(val1, val)
end;
procedure R(i: integer; var s: integer);
var op: char;
    val: integer;
begin
    if sym in ['+', '-'] then begin
        op := lexop;
        T(val1);
        if op = '+' then
            R(i + val1, s)
        else
            R(i - val1, s)
    end
    else
        s := i
end;
procedure T(var val: integer);
begin
    if sym = NUM then begin
        val := lexival;
        sym := lex
    end
    else
        error;
end;
```

Obr. 4.13: Implementace překladového schematu rekurzivním sestupem

Pokud pracujeme s dědičnými atributy, můžeme využít toho, že během analýzy nonterminálu Y v pravidle $A \to XY$ jsou na zásobníku stále k dispozici atributy symbolu X. Pokud je například dědičný atribut $Y.i$ symbolu i definován pravidlem $Y.i := X.s$, kde $X.s$ je atribut symbolu X, můžeme místo hodnoty $Y.i$ všude použít $X.s$. Důležité je, aby tento atribut byl v zásobníku vždy na stejném místě.

Příklad 4.13. Uvažujme následující překladové schéma pro deklarace proměnných typu *integer* a *real*.

$$
\begin{array}{lll}
D \to T & & \{\ L.in := T.type\ \} \\
\quad L & & \\
T \to \textbf{int} & & \{\ T.type := integer\ \} \\
T \to \textbf{real} & & \{\ T.type := real\ \} \\
L \to & & \{L_1.in := L.in\ \} \\
\quad L_1\ ,\ \textbf{id} & & \{\ addtype(\textbf{id}.entry, L.in)\ \} \\
L \to \textbf{id} & & \{\ addtype(\textbf{id}.entry, L.in)\ \}
\end{array}
$$

V okamžiku redukce libovolné pravé strany nonterminálu L je na zásobníku symbol T bezprostředně před touto pravou stranou. Místo atributu $L.in$, který je definován kopírovacím pravidlem $L.in := T.type$, tedy můžeme použít přímo atributu $T.type$. Uvedené schéma můžeme implementovat pomocí atributového zásobníku *val* tak, jak ukazuje obr. 4.14. Stejně jako na obr. 4.8 proměnná *top* obsahuje současný index vrcholu zásobníku a *ntop* index vrcholu zásobníku po provedení redukce. ■

PRAVIDLO	SÉMANTICKÁ AKCE
$D \to T\ L\ ;$	
$T \to \textbf{int}$	$val[ntop] := integer$
$T \to \textbf{real}$	$val[ntop] := real$
$L \to L\ ,\ \textbf{id}$	$addtype(val[top], val[top-3])$
$L \to \textbf{id}$	$addtype(val[top], val[top-1])$

Obr. 4.14: Implementace dědičných atributů při analýze zdola nahoru

Používáme-li pro generování syntaktického analyzátoru programu yacc, můžeme k atributům symbolů, které leží na zásobníku před pravou stranou redukovaného pravidla, přistupovat stejně jako k atributům symbolů redukovaného pravidla pomocí zápisu $\$i$, kde i je index symbolu. Tento index je roven nule pro první symbol před redukovanou pravou stranou, -1 pro předcházející atd. Schéma z příkladu 4.13 tedy můžeme pro yacc zapsat tak, jak ukazuje obr. 4.15.

```
%term INT REAL ID
%%
D  :  T  L ;
T  :  INT  { $$ = integer; }
   |  REAL { $$ = real; } ;
L  :  L , ID  { addtype($3, $0); }
   |  ID      { addtype($1, $0); } ;
```

Obr. 4.15: Použití dědičných atributů v zápisu pro yacc

V případě, že používáme atributy různých typů, nedovede yacc odvodit sám typ atributu, který nepatří symbolu v pravidle, a je tedy třeba tento typ uvést explicitně zápisem $\$<typ>i$. Yacc také umožňuje zápis sémantických akcí na libovolné místo pravé strany pravidla; případnou transformaci nahrazením sémantické akce markerem provede automaticky. Opět pokud má taková vnitřní akce syntetizovaný atribut a používáme-li typovaných atributů, je třeba uvést typ při všech odkazech na tento atribut.

Kapitola 5

Tabulka symbolů

V tabulce symbolů se uschovávají informace o pojmenovaných objektech, deklarovaných explicitně (uživatelské typy, proměnné, procedury, návěští atd.) nebo implicitně (standardní typy, procedury a funkce, pomocné proměnné vytvořené překladačem atd.). Tyto informace se využívají zejména k následujícím účelům:

- řešení kontextových vazeb v programu (vztah mezi deklarací a použitím objektu), které nelze popsat bezkontextovou gramatikou,
- provádění typové kontroly a
- generování intermediárního a cílového kódu.

Jednotlivé atributy objektů v tabulce symbolů jsou dány buď zdrojovým jazykem (např. jméno, druh, typ, počet parametrů procedury) nebo cílovým jazykem (např. velikost, adresa).

Tabulka symbolů se může vytvářet buď během sémantické analýzy a generování mezikódu — v tom případě předává lexikální analyzátor všechna jména jako řetězce znaků, — nebo se může vytvářet již během lexikální analýzy, kdy jsou jména objektů reprezentována v průběhu celého překladu pouze jako ukazatele do tabulky. Samozřejmě ve druhém případě musí sémantická analýza doplnit do tabulky zbývající údaje, které nemohou být po lexikální analýze ještě známé. Při jednoprůchodovém překladu může lexikální analyzátor přímo vyhledávat nalezené identifikátory v tabulce a umožnit syntaktické analýze využívat pro rozhodování některých kontextově závislých informací, např. místo symbolu pro identifikátor vrátit speciální symbol pro identifikátor proměnné nebo procedury. Taková interakce lexikálního analyzátoru s tabulkou symbolů může vést ke zjednodušení gramatiky a zlepšení detekce a zotavení se po kontextově závislých chybách, na druhé straně se ale snižuje modularita překladače.

5.1 Informace v tabulce symbolů

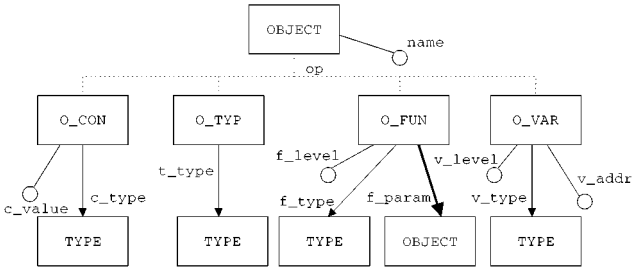
Kromě jmen objektů obsahuje tabulka symbolů — jak již bylo uvedeno na začátku této kapitoly — další informace potřebné pro činnost překladače. Strukturu těchto informací můžeme vyjádřit speciálním grafem, převzatým z teorie databázových systémů, tzv. *E–R grafem* (Entity–Relationship Graph — viz [9]). Tento graf vyjadřuje sémantické vztahy mezi jednotlivými objekty a dá se v překladači přímo implementovat pomocí dynamických datových struktur, jak si dále ukážeme. E–R graf má při překladu mnohem širší použití než jen pro popis informací v

tabulce symbolů, ve skutečnosti umožňuje definovat úplný *sémantický model programu*, kterým můžeme reprezentovat jak deklarace, tak i příkazy nebo výrazy v programu (viz článek 8.1.1).

E–R graf je tvořen dvěma množinami uzlů. Jedna množina uzlů představuje základní sémantické *entity* (pojmenované objekty, typy, příkazy, výrazy atd.) a druhá množina uzlů představuje *atributy* entit. Hrany spojující jednotlivé entity vyjadřují *relace* mezi entitami. Relace mohou být typu 1:1 (např. relace "typ proměnné" přiřazuje entitě "proměnná" její právě jeden typ) nebo typu 1:N (např. relace "parametr procedury" přiřazuje entitě "procedura" uspořádanou množinu objektů, reprezentujících její parametry). Atributy jsou spojeny hranami s uzly, k nimž patří (např. entita "proměnná" může mít jako atribut svou relativní adresu).

Graficky budeme entity znázorňovat obdélníky, relace 1:1 slabšími, relace 1:N silnějšími šipkami a atributy malými kroužky spojenými s entitami hranou. Jeden atribut může odlišovat různé varianty jediné entity (např. "objekt" může být "proměnná", "procedura", "návěští" atd.) — v tom případě tyto varianty nakreslíme jako samostatné entity spojené se společnou částí tečkovanými čarami.

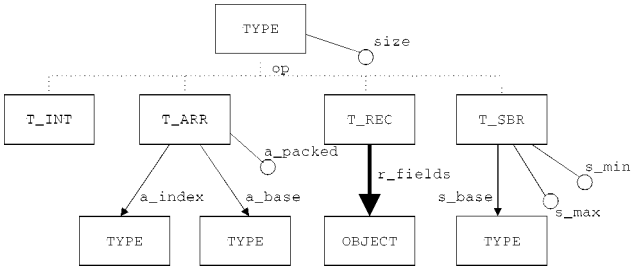
Příklad 5.1. Obr. 5.1 představuje část E–R grafu pro pojmenované objekty v jazyce Pascal. Tabulka symbolů bude v tomto případě uchovávat informace o entitách `OBJECT`, jejichž atribut `name` bude představovat vyhledávací klíč. Objekty mohou být konstanty, typy, funkce nebo proměnné, přičemž atribut `name` představuje jméno pojmenovaného objektu a atribut `op` rozlišuje druh objektu. Všechny uvedené objekty mají relací 1:1 definován typ, funkce má navíc seznam parametrů reprezentovaný relací `f_param` typu 1:N.



Obr. 5.1: E–R graf pro objekty jazyka Pascal

Další entitou, která se na obr. 5.1 používá, je `TYPE`, reprezentující datový typ. Pro tuto entitu můžeme vytvořit stejným postupem graf, jehož část je na obr. 5.2. ■

Implementace E–R grafu pomocí dynamických datových struktur je již jednoduchá. Každou entitu budeme reprezentovat jedním záznamem, který bude obsahovat společné atributy a relace a případně seznam jednotlivých variant této entity. Vazby typu 1:1 můžeme definovat jako ukazatele na příslušné typy entit, vazby 1:N jako ukazatele na první položku seznamu entit.



Obr. 5.2: E–R graf pro datové typy jazyka Pascal

Příklad 5.2. Entitu `OBJECT` z příkladu 5.1 můžeme v Pascalu reprezentovat následujícími datovými typy:

```
type
   Objects = ( O_CON, O_TYP, O_FUN, O_VAR );
   ObjList = record
                ent:  ↑ObjEnt;
                next: ↑ObjList;
             end;
   ObjEnt  = record
                name: String;
                case op: Objects of
                   O_CON: ( c_value: Value;
                            c_type:  ↑TypeEnt );
                   O_TYP: ( t_type:  ↑TypeEnt );
                   O_FUN: ( f_level: Integer;
                            f_type:  ↑TypeEnt;
                            f_param: ↑ObjList );
                   O_VAR: ( v_level: Integer;
                            v_addr:  Integer;
                            v_type:  ↑TypeEnt );
                end;
   TypeEnt = ...
```

Při dalším zpracování takto reprezentovaného modelu deklarací je třeba mít stále na paměti, že výsledná datová struktura — i když se to tak jeví z uvedených příkladů — nemusí být stromová. Například samotná reprezentace datového typu `ObjList` vede k cyklu v grafu (obsahuje ukazatel sama na sebe). Pro průchod sémantickým grafem se proto musejí využívat poněkud upravené algoritmy pro zpracování stromů. ■

5.2 Organizace tabulky symbolů

5.2.1 Operace nad tabulkou symbolů

Dvě nejběžněji prováděné operace nad tabulkou symbolů jsou operace *vkládání* (insertion) a *vyhledávání* (lookup, retrieval).

Operace vkládání do tabulky obecně nejprve zjistí, zda ukládaná hodnota klíče (v tomto případě objekt se stejným jménem) již v tabulce není. Pokud ne, vytvoří se nový záznam a zařadí se do tabulky. V opačném případě se může nahlásit chyba, např. "vícenásobně deklarovaný identifikátor." U některých jazyků však nalezení jména v tabulce nemusí znamenat chybový stav, např. v následujících případech:

- deklarace procedury v Pascalu, jejíž záhlaví už bylo uvedeno dříve s direktivou *forward*,
- deklarace objektu, který byl už v programu použit, a kterému byly přiděleny implicitní atributy (např. funkce nebo návěští příkazu v jazyce C).

Operace vyhledání v tabulce obvykle vrátí informaci o tom, zda se objekt s požadovaným jménem v tabulce nachází, a v případě, že ano, vrátí rovněž nalezený objekt. Pokud objekt v tabulce není a zdrojový jazyk umožňuje implicitní deklarace, vytvoří se nový objekt s implicitními atributy, zařadí se do tabulky a vrátí se stejně, jako by v tabulce již byl.

V následujících odstavcích provedeme pouze přehled nejpoužívanějších metod. Implementace konkrétních algoritmů byla náplní kursu Programovací techniky (viz [8]).

5.2.2 Implementace tabulek pro jazyky bez blokové struktury

Pro jazyky bez blokové struktury vystačíme s jediným adresovým prostorem pro všechny položky. Některé z dále uvedených metod se rovněž používají pro vyhledávání v tabulce. Základní implementační metody jsou:

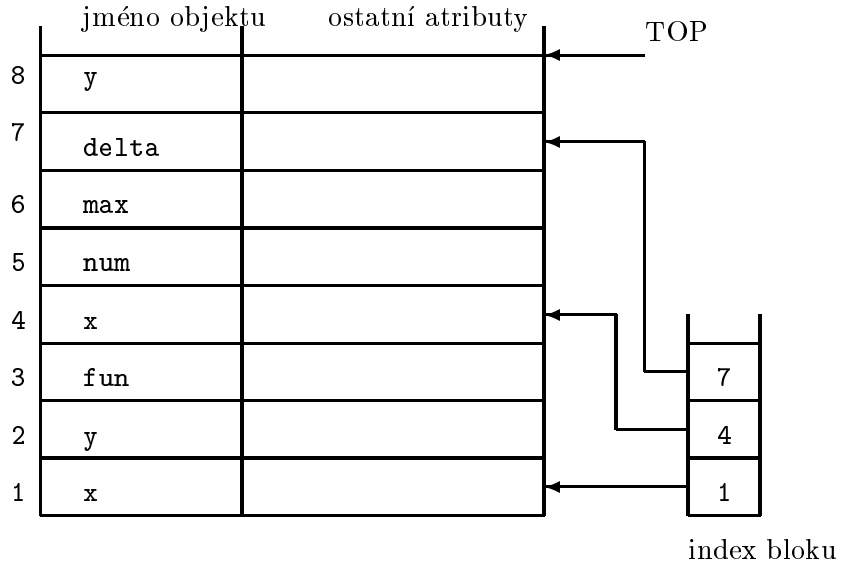
- *Neseřazené tabulky.* Neseřazené tabulky (pole, seznamy) jsou z hlediska implementace nejjednodušší. Položky do nich vkládáme v tom pořadí, jak jsou deklarované. Ukládání i vyhledávání má však časovou náročnost $\mathcal{O}(n)$, kde n je počet položek v tabulce. Tato organizace se dá použít pouze tehdy, očekáváme-li malý počet položek.
- *Seřazené tabulky s binárním vyhledáváním.* Použijeme-li pro tabulku symbolů seřazené pole, můžeme snížit časovou náročnost vyhledávání na $\mathcal{O}(\log_2 n)$, ovšem nezmění se časová náročnost vkládání, neboť musíme stále zajišťovat seřazení tabulky. Binární vyhledávání v seřazeném poli je výhodné právě pro tabulky klíčových slov, které jsou statické.
- *Stromově strukturované tabulky.* Stromové uspořádání tabulky symbolů redukuje dobu vkládání na $\mathcal{O}(\log_2 n)$. Doba vyhledávání se pohybuje mezi $\mathcal{O}(n)$ a $\mathcal{O}(\log_2 n)$, v závislosti na struktuře stromu. Tato doba je konstantní pro optimálně vyvážené stromy, které však vyžadují značně složité algoritmy vkládání. Proto se velmi často používají různá suboptimální řešení, nejčastěji *AVL stromy.*
- *Tabulky s rozptýlenými položkami.* Z hlediska doby vyhledávání jsou nejvýhodnějším řešením tabulky s rozptýlenými položkami, u nichž doba vyhledávání je do značné míry nezávislá na počtu záznamů v tabulce (závislost se projevuje až při vysokém zaplnění, kterému

se dá předejít vhodnou volbou velikosti tabulky). Nevýhody této organizace jsou především v problematickém ošetření přeplnění tabulky, velkých nárocích na paměť a v tom, že tabulka neumožňuje systematický průchod položkami v abecedním pořadí.

5.2.3 Implementace blokově strukturované tabulky symbolů

Pro jazyky s blokovou strukturou jako Pascal, C nebo Modula-2 musí být k dispozici ještě další dvě operace, které označíme jako `tabopen` a `tabclose`. Operace `tabopen` se volá vždy na začátku nového bloku deklarací a operace `tabclose` na konci bloku. Tyto operace zajišťují rozlišování jednotlivých úrovní deklarací a umožňují uchovávat v tabulce několik různých objektů označených stejnými jmény za předpokladu, že byly deklarovány na různých úrovních. Operace vkládání a vyhledávání musejí proto splňovat ještě tyto dodatečné podmínky:

- při vkládání se pracuje pouze s naposledy otevřenou úrovní tabulky, případné další výskyty téhož jména na některé nižší úrovni se neberou v úvahu;
- při vyhledávání se prohledávají postupně všechny úrovně tabulky od nejvyšší úrovně k nejnižší a vrátí se objekt odpovídající prvnímu nalezenému výskytu hledaného jména.

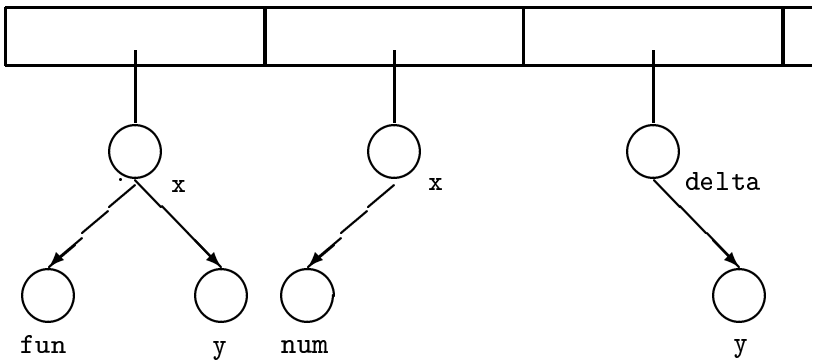


Obr. 5.3: Příklad zásobníkové organizace tabulky symbolů s blokovou strukturou

Implementace blokově strukturované tabulky symbolů je obvykle založena na některé z metod, které byly uvedeny v předchozím odstavci. Vzhledem k tomu, že každá úroveň tabulky symbolů se uzavírá až tehdy, jsou-li uzavřené všechny vnořené úrovně, je přirozenou reprezentací blokově strukturované tabulky zásobník. V praxi se nejčastěji užívají tyto kombinace:

- *Zásobníková tabulka symbolů.* Jde o nejjednodušší organizaci tabulky, kdy jsou záznamy jednoduše umisťovány na vrchol zásobníku tak, jak přichásejí jednotlivé deklarace symbolů. Kromě zásobníku položek se ještě udržuje zásobník indexů úrovní, který ukazuje odkaz

vždy na první položku dané úrovně (viz obr. 5.3). Operace `tabopen` pouze uloží na jeho vrchol současný index vrcholu zásobníku položek a operace `tabclose` vrátí index zásobníku položek na hodnotu, která leží na vrcholu zásobníku indexů. Při vyhledávání se prochází zásobník od vrcholu směrem zpět, při ukládání se hledají případné předchozí výskyty jména pouze na vrcholu zásobníku, až po naposledy uložený index. Tato organizace je velmi podobná nesetříděné tabulce symbolů včetně jejích nevhodných časových charakteristik, proto se dá použít pouze tam, kde se neočekává velký počet ukládaných položek.

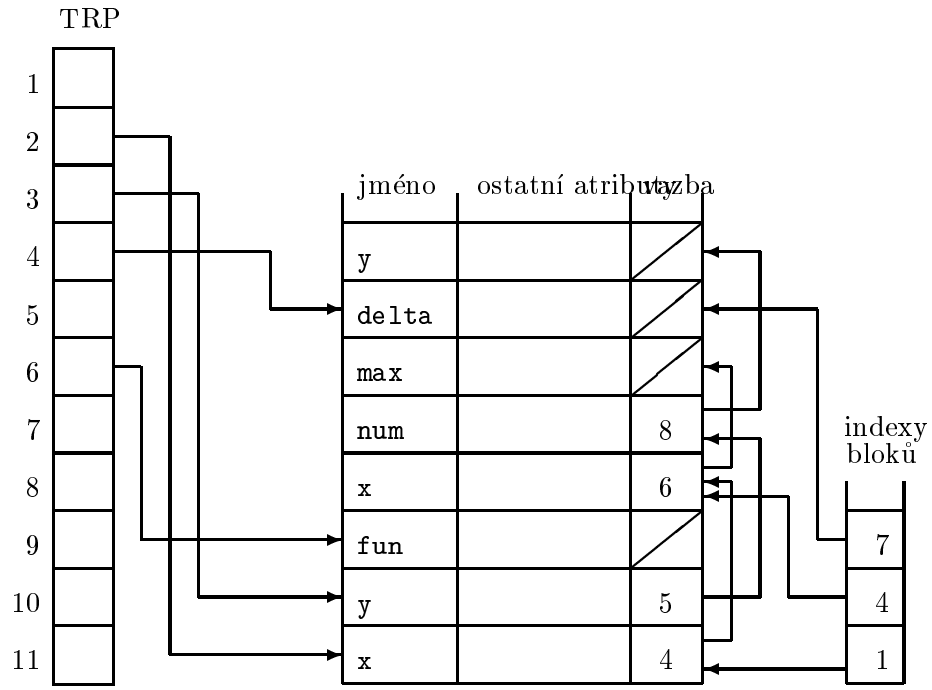


Obr. 5.4: Příklad stromově organizované tabulky symbolů s blokovou strukturou

- *Kombinace zásobníku a stromu.* Při této organizaci udržujeme podobně jako v předchozím případě zásobník otevřených úrovní tabulky symbolů, ovšem tento zásobník nyní bude obsahovat odkazy na kořenové uzly stromů pro jednotlivé úrovně (viz obr. 5.4). Každé otevřené úrovni nyní přísluší samostatná tabulka symbolů, organizovaná jako strom. Při vkládání se pracuje pouze se stromem, na který ukazuje položka na vrcholu zásobníku úrovní, při vyhledávání se postupně procházejí jednotlivé úrovně počínaje naposledy otevřenou úrovní. Tato metoda je zvláště vhodná pro velký počet položek v tabulce, pokud jsme omezováni velikostí paměti.

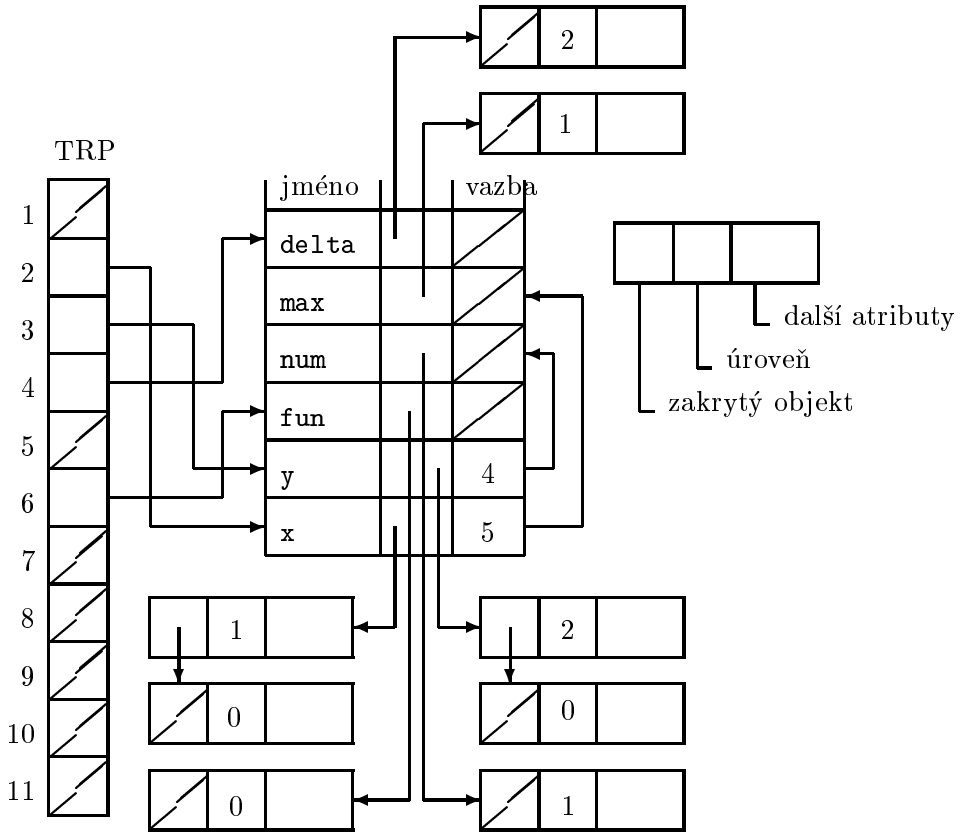
- *Kombinace zásobníku a tabulky s rozptýlenými položkami.* Použití tabulky s rozptýlenými položkami pro blokově strukturované jazyky není příliš zřejmé, tento typ tabulky nezachovává pořadí položek a nemůže samostatně zajistit určitý způsob procházení tabulkou. Je však možné použít oddělený prostor pro položky a vlastní tabulku organizovat pouze jako tabulku ukazatelů na položky (viz obr. 5.5). V tom případě můžeme podobně jako v první uvedené metodě ukládat do zásobníku index první přidělené položky každé otevřené úrovně. Tím máme k dispozici informaci o příslušnosti položek tabulky do jednotlivých bloků, kterou můžeme využít při vyhledávání a vkládání. Operace `tabclose` kromě toho, že obnoví index vrcholu zásobníku položek, musí rovněž odstranit všechny příslušné odkazy a nahradit je příznaky neplatné položky (rušením položek v tabulce s rozptýlenými položkami se zabývá učební text [8]). Tato metoda vyžaduje při vkládání a vyhledávání projít celým řetězcem synonym a vyhledat v něm všechny výskyty téhož jména.

- *Jednoúrovňová blokově strukturovaná tabulka symbolů.* Pravděpodobně nejefektivnější variantou blokově strukturované tabulky symbolů s rozptýlenými položkami využívá zásobníku pro ukládání všech existujících deklarací konkrétního identifikátoru (viz 5.6), zatímco



Obr. 5.5: Příklad blokově strukturované tabulky s rozptýlenými položkami

hlavní vyhledávací mechanismus je implementován jedinou společnou vyhledávací tabulkou pro všechny úrovně. Je-li při operaci vkládání v tabulce nalezena položka se shodným jménem, avšak deklarovaná v nadřazené úrovni, přidá se do tabulky nová položka, na kterou se přesměruje původní odkaz, a do nové položky se uschová adresa zakryté položky. Tím se při vyhledávání zajistí, že budou přístupná pouze ta jména, která jsou zároveň dostupná na současné úrovni deklarací v programu. Operace `tabclose` musí v tabulce vyhledat všechny položky patřící do právě uzavírané úrovně, obnovit odkazy na zakrytá jména, případně zcela z tabulky odstranit odkazy na jména, která nebyla deklarována v žádném nadřazeném bloku. Podobná organizace se dá využít i pro implementaci tabulky pomocí binárních vyhledávacích stromů. Její hlavní výhoda je v tom, že doba vyhledávání není závislá na deklarační úrovni hledaného jména (vyhledávání probíhá paralelně na všech úrovních).



Obr. 5.6: Příklad jednoúrovňové blokově strukturované tabulky symbolů

Kapitola 6

Struktura programu v době běhu

Ještě než začneme uvažovat generování kódu, musíme definovat vztah mezi statickým zdrojovým textem programu a akcemi, které se musejí provést v době běhu programu. Během zpracování může totéž jméno ve zdrojovém textu označovat různé objekty na cílovém počítači. Tato kapitola se bude zabývat vztahem mezi jmény a datovými objekty.

Přidělování a uvolňování paměti pro datové objekty má na starosti *systém řízení programu v době běhu* (run-time system), tvořený podprogramy zaváděnými společně s cílovým programem. Návrh řídicího systému je silně ovlivňován sémantikou procedur. V této kapitole se budeme zabývat technikami, které jsou využitelné pro jazyky jako je C, Pascal nebo Modula-2.

Každé provedení procedury nazýváme její *aktivací*. Je-li procedura rekurzivní, může v jednom okamžiku existovat zároveň několik jejích aktivací. Každé volání procedury v Pascalu vede k aktivaci, která může manipulovat s datovými objekty přidělenými speciálně pro její potřebu.

Reprezentace datových objektů v době běhu je dána jejich typem. Často lze elementární datové typy jako jsou znaky, celá a reálná čísla reprezentovat na cílovém počítači ekvivalentními datovými objekty. Složené datové typy jako jsou pole, řetězce a struktury, se obvykle reprezentují jako kolekce primitivních objektů.

6.1 Podprogramy

Většina současných procedurálních programovacích jazyků umožňuje vytváření strukturovaných programů, ve kterých je základním pojmem *podprogram* jako samostatná programová jednotka, představující abstrakci nějaké akce. Abychom byli konkrétní, budeme předpokládat, že zdrojový program je tvořen procedurami a funkcemi jako v Pascalu.

6.1.1 Statická a dynamická struktura podprogramů

Definice podprogramu je deklarace, která ve své nejjednodušší formě váže identifikátor s příkazem. Tento identifikátor je *jméno podprogramu* a příkaz je *tělo podprogramu*. Například úsek programu v Pascalu na obr. 6.1 obsahuje na řádcích 3–9 definici podprogramu se jménem `fib`; tělo podprogramu je na řádcích 5–8. Podprogramy, které vracejí hodnotu, se nazývají *funkce*, ostatní podprogramy se nazývají *procedury*. Celý program lze rovněž chápat jako podprogram volaný programy operačního systému počítače.

Vyskytne-li se jméno podprogramu uvnitř proveditelného příkazu, říkáme, že se podprogram v tomto bodě *volá*. Volání podprogramu provede jeho tělo. Hlavní program na řádcích 16–19

```
(1)   program table(input,output);
(2)      var max : integer;

(3)      function fib(n: integer): integer;
(4)         begin
(5)            if n < 2 then
(6)               fib := 1
(7)            else
(8)               fib := fib(n-2) + fib(n-1)
(9)         end;

(10)     procedure printtab(n: integer);
(11)        var i : integer;
(12)        begin
(13)           for i := 1 to n do
(14)              writeln( i:3, fib(i):6 );
(15)        end;

(16)     begin
(17)        read(max);
(18)        printtab(max);
(19)     end.
```

Obr. 6.1: Program v Pascalu pro tisk tabulky Fibonacciho čísel

v obr. 6.1 volá na řádku 18 proceduru `printtab`. Volání procedur má obvykle charakter příkazu, zatímco volání funkcí se vyskytuje jako součást výrazu.

Některé identifikátory v definici podprogramu jsou speciální a nazývají se *formální parametry* podprogramu. Identifikátor `n` je formálním parametrem procedury `fib`. Volanému podprogramu můžeme předat argumenty, nazývané také *skutečné parametry*; tyto argumenty nahrazují formální parametry podprogramu v jeho těle. Vztahem mezi skutečnými a formálními argumenty se budeme zabývat v článku 6.5. Na řádku 14 v obr. 6.1 je volání `fib` se skutečným parametrem `i`.

Každé provedení těla podprogramu nazýváme aktivací podprogramu. *Doba života* aktivace podprogramu `p` je posloupnost kroků mezi prvním a posledním krokem provádění těla podprogramu, včetně času stráveného prováděním podprogramů volaných z `p`, jimi volaných podprogramů atd.

Jsou-li a a b aktivace podprogramů, potom jejich doby života se buď nepřekrývají, nebo jsou do sebe zanořené. To znamená, že začne-li b ještě před ukončením a, musí řízení opustit b dříve než a. Tato vlastnost se dá využít při přidělování prostoru pro lokální proměnné podprogramů na zásobníku. Podprogram je *rekurzivní*, jestliže jeho nová aktivace může začít ještě předtím, než se ukončí jeho dřívější aktivace.

Podprogramy představují prostředek pro strukturalizaci programu. Na tuto strukturalizaci můžeme pohlížet ze dvou stran: jako na statické členění textu programu do samostatných jednotek, nebo jako na hierarchii aktivních podprogramů v době běhu programu.

V jazycích jako je Pascal nebo Modula-2 mohou být uvnitř podprogramů deklarovány další podprogramy, které jsou v nich lokální. Každý podprogram má přiděleno číslo odpovídající jeho

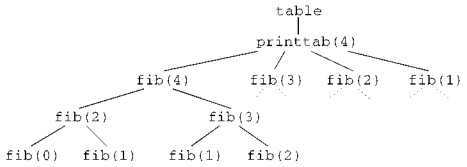
statické úrovni zanoření. Hlavní program má statickou úroveň 0, podprogramy v něm deklarované úroveň 1 atd. Například všechny funkce v jazyce C mají statickou úroveň 1.

Při běhu přeloženého programu dochází k volání jednotlivých podprogramů, které definuje implicitně *dynamickou úroveň zanoření*. Dynamickou strukturu programu můžeme znázornit *aktivačním stromem*, pro který platí následující pravidla:

1. každý uzel reprezentuje aktivaci podprogramu,
2. kořen reprezentuje aktivaci hlavního programu,
3. uzel a je přímým předchůdcem uzlu b, právě když se řízení předává z aktivace b do a,
4. uzel a je uveden vlevo od uzlu b, právě když doba života a předchází dobu života b.

Dynamická úroveň zanoření konkrétní aktivace podprogramu je potom rovna vzdálenosti příslušného uzlu od kořene aktivačního stromu.

Příklad 6.1. Aktivační strom na obr. 6.2 byl vytvořen pro program `table` z obr. 6.1 pro vstupní hodnotu `max` rovnou 4. Kořen stromu je tvořen hlavním programem, pod nímž následuje aktivace procedury `printtab` a dále jednotlivá rekurzivní volání funkce `fig`. ■



Obr. 6.2: Aktivační strom

Při běhu programu má každá aktivace podprogramu obvykle k dispozici vlastní oblast paměti pro lokální proměnné a další pomocné údaje (obsah registrů v okamžiku volání, návratová adresa z podprogramu apod.). Tato oblast paměti se nazývá *aktivační záznam* podprogramu. Aktivační záznamy mohou mít v případě, že zdrojový jazyk neumožňuje rekurzivní volání, přidělenu statickou oblast paměti nebo se mohou uchovávat v zásobníku. Při volání podprogramu se na vrchol řídicího zásobníku uloží nový aktivační záznam, který se odstraní při návratu zpět. Je-li na vrcholu řídicího zásobníku aktivační záznam pro uzel n aktivačního stromu, potom zbytek zásobníku obsahuje aktivační záznamy všech nadřazených uzlů v cestě od kořene k uzlu n. Blíže se budeme organizací paměti v době běhu zabývat v dalším článku.

6.2 Organizace paměti

Přeložený program dostane od operačního systému počítače k dispozici blok paměti, který obecně může být rozdělen na následující části:

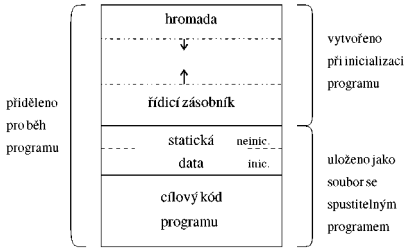
- vygenerovaný cílový kód,

- statická data,
- řídicí zásobník,
- hromada.

Velikost vygenerovaného kódu je známa již v době překladu, takže jej může překladač umístit do staticky definované oblasti, obvykle na začátek přiděleného paměťového prostoru. Rovněž velikost statických datových objektů může být známa již v době překladu a překladač je může umístit za program nebo uložit dokonce jako součást programu. Například v jazyce Fortran lze všem proměnným vyhradit prostor ve statické oblasti paměti, neboť neumožňuje rekurzivní volání podprogramů a pracuje pouze s daty, jejichž umístění lze definovat staticky v době překladu.

Jazyky umožňující rekurzivní volání procedur (C, Pascal) využívají pro aktivace podprogramů řídicího zásobníku, do kterého se ukládají jednotlivé aktivační záznamy. Strukturou aktivačního záznamu se budeme zabývat později.

Pro účely dynamického přidělování paměti (explicitně vyžádaného voláním příslušných funkcí nebo implicitně při přidělování paměti například pro pole s dynamickými rozměry) se používá zvláštní část paměti zvané *hromada*. Vzhledem k tomu, že se velikosti použité části paměti pro zásobník a hromadu v průběhu činnosti programu mohou značně měnit, je výhodné pro obě části využít opačné konce společné části paměti — viz obr. 6.3. Nedostatek paměti se rozpozná tehdy, jestliže ukazatel konce některé oblasti překročí hodnotu ukazatele konce druhé oblasti.



Obr. 6.3: Organizace paměti při běhu programu

6.3 Strategie přidělování paměti

Pro datové oblasti, jimiž jsme se zabývali v předchozím článku, se používají následující hlavní metody přidělování paměti:

- statické přidělení paměti v době překladu,
- přidělování paměti na zásobníku a

- přidělování paměti z hromady.

V dalších odstavcích se zaměříme na přidělování paměti pro aktivační záznamy podprogramů.

6.3.1 Statické přidělování

Při statickém přidělování paměti jsou všem objektům v programu přiděleny adresy již v době překladu. Při kterémkoliv volání podprogramu jsou jeho lokální proměnné vždy na stejném místě, což umožňuje zachovávat hodnoty lokálních proměnných nezměněné mezi různými aktivacemi podprogramu. Statická alokace proměnných však klade na zdrojový jazyk určitá omezení. Údaje o velikosti a počtu všech datových objektů musejí být známy již v době překladu, rekurzivní podprogramy mají velmi omezené možnosti, neboť všechny aktivace podprogramu sdílejí tytéž proměnné, a konečně nelze vytvářet dynamické datové struktury.

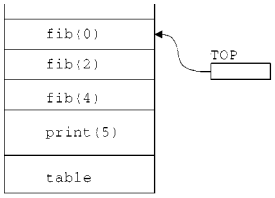
Jedním z jazyků, které používají statické přidělování paměti, je Fortran. Program ve Fortranu se skládá z hlavního programu, podprogramů a funkcí. Aktivační záznamy podprogramů mohou být umístěny dokonce přímo v kódu, což se používalo běžně u starších počítačů.

6.3.2 Přidělování na zásobníku

Přidělování paměti pro aktivační záznamy na zásobníku se používá běžně u jazyků, které umožňují rekurzivní volání podprogramů nebo které používají staticky do sebe zanořené podprogramy. Paměť pro lokální proměnné je přidělena při aktivaci podprogramu vždy na vrcholu zásobníku a při návratu je opět uvolněna. To ale zároveň znamená, že hodnoty lokálních proměnných se mezi dvěma aktivacemi podprogramu nezachovávají.

Při implementaci přidělování paměti na zásobníku bývá jeden registr vyhrazen jako ukazatel na začátek aktivačního záznamu na vrcholu zásobníku. Vzhledem k tomuto registru se pak počítají všechny adresy datových objektů, které jsou umístěny v aktivačním záznamu. Naplnění registru a přidělení nového aktivačního záznamu je součástí *volací posloupnosti*, obnovení stavu před voláním se provádí během *návratové posloupnosti*. Volací (a návratové) posloupnosti se od sebe v různých implementacích liší. Jejich činnost bývá rozdělena mezi volající a volaný program; obvykle volající program určí adresu začátku nového aktivačního záznamu (k tomu potřebuje znát velikost záznamu vlastního), přesune do něj předávané argumenty a spustí volaný podprogram zároveň s uložením návratové adresy do určitého registru nebo na známé místo v paměti. Volaný podprogram nejprve uschová do svého aktivačního záznamu stavovou informaci (obsahy registrů, stavové slovo procesoru, návratovou adresu), inicializuje svá lokální data a pokračuje zpracováním svého těla. Při návratu opět volaný podprogram uloží hodnotu výsledku do registru nebo do paměti, obnoví uschovanou stavovou informaci a provede návrat do volajícího programu. Ten si převezme návratovou hodnotu a tím je volání podprogramu ukončeno. Na obr. 6.4 je uveden stav řídicího zásobníku při vyhodnocování nejlevějšího koncového uzlu aktivačního stromu z obr. 6.2.

Umožňuje-li zdrojový jazyk předávat podprogramům datové struktury, jejichž velikost není známa v době překladu (např. pole, jehož počet prvků je dán hodnotou jiného parametru), je třeba uvedenou strategii poněkud modifikovat. V části aktivačního záznamu, kde jsou umístěny parametry, se vyhradí pouze místo pro deskriptor objektu s ukazatelem na jeho skutečnou hodnotu a případně ještě dalšími informacemi, a pro vlastní objekt se vyhradí místo samostatně až za všemi položkami s pevnou délkou. K hodnotě objektu se pak přistupuje nepřímo přes deskriptor.



Obr. 6.4: Řídicí zásobník

6.3.3 Přidělování z hromady

Strategie přidělování na zásobníku je nepoužitelná, pokud mohou hodnoty lokálních proměnných přetrvávat i po ukončení aktivace, případně pokud aktivace volaného podprogramu může přežít aktivaci volajícího. V těchto případech přidělování a uvolňování aktivačních záznamů se mohou překrývat, takže nemůžeme paměť organizovat jako zásobník.

Aktivační záznamy se mohou v těchto nejobecnějších situacích přidělovat z volné oblasti paměti (hromady), která se jinak používá pro dynamické datové struktury vytvářené uživatelem. Přidělené aktivační záznamy se uvolňují až tehdy, pokud se ukončí aktivace příslušného podprogramu nebo pokud už nejsou lokální data potřebná.

Při použití této strategie se pro vlastní přidělování a uvolňování paměti používají stejné techniky jako pro dynamické proměnné.

6.4 Metody přístupu k nelokálním objektům

V předchozích odstavcích jsme se zabývali různými metodami přidělování paměti pro lokální data podprogramů. Nebrali jsme však do úvahy existenci globálních dat — globálních datových objektů přístupných v rámci celého programu, případně lokálních proměnných ve staticky nadřazených podprogramech.

Data, která jsou globální v celém programu, mají charakter statických dat a může být pro ně použito techniky statického přidělování paměti. Adresy těchto objektů jsou známy již v době překladu. Například v jazyce C existují pouze globální data a lokální data jednotlivých funkcí, které do sebe nemohou být staticky zanořené.

Pro podprogramy, které jsou staticky zanořené do jiných podprogramů, musíme zajistit možnost přístupu k lokálním proměnným nadřazených bloků, tj. k jejich aktivačním záznamům. Nejjednodušším řešením je rozšíření aktivačního záznamu o ukazatel na aktivační záznam bezprostředního staticky nadřazeného podprogramu (*přístupový ukazatel*). Odkazuje-li se příkaz v proceduře `p` na statické úrovni n_p na proměnnou `a` na statické úrovni n_a, se musí nejprve projít $n_p - n_a$ přístupovými ukazateli, čímž získáme adresu aktivačního záznamu obsahujícího proměnnou `a`. Tuto adresu pak můžeme již přímo použít pro zpřístupnění proměnné `a`, neboť její relativní adresa v aktivačním záznamu je známa.

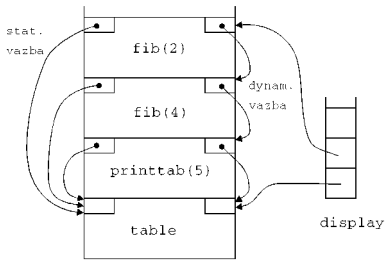
Kód pro vytvoření přístupových ukazatelů je součástí volací posloupnosti podprogramu.

Předpokládejme, že procedura `p` na statické úrovni n_p volá proceduru `x` na statické úrovni n_x. Postup při vytváření přístupového ukazatele závisí na tom, zda je či není volaná procedura zanořená do volající.

1. Je-li $n_p < n_x$, je `x` zanořená mnohem hlouběji než `p` a musí tedy být deklarovaná uvnitř `p` (jinak by nebyla přístupná). Přístupový ukazatel volané procedury v tomto případě bude ukazovat na přístupový ukazatel volající procedury.

2. Je-li $n_p \geq n_x$, musí být nadřazené bloky jak volané, tak volající procedury na úrovních $1, 2, \ldots, n_x - 1$ stejné. Následuje-li volající procedura $n_p - n_x + 1$ přístupových ukazatelů, dostane se na nejvyšší úroveň, která staticky zahrnuje obě procedury, volající i volanou. Přístupový ukazatel volané procedury se pak nastaví tak, aby ukazoval na ukazatel nalezeného bloku.

Uvedená metoda zpřístupnění globálních objektů vyžaduje při každém přístupu ke globálnímu objektu generovat instrukce pro průchod přístupovými ukazateli. Tento proces se dá zrychlit, pokud udržujeme v paměti pole d ukazatelů na aktivační záznamy, zvané *display*. Obsah tohoto pole je vždy takový, že hodnota $d[i]$ udává adresu aktivačního záznamu podprogramu na statické úrovni i (viz obr. 6.5). Při volání podprogramu na statické úrovni i nejprve musíme uschovat do nového aktivačního záznamu starou hodnotu $d[i]$ a potom nastavit $d[i]$ tak, aby ukazoval na nový aktivační záznam. Před ukončením aktivace pouze obnovíme uschovanou hodnotu $d[i]$.



Obr. 6.5: Přístupové ukazatele a display

Display může být implementován různými způsoby. Pokud má cílový počítač dostatečný počet registrů, může být display tvořen posloupností vybraných registrů; tím se značně zjednoduší přístup k nelokálním proměnným, zvláště má-li cílový počítač instrukce s adresou danou součtem obsahu registru a nějaké konstanty. Překladač může na základě analýzy programu zjistit nejvyšší statickou úroveň zanoření, a tím i požadovaný počet registrů pro display, takže zbývající registry se mohou použít pro výpočty.

6.5 Předávání parametrů do podprogramů

Parametry podprogramu mají obvykle přidělen prostor v aktivačním záznamu. Do tohoto prostoru se při volání podprogramu umístí skutečné parametry — hodnoty, adresy, případně jiné datové struktury zpřístupňující předávaný parametr. To, co se konkrétně předává, závisí na typu a požadovaném způsobu předávání.

V této části se budeme zabývat několika technikami předávání parametrů. Na základě způsobu implementace můžeme tyto techniky rozdělit do tří skupin:

- *předávání hodnotou (kopírováním), výsledkem a hodnotou-výsledkem*
  Hodnota skutečného parametru se zkopíruje do formálního parametru nebo se výsledná hodnota formálního parametru zkopíruje zpět do skutečného parametru.

- *předávání odkazem (var)*
  Parametry předávané odkazem se reprezentují jako adresa skutečného parametru. Změna takového formálního parametru vede k bezprostřední změně skutečného parametru.

- *předávání jménem*
  Parametry předávané jménem se podle potřeby vyhodnocují při všech odkazech. Jejich zpracování je blízké zpracování makrodefinic.

- *předávání procedur a funkcí*
  Parametry, které představují procedury nebo funkce, se předávají jako deskriptory podprogramů; tyto deskriptory obsahují kromě adresy vstupního bodu podprogramu též vazbu reprezentující prostředí, v němž se má podprogram provádět.

6.5.1 Předávání parametrů hodnotou a výsledkem

Při předávání hodnotou se do aktivačního záznamu podprogramu zkopíruje hodnota skutečného parametru a veškeré výpočty uvnitř podprogramu se provádějí s touto kopií. To znamená, ze hodnota skutečného parametru se při tomto způsobu předávání nezmění. Parametry předávané hodnotou můžeme považovat za vstupní parametry podprogramu. Podobně při předávání výsledkem se v podprogramu pracuje stále s lokální hodnotou formálního parametru, která se při návratu z podprogramu okopíruje do skutečného parametru (skutečným parametrem tedy musí být L-hodnota, tj. taková hodnota, která může stát na levé straně přiřazení). Parametry předávané výsledkem mohou být pouze výstupními parametry. Kombinací obou metod získáme zároveň vstupní i výstupní parametr.

Tento způsob předávání parametrů můžeme implementovat jednoduše v místě volání, kdy přesuneme hodnotu parametru do nebo z aktivačního záznamu volaného podprogramu. Uvnitř podprogramu s takovým parametrem zacházíme stejně jako s kteroukoliv jinou lokální proměnnou. Poněkud odlišný přístup je třeba volit při předávání polí nebo řetězců. Zde se často využívá nepřímého přístupu přes přístupový vektor (deskriptor), který obsahuje adresu začátku pole nebo řetězce a případně i další údaje, jako počet prvků pole, délku řetězce nebo rozsahy indexů. Takto je možné implementovat i předávání polí a řetězců proměnné délky. Velikost přístupového vektoru je známa v době překladu a je tedy možné pro něj vyhradit pevné místo v aktivačním záznamu. Skutečná hodnota pak může být uložena na jiném místě, např. v oblasti pro dynamické proměnné. Při předávání záznamů můžeme přesunout přímo hodnotu záznamu nebo předat jen jeho adresu a nechat vlastní přesun na volaném podprogramu.

6.5.2 Předávání parametrů odkazem

Při této metodě předávání parametrů umístí volající do aktivačního záznamu volaného podprogramu pouze adresu předávané l-hodnoty. Uvnitř podprogramu se pak všechny odkazy na takový formální parametr zpracovávají jako nepřímé. Pro pole můžeme předat přímo adresu začátku pole nebo adresu přístupového vektoru. Předávání parametrů odkazem se dá jednoduše nahradit předáváním adres parametrů hodnotou, například jako je to definováno v jazyce C. Pokud však takový jazyk nemá dostatečně silnou typovou kontrolu, může velmi často docházet k chybám, například pokud programátor předá místo ukazatele přímo hodnotu nebo naopak pokud místo hodnoty formálního parametru pracuje s jeho adresou.

Příklad 6.2. Následující podprogram v jazyce C provádí záměnu hodnot dvou proměnných, jejichž adresy jsou předávány hodnotou. Všechny výskyty parametrů ve výrazech musejí explicitně obsahovat dereferenci ukazatele.

```
void swap(int *x, int *y)
{
    int temp;
    temp = *x; *x = *y; *y = temp;
}
```

■

6.5.3 Předávání parametrů jménem

Metoda předávání parametrů jménem byla použita například v jazyce Algol 60. Je-li jako skutečný parametr předán výraz, např. odkaz na prvek pole `a[i]`, závisí v každém okamžiku jeho hodnota nejen na obsahu pole `a`, ale i na hodnotě proměnné `i`. Každý výskyt formálního parametru předávaného hodnotou v textu podprogramu se vlastně nahradí textově hodnotou skutečného parametru, jako by šlo o makrodefinici.

Příklad 6.3. Volání `swap(i, a[i])` podprogramu z příkladu 6.2 by se provedlo tak, jako bychom zapsali

```
temp := i; i := a[i]; a[i] := temp
```

To znamená, že při volání jménem se sice `i` nastaví na `a[i]` tak, jak očekáváme, avšak počáteční hodnotu I_0 proměnné `i` uloží do `a[a[`I_0`]]` a ne do `a[`I_0`]`. Lze ukázat, že pokud se používá předávání jménem, nelze správně pracující verzi procedury `swap` vůbec napsat. ■

Implementace předávání parametru jménem je značně obtížná. Pro každý takový parametr musíme vygenerovat podprogram pro jeho vyhodnocení. Další komplikací je, že vyhodnocení parametru musí probíhat v prostředí volajícího podprogramu (například pro odkazy na proměnné se musí použít tabulka symbolů platná v místě volání). Podprogramu se tedy předává dvojice hodnot — adresa podprogramu pro vyhodnocení parametru a adresa definující prostředí v místě volání. Vzhledem k problematické implementaci se dnes metoda předávání parametrů jménem nepoužívá, je však zajímavá z hlediska vývoje jazyků a implementačních technik. Tato metoda je také velice blízká technice tzv. *otevřených* (inline) *podprogramů*, tj. podprogramů, jejichž tělo se vždy rozvine v místě volání.

6.5.4 Předávání procedur a funkcí

Při předávání podprogramu jako parametru musíme v jazycích, které umožnují zanořování podprogramů, řešit obdobný problém jako při předávání parametrů jménem. Nestačí pouze předat adresu začátku podprogramu — předávaný podprogram musí mít v okamžiku volání připraveno totéž prostředí, jako by byl volaný v místě předávání. Jedná se především o vazby zajišťující přístup ke staticky nadřazeným lokálním proměnným.

```
procedure A;
   var m: real;

   procedure B(procedure P);
   begin
      P
   end;

   procedure C;
      var x: real;
      procedure D;
      begin
         x := 3.25;
      end;
      procedure E;
      begin
         B(D)
      end;
   begin
      E
   end;
begin
   C
end;
```

Obr. 6.6: Předávání procedury `D` jako parametru

Například v programu na obr. 6.6 procedura `E` volá proceduru `B` a předává jí jako parametr proceduru `D`. Procedura `D` musí mít přístupné proměnné `m` a `x`, avšak v místě jejího volání (v těle procedury `B`) je přístupná pouze proměnná `m`. Proto musí překladač zajistit kromě předání adresy `D` také předání ukazatele na aktivační záznam procedury `C` a při volání formální procedury zajistit potřebné vazby.

Kapitola 7

Typová kontrola

Překladač musí kontrolovat, zda zdrojový program dodržuje jak syntaktické, tak sémantické konvence zdrojového jazyka. Tato kontrola, zvaná statická kontrola (pro odlišení od dynamické kontroly během provádění cílového programu), zajišťuje detekci a ohlášení určitých druhů programátorských chyb. Příklady statických kontrol mohou být:

- *Typová kontrola.* Překladač by měl ohlásit chybu, pokud se nějaký operátor aplikuje na nekompatibilní operandy; například tehdy, jestliže se sečítá proměnná typu pole s proměnnou typu funkce.
- *Kontrola toku řízení.* Příkazy, které způsobí, že tok řízení opustí určitou konstrukci, musí mít určité místo, na které se má řízení přenést. Například příkaz `break` v C způsobí, že tok řízení opustí nejmenší obklopující příkaz `while`, `for` nebo `switch`; chyba nastane, pokud takový obklopující příkaz neexistuje.
- *Kontrola jedinečnosti.* Mohou nastat situace, kdy určitý objekt musí být deklarován právě jednou. Například v Pascalu musí být identifikátor deklarován jedinečně, návěští v příkazu case musejí být navzájem různá a prvky výčtového typu se nemohou opakovat.
- *Kontroly vztahující se ke jménům.* Někdy se určité jméno musí vyskytnout dvakrát nebo vícekrát. Například v jazyku Modula-2 musí být jméno procedury uvedeno znovu na jejím konci. Překladač musí zkontrolovat, zda je na obou místech použito totéž jméno.

V této kapitole se zaměříme na typovou kontrolu. Jak naznačují uvedené příklady, mnoho statických kontrol je rutinních a mohou se implementovat metodami z předchozí kapitoly. Některé z nich lze zahrnout do jiných činností. Například při vkládání informací do tabulky symbolů můžeme zkontrolovat, zda je jméno deklarováno jedinečně. Mnoho překladačů Pascalu kombinuje statickou kontrolu a generování intermediárního kódu se syntaktickou analýzou. Pro složitější konstrukce, jako jsou např. v jazyku Ada, může být vhodnější mít oddělený průchod provádějící typové kontroly mezi syntaktickou analýzou a generováním intermediárního kódu.

Podsystém typové kontroly ověřuje, zda typy konstrukcí odpovídají typům očekávaným z jejich kontextu. Například standardní aritmetický operátor `mod` jazyka Pascal vyžaduje celočíselné operandy, takže typová kontrola musí ověřit, zda oba operandy `mod` mají typ integer. Podobně musí typová kontrola prověřit, zda je operátor dereference aplikován na ukazatel, že indexování se provádí pouze pro pole, že uživatelem definovaná funkce se aplikuje na správný počet a typ argumentů atd.

Informace o typech, získaná během typové kontroly, může být požadována při generování kódu. Například aritmetické operátory jako je + se obvykle aplikují buď na celá nebo na reálná čísla, a musíme tedy na základě kontextu rozhodnout, o který význam operátoru + se jedná. Symbol reprezentující v různých kontextech různé operace se nazývá *přetížený*. Přetěžování může být doprovázeno implicitní konverzí typů, kdy překladač doplňuje operátor pro konverzi operandu na typ očekávaný podle kontextu.

Odlišným pojmem od přetěžování je *polymorfismus*. Polymorfické funkce a procedury mohou při každém volání pracovat s argumenty jiných typů. Např. v jazyce Pascal můžeme proceduru `writeln` považovat za polymorfickou, neboť jejími argumenty mohou být celočíselné, reálné, booleovské výrazy, znaky nebo řetězce. V závislosti na typu skutečného argumentu se teprve vybírá konkrétní algoritmus pro zobrazení hodnoty.

7.1 Typové systémy

Návrh podsystému typové kontroly jazyka je založen na informacích o syntaktických konstrukcích jazyka a pravidlech pro přiřazování typů jazykovým konstrukcím. Tato pravidla mohou mít například následující formu:

- "Jsou-li oba operandy aritmetických operací sčítání, odčítání a násobení typu integer, je výsledek typu integer."
- "Výsledek unárního operátoru & je ukazatel na objekt, ke kterému se vztahuje operand. Je-li typ operandu '...', je typ výsledku 'ukazatel na ...'."

V uvedených úsecích se implicitně předpokládá, že s každým výrazem je svázán jeho typ. Typy navíc mohou mít určitou strukturu; typ "ukazatel na ..." je vytvořen z typu "...", na který se odkazuje.

V běžných programovacích jazycích jsou k dispozici obvykle dvě skupiny datových typů: základní nebo složené. Základní typy jsou atomické typy, z hlediska programátora bez další vnitřní struktury. V Pascalu jsou například základními typy *boolean*, *char*, *integer* a *real*. Intervaly jako 1..10 a výčtové typy jako

```
(violet, indigo, blue, green, yellow, orange, red)
```

lze považovat za základní typy. Pascal programátorovi dovoluje vytvářet podle potřeby další typy ze základních a dříve definovaných složených typů; příkladem jsou pole, záznamy a množiny. Jako složené typy lze navíc chápat i ukazatele a funkce.

7.1.1 Typové výrazy

Typ jazykové konstrukce lze popsat *typovým výrazem*. Neformálně je typový výraz buď základní typ nebo je vytvořen aplikací operátoru zvaného konstruktor typu na jiné typové výrazy. Soubor základních typů a konstruktorů je dán definicí jazyka.

V této kapitole budeme používat následující definice typového výrazu:

1. Základní typ je typový výraz. Mezi základními typy jsou *boolean*, *char*, *integer* a *real*. Speciální základní typ *type_error* signalizuje chybu během typové kontroly. Konečně základní typ *void* označuje "nepřítomnost hodnoty" a dovoluje přiřadit datový typ i procedurám a příkazům.

2. Vzhledem k tomu, že typové výrazy mohou být pojmenované, je jméno typu typovým výrazem. Příklad použití jmen typů je dále v 3(c).

3. Typový konstruktor aplikovaný na typový výraz je typovým výrazem. Mezi konstruktory patří:

   a) *Konstruktor pole.* Je-li T typový výraz, pak $array(I, T)$ je typovým výrazem, jenž označuje pole prvků typu T s indexovou množinou I. Typ I je často intervalem celých čísel. Například deklarace v Pascalu

   ```
   var A: array [1..10] of integer;
   ```

   spojuje se jménem `A` typový výraz $array(1..10, integer)$.

   b) *Součin typů.* Jsou-li T_1 a T_2 typové výrazy, potom jejich kartézský součin $T_1 \times T_2$ je typovým výrazem. Předpokládáme, že $\times$ je zleva asociativní.

   c) *Záznamy.* Rozdíl mezi záznamem a součinem je ten, že složky záznamu jsou pojmenované. Typový konstruktor *record* bude aplikován na n-tici tvořenou jmény složek a typy složek. Například úsek programu v Pascalu:

   ```
   type row = record
                 address: integer;
                 lexeme: array [1..15] of char
              end;
   var table: array [1..101] of row;
   ```

   deklaruje jméno typu `row` představujícího typový výraz

   $$record((\texttt{address} \times integer) \times (\texttt{lexeme} \times array(1..15, char)))$$

   a proměnnou `table` jako pole záznamů tohoto typu.

   d) *Ukazatele.* Je-li T typový výraz, potom $pointer(T)$ je typový výraz označující typ "ukazatel na objekt typu T." Například opět v Pascalu deklarace

   ```
   var p: ↑row
   ```

   deklaruje proměnnou `p` s typem $pointer(row)$.

   e) *Funkce.* Z matematického hlediska funkce zobrazuje prvky jedné množiny, definičního oboru, do jiné množiny, oboru hodnot. Funkce v programovacích jazycích můžeme chápat jako zobrazení zdrojového typu D (domain) do cílového typu R (range). Typ takové funkce budeme zapisovat typovým výrazem $D \to R$. Například standardní funkce `mod` jazyka Pascal má zdrojový typ $int \times int$, tj. dvojici celých čísel, a cílový typ int. Za předpokladu, že $\times$ má vyšší prioritu než $\to$ a že $\to$ je asociativní zprava, tedy má `mod` typ

   $$int \times int \to int$$

   Jako další příklad vezmeme deklaraci z Pascalu

   ```
   function f(a, b: char): ↑integer; ...
   ```

   která říká, že zdrojovým typem funkce `f` je $char \times char$ a cílovým typem je $pointer(integer)$. Typ `f` je tedy označen typovým výrazem

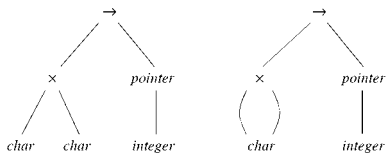
   $$char \times char \to pointer(integer)$$

Z implementačních důvodů jsou často kladena omezení na typ, jenž může funkce vracet; např. v jazyce C nelze vracet pole nebo funkce. Existují však jazyky, z nichž Lisp je nejvýraznějším příkladem, které dovolují, aby funkce vracely objekty libovolných typů, takže můžeme např. definovat funkci `g` typu

$$(integer \rightarrow integer) \rightarrow (integer \rightarrow integer),$$

Funkce `g` tedy má jako argument funkci zobrazující celé číslo na celé číslo, a tato funkce produkuje jako výsledek jinou funkci stejného typu. Zpracování takovýchto funkcí (tzv. *funkcí vyššího řádu*) je typické pro funkcionální jazyky.

Výhodnou metodou reprezentace typových výrazů je použití grafu. Během překladu definice typu můžeme pro typový výraz sestrojit strom nebo DAG, jehož vnitřními uzly budou konstruktory typu a listy budou základními typy, jmény typů a typových proměnných (viz obr. 7.1). Obdobnou reprezentací je grafový model, uvedený na obr. 5.2.



Obr. 7.1: Strom a DAG pro výraz $char \times char \rightarrow pointer(integer)$

Typový systém je soubor pravidel pro přiřazování typových výrazů různým částem programu; v této kapitole jej budeme implementovat pomocí syntaxí řízeného překladu. Různými překladači téhož jazyka mohou být implementovány různé typové systémy. Například v systému Unix jsou pro původní verzi jazyka C k dispozici dva programy s odlišnými typovými systémy. Program `lint` provádí pouze statickou kontrolu programu bez jeho překladu, ovšem na základě mnohem přísnějšího typového systému než překladač `cc`, a tím umožňuje odhalení programátorských chyb, které samy o sobě nejsou v rozporu s definicí jazyka C.

Příklad 7.1. Jako příklad implementace typové kontroly použijeme jednoduchý jazyk, ve kterém musí být typ každého identifikátoru deklarován před jeho použitím. Jazyk má následující gramatiku:

$$\begin{array}{lcl} P & \rightarrow & D \ ; \ E \\ D & \rightarrow & D \ ; \ D \mid \textbf{id} \ : \ T \\ T & \rightarrow & \textbf{char} \mid \textbf{integer} \mid \textbf{array} \ [\ \textbf{num} \ ] \ \textbf{of} \ T \mid \uparrow T \\ E & \rightarrow & \textbf{literal} \mid \textbf{num} \mid \textbf{id} \mid E \ \textbf{mod} \ E \mid E \ [\ E \ ] \mid E\hat{} \end{array}$$

Základními typy jazyka jsou *char* a *integer*, typ *type_error* se používá pouze pro signalizaci typové chyby. Pro jednoduchost předpokládáme, že index pole začíná vždy od hodnoty 1. Překladové schéma na obr. 7.2 popisuje budování typových výrazů, deklaraci proměnných a typovou

kontrolu výrazů. Po vhodné modifikaci gramatiky můžeme toto schéma použít jak pro překlad shora dolů, tak i pro překlad zdola nahoru.

$P \to D\ ;\ E$	
$D \to D\ \ D$	
$D \to \mathbf{id}\ :\ T$	{ $addtype(\mathbf{id}.entry, T.type)$ }
$T \to \mathbf{char}$	{ $T.type := char$ }
$T \to \mathbf{integer}$	{ $T.type := integer$ }
$T \to \hat{}\,T_1$	{ $T.type := pointer(T_1.type)$ }
$T \to \mathbf{array}\ [\ \mathbf{num}\ ]\ \mathbf{of}\ T_1$	{ $T.type := array(1..\mathbf{num}.val, T_1.type)$ }
$E \to \mathbf{literal}$	{ $E.type := char$ }
$E \to \mathbf{num}$	{ $E.type := integer$ }
$E \to \mathbf{id}$	{ $E.type := lookup(\mathbf{id}.entry)$ }
$E \to E_1\ \mathbf{mod}\ E_2$	{ $E.type :=$ **if** $E_1.type = integer$ **and** $E_2.type = integer$ **then** $integer$ **else** $type\_error$ }
$E \to E_1\ [\ E_2\ ]$	{ $E.type :=$ **if** $E_2.type = integer$ **and** $E_1.type = array(s,t)$ **then** t **else** $type\_error$ }
$E \to E_1\ \hat{}$	{ $E.type :=$ **if** $E_1.type = pointer(t)$ **then** t **else** $type\_error$ }

Obr. 7.2: Překladové schéma pro typovou kontrolu deklarací a výrazů

V uvedeném překladovém schematu akce $addtype(\mathbf{id}.entry, T.type)$ do položky tabulky symbolů specifikované syntetizovaným atributem $entry$ uloží typ identifikátoru **id** z deklarace. Syntetizovaný atribut *type* nonterminálu E udává typ odpovídajícího výrazu. Pro zjištění typu, který je svázán s položkou tabulky symbolů e, používáme funkce $lookup(e)$

Při kontrole operátoru **mod** ve výrazu požadujeme, aby oba operandy měly typ *integer*. V odkazu na prvek pole $E_1[E_2]$ musí mít indexový výraz E_2 typ *integer*; typ výsledku t je potom dán typem prvku pole, který získáme z konstruktoru $array(s,t)$. Pro výraz $E\hat{}$ požadujeme, aby jeho operandem byl ukazatel; typ t celého výrazu opět získáme z konstruktoru $pointer(t)$. Toto překladové schéma můžeme podobným způsobem rozšířit o další typy a operátory.

7.1.2 Statická a dynamická kontrola typů

Kontrole prováděné překladačem říkáme statická, zatímco kontroly prováděné při běhu programu se nazývají dynamické. V principu je možné všechny kontroly provádět až dynamicky, pokud cílový kód ponese s hodnotou prvku zároveň i jeho typ. Z hlediska efektivity spolehlivosti programu je však vhodnější provádět v době překladu co největší počet kontrol.

Spolehlivý typový systém (sound type system) vylučuje potřebu dynamické kontroly typových chyb, neboť dovoluje staticky zajistit, že takové chyby nemohou za běhu cílového programu nastat. To znamená, že pokud nějaký spolehlivý typový systém přiřadí části programu jiný typ než *type_error*, potom při běhu cílového kódu vygenerovaného z této části programu nemůže

nastat typová chyba. Jazyk je *přísně typovaný* (strongly typed), pokud jeho překladač může zaručit, že program, který příjme, se bude provádět bez typových chyb.

V praxi se však mohou některé kontroly provádět výlučně dynamicky. Například pokud nejprve deklarujeme

```
table: array [0..255] of char;
i: integer;
```

a potom počítáme `table[i]`, nemůže překladač obecně zaručit, že při provádění programu bude hodnota `i` ležet v intervalu 0 až 255. Pouze v některých programech lze pomocí technik analýzy toku dat zda je `i` v určitých mezích. Žádná technika to však nemůže provést správně ve všech případech.

7.1.3 Zotavení po chybě při typové kontrole

Vzhledem k tomu, že typová kontrola má schopnost zachycovat chyby v programech, je pro podsystém typové kontroly důležité, aby při výskytu chyby provedl něco rozumného. Nejdříve ze všeho musí překladač ohlásit podstatu a pozici chyby. Při typové kontrole vyžadujeme, aby došlo k zotavení a mohl se kontrolovat i zbytek programu. Zotavení musí být zabudováno již od počátku do typového systému.

Zavedení zpracování chyb může vést k typovému systému, který jde mnohem dále než systém nutný pouze ke specifikaci správných programů. Například nastala-li již chyba, nemůžeme znát typ nesprávně vytvořeného úseku programu. Zacházení s neúplnými informacemi vyžaduje techniky podobné metodám potřebným v jazycích, které nevyžadují deklaraci identifikátorů před jejich použitím. K zajištění konzistentního použití nedeklarovaných nebo zjevně nesprávně deklarovaných identifikátorů lze použít typových proměnných, představujících neznámý datový typ.

7.2 Ekvivalence typových výrazů

Během typové kontroly často vyžadujeme, aby dva datové typy byly ekvivalentní. Pojem ekvivalence datových typů však prozatím nebyl přesně definován; není například zřejmé, zda dva různě pojmenované typy se shodnou vnitřní strukturou jsou či nejsou ekvivalentní. V programovacích jazycích se setkáváme v podstatě se dvěma základními přístupy. *Ekvivalence podle jmen* považuje každý pojmenovaný typ za jedinečný, odlišný od všech ostatních pojmenovaných či nepojmenovaných typů; dva typové výrazy jsou ekvivalentní podle jména právě tehdy, jsou-li identické. Při zjišťování *ekvivalence podle struktury* nejprve nahradíme všechna jména odpovídajícími typovými výrazy; dva typové výrazy považujeme za ekvivalentní, jestliže po tomto nahrazení mají oba výrazy stejnou vnitřní strukturu.

Příklad 7.2. Uvažujme následující úsek deklarací v jazyce Pascal:

```
type link = ↑ cell;
var  next : link;
     last : link;
     p    : ↑ cell;
     q, r : ↑ cell;
```

Identifikátor `link` je zde jménem typu `↑cell`. Zajímá nás, zda typy proměnných `next`, `last`, `p`, `q` a `r` jsou či nejsou identické. Proměnným `next` a `last` je přiřazen typový výraz `link`,

ostatním proměnným výraz *pointer*(`cell`). Je-li implementována ekvivalence podle jmen, mají proměnné `next` a `last` stejný typ, neboť jim odpovídající typové výrazy jsou identické. Podobně proměnné `p`, `q` a `r` mají stejný typ, ovšem odlišný od typu proměnné `next`. Uvažujeme-li však strukturální ekvivalenci, jsou typy všech proměnných stejné, neboť po nahrazení jména typu `link` odpovídajícím typovým výrazem *pointer*(`cell`) z jeho definice dostaneme pro všechny proměnné výrazy se stejnou vnitřní strukturou.

V některých implementacích se k ekvivalenci podle jmen přistupuje poněkud odlišným způsobem. Každému výskytu nepojmenovaného typu se přiřadí implicitní jméno, které tento výskyt odlišuje od všech ostatních výskytů téhož nepojmenovaného typu. V našem příkladě by tedy proměnná `p` mohla mít jiný typ než proměnné `q` a `r`. Tento přístup podstatně zjednodušuje implementaci ekvivalence typů, neboť pokud například reprezentujeme typy proměnných pomocí ukazatelů na datové struktury popisující konkrétní výskyt typu, můžeme za ekvivalentní datové typy považovat ty, které jsou reprezentovány stejnými ukazateli.

Pro testování strukturální ekvivalence můžeme použít algoritmu obdobnému tomu, který je uveden na obr. 7.3. Funkce $sequiv(s,t)$ vrátí hodnotu true, pokud jsou typové výrazy s a t strukturálně ekvivalentní, a hodnotu false v opačném případě.

```
function sequiv(s,t): boolean;
begin
    if s a t jsou stejné základní typy then
        return true
    else if s = array(s1, s2) and t = array(t1, t2) then
        return sequiv(s1, t1) and sequiv(s2, t2)
    else if s = s1 × s2 and t = t1 × t2 then
        return sequiv(s1, t1) and sequiv(s2, t2)
    else if s = pointer(s1) and t = pointer(t1) then
        return sequiv(s1, t1)
    else if s = s1 → s2 and t = t1 → t2 then
        return sequiv(s1, t1) and sequiv(s2, t2)
    else
        return false
end
```

Obr. 7.3: Testování strukturální ekvivalence typových výrazů

V některých implementacích překladačů se pro kódování typových výrazů používají i jiné datové struktury než graf. Datový typ může být zakódován jako posloupnost bitů tvořená kódem základního datového typu, ke kterému se přidávají kódy typových konstruktorů v pořadí jejich aplikace. Výhodou tohoto přístupu je úsporná reprezentace a jednodušší testování strukturální ekvivalence, neboť dva strukturálně odlišné datové typy nemohou mít stejnou bitovou reprezentaci. Naopak nevýhodou je omezení přípustné složitosti datových typů, které může programátor používat, obvykle délkou slova procesoru.

Při implementaci ekvivalence podle struktury musíme uvažovat i možnost rekurzivní definice typu — např. datový typ záznam může v sobě obsahovat ukazatel na jiný záznam téhož typu. Je-li datový typ v překladači reprezentován grafem, obdržíme po nahrazení jmen typů odpovídajícími grafy cyklický graf, a musíme tedy zajistit, aby se algoritmus zjišťující strukturální

ekvivalenci typů choval korektně i v tomto případě.

7.3 Typové konverze

Uvažujme výraz `x+i`, kde `x` je typu real a `i` typu integer. Vzhledem k tomu, že reprezentace obou typů v počítači je odlišná a že počítač pro operace nad celými a reálnými čísly používá jiné instrukce, musí překladač nejprve zajistit konverzi jednoho z operandů na společný datový typ. To, zda tato konverze je implicitní nebo musí být explicitně zapsána programátorem, závisí na definici jazyka. Podobně musí být definována pravidla pro přiřazování hodnot do proměnných různých typů. Například v jazyce Pascal se při přiřazení celočíselné hodnoty do reálné proměnné provede implicitní konverze přiřazované hodnoty na typ real, ovšem při přiřazení reálného výrazu do celočíselné proměnné musí programátor explicitně definovat požadovanou konverzi voláním funkce *trunc* nebo *round*.

Implicitní konverze jednoho datového typu na druhý (často také zvané *koerce*) provádí překladač automaticky. Obvykle jsou tyto konverze omezeny na případy, kdy nemůže dojít ke ztrátě informace, např. konverze celého čísla na reálné. *Explicitní konverze* datových typů požaduje programátor obvykle ve formě volání určitých standardních funkcí nebo pomocí operátorů konverze. Například v jazyce Pascal funkce `ord` převádí znaky na celá čísla a funkce `chr` naopak celá čísla na znaky, zatímco v jazyce C se tato konverze provádí implicitně. V jazyce Ada jsou všechny konverze explicitní, čímž se zajistí skutečně důsledná typová kontrola a odhalení případných chyb v důsledku nesprávně zapsaných výrazů.

7.4 Přetěžování funkcí a operátorů

Přetížený symbol je takový, který má různý význam v závislosti na kontextu, ve kterém je použit. Ve výrazech je například přetížen symbol `+`, protože ve výrazu `A + B` může mít různý význam v závislosti na typech operandů `A` a `B`. V jazyce Ada jsou přetížené závorky (); výraz `A(I)` může být odkaz na `I`-tý prvek pole `A`, volání funkce `A` s parametrem `I` nebo explicitní konverze výrazu `I` na typ `A`.

Přetížení se nazývá *vyřešené*, pokud se nám podaří nalézt jednoznačný význam pro určitý výskyt přetíženého symbolu. U běžných programovacích jazyků, kde přetížení nastává pouze u standardních operátorů, není obvykle nalezení jednoznačného významu obtížné. V jazycích jako je Ada nebo C++ však může docházet k velmi komplikovaným situacím, kdy podvýraz nějakého výrazu může mít množinu možných typů a kdy pro vyřešení přetížení potřebujeme znát širší kontext.

Příklad 7.3. V jazyce Ada je jednou ze standardních interpretací operátoru `*` násobení dvou celých čísel. Tento operátor můžeme přetížit deklaracemi jeho dalších významů, např.

```
function "*" ( i, j : integer ) return complex;
function "*" ( x, y : complex ) return complex;
```

Po uvedených deklaracích množina možných typů operátoru `*` zahrnuje

$$\begin{aligned} integer \times integer &\rightarrow integer \\ integer \times integer &\rightarrow complex \\ complex \times complex &\rightarrow complex \end{aligned}$$

Za předpokladu, že konstanty `2`, `3` a `5` jsou pouze typu integer, může mít podvýraz `3*5` typ *integer* nebo *complex*, v závislosti na kontextu. Je-li úplný výraz `2*(3*5)`, musí být `3*5` typu *integer*, neboť operátor `*` může mít buď oba operandy typu *integer* nebo oba operandy typu *complex*. V jazyce C++ může být tato situace ještě komplikovaná tím, že programátor může definovat funkce pro implicitní konverzi typu *integer* na *complex*; tehdy by se po implicitní konverzi hodnoty `2` na typ *complex* mohl celý výraz vyhodnotit jako výraz typu *complex* a výsledný typ by byl opět nejednoznačný. Zpracování přetížených symbolů je obecně značně složitý problém; některé algoritmy, které se pro řešení přetížení používají, je možno nalézt v [3].

7.5 Polymorfické procedury a funkce

Obyčejné procedury a funkce umožňují provedení svého těla pouze s parametry pevných typů, které jsou uvedeny v deklaraci podprogramu nebo jsou dány implicitními konvencemi. Typy parametrů polymorfických procedur a funkcí naopak mohou být při každém volání podprogramu odlišné. V běžných programovacích jazycích se s polymorfismem setkáváme například u standardních operátorů pro indexování polí, volání funkcí a manipulaci s ukazateli. Například v jazyce C je-li ve výrazu `&x` operand `x` typu "...," je výsledek typu "ukazatel na ...." Za symbol "..." můžeme dosadit libovolný typ, takže operátor `&` je v jazyce C polymorfický.

Polymorfické procedury a funkce jsou z hlediska programátorského velmi efektivním prostředkem pro vyjadřování obecných algoritmů. Například potřebujeme-li v Pascalu funkci pro zjištění délky seznamu celých nebo reálných čísel, musíme stejný algoritmus zapsat dvakrát, přičemž lišit se budou pouze deklarace typu parametru funkce. Výhodnější by bylo použít polymorfické funkce, která by umožňovala výpočet délky seznamu prvků libovolného typu (který ve vlastním výpočtu nehraje žádnou roli).

Abychom mohli specifikovat typy polymorfických funkcí, musíme v typových výrazech použít *typové proměnné*. Typové proměnné budeme označovat písmeny řecké abecedy $\alpha, \beta, \ldots$ a budou reprezentovat vždy konkrétní neznámý typ. Například operátor `&` jazyka C bude mít typ

$$\alpha \rightarrow pointer(\alpha)$$

.

V programovacích jazycích, které nevyžadují explicitní definice typů proměnných a funkcí (například ve funkcionálních jazycích jako je jazyk ML), musíme typy jednotlivých jazykových konstrukcí určovat na základě kontextu. Tento proces se nazývá *inference typů*. Například ve funkci

```
fun length(lptr) =
        if null(lptr) then 0
        else length(tl(lptr)) + 1;
```

se na druhém řádku volá standardní funkce `null`, která je typu $list(\alpha) \rightarrow boolean$, kde *list* je konstruktor seznamu. Odtud je zřejmé, že `lptr` musí být typu $list(\alpha)$, kde α je nějaký (libovolný) typ. Jako výsledek se vrací celočíselná konstanta `0`, proto je výsledek funkce `length` typu *integer*. Funkce `length` má tedy typ $list(\alpha) \rightarrow integer$. Na třetím řádku můžeme už jenom provést na základě znalosti typu funkcí `tl` a `length` kontrolu, zda je uvedený výraz typově správný.

Inference typů lze využít i v překladačích klasických jazyků pro doplňování chybějících informací v době překladu. Například v jazyce C můžeme z volání funkce odvodit typy jejích operandů a výsledku a později, v okamžiku její definice, zkontrolovat, zda je tato definice konzistentní s předchozími voláními.

7.5.1 Unifikace typových výrazů

Při inferenci typů je základním problémem nalezení společné instance dvou typových výrazů s typovými proměnnými — jejich *unifikace*. Unifikaci můžeme definovat pomocí funkce S zvané *substituce*, která proměnným přiřazuje výrazy. Zápis $S(e)$ představuje výraz získaný tak, že všechny proměnné α obsažené v e nahradíme hodnotou $S(\alpha)$. Potom S je unifikátorem pro e a f, právě když $S(e) = S(f)$.

Máme-li dva typové výrazy e a f, hledáme takovou nejobecnější substituci proměnných v nich obsažených, aby po této substituci oba výrazy byly ekvivalentní. Výsledkem unifikace může být buď tato substituce, nebo zjištění, že společná instance výrazů neexistuje. Speciálním případem unifikace je testování ekvivalence dvou typových výrazů; pokud výrazy e a f neobsahují proměnné, je možné je unifikovat právě tehdy, jestliže jsou ekvivalentní.

Příklad 7.4. Uvažujme následující dva typové výrazy:

$$\begin{aligned} ((\alpha_1 \to \alpha_2) \times list(\alpha_3)) &\to list(\alpha_2) \\ ((\alpha_3 \to \alpha_4) \times list(\alpha_3)) &\to \alpha_5 \end{aligned}$$

Pro tyto výrazy můžeme najít substituci S takovou, že $S(\alpha_1) = S(\alpha_3) = \alpha_3, S(\alpha_2) = S(\alpha_4) = \alpha_2$, $S(\alpha_5) = list(\alpha_2)$, která zobrazuje e a f na výraz

$$S(e) = S(f) = ((\alpha_3 \to \alpha_2) \times list(\alpha_3)) \to list(\alpha_2)$$

■

Jeden z možných unifikačních algoritmů je uveden v [3]; podobné algoritmy se používají při vyhodnocování programů v logických programovacích jazycích (např. Prolog) nebo obecně při řešení problémů z oblasti umělé inteligence.

Kapitola 8

Generování intermediárního kódu

V analyticko-syntetickém modelu překladu převádí přední část překladače zdrojový program do intermediární reprezentace, ze které dále zadní část překladače generuje cílový kód. Je samozřejmě možné — a také se tak často postupuje — přeložit zdrojový program přímo do cílového jazyka. Překlad využívající nějakého strojově nezávislého mezikódu má však své výhody:

1. Zjednodušuje se přepracování překladače pro jiný cílový jazyk (retargeting). Stačí vytvořit pouze novou koncovou část.

2. Mezikód lze optimalizovat s využitím metod strojově nezávislé optimalizace.

V této kapitole si ukážeme použití metod syntaxí řízeného překladu pro překlad základních programových konstrukcí jako jsou deklarace, přiřazení a řídicí příkazy do intermediárního kódu. Většina uvedených metod se dá použít během překladu zdola nahoru nebo shora dolů, takže generování intermediárního kódu se dá podle potřeby začlenit do syntaktické analýzy.

8.1 Intermediární jazyky

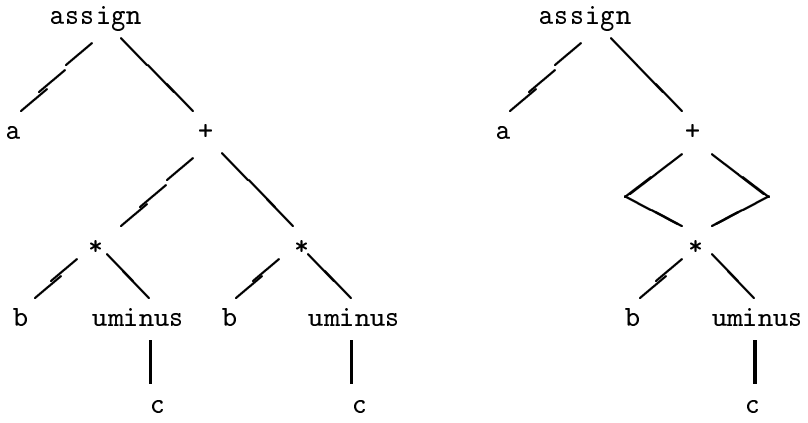
Jako intermediární reprezentace programu se používají nejčastěji stromy (případně obecné grafy) a zásobníkový nebo tříadresový kód. Výběr mezikódu je často dán požadavky na efektivitu jeho dalšího zpracování. Například pro rozsáhlejší optimalizace je výhodnější použít tříadresového kódu místo zásobníkového. Naopak zásobníkový kód může být výhodnější v překladačích generujících kód pro počítače se zásobníkovou architekturou.

8.1.1 Grafová reprezentace

Za přirozenou grafovou reprezentaci programu můžeme považovat přímo syntaktický strom nebo DAG. Na obr. 8.2 je znázorněn strom a DAG pro přiřazovací příkaz `a := b * -c + b * -c`.

Pomocí grafu se často v překladači reprezentují deklarace, které se neobjevují přímo v mezikódu (viz odstavec 5.1), a výrazy, jejichž kód se někdy může v mezikódu vyskytovat na jiném místě, než kde byl výraz uveden ve zdrojovém programu. Například pro příkaz cyklu `for` jazyka C

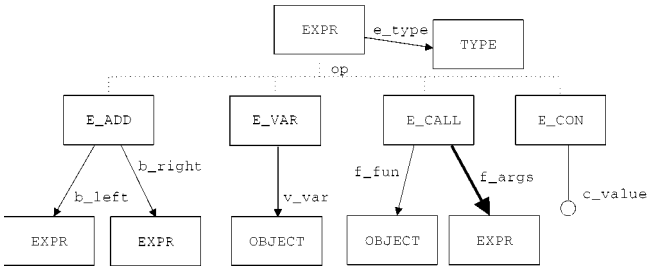
```
for(p=first; p; p=p->next) print(p);
```



(a) Syntaktický strom (b) DAG

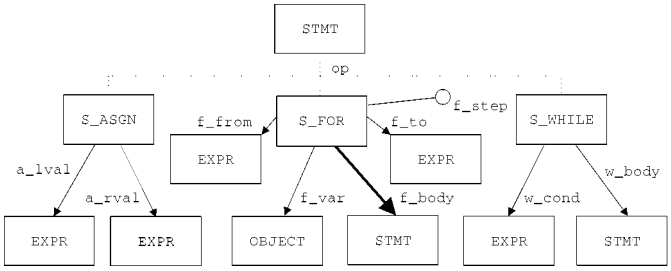
Obr. 8.1: Grafická reprezentace výrazu `a := b * -c + b * -c`

se kód pro vyhodnocení výrazu `p=p->next` může vygenerovat až za konec těla cyklu a je tedy nutné nějakým způsobem uchovat výraz až do okamžiku, kdy bude tělo cyklu zpracováno. Překlad výrazu může probíhat dvoufázově: nejprve se vytvoří jeho grafová reprezentace, a pak se tento graf ve vhodném okamžiku převede například do tříadresového kódu.



Obr. 8.2: E–R model výrazu

Pro obecnou reprezentaci jak výrazů, tak i příkazů programu můžeme rovněž použít E–R modelu z článku 5.1. Část sémantického grafu pro typické výrazy je uvedena na obr. 8.2, na obr. 8.3 je znázorněna struktura některých příkazů jazyka Pascal.



Obr. 8.3: E–R model příkazu

```
VAR  b          ; ... (b)
VAR  c          ; ... (b) (c)
INV             ; ... (b) (-c)
MUL             ; ... (b * -c)
VAR  b          ; ... (b * -c) (b)
VAR  c          ; ... (b * -c) (b) (c)
INV             ; ... (b * -c) (b) (-c)
MUL             ; ... (b * -c) (b * -c)
ADD             ; ... (b * -c + b * -c)
ASG  a          ; ...
```

Obr. 8.4: Zásobníkový kód pro výraz `a := b * -c + b * -c`

8.1.2 Zásobníkový kód

Postfixová notace, ze které vychází zásobníkový kód, představuje linearizovaný zápis syntaktického stromu; je to seznam uzlů, ve kterém je uzel stromu uveden vždy bezprostředně za svými přímými následníky. Postfixový zápis syntaktického stromu z obr. 8.2(a) je

```
a b c uminus * b c uminus * + assign
```

Postfixová notace neobsahuje explicitně hrany syntaktického stromu. Ty se dají zpětně odvodit z pořadí uzlů a z počtu jejich operandů.

Zásobníkový kód je tvořen posloupností příkazů, které obecně definují posloupnost akcí nad zásobníkem. Každá z těchto akcí představuje buď vložení hodnoty proměnné nebo konstanty na vrchol zásobníku, provedení určité operace nebo uložení hodnoty ze zásobníku do proměnné. Operandy a výsledky operací jsou obvykle uloženy na zásobníku. Příkaz `a := b * -c + b * -c` můžeme v zásobníkovém kódu zapsat například tak, jak ukazuje obr. 8.4. V poznámce je u každé instrukce zásobníkového kódu uveden obsah zásobníku po jejím provedení. Pořadí operandů a operátorů je stejné jako v postfixové notaci, ovšem postfixová notace operandy od operátorů formálně nerozlišuje.

8.1.3 Tříadresový kód

Tříadresový kód je posloupnost příkazů, které mají obecně tvar

`x := y` *op* `z`

kde `x`, `y` a `z` jsou jména, konstanty nebo překladačem vytvořené dočasné objekty; *op* představuje libovolný operátor, např. některý z aritmetických nebo logických operátorů. V operandech nemohou být žádné další výrazy, příkaz obsahuje vždy jen jediný operátor. Proto musí být složitější výrazy rozloženy na své nejjednodušší složky s použitím dočasných proměnných vytvořených překladačem. Zde je vidět zásadní rozdíl mezi zásobníkovým a tříadresovým kódem. Zásobníkový kód se odkazuje na operandy implicitně, na základě jejich pozice, zatímco tříadresový kód všechny operandy pojmenovává. Tím se značně zjednodušují optimalizace tříadresového mezikódu, při nichž se mohou jednotlivé příkazy navzájem libovolně přesouvat.

Pojmenování kódu vychází z toho, že každý příkaz obvykle obsahuje tři adresy, dvě pro operandy a jednu pro výsledek. Při implementaci mohou tyto adresy znamenat například ukazatele do tabulky symbolů na příslušné objekty.

Tříadresový kód je linearizovanou reprezentací syntaktického stromu nebo DAG, ve které jména generovaná překladačem odpovídají vnitřním uzlům grafu. Syntaktický strom a DAG z obr. 8.1 jsou na obr. 8.5 zapsány v tříadresovém kódu. Jména proměnných se mohou v zápisu používat přímo, proto zde nejsou žádné příkazy, které by reprezentovaly listy původního grafu.

```
t1 := - c
t2 := b * t1
t3 := - c
t4 := b * t3
t5 := t2 + t4
a  := t5
```

(a) Kód pro syntaktický strom

```
t1 := - c
t2 := b * t1
t5 := t2 + t2
a  := t5
```

(b) Kód pro DAG

Obr. 8.5: Tříadresový kód pro strom a DAG z obr. 8.1

Typy příkazů tříadresového kódu

Příkazy tříadresového kódu jsou podobné příkazům jazyka asembleru. Mohou být označeny symbolickým návěštím, které se využívá v příkazech pro změnu toku řízení. Transformace symbolického jména na index příkazu v jeho vnitřní reprezentaci se provádí buď v samostatném průchodu, nebo metodou backpatching, kterou se budeme zabývat v odstavci 8.7.

V dalším textu budeme používat následující nejčastější tříadresové příkazy:

- Přiřazovací příkazy ve tvaru `x := y` *op* `z`, kde *op* je binární aritmetický nebo logický operátor.
- Přiřazovací příkazy ve tvaru `x :=` *op* `y`, kde *op* je unární operátor (unární minus, logická negace, operátory pro konverzi datových typů apod.).
- Kopírovací příkazy ve tvaru `x := y`.

- Nepodmíněný skok `goto L`.
- Podmíněné skoky ve tvaru `if x` *relop* `y goto L`, které se provedou tehdy, je-li splněna relace *op* mezi hodnotami `x` a `y`.
- Příkazy `param x` a `call p,`n pro volání procedury a `return y` s volitelnou hodnotou `y` reprezentující návratovou hodnotu. Typická posloupnost těchto příkazů pro volání procedury `p(`$\mathtt{x}_1$`, `$\mathtt{x}_2$`, ..., `$\mathtt{x}_n$`)` je

```
param x1
param x2
   ...
param xn
call p,n
```

  kde n je počet skutečných parametrů předávaných proceduře.
- Přiřazení s indexováním ve tvaru `x:=y[i]` nebo `x[i]:=y`.
- Přiřazení adres a nepřímý přístup přes ukazatel ve tvaru `x:=&y`, `x:=*y` a `*x:=y`. První z těchto příkazů uloží do `x` adresu objektu `y`, další uloží do `x` hodnotu, jejíž adresa je v proměnné `y` a poslední uloží na adresu, která je v proměnné `x` hodnotu `y`.

Výběr operátorů je velmi důležitou součástí návrhu intermediárního kódu. Soubor operátorů musí být dostatečně bohatý, aby se jím daly vyjádřit všechny operace zdrojového jazyka. Menší počet operátorů zjednodušuje implementaci generátoru kódu, avšak vede k podstatně delším úsekům mezikódu, které se dále musí optimalizovat.

Implementace tříadresových příkazů

Tříadresové příkazy jsou abstraktní formou intermediárního kódu. V překladači se tyto příkazy mohou implementovat jako záznamy s položkami pro operátor a operandy. Obvykle se pro ně používá jedna z následujících reprezentací:

- *Čtveřice (quadruples).* Čtveřice je struktura se čtyřmi položkami, které označíme *op*, $arg1$, $arg2$ a *result*. Položka *op* obsahuje kód operátoru, $arg1$ a $arg2$ operandy a *result* výsledek. Některé příkazy nemusejí využívat všechny položky, např. unární operátory nevyužívají $arg2$.
- *Trojice (triples).* V této reprezentaci datová struktura reprezentující příkaz neobsahuje položku pro výsledek. Výsledek je v operandech dalších příkazů reprezentován číslem příslušné trojice.
- *Nepřímé trojice (indirect triples).* Nevýhodou předchozí reprezentace je, že se jednotlivé trojice nemohou jednoduše přesouvat nebo rušit, například během optimalizace kódu. Proto se může využít ještě dalšího pomocného pole, které obsahuje pouze ukazatele na jednotlivé trojice a které definuje jejich skutečné pořadí.

	op	*arg*1	*arg*2	*result*
(0)	`uminus`	`c`		`t1`
(1)	`*`	`b`	`t1`	`t2`
(2)	`uminus`	`c`		`t3`
(3)	`*`	`b`	`t3`	`t4`
(4)	`+`	`t2`	`t4`	`t5`
(5)	`assign`	`t5`		`a`

(a) Čtveřice

	op	*arg*1	*arg*2
(0)	`uminus`	`c`	
(1)	`*`	`b`	(0)
(2)	`uminus`	`c`	
(3)	`*`	`b`	(2)
(4)	`+`	(0)	(3)
(5)	`assign`	`a`	(4)

(b) Trojice

Obr. 8.6: Reprezentace tříadresových příkazů trojicemi a čtveřicemi

	příkaz
(0)	(14)
(1)	(15)
(2)	(16)
(3)	(17)
(4)	(18)
(5)	(19)

	op	*arg*1	*arg*2
(14)	`uminus`	`c`	
(15)	`*`	`b`	(14)
(16)	`uminus`	`c`	
(17)	`*`	`b`	(16)
(18)	`+`	(15)	(17)
(19)	`assign`	`a`	(18)

Obr. 8.7: Reprezentace tříadresových příkazů nepřímými trojicemi

8.2 Deklarace

8.2.1 Deklarace proměnných

Při zpracování deklarací je základním úkolem překladače vytváření tabulky symbolů. S tím obvykle souvisí i shromažďování informací o datových typech a velikostech jejich reprezentace a přidělování adres proměnným a složkám záznamů, což již není nezávislé na generovaném cílovém kódu. Překladač musí uvažovat nejen konkrétní velikosti objektů různých typů, ale také další požadavky definované architekturou cílového počítače, například zarovnávání. Při vytváření přemístitelného kódu je adresa objektu definována vždy dvěma údaji: příslušností do určité samostatně adresované skupiny objektů (např. globální a lokální proměnné nebo procedury, konstanty, externí proměnné) a relativní adresou vzhledem k začátku této skupiny. Pro každou takovou skupinu můžeme udržovat samostatný čítač adres, který se při deklaraci objektu patřícího do příslušné skupiny vždy zvýší o velikost datového typu objektu.

Příklad 8.1. Překladové schéma na obr. 8.8 popisuje překlad posloupnosti deklarací ve tvaru **id**: T. Současná relativní adresa pro deklarované proměnné je uložena v proměnné *offset* a je na začátku nastavena na nulu. Procedura $enter(name, type, offset)$ vytvoří novou položku tabulky symbolů pro proměnnou *name* typu *type*, které bude přidělena relativní adresa *offset*. Syntetizované atributy *type* a *width* nonterminálu T představují typ a jeho velikost. Typ je reprezentován grafem, jehož uzly se vytvářejí ze základních typů *integer* a *real* funkcemi *array* a *pointer*. Předpokládáme, že hodnoty typu *integer* a ukazatele vyžadují 4 slabiky a hodnoty typu *real* 8 slabik paměti. ■

Inicializace proměnné *offset* ve schématu na obr. 8.8 má tvar

$$P \to \{offset := 0\}D \tag{8.1}$$

$P \rightarrow$ D	$\{\ offset := 0\ \}$
$D \rightarrow D\ ;\ D$	
$D \rightarrow$ **id**: T	$\{\ enter(\mathbf{id}.name, T.type, offset);$ $offset := offset + T.width\ \}$
$T \rightarrow$ **integer**	$\{\ T.type := integer;$ $T.width := 4\ \}$
$T \rightarrow$ **real**	$\{\ T.type := real;$ $T.width := 8\ \}$
$T \rightarrow$ **array** [**num**] **of** T_1	$\{\ T.type := array(\mathbf{num}.val, T_1.type);$ $t.width := \mathbf{num}.val \times T_1.width\ \}$
$T \rightarrow \uparrow T_1$	$\{\ T.type := pointer(T_1.type);$ $T.width := 4\ \}$

Obr. 8.8: Výpočet typů a relativních adres v deklaracích

Pomocí nonterminálů generujících prázdný řetězec (markerů) můžeme taková pravidla přepsat do tvaru, kdy jsou všechny akce na konci pravidel. Např. s využitím nonterminálu M přepíšeme (8.1) na

$$\begin{array}{lcll} P & \rightarrow & M\ D & \\ M & \rightarrow & \epsilon & \{offset := 0\} \end{array}$$

Přesun akcí na konec pravidel umožňuje provádět překlad zdola nahoru, kdy se sémantické akce provádějí během redukce pravé strany pravidla.

8.2.2 Deklarace v jazycích s blokovou strukturou

V jazycích jako je Pascal nebo C mohou být jednotlivé bloky deklarací do sebe zanořené. Na začátku zanořeného bloku deklarací se dočasně potlačí zpracování deklarací nadřazeného bloku, ve kterém se pokračuje až po uzavření zanořeného bloku. V kapitole 5 jsme pro tento účel zavedli operace `tabopen` a `tabclose`, které otevíraly a zavíraly jednu úroveň blokově strukturované tabulky symbolů. Následující příklad ukazuje, jak se bloková struktura jazyka odrazí ve zpracování deklarací.

Příklad 8.2. Jazyk deklarací z příkladu 8.1 rozšíříme o pravidlo

$$D \rightarrow \mathbf{proc}\ \mathbf{id}\ ;\ D\ ;\ S$$

umožňující deklarovat proceduru s lokálními deklaracemi. Na začátku vnořeného bloku deklarací musíme nejprve uschovat současnou hodnotu čítače *offset* (použijeme k tomu atributu markeru M), nastavit tento čítač na nulu a otevřít novou úroveň tabulky symbolů. Po ukončení těla bloku naopak uzavřeme současnou úroveň a obnovíme původní hodnotu čítače viz obr. 8.9. ■

Jazyk C sice neumožňuje do sebe vkládat deklarace funkcí, avšak dovoluje do sebe zanořovat bloky deklarací proměnných. Všechny proměnné v zanořených blocích spolu sdílejí společnou oblast paměti; jejich relativní adresy se počítají od začátku oblasti lokálních proměnných funkce, v níž jsou deklarované. To znamená, že při vstupu do bloku musíme nechat hodnotu *offset* beze

$$
\begin{array}{ll}
D \rightarrow \textbf{proc id}; M\ D\ ;\ S & \{\ tabclose();\\
 & \quad offset := M.offset\ \}\\
M \rightarrow & \{\ M.offset = offset;\\
 & \quad offset = 0;\\
 & \quad tabopen()\ \}
\end{array}
$$

Obr. 8.9: Zpracování zanořených deklarací

změny. Při výstupu z bloku můžeme obnovit původní hodnotu a případně tak využít uvolněné paměti pro další proměnné.

Podobným způsobem jako lokální proměnné se zpracovávají také deklarace položek záznamů. Na začátku deklarace záznamu se rovněž otevře nová úroveň tabulky symbolů a vynuluje se čítač adres, při ukončení záznamu se však musí deklarace položek, které se při zpracování těla záznamu uložily do tabulky, uchovat jako atribut datového typu záznam. Tyto deklarace se totiž budou dále používat při odkazech na složky záznamu ve výrazech. Nejjednodušší implementace úschovy položek záznamu je při použití tabulky symbolů strukturované jako zásobník stromů — uschová se ukazatel na kořen stromu pro poslední otevřenou úroveň tabulky. Deklarace položek záznamů s variantami (v Pascalu) nebo unií (v jazyce C) probíhá obdobně, pouze se po deklaraci nové složky nezvyšuje čítač adres a tím se všem odpovídajícím položkám přidělí totéž místo. Pouze je třeba sledovat délku největší položky, která se stane délkou celého datového typu.

S deklaracemi položek záznamů také souvisí zpracování příkazu **with** jazyka Pascal. Tento příkaz zpřístupní současně všechny složky určitého záznamu. Příkaz **with** se dá implementovat tak, že znovu otevřeme novou úroveň deklarací a vložíme do ní část tabulky symbolů, kterou jsme uschovali při dokončení deklarace záznamu. Po ukončení platnosti příkazu **with** opět tuto úroveň zrušíme.

8.3 Přiřazovací příkazy a výrazy

Pro překlad celočíselných aritmetických výrazů a přiřazení do jednoduchých proměnných můžeme použít schématu z obr. 8.10. V tomto schématu se používá funkce *lookup* pro vyhledání proměnné v tabulce symbolů na základě jejího jména; pokud se jméno v tabulce nenajde, funkce vrátí hodnotu *nil*. Funkce *newtemp* vrátí ukazatel na nově vytvořenou dočasnou proměnnou. Dočasné proměnné mohou být obecně uloženy rovněž v tabulce symbolů, pokud jim přidělíme speciální jména, která nemohou být použita programátorem pro proměnné v programu.

Sémantické akce na obr. 8.10 používají pro výstup tříadresových příkazů procedury *emit*, jejíž parametry uvádíme poněkud zjednodušeně buď jako řetězcové konstanty, nebo jako jména atributů, jejichž příslušné hodnoty se mají předat na výstup.

Uvedené překladové schéma se dá použít i v případě, že pracujeme s jazykem, který má blokovou strukturu, neboť jediná změna nastane v implementaci funkce *lookup* (viz kapitola 5). Dočasné proměnné se považují vždy za lokální proměnné podprogramu, ve kterém jsou použity.

8.3.1 Přidělování dočasných proměnných

Pro přidělování dočasných proměnných se dají použít dvě odlišné strategie. Pro optimalizující překladače je výhodné, pokud každé volání *newtemp* vrátí nové jméno, odlišné od všech předchozích. To však může mít za následek přeplňování tabulky symbolů (nebo obecně pracovní paměti) informacemi, které se používají jen velmi krátce.

$S \to \mathbf{id} := E$ { $p := lookup(\mathbf{id}.name)$;
if $p \neq nil$ **then**
$emit(p\ ':='\ E.place)$
else $error$ }

$E \to E_1\ +\ E_2$ { $E.place := newtemp$;
$emit(E.place\ ':='\ E_1.place\ '+'\ E_2.place)$ }

$E \to E_1\ *\ E_2$ { $E.place := newtemp$;
$emit(E.place\ ':='\ E_1.place\ '*'\ E_2.place)$ }

$E \to -\ E_1$ { $E.place := newtemp$;
$emit(E.place\ ':='\ '\texttt{uminus}'\ E_1.place)$ }

$E \to (\ E_1\ )$ { $E.place := E_1.place$ }

$E \to \mathbf{id}$ { $p := lookup(\mathbf{id}.name)$;
if $p \neq nil$ **then**
$E.place := p$
else $error$ }

Obr. 8.10: Překladové schéma pro překlad aritmetických výrazů a přiřazení

Další možností, která se využívá zejména u jednoprůchodových překladačů, je vícenásobné využívání dočasných proměnných. Ze schematu na obr. 8.10 je zřejmé, že např. překladem výrazu $E_1\ +\ E_2$ vznikne kód ve tvaru

```
vypočti E1 do proměnné t1
vypočti E2 do proměnné t2
t := t1 + t2
```

po jehož vyhodnocení již nejsou proměnné `t1` a `t2` dále potřebné. Doba života všech dočasných proměnných použitých pro vyhodnocení E_1 je vlastně zanořena do doby života proměnné `t`, takže je možné upravit funkci *newtemp* tak, že pro přidělování dočasných proměnných využívá zásobníku. Rovněž je možné do schematu zařadit explicitní volání procedury pro uvolnění dočasné proměnné; tato proměnná se zařadí do seznamu volných dočasných proměnných, odkud se pak může znovu použít při dalším volání *newtemp*.

Přidělování dočasných proměnných je poněkud komplikovanější, pokud jim může být přiřazena hodnota více než jedenkrát, např. v podmíněném výrazu `a > b ? a : b` v jazyce C se musí hodnoty obou větví dostat do téže proměnné. Podobný problém nastává tehdy, pokud provádíme optimalizaci společných podvýrazů; tehdy se může hodnota jedné dočasné proměnné používat na více místech.

8.3.2 Adresování prvků polí

Pole obsahují vždy prvky stejného typu, které se mohou umístit bezprostředně jeden za druhým do společného bloku paměti. Je-li velikost každého prvku w, je i-tý prvek pole `A` uložen na adrese

$$base + (i - low) \times w \tag{8.2}$$

kde low je dolní mez indexu pole a $base$ je relativní adresa přidělené oblasti paměti (neboli relativní adresa prvku `A[`low`]`). Výraz (8.2) můžeme přepsat do tvaru, který umožňuje jeho částečné vyhodnocení již v době překladu:

$$i \times w + (base - low \times w)$$

Podvýraz $c = base - low \times w$ se dá vypočítat v okamžiku deklarace pole a uložit do tabulky symbolů pro `A`. Při generování kódu pro přístup k prvku `A[`i`]` získáme jeho relativní adresu jednoduše přičtením $i \times w$ k c.

Stejnou úvahu můžeme provést pro vícerozměrná pole. Dvojrozměrná pole se obvykle ukládají v paměti po řádcích, kdy můžeme pro výpočet relativní adresy prvku `A[`i_1`,`i_2`]` použít výrazu

$$base + ((i_1 - low_1) \times n_2 + i_2 - low_2) \times w$$

kde low_1 a low_2 jsou dolní meze indexů i_1 a i_2 a n_2 je počet sloupců pole, tj. $n_2 = high_2 - low_2 + 1$), kde $high_2$ je horní mez indexu i_2. Uvedený výraz můžeme opět přepsat do tvaru, ve kterém je oddělena konstantní a proměnná část adresy, jako

$$((i_1 \times n_2) + i_2) \times w + (base - ((low_1 \times n_2) + low_2) \times w) \tag{8.3}$$

Druhý operand tohoto součtu může být vypočten již v době překladu.

Zobecněním výrazu 8.3 pro k-rozměrné pole uložené tak, že se poslední index mění nejrychleji, dostaneme pro relativní adresu prvku `A[`i_1`,`i_2`,...,`i_k`]` následující výraz (*mapovací funkci*):

$$\begin{aligned} &((\cdots((i_1 n_2 + i_2)n_3 + i_3)\cdots)n_k + i_k) \times w \\ +&base - ((\cdots((low_1 n_2 + low_2)n_3 + low_3)\cdots)n_k + low_k) \times w \end{aligned} \tag{8.4}$$

Vzhledem k tomu, že pro j-tý index předpokládáme pevnou hodnotu $n_j = high_j - low_j + 1$), můžeme výraz na druhém řádku v (8.4) vypočítat v době překladu a uložit do položky tabulky symbolů pro `A`. V jazyce C je celý výpočet jednodušší, neboť dolní mez všech indexů je vždy nulová, takže konstantní část výrazu (8.4) je vždy rovna pouze *base*.

Překladové schéma na obr. 8.11 popisuje překlad přiřazovacích příkazů s aritmetickými výrazy a indexy. Toto schéma přímo implementuje výpočet podle vztahu (8.4). Oproti schématu z obr. 8.10 je operandem, resp. levou stranou přiřazení místo symbolu **id** nonterminál L představující l-hodnotu (tj. hodnotu, která může stát na levé straně přiřazení). Pro tento nonterminál můžeme zavést následující pravidla:

$$\begin{aligned} L &\rightarrow \textbf{id}\ [\ Elist\ ] \mid \textbf{id} \\ Elist &\rightarrow Elist\ ,\ E \mid E \end{aligned}$$

Pro vlastní výpočet je však výhodnější tato pravidla přepsat do tvaru, kdy máme během zpracování indexů již k dispozici ukazatel na položku tabulky symbolů pro indexované pole:

$$\begin{aligned} L &\rightarrow Elist\ ] \mid \textbf{id} \\ Elist &\rightarrow Elist\ ,\ E \mid \textbf{id}\ [\ E \end{aligned}$$

Ve schematu na obr. 8.11 se tento ukazatel předává jako atribut $Elist.array$. Dále se pro počet dimenzí (indexových výrazů) používá atributu $Elist.ndim$. Funkce $limit(array, j)$ vrací hodnotu n_j, počet prvků v j-té dimenzi pole, na jehož záznam v tabulce symbolů ukazuje $array$. Funkce $c(array)$ vrátí konstantní část výrazu (8.4) pro pole $array$. Atribut $Elist.place$ obsahuje jméno dočasné proměnné, do které byla uložena hodnota vypočtená z indexových výrazů v $Elist$.

Nonterminál L, reprezentující l-hodnotu, má dva atributy, $L.place$ a $L.offset$. Je-li L jednoduchá proměnná, obsahuje $L.place$ ukazatel na příslušnou položku tabulky symbolů a $L.offset$ je **null**. V opačném případě ukazuje $L.place$ na položku tabulky symbolů pro pole a $L.offset$

(1)	$S \to L\ :=\ E$	{ **if** $L.offset\ =$ **null then** /* L je jednoduchá proměnná */ $emit(L.place\ ':='\ E.place)$; **else** $emit(L.place\ '['\ L.offset\ ']'\ ':='\ E.place)$ }
(2)	$E \to E_1\ +\ E_2$	{ $E.place := newtemp$; $emit(E.place\ ':='\ E_1.place\ '+'\ E_2.place)$}
(3)	$E \to (\ E_1\ )$	{ $E.place := E_1.place$ }
(4)	$E \to L$	{ **if** $L.offset\ =$ **null then** /* L je jednoduchá proměnná */ $E.place := L.place$ **else begin** $E.place := newtemp$; $emit(E.place\ ':='\ L.place\ '['\ L.offset\ ']')$ **end** }
(5)	$L \to Elist\ ]$	{ $L.place := newtemp$; $L.offset := newtemp$; $emit(L.place\ ':='\ c(Elist.array))$; $emit(L.offset\ ':='\ Elist.place\ '*'\ width(Elist.array))$ }
(6)	$L \to$ **id**	{ $L.place :=$ **id**.$place$; $L.offset :=$ **null** }
(7)	$Elist \to Elist_1\ ,\ E$	{ $t := newtemp$; $m := Elist_1.ndim + 1$; $emit(t\ ':='\ Elist_1.place\ '*'\ limit(Elist_1.array, m))$; $emit(t\ ':='\ t\ '+'\ E.place)$; $Elist.array := Elist_1.array$; $Elist.place := t$; $Elist.ndim := m$ }
(8)	$Elist \to$ **id** $[\ E$	{ $Elist.array :=$ **id**.$place$; $Elist.place := E.place$; $Elist.ndim := 1$ }

Obr. 8.11: Překladové schéma pro výrazy s indexy

na položku pro dočasnou proměnnou, do které byla uložena vypočtená relativní adresa prvku pole.

Příklad 8.3. Nechť `A` je pole 10×20 s dolními mezemi indexů $low_1 = low_2 = 1$. Počty prvků v jednotlivých dimenzích jsou tedy $n_1 = 10$ a $n_2 = 20$. Nechť velikost prvku w je 4. Přiřazení `x := A[y,z]` se přeloží do následující posloupnosti tříadresových příkazů:

```
t1 := y * 20
t1 := t1 + z
t2 := c          /*  konstanta c = base_A − 84  */
t3 := 4 * t1
t4 := t2[t3]
x  := t4
```

V příkladu jsme použili místo atributu **id**.*place* přímo jméno proměnné. ■

8.3.3 Konverze typů během přiřazení

V uvedených příkladech překladových schémat jsme zatím uvažovali pouze aritmetické výrazy tvořené operandy téhož typu. V praxi se však běžně pracuje se smíšenými výrazy, u nichž musí překladač buď vygenerovat příslušné implicitní typové konverze, nebo musí nahlásit chybu.

$E.place := newtemp$;
if $E_1.type = integer$ **and** $E_2.type = integer$ **then begin**
    $emit(E.place\ ':='\ E_1.place\ '\texttt{int}+'\ E_2.place)$;
    $E.type := integer$
end
else if $E_1.type = real$ **and** $E_2.type = real$ **then begin**
    $emit(E.place\ ':='\ E_1.place\ '\texttt{real}+'\ E_2.place)$;
    $E.type := real$
end
else if $E_1.type = integer$ **and** $E_2.type = real$ **then begin**
    $u := newtemp$;
    $emit(u\ ':='\ '\texttt{inttoreal}'\ E_1.place)$;
    $emit(E.place\ ':='\ u\ '\texttt{real}+'\ E_2.place)$;
    $E.type := real$
end
else if $E_1.type = real$ **and** $E_2.type = integer$ **then begin**
    $u := newtemp$;
    $emit(u\ ':='\ '\texttt{inttoreal}'\ E_2.place)$;
    $emit(E.place\ ':='\ E_1.place\ '\texttt{real}+'\ u)$;
    $E.type := real$
end
else
    $E.type := type\_error$;

Obr. 8.12: Sémantická akce pro pravidlo $E \to E_1\ +\ E_2$

Uvažujme například jednoduché rozšíření aritmetického výrazu o datové typy *real* a *integer* s možností implicitní konverze celočíselného operandu na reálný ve smíšených výrazech. K tomu musíme zavést nový atribut *E.type*, jehož hodnota *real* nebo *integer* reprezentuje typ příslušného výrazu. Sémantická pravidla ve schématech na obr. 8.10 a 8.11 musíme rozšířit o výpočet tohoto atributu a generování odpovídajících konverzí a aritmetických operátorů. Pro převod hodnoty `y` typu integer na reálnou hodnotu `x` budeme generovat na vhodných místech instrukci `x := inttoreal y` a budeme rozlišovat typ prováděné aritmetické operace. Například pro pravidlo $E \to E_1\ +\ E_2$ můžeme použít sémantickou akci z obr. 8.12.

V reálné implementaci výrazů se smíšenými operandy se nevytváří samostatné pravidlo pro každý operátor; spíše se používá společný podprogram, jehož jedním parametrem je operátor, pro který se má vygenerovat kód. Často se generování kódu pro výrazy také řeší pomocí tabulek — například můžeme použít tabulku, ze které pro zadané typy operandů získáme informaci o tom, zda je tato kombinace přípustná a zda generovat určitou typovou konverzi pro některý z operandů.

Příklad 8.4. Za předpokladu, že proměnné `x` a `y` mají typ *real* a `i` a `j` typ *integer*, můžeme pro vstup `x := y + i * j` vygenerovat kód

```
t1 := i int+ j
t3 := inttoreal t1
t2 := y real+ t3
x  := t2
```

■

8.4 Booleovské výrazy

Booleovské výrazy se v programovacích jazycích používají ke dvěma hlavním účelům — pro výpočet logických hodnot a (především) jako podmínky v příkazech pro změnu toku řízení jako je např. podmíněný příkaz nebo příkaz cyklu.

Booleovské výrazy jsou tvořené operátory jako **and**, **or** nebo **not** a operandy, kterými mohou být booleovské konstanty, proměnné nebo relační výrazy. V některých jazycích mohou být operandy booleovských výrazů hodnoty i jiných typů. Pro další výklad budeme vycházet z této gramatiky booleovského výrazu:

$$E \to E \textbf{ or } E \mid E \textbf{ and } E \mid (E) \mid \textbf{id relop id} \mid \textbf{true} \mid \textbf{false}$$

Terminální symbol **relop** bude mít atribut *op* určující jeden ze šesti relačních operátorů.

Pro reprezentaci booleovských hodnot a překlad booleovských výrazů se používají dvě základní metody. První metoda reprezentuje logické hodnoty jako čísla a vyhodnocuje booleovské výrazy stejně jako aritmetické. Pro kódování booleovských hodnot se často používá 0 jako false a 1 nebo nenulová hodnota jako true.

Druhou základní metodou je reprezentace booleovských výrazů tokem řízení, tj. pozicí dosaženou v programu. Tato metoda je zvláště výhodná pro implementaci řídicích příkazů, neboť umožňuje jejich efektivnější vyhodnocování — pokud např. ve výrazu E_1 **or** E_2 zjistíme, že hodnota jednoho operandu je true, nemusíme již vyhodnocovat druhý operand.

To, zda můžeme použít první nebo druhou metodu, je dáno sémantikou implementovaného programovacího jazyka. Dovoluje-li jazyk ponechat některé části výrazu nevyhodnocené, může překladač provést optimalizaci booleovského výrazu a vyhodnotit jen tu část výrazu, kterou potřebuje. Pokud však některý z operandů má vedlejší účinek (např. se v něm volá funkce, která mění hodnotu nějaké globální proměnné), můžeme při zkráceném vyhodnocení výrazu dostat neočekávané výsledky. Obecně nelze říci, která z obou metod je výhodnější; některé překladače umožňují pomocí parametrů metodu překladu určit nebo dovedou vybrat vhodnou metodu na základě analýzy každého konkrétního výrazu.

8.4.1 Reprezentace booleovských výrazů číselnou hodnotou

Za předpokladu, že budeme logické hodnoty reprezentovat čísly 0 a 1 a provádět úplné vyhodnocení, můžeme výraz `a or b and not c` přeložit do tříadresového kódu jako

```
t1 := not c
t2 := b and t1
t3 := a or t2
```

Pokud výraz obsahuje relační operátor, musíme z výsledku relace nejprve odvodit příslušnou číselnou hodnotu. Například výraz `x < y` se přeloží jako

```
100: if x < y goto 103
101: t1 := 0
102: goto 104
103: t1 := 1
```

Překladové schéma pro generování tříadresového kódu pro booleovské výrazy je na obr. 8.13. Předpokládáme, že procedura *emit* zapisuje do výstupního souboru tříadresové příkazy a že proměnná *neststat* obsahuje index následujícího příkazu tříadresového kódu, zvyšovaný procedurou *emit*.

$E \to E_1$ **or** E_2	{	$E.place := newtemp$;
		$emit(E.place$ ':=' $E_1.place$ 'or' $E_2.place)$ }
$E \to E_1$ **and** E_2	{	$E.place := newtemp$;
		$emit(E.place$ ':=' $E_1.place$ 'and' $E_2.place)$ }
$E \to$ **not** E_1	{	$E.place := newtemp$;
		$emit(E.place$ ':=' 'not' $E_1.place)$ }
$E \to (E_1)$	{	$E.place := E_1.place$}
$E \to$ **id**$_1$ **relop** **id**$_2$	{	$E.place := newtemp$;
		$emit($'if' **id**$_1.place$ **relop**$.op$ **id**$_2.place$ 'goto' $nextstat + 3)$;
		$emit(E.place$ ':=' '0'$)$;
		$emit($'goto' $nextstat + 2)$;
		$emit(E.place$ ':=' '1'$)$ }
$E \to$ **true**	{	$E.place := newtemp$;
		$emit(E.place$ ':=' '1'$)$ }
$E \to$ **false**	{	$E.place := newtemp$;
		$emit(E.place$ ':=' '0'$)$ }

Obr. 8.13: Překladové schéma pro číselnou reprezentaci booleovských výrazů

Příklad 8.5. Na základě schematu z obr. 8.13 můžeme pro booleovský výraz `a < b or c < d and e < f` vygenerovat kód uvedený na obr. 8.14. ■

```
100: if a < b goto 103        107: t2 := 1
101: t1 := 0                  108: if e < f goto 111
102: goto 104                 109: t3 := 0
103: t1 := 1                  110: goto 112
104: if c < d goto 107        111: t3 := 1
105: t2 := 0                  112: t4 := t2 and t3
106: goto 108                 113: t5 := t1 or t4
```

Obr. 8.14: Překlad výrazu `a < b or c < d and e < f`

8.4.2 Zkrácené vyhodnocování booleovských výrazů

Metoda zkráceného vyhodnocování booleovských výrazů umožňuje zpracovat booleovské výrazy bez jejich úplného vyhodnocení. Při této metodě se logické hodnoty nereprezentují jako data; každé kombinaci logických hodnot operandů ve výrazu místo toho odpovídá určitá pozice ve

vygenerovaném kódu, na kterou se program dostane pomocí podmíněných a nepodmíněných skoků. Například na obr. 8.14 můžeme hodnotu proměnné `t1` odvodit z toho, zda se dostaneme na řádek 101 nebo 103 a podle toho pokračovat ve vyhodnocování zbytku výrazu; samotná hodnota proměnné `t1` je redundantní.

Překladové schéma na obr. 8.15 využívá pro zkrácené vyhodnocení výrazu dědičných atributů $E.true$ a $E.false$, které obsahují jméno návěští, na které má program přejít, je-li hodnota příslušného podvýrazu true, resp. false. Nonterminál M s atributem $M.lab$ slouží pouze pro vygenerování návěští před vyhodnocením druhého operandu binárního operátoru. Funkce *newlabel* při každém volání vrátí nové, ještě nepoužité návěští.

$E \to E_1$ **or** M E_2	{	$E_1.true := E_2.true := E.true$;
		$E_1.false := M.lab := newlabel$;
		$E_2.false := E.false$ }
$E \to E_1$ **and** M E_2	{	$E_1.false := E_2.false := E.false$;
		$E_1.true := M.lab := newlabel$;
		$E_2.true := E.true$ }
$E \to$ **not** E_1	{	$E_1.true := E.false$;
		$E_2.false := E.true$ }
$E \to$ (E_1)	{	$E_1.true := E.true$;
		$E_1.false := E.false$ }
$E \to$ **id**$_1$ **relop** **id**$_2$	{	$gen($`'if'` **id**$_1.place$ **relop**$.op$ **id**$_2.place$ `'goto'` $E.t)$;
		$gen($`'goto'` $E.false)$ }
$E \to$ **true**	{	$gen($`'goto'` $E.true)$ }
$E \to$ **false**	{	$gen($`'goto'` $E.false)$ }
$M \to \epsilon$	{	$gen(M.lab$ `':'`$)$ }

Obr. 8.15: Překladové schéma pro zkrácené vyhodnocení booleovských výrazů

Pro každý relační operátor se vygeneruje podmíněný skok na návěští $E.true$ a nepodmíněný skok na návěští $E.false$. Všechny další operace spočívají pouze ve vhodném vytváření a kombinování návěští pro tyto dva skoky. Například předpokládejme, že máme výraz ve tvaru E_1 **and** E_2. Má-li E_1 hodnotu false, bude mít i celý výraz E hodnotu false a můžeme tedy pro $E_1.false$ použít hodnotu $E.false$. Má-li E_1 hodnotu true, musíme vyhodnotit ještě E_2, takže použijeme $E_1.true$ jako návěští prvního příkazu pro E_2. Pro vyhodnocení E_2 pak již můžeme použít stejná návěští jako pro celý výraz E. Podobná úvaha platí i pro operátor **or**. Pro výraz ve tvaru **not** E nepotřebujeme dokonce vůbec žádný kód, pouze vyměníme úlohy atributů $E.true$ a $E.false$.

Příklad 8.6. Na základě schematu z obr. 8.15 můžeme pro booleovský výraz `a < b or c < d and e < f` vygenerovat kód uvedený na obr. 8.16. ■

Kód vygenerovaný podle schématu z obr. 8.15 není optimální. Například na obr. 8.16 lze vypustit druhý řádek, aniž se nějak ovlivní funkce programu. Vygenerovaný kód lze optimalizovat buď dodatečně, nebo je možné do překladového schematu začlenit akce, které budou určité optimalizace provádět již během generování. Často lze kód například vylepšit obrácením testované podmínky, např. přepíšeme-li třetí řádek na obr. 8.16 do tvaru

```
L1: if c >= d goto Lfalse
```

můžeme vypustit i následující skok na návěští `Lfalse`.

```
    if a < b goto Ltrue
    goto L1
L1: if c < d goto L2
    goto Lfalse
L2: if e < f goto Ltrue
    goto Lfalse
```

Obr. 8.16: Zkrácený překlad výrazu `a < b or c < d and e < f`

V některých jazycích, jako je např. jazyk C, se booleovské výrazy mohou libovolně kombinovat s ostatními aritmetickými výrazy. Schéma pro překlad takových kombinovaných výrazů musí zajistit přechod mezi reprezentací tokem řízení a číselnou reprezentací logických hodnot. Obr. 8.17 ukazuje dvě taková pravidla pro aritmetický výraz AE a booleovský výraz BE.

$$
\begin{array}{lll}
AE \to BE & \{ & BE.true := newlabel; \\
 & & BE.false := newlabel; \\
 & & templab := newlabel; \\
 & & AE.place := newtemp; \\
 & & gen(BE.true \texttt{ ':'}); \\
 & & gen(AE.place \texttt{ ':=' '1'}); \\
 & & gen(\texttt{'goto'}\ templab); \\
 & & gen(BE.false \texttt{ ':'}); \\
 & & gen(AE.place \texttt{ ':=' '0'}); \\
 & & gen(templab \texttt{ ':'}) \ \} \\
BE \to AE & \{ & gen(\texttt{'if'}\ AE.place\ \texttt{'<>' '0' 'goto'}\ BE.true); \\
 & & gen(\texttt{'goto'}\ BE.false)\}
\end{array}
$$

Obr. 8.17: Pravidla pro překlad smíšených booleovských výrazů

8.5 Příkazy pro změnu toku řízení

Nyní se pokusíme předvedené metody překladu booleovských výrazů začlenit do překladu řídicích příkazů. Budeme uvažovat příkazy generované následující gramatikou:

$$
\begin{array}{rcl}
S & \to & \textbf{if}\ E\ \textbf{then}\ S_1 \\
 & | & \textbf{if}\ E\ \textbf{then}\ S_1\ \textbf{else}\ S_2 \\
 & | & \textbf{while}\ E\ \textbf{do}\ S_1
\end{array}
$$

V těchto pravidlech je vždy E booleovský výraz. Při překladu s úplným vyhodnocením bude mít E syntetizovaný atribut $E.place$, jméno proměnné obsahující hodnotu výrazu, při zkráceném překladu tokem řízení bude mít E naopak dva *dědičné* atributy obsahující návěští pro hodnotu true ($E.true$) a false ($E.false$), stejně jako v předcházejících odstavcích.

Překladové schéma na obr. 8.18 vychází z předpokladu, že booleovské výrazy se překládají zkráceně. Pro vkládání návěští a skoků do řídicích konstrukcí se zde používají nonterminály M a N s dědičným atributem *lab* označujícím jméno návěští pro definici nebo skok. Nonterminál N zajišťuje v úplném příkazu **if** přeskok části za **else** při splnění podmínky.

$S \to$ **if** E **then** $M\ S_1$	{	$E.true := newlabel;$ $E.false := newlabel;$ $M.lab := E.true;$ $gen(E.false\ \texttt{':'})$ }
$S \to$ **if** E **then** $M\ S_1$ **else** $N\ M\ S_2$	{	$E.true := newlabel;$ $E.false := newlabel;$ $M_1.lab := E.true;$ $M_2.lab := E.false;$ $N.lab := newlabel;$ $gen(N.lab\ \texttt{':'})$ }
$S \to$ **while** $M_1\ E$ **do** $M_2\ S_1$	{	$M_1.lab := newlabel;$ $M_2.lab := E.true;$ $gen(\texttt{'goto'}\ M_1.lab);$ $gen(E.false\ \texttt{':'})$ }
$M \to \epsilon$	{	$gen(M.lab\ \texttt{':'})$ }
$N \to \epsilon$	{	$gen(\texttt{'goto'}\ N.lab)$ }

Obr. 8.18: Překladové schéma pro řídicí příkazy

Příklad 8.7. Uvažujme příkaz

```
while a < b do
   if c < d then
      x := y + z
   else
      x := y - z
```

Na základě schémat z obr. 8.10, 8.15 a 8.18 můžeme pro tento příkaz vygenerovat následující tříadresový kód:

```
L1:  if a < b goto L2
     goto Lnext
L2:  if c < d goto L3
     goto L4
L3:  t1 := y + z
     x := t1
     goto L1
L4:  t2 := y - z
     x := t2
     goto L1
Lnext:
```

Pokud obrátíme směr relací v prvním a třetím řádku, můžeme vypustit za nimi následující nepodmíněné skoky. ■

8.6 Selektivní příkazy

Selektivní příkazy typu switch nebo case jsou dostupné v mnoha jazycích. Představují vlastně zobecněný příkaz if s více variantami. Obecně mají tyto příkazy strukturu obdobnou jako na

obr. 8.19. Výraz E v záhlaví selektivního příkazu se vyhodnotí a v případě, že se jeho hodnota rovná některé z uvedených konstant V_i, provede se příkaz S_i uvedený za konstantou. V opačném případě, pokud je uvedena varianta **default**, se provede implicitní příkaz S_{def}.

switch E
  begin
    case V_1 : S_1
    case V_2 : S_2
      . . .
    case V_n : S_n
    default : S_{def}
  end

Obr. 8.19: Struktura selektivního příkazu

Selektivní příkaz můžeme implementovat mnoha různými způsoby v závislosti na intervalu, ve kterém leží uvedené konstanty, velikosti mezer mezi konstantami (počtu nevyužitých hodnot uvnitř intervalu mezi největší a nejmenší konstantou) a preferovaných vlastnostech vygenerovaného kódu (optimalizace na čas nebo velikost kódu). Obecně se používají následující metody:

- *Posloupnost podmíněných skoků.* Tato nejjednodušší varianta je výhodná pouze při malém počtu položek; každý podmíněný skok testuje jednu uvedenou variantu.
- *Tabulka dvojic.* Vytvoříme vyhledávací tabulku obsahující vždy hodnotu konstanty a návěští začátku příkazu pro tuto variantu. V tabulce pak můžeme vyhledávat některou z běžných metod. Není-li hodnota výrazu v tabulce nalezena, provede se implicitní varianta. Pro velké počty návěští se někdy používá tabulek s rozptýlenými položkami.
- *Rozskoková tabulka.* V případě, že hodnoty konstant dostatečně hustě zaplňují určitý interval hodnot, například $< i_{min}, i_{max} >$, můžeme vytvořit pole návěští obsahující v j-tém prvku návěští pro hodnotu $i_{min} + j$. Při vyhodnocení selektivního příkazu se nejprve zjistí, zda hodnota výrazu e leží v intervalu $< i_{min}, i_{max} >$ a pokud ano, provede se nepřímý skok na návěští uložené v $(e - i_{min})$-té položce tabulky. Nevyužité pozice v tabulce se zaplní návětším pro implicitní variantu nebo chybu.

V praxi se rovněž používají modifikace uvedených metod nebo jejich kombinace. Například můžeme kromě podmíněného skoku testujícího rovnost hodnoty výrazu s některou konstantou vygenerovat zároveň odskok, je-li hodnota menší, a dále pak testovat odděleně již menší podmnožiny hodnot (jde vlastně o implementaci binárního vyhledávacího stromu pomocí toku řízení). Po vymezení dostatečně kompaktní podmnožiny hodnot pak můžeme pro konečné vyhodnocení použít rozskokové tabulky.

Překlad selektivního příkazu do tříadresového kódu má strukturu jako na obr. 8.20. Po vyhodnocení výrazu E se provede odskok na vlastní tělo selektivního příkazu, čímž se přeskočí kód vygenerovaný pro jednotlivé varianty. Každá varianta je pak označena návěštím a končí skokem na příkaz následující za selektivním příkazem (V jazyce C se tento skok generuje pouze pro explicitně zapsaný příkaz `break`).
Část kódu uvedená mezi návěštími `test` a `next` odpovídá vlastnímu selektivnímu příkazu a její struktura tedy závisí na zvolené metodě překladu. Pokud nechceme tento problém řešit již při

```
        kód pro vyhodnocení E do proměnné t
        goto test
L1:     kód pro S1
        goto next
L2:     kód pro S2
        goto next
          ...
Ln:     kód pro Sn
        goto next
test:if t = V1 goto L1
        if t = V2 goto L2
          ...
        if t = Vn goto Ln
next:
```

Obr. 8.20: Překlad selektivního příkazu

generování mezikódu, lze rozšířit mezikód o speciální příkazy reprezentující selektivní příkaz, např. takto:

```
test: select t,Ldef
       case   V1,L1
       case   V2,L2
       ...
       case   Vn,Ln
next:
```

Příkaz `select` má jako parametr odkaz na místo, kde je uložena hodnota vypočteného výrazu a návěští pro implicitní variantu, příkaz `case` definuje hodnotu konstanty a návěští příslušného příkazu pro každou uvedenou variantu. Konec seznamu příkazů `case` je jednoduše rozpoznatelný podle návěští, které musí vždy následovat. O skutečné metodě překladu se pak může rozhodnout až při generování kódu, kdy lze například využít některých speciálních instrukcí procesoru.

8.7 Backpatching

V předchozích odstavcích jsme si předvedli několik různých metod generování mezikódu pro řídicí příkazy. Nezabývali jsme se však vlastním přiřazením adres pro jednotlivá návěští, obcházeli jsme jej použitím symbolických jmen. Při víceprůchodovém překladu se toto přiřazení může provést v samostatném průchodu vygenerovaným kódem, kdy už jsou známy všechny vygenerované instrukce i rozmístění jednotlivých návěští. Pokud se ale překlad provádí jednoprůchodově, dostáváme se velmi často do situace, kdy neznáme v určitém bodě cílovou adresu návěští, neboť kód, který bude tímto návěštím označen, ještě nebyl vygenerován.

Tento problém můžeme řešit tak, že necháme adresu návěští prázdnou a udržujeme seznam adres, ze kterých se na každé takové návěští odkazujeme. V okamžiku, kdy dosáhneme místa obsahujícího definici návěští, zpětně jeho adresu doplníme do všech míst uvedených v seznamu. Tato metoda zpětných oprav se nazývá *backpatching*. Implementaci si ukážeme na jednoprůchodovém překladu řídicích příkazů s logickými výrazy vyhodnocovanými zkráceně.

Budeme předpokládat, že generujeme čtveřice uložené v poli; návěští pak budou indexy do pole čtveřic. Adresa následující volné čtveřice bude uložena v proměnné *nextquad*. Pro manipulaci se seznamy návěští budeme používat následující podprogramy:

1. *makelist*(i) vytvoří nový seznam obsahující pouze index i; vrátí ukazatel na vytvořený seznam.
2. *merge*(p_1, p_2) spojí dva seznamy, na které ukazuje p_1 a p_2 do jediného a vrátí ukazatel na takto vytvořený nový seznam.
3. *backpatch*(p, i) vloží i jako adresu návěští do všech příkazů obsažených v seznamu, na který ukazuje p.

8.7.1 Booleovské výrazy

Při překladu booleovských výrazů doplníme do gramatiky nonterminál (marker) M, který bude sloužit pro uschování současné hodnoty čítače čtveřic *nextquad*. Upravená gramatika, pro kterou budeme dále vytvářet překladové schéma, je následující:

(1) $E \rightarrow E_1$ **or** M E_2
(2) | E_1 **and** M E_2
(3) | **not** E_1
(4) | (E_1)
(5) | **id**$_1$ **relop** **id**$_2$
(6) | **true**
(7) | **false**
(8) $M \rightarrow \epsilon$

Nonterminál E používá syntetizované atributy $truelist$ a $falselist$, obsahující seznamy neúplných instrukcí s odskoky při hodnotě výrazu true a false. Sémantické akce vycházejí z následující úvahy: Například je-li v pravidle $E \rightarrow E_1$ **and** M E_2 hodnota E_1 false, je i hodnota E false, takže všechny příkazy v $E_1.falselist$ se stanou částí $E.falselist$. Je-li však E_1 true, musíme nejprve testovat E_2, takže cílovou adresou pro instrukce v $E_1.truelist$ musí být adresa začátku kódu pro vyhodnocení E_2. Tu získáme pomocí markeru M, jehož atribut $M.quad$ obsahuje právě index prvního příkazu pro E_2. S pravidlem $M \rightarrow \epsilon$ je svázána sémantická akce

$$\{\ M.quad := nextquad\ \}$$

. Hodnota čítače *nextquad* uschovaná tímto nonterminálem bude sloužit pro zpětné opravy adres v seznamu $E_1.truelist$ v okamžiku, kdy dosáhneme konce pravidla pro operátor **and**. Celé překladové schéma pro booleovské výrazy je uvedeno na obr. 8.21.
Sémantická akce (5) generuje dva skoky, podmíněný a nepodmíněný. Oba tyto skoky směřují na dosud neznámou adresu, proto se jejich adresy zařadí do seznamů $E.truelist$ a $E.falselist$. V pravidlech (6) a (7) se reprezentuje konstanta **true** a **false** jako nepodmíněný skok, jehož adresa se opět zařadí do příslušného seznamu neúplných skoků; druhý seznam v tomto případě bude prázdný. Uvedené schéma může být použito přímo při překladu zdola nahoru, neboť všechny sémantické akce se vyskytují na konci pravidel a mohou se tedy provádět zároveň s redukcemi.

Příklad 8.8. Uvažujme opět výraz `a<b or c<d and e<f` jako v příkladu 8.6. Odpovídající ohodnocený derivační strom je uveden na obr. 8.22 (názvy atributů jsou zde zkráceny). Pro podvýraz `a<b` se na základě pravidla (5) vygenerují dvě čtveřice

(1)	$E \to E_1$ **or** $M\ E_2$	{ $backpatch(E_1.falselist, M.quad)$; $E.truelist := merge(E_1.truelist, E_2.truelist)$; $E.falselist := E_2.falselist$ }
(2)	$E \to E_1$ **and** $M\ E_2$	{ $backpatch(E_1.truelist, M.quad)$; $E.truelist := E_2.truelist$; $E.falselist := merge(E_1.falselist, E_2.falselist)$ }
(3)	$E \to$ **not** E_1	{ $E.truelist := E_1.falselist$; $E.falselist := E_1.truelist$ }
(4)	$E \to (\ E_1\ )$	{ $E.truelist := E_1.truelist$; $E.falselist := E_1.falselist$ }
(5)	$E \to$ **id**$_1$ **relop** **id**$_2$	{ $E.truelist := makelist(nextquad)$; $E.falselist := makelist(nextquad + 1)$; $emit($`'if'` **id**$_1.place$ **relop**$.op$ **id**$_2.place$ `'goto _'`$)$; $emit($`goto _`$)$ }
(6)	$E \to$ **true**	{ $E.truelist := makelist(nextquad)$; $E.falselist :=$ **nil**; $emit($`'goto _'`$)$ }
(7)	$E \to$ **false**	{ $E.falselist := makelist(nextquad)$; $E.truelist :=$ **nil**; $emit($`'goto _'`$)$ }
(8)	$M \to \epsilon$	{ $M.quad := nextquad$ }

Obr. 8.21: Překladové schéma pro booleovské výrazy s backpatchingem

```
100: if a < b goto _
101: goto _
```

Nonterminál M v pravidle (1) zaznamená hodnotu *nextquad*, která je nyní 102. Redukce `c<d` na E podle pravidla (5) vygeneruje čtveřice

```
102: if c < d goto _
103: goto _
```

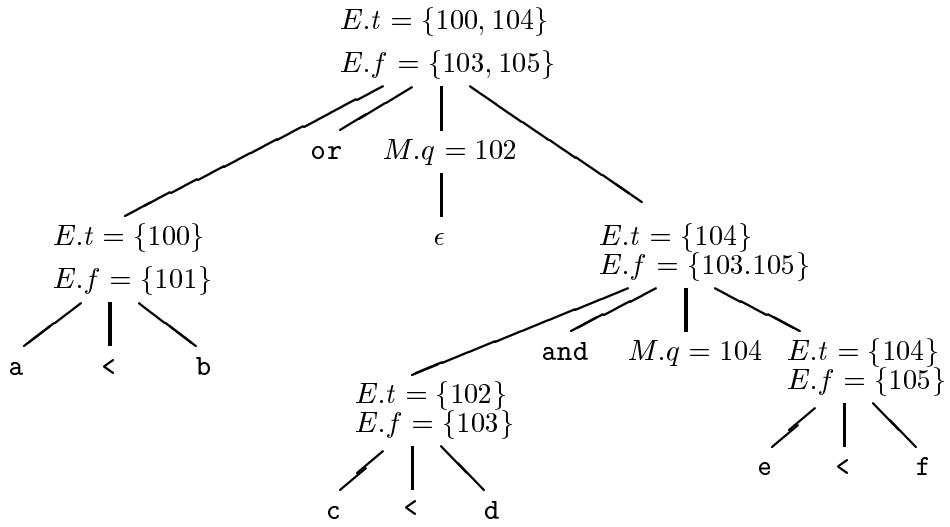
Nyní máme k dispozici nonterminál E_1 z pravidla (2). Následující marker M zaznamená současnou hodnotu *nextquad*, tj. 104. Redukce `e<f` na E opět podle pravidla (5) vygeneruje

```
104: if y < f goto _
105: goto _
```

Dále nastane redukce podle pravidla $E \to E_1$ **and** $M\ E_2$. Odpovídající sémantická akce volá proceduru $backpatch(\{102\}, 104)$, která doplní adresu 104 do příkazu 102. Konečně redukce podle pravidla $E \to E_1$ **or** $M\ E_2$ volá $backpatch(\{101\}, 102)$, která doplní adresu 102 do příkazu 101. Tím dostaneme konečnou posloupnost příkazů ve tvaru

```
100: if a < b goto _
101: goto 102
102: if c < d goto 104
103: goto _
104: if e < f goto _
105: goto _
```

přičemž pro nonterminál E reprezentující celý výraz budeme mít atributy $E.falselist =$



Obr. 8.22: Ohodnocený derivační strom pro `a<b or c<d and e<f`

$\{103, 105\}$ a $E.truelist = \{100, 104\}$. To znamená, že celý výraz má hodnotu true, provedou-li se skoky v příkazech 100 nebo 104, a hodnotu false, provedou-li se skoky v příkazech 103 nebo 105. Cílové adresy těchto skoků se doplní dále během překladu v závislosti na tom, co se má udělat při které hodnotě výrazu. ■

8.7.2 Překlad řídicích příkazů

Nyní předvedeme, jakým způsobem se dá backpatching použít při jednoprůchodovém překladu řídicích příkazů. Pro překlad budeme používat následující gramatiku:

(1) $S \rightarrow$ **if** E **then** S
(2) | **if** E **then** S **else** S
(3) | **while** E **do** S
(4) | **begin** L **end**
(5) | A
(6) $L \rightarrow L$; S
(7) | S

Nonterminál S označuje příkaz, L seznam příkazů, A přiřazovací příkaz a E booleovský výraz. Pro booleovský výraz použijeme výsledků předchozího odstavce, struktura přiřazovacího příkazu, případně dalších jednoduchých příkazů nás nyní nebude zajímat.

Pro překlad zvolíme opět ten přístup, že budeme zpětně doplňovat adresy skoků v tom okamžiku, kdy dosáhneme jejich cílového příkazu. Kromě dvou seznamů adres pro booleovské výrazy budeme rovněž potřebovat seznam adres skoků na kód, který následuje za příkazy generovanými nonterminály S a L; na tento seznam bude ukazovat atribut *nextlist*.

Při překladu příkazu $S \rightarrow$ **while** E **do** S_1 budeme potřebovat dvě návěští: jedno pro označení začátku celého příkazu S a jedno pro tělo cyklu S_1. Adresy těchto návěští opět zaznamenáme pomocí dvou výskytů markeru M na příslušných pozicích v pravidle:

$$S \rightarrow \textbf{while}\ M_1\ E\ \textbf{do}\ M_2\ S_1$$

S nonterminálem M bude opět svázáno pravidlo, které do atributu $M.quad$ uloží hodnotu čítače *nextquad*. Po zpracování těla S_1 se řízení vrací na začátek cyklu, takže při redukci **while** M_1 E **do** M_2 S_1 na S přepíšeme cílové adresy skoků ze seznamu $S_1.nextlist$ na $M_1.quad$. Vzhledem k tomu, že posledním příkazem S_1 nemusí být skok, musíme rovněž za tělo S_1 doplnit explicitní skok na začátek kódu pro E. Nakonec do příkazů v seznamu $E.truelist$ doplníme adresu začátku S_1, tj. $M_2.quad$ a $E.falselist$ se stane hodnotou atributu $S.nextlist$ celého příkazu S.

Překlad příkazu **if** E **then** S_1 **else** S_2 je ještě poněkud složitější, neboť potřebujeme za kód vygenerovaný z S_1 vložit skok přes kód pro S_2. K tomu využijeme dalšího markeru N s atributem $N.nextlist$, seznamem obsahujícím pouze adresu čtveřice s příkazem `goto _`, který vygeneruje sémantická akce spojená s N. Úplné překladové schéma pro řídicí příkazy je uvedeno na obr. 8.23.

(1) $S \rightarrow$ **if** E **then** M_1 S_1 N **else** M_2 S_2
$\{$ $backpatch(E.truelist, M_1.quad)$;
$backpatch(E.falselist, M_2.quad)$;
$S.nextlist := merge(S_1.nextlist, merge(N.nextlist, S_2.nextlist))$ $\}$

(2) $N \rightarrow \epsilon$ $\{$ $N.nextlist := makelist(nextquad)$;
$emit(\texttt{'goto \_'})$ $\}$

(3) $M \rightarrow \epsilon$ $\{$ $M.quad := nextquad$ $\}$

(4) $S \rightarrow$ **if** E **then** M S_1
$\{$ $backpatch(E.truelist, M.quad)$;
$S.nextlist := merge(E.falselist, S_1.nextlist)$ $\}$

(5) $S \rightarrow$ **while** M_1 E **do** M_2 S_1
$\{$ $backpatch(S_1.nextlist, M_1.quad)$;
$backpatch(E.truelist, M_2.quad)$;
$S.nextlist := E.falselist$;
$emit(\texttt{'goto'}\ M_1.quad)$ $\}$

(6) $S \rightarrow$ **begin** L **end** $\{$ $S.nextlist := L.nextlist$ $\}$

(7) $S \rightarrow A$ $\{$ $S.nextlist :=$ **nil** $\}$

(8) $L \rightarrow L_1$; M S $\{$ $backpatch(L_1.nextlist, M.quad)$;
$L.nextlist := S.nextlist$ $\}$

(9) $L \rightarrow S$ $\{$ $L.nextlist := S.nextlist$ $\}$

Obr. 8.23: Překladové schéma pro řídicí příkazy s backpatchingem

Při zpracování návěští a příkazů skoku jako součástí zdrojového jazyka je třeba provádět navíc určité kontroly, například zda existuje právě jedna definice návěští, nebo zda cílová adresa skoku nemíří dovnitř složené konstrukce. Některé jazyky, jako např. Pascal, dále vyžadují, aby všechna návěští byla předem deklarována.

Pokud překladač rozpozná příkaz skoku, např. `goto L`, musí zkontrolovat, zda je v rozsahu platnosti návěští `L` právě jedna jeho definice. Pokud se návěští ještě nevyskytlo, uloží se do tabulky symbolů a přiřadí se mu jako atribut seznam tvořený adresou vygenerované čtveřice s příkazem skoku. Při všech dalších výskytech návěští v příkazu skoku ještě před jeho definicí se do tohoto seznamu pouze přidávají adresy odpovídajících skokových instrukcí. V okamžiku, kdy se vyskytne definice takto již použitého návěští, zavolá se procedura *backpatch*, které se předá seznam nedokončených skokových instrukcí a současná hodnota *nextquad*. Hodnota *nextquad* se

rovněž uloží do tabulky symbolů jako skutečná adresa návěští, která se pak může v následujících příkazech skoku použít přímo.

8.7.3 Volání podprogramů

Podprogramy jsou tak důležité a často používané programové konstrukce, že je nutné, aby pro volání a návraty z podprogramů generoval překladač co nejefektivnější kód. V některých případech se mohou akce spojené s předáváním řízení mezi podprogramy provádět ve spolupráci se systémem řízení běhu programu.

Jak již bylo uvedeno v kapitole 6, volání podprogramu je obvykle standardizované ve tvaru určité volací posloupnosti. I když se volací posloupnosti od sebe liší i pro jednotlivé implementace jednoho programovacího jazyka, jsou obvykle tvořeny následujícími akcemi:

Je-li zavolán podprogram, musí se přidělit prostor pro alokační záznam volaného podprogramu. Musí se vyhodnotit předávané argumenty a dát je k dispozici volanému podprogramu na určitém známém místě. Dále je třeba upravit vazební ukazatele tak, aby měl volaný podprogram zajištěn přístup ke správným datům v nadřazených blocích. Stav volajícího podprogramu musí být uložen tak, aby mohl být znovu obnoven při návratu. Na známé místo se také uloží návratová adresa, místo, na které se vrátí řízení po ukončení volaného podprogramu. Návratová adresa je obvykle místo následující za příkazem volání. Nakonec se musí vygenerovat skok na začátek volaného podprogramu.

Při návratu z podprogramu se rovněž musejí provést určité pevně stanovené akce. Je-li podprogram funkcí, je třeba uložit na známé místo výsledek. Dále se musí obnovit aktivační záznam volajícího podprogramu a provést skok na návratovou adresu.

Mezi úkoly volajícího a volaného podprogramu není žádná přesná hranice. Často se tyto úkoly rozdělují na základě vlastností zdrojového jazyka, cílového počítače a požadavků operačního systému.

Kapitola 9

Optimalizace

Pojem *optimalizace programu* úzce souvisí s přirozeným chápáním ekvivalence mezi programy. Dva programy jsou ekvivalentní tehdy, když pro každý soubor vstupních údajů z množiny možných vstupních údajů dávají stejné výstupní údaje (výsledky). Prvotním cílem kompilačního překladače je samozřejmě zachovat tuto ekvivalenci mezi každým zdrojovým a cílovým programem. K danému zdrojovému programu však existuje nekonečně mnoho ekvivalentních cílových programů, které se liší např. délkou, rychlostí výpočtu, nároky na paměť počítače při výpočtu, nebo dokonce algoritmy, které jsou implementovány zdrojovým a cílovým programem.

Je dokázáno, že problém nalezení nejlepšího ekvivalentního cílového programu pro každý zdrojový program podle určité účelové funkce je algoritmicky neřešitelný. Proto ve skutečnosti žádný překladač nemůže provádět dokonalou optimalizaci generovaného programu. Termín optimalizace se v této souvislosti tradičně používá pro určitá vylepšení cílového programu, bez nichž by byl výsledný program pomalejší při výpočtu, nebo měl větší nároky na paměť, případně obojí. Proces optimalizace obvykle nezahrnuje transformace zdrojového programu, které by měnily programátorem stanovený algoritmus řešení nebo jeho implementaci ve zdrojovém jazyce. Vzniká tak otázka, co je zdrojem optimalizace a proč není optimalizace samozřejmou součástí překladu.

Prostředky většiny vyšších programovacích jazyků jsou charakterizovány strojovou nezávislostí, která je dosažena abstrakcí od konkrétního výpočetního prostředí, v němž budou programy prováděny. Práce s abstraktními objekty, umožňující potlačit detaily popisu výpočetních procesů počítače, podstatně usnadňuje zápis programu. Na druhé straně je však efektivnost a výkonnost programu do značné míry závislá na tom, jak se používá registrů počítače (minimalizace přesunů mezi registry a paměti) a jak se používá specifických vlastností instrukcí počítače, tedy prostředků, se kterými programátor nepracuje. Pro překladače tak vzniká obtížný úkol generovat takový cílový program ekvivalentní zdrojovému programu, který je srovnatelný v kvalitě využívání konkrétních prostředků cílového počítače s "ručně" psaným programem.

Je zřejmé, že kvalitu cílového programu určuje dominujícím způsobem generátor cílového programu. Na základě syntaktické a sémantické analýzy a případného překladu do některého vnitřního jazyka vytváří generátor cílového programu posloupnosti instrukcí implementující operace získané analýzou zdrojového programu. Není snadné stanovit jednoznačně hranici mezi činnostmi, které jsou v kompetenci dobrého generátoru cílového programu a činnostmi, které lze kvalifikovat jako optimalizaci cílového programu. Běžně však sémantická analýza prováděná v rámci syntaxí řízeného překladu nedává vyčerpávající informaci o kontextových vazbách z hlediska řídících a hlavně údajových toků v překládaném programu. Tyto vazby mohou značně

ovlivnit tvar generovaného programu. Jejich využití vyžaduje vytvoření speciální struktury — grafu toku řízení programu a jeho analýzu, což jsou obvykle činnosti spojené s optimalizací programu.

Optimalizace prováděné v rámci generování cílového programu berou v co největší míře v úvahu specifika architektury a instrukčního souboru cílového počítače a jsou proto označovány jako *strojově závislé optimalizace*.

Existuje významná třída optimalizací, které je možno provádět nad vnitřním tvarem překládaného programu. Tyto optimalizace lze charakterizovat jako transformace převádějící přeložený program ve vnitřním tvaru na ekvivalentní program v témže vnitřním tvaru, na jehož základě bude generován lepší cílový program. Takové optimalizační transformace odstraňují neefektivní nebo zbytečné operace, které byly v procesu vytváření vnitřního tvaru programu vygenerovány v důsledku neznalosti již zmíněných kontextových vazeb mezi proměnnými. Dále mění pořadí operací nebo samotné operace s cílem získat rychlejší program.

Poněvadž tyto transformace probíhají nad relativně strojově nezávislou reprezentací zdrojového programu, nazývají se *strojově nezávislé optimalizace*. Mezi typické strojově nezávislé optimalizace patří:

- odstranění výpočtů s konstantami (constant folding),
- eliminace společných výrazů,
- přesun invariantních výpočtů před cyklus,
- redukce ceny operace.

Optimalizace programu bývá velmi nákladnou činností překladače jak z hlediska prodloužení celkové doby překladu programu, tak z hlediska práce a úsilí při realizaci optimalizujícího překladače. Provedení určité optimalizace umožňuje obvykle provést další optimalizaci téhož nebo jiného typu a celý proces má tak iterační charakter. Nákladná je rovněž analýza taku údajů. Někdy se proto odlišují *lokální optimalizace*, které probíhají pouze nad sekvencemi operací (bez skoků a větvení) a *globální optimalizace*, které využívají kontextu celého programu a vyžadují globální analýzu toku údajů. Často strojově nezávislé optimalizace představují samostatný (volitelný) průchod překladače.

Základní kritéria, podle kterých se posuzuje úspěšnost optimalizace, jsou:

- délka generovaného cílového programu,
- rychlost výpočtu,
- požadovaná paměť pro údaje.

Ne vždy je nejdůležitější rychlost výpočtu. Na malých počítačích může být paměťová náročnost programu stejně důležitá, ne-li důležitější. Ani v případě, že máme k dispozici dostatek paměti, není vždy ekonomické provádět optimalizace rychlosti výpočtu. Ušetřená doba výpočtu programu v předpokládaném počtu jeho použití by měla převyšovat dobu, která připadla na optimalizaci. Tato podmínka nemusí být vždy splněna. Uvažme například typické studenské programy, které se často provádějí pouze jedenkrát. Z těchto všech důvodů existují v praxi různé verze téhož překladače nebo častěji, možnosti volby v témže překladači, které příslušejí různému stupni a různým typům optimalizace.

Platí dvě důležité zásady. Efekt nákladných optimalizací může být zcela zastíněn neefektivním generováním cílového programu. Druhá zásada říká, že podstatného zrychlení programu často dosáhneme změnou algoritmu. Takové vylepšení lze však stěží běžně očekávat v rámci optimalizace prováděné překladačem.

Podíl optimalizací a výběru algoritmu ilustruje názorně tento příklad [3].

Předpokládejme, že v našem programu pro seřazení n údajů podle velikosti jsme implementovali jednoduchý algoritmus (např. "bubblesort") a že po překladu získáme (neoptimalizovaný) strojový program B s délkou výpočtu $100\,n^2$ mikrosekund. Přeložíme-li zdrojový program optimalizujícím překladačem můžeme dosáhnout dvojnásobného zrychlení programu a získat program B_0 s délkou výpočtu $50\,n^2$ mikrosekund. Nyní předpokládejme, že místo optimalizací použijeme jiný algoritmus řazení (např. "quicksort") a že získáme (neoptimalizovaný) program Q s délkou výpočtu $500\,n\,\log(n)$ mikrosekund. Konkrétní délky výpočtu programů B, B_0 a Q pro $n = 100$ a $n = 1000$ udává tab. 9.1.

	n = 100	n = 1000
původní program	1 s	100 s
optimalizovaný program B_0	0,5 s	50 s
program Q s rychlejším algoritmem	0,1 s	1,5 s

Tab. 9.1: Efekt optimalizace a záměny algoritmu

Vidíme, že pro $n = 100$ je neoptimalizovaný program Q pětkrát rychlejší než optimalizovaný program B_0, pro $n = 1000$ je 33 krát rychlejší a pro n větší než 1000 je zrychlení programu řazení ještě výraznější.

V této kapitole se zaměříme na strojově nezávislé optimalizace nad programem ve vnitřním jazyce překladače. V příkladech a v popisech některých algoritmů budeme používat vnitřní jazyk tříadresových instrukcí zavedený v kap. 8. Problémy spojené se strojově závislými optimalizacemi jsou diskutovány v kap. 10, která pojednává o generování cílového programu.

Celá kapitola je rozdělena do čtyř vzájemně souvisejících částí, které se zabývají grafem toku řízení programu, základními typy strojově nezávislých optimalizací, lokální optimalizací v sekvencích příkazů a základy analýzy toku údajů.

9.1 Graf toku řízení programu

Nyní zavedeme důležitou reprezentaci vnitřního tvaru programu, která umožňuje zobrazit informaci o všech možných výpočetních posloupnostech na úrovni příkazů (instrukcí) vnitřního tvaru programu. Touto reprezentací je orientovaný graf, jehož vrcholy příslušejí nikoliv jednotlivým příkazům, jak je tomu u výpočetního grafu programu, ale posloupnostem příkazů, které se nazývají základní bloky. Hrany grafu toku řízení programu pak reprezentují relaci následnosti z hlediska potenciálního přenosu řízení mezi základními bloky.

Vyjádřeno přesněji, *základní blok* je posloupnost po sobě následujících příkazů, jejíž provádění může začít pouze prvním příkazem a neobsahuje, s výjimkou posledního příkazu, příkaz větvení, skoku nebo zastavení výpočtu. Za každých okolností se příkazy tvořící základní blok provedou vždy všechny.

Poněvadž řízení uvnitř základního bloku je přísně sekvenční, lze základní blok z hlediska toku řízení chápat jako nedělitelnou jednotku a základní blok tak může být reprezentován jediným vrcholem grafu.

Celý graf toku řízení programu, dále jen graf toku řízení, je orientovaný graf $GTR = (\mathcal{B}, H, \sigma)$, kde

$\mathcal{B}$ je množina základních bloků,

H je množina hran,

σ je zobrazení $\sigma : H \to \mathcal{B}\times\mathcal{B}$ takové, že $\sigma(h) = (B_i, B_j)$ právě když blok B_j je potenciálním následníkem bloku B_i z hlediska toku řízení, $h \in H$ a $B_i, B_j \in \mathcal{B}$.

Symboly , (B) a , $^{-1}(B)$ budeme označovat *množinu následníků* a *množinu předchůdců* vrcholu B:

$$, (B) = \{B' : \exists h \in H \text{ takové, že } \sigma(h) = (B, B')\}$$

$$, ^{-1}(B) = \{B' : \exists h \in H \text{ takové, že } \sigma(h) = (B, B')\}.$$

Důležitou charakteristikou grafu toku řízení je skutečnost, že hrany nenesou žádnou další informaci o povaze přechodu mezi základními bloky (zda je nepodmíněný nebo přísluší hodnotě true či false určitého booleovského výrazu). Z tohoto důvodu nemá smysl uvažovat případné násobné (rovnoběžné) hrany mezi dvěma vrcholy a proto není třeba hrany explicitně pojmenovávat. Každou hranu lze jednoznačně reprezentovat uspořádanou dvojicí (B_i, B_j).

Příklad 9.1. Na obr. 9.1 je posloupnost příkazů n-adresového vnitřního jazyka, která přísluší překladu příkazů

```
while (A <= B) and (C <=  D) or (X=Y) do
   A:=A+1;
X:=Y-1;
```

Příslušný graf toku řízení je zobrazen na obr. 9.2. ■

```
(1) if A > B goto 4        (6) A := T1
(2) if C > D goto 4        (7) goto 1
(3) goto 5                 (8) T2 := Y - 1
(4) if X <> Y goto 8       (9) X := T2
(5) T1 := A + 1
```

Obr. 9.1: Tříadresový kód

Dříve než popíšeme algoritmus konstrukce grafu toku řízení, zabývejme se otázkou rozkladu vnitřního tvaru programu do základních bloků.

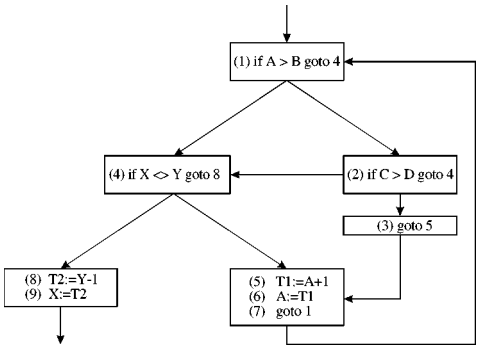
Pro identifikaci základních bloků je klíčovou otázkou nalezení jejich *úvodních* (prvních) příkazů. Je-li znám úvodní příkaz základního bloku, pak celý základní blok tvoří posloupnost po sobě jdoucích příkazů zahrnující všechny příkazy až po první výskyt úvodního příkazu jiného základního bloku.

Množinu úvodních příkazů základních bloků však lze nalézt snadno podle těchto pravidel:

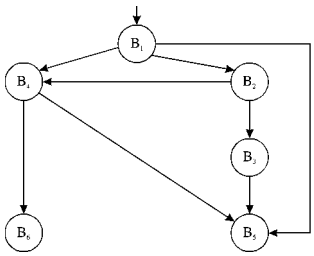
1) První příkaz programu je úvodní příkaz.

2) Příkaz, na který je přenášeno řízení podmíněným nebo nepodmíněným skokem, je úvodní příkaz.

a) detailní tvar



b) redukovaný tvar



Obr. 9.2: Graf toku řízení

3) Příkaz bezprostředně následující za podmíněným příkazem je úvodní příkaz.

Po vytvoření všech základních bloků mohou ve výchozím programu (ve vnitřním tvaru) existovat příkazy, které nepatří do žádného základního bloku. Tyto příkazy jsou z hlediska toku řízení programu nedostupné (nemohou být nikdy provedeny) a mohou být proto překladačem ignorovány při generování cílového programu.

Algoritmus 9.1. (Konstrukce grafu toku řízení)
Vstup: Vnitřní tvar programu.
Výstup: Graf toku řízení.
Metoda:

- Vytvoř množinu $\mathcal{B} = \{B_1, B_2, \ldots, B_n\}$ základních bloků programu. Tato množina tvoří množinu vrcholů hledaného grafu.
- Vrchol B_1 odpovídající základnímu bloku, který obsahuje první příkaz programu, je počátečním vrcholem grafu toku řízení.
- Pro libovolné dva vrcholy B_i a B_j je (B_i, B_j) hranou grafu toku tehdy a jen tehdy, jestliže platí jedna z podmínek:
  a) posledním příkazem bloku B_i je nepodmíněný nebo podmíněný příkaz skoku na první příkaz bloku B_j,
  b) posledním příkazem bloku B_i je podmíněný příkaz skoku, za kterým bezprostředně následuje první příkaz bloku B_j.

Graf toku řízení je ve fázi optimalizace využíván k několika důležitým činnostem při analýze optimalizovaného programu i syntéze "optimálního" programu. Předně umožňuje, jak jsme již uvedli, nalézt nedostupné úseky programu. Dále se používá k detekci cyklů. Optimalizace provedené nad příkazy základních bloků, které tvoří cyklus, jsou pro zrychlení cílového programu nejvýraznější. V této souvislosti se uvádí často *pravidlo "90-10"*, které říká, že statisticky pouze 10% kódu celého programu spotřebuje 90% celkové doby výpočtu programu. Těchto 10% kódu samozřejmě přísluší cyklům.

Graf toku řízení hraje rovněž důležitou úlohu při globální analýze toku údajů. Údajové vazby mezi proměnnými různých základních bloků jsou závislé na hodnotách, které tyto proměnné mají při provádění programu. Graf toku řízení stanovuje možné cesty výpočtu. Konečně některé optimalizační transformace mění pořadí tak, že mění graf toku řízení. Typickým příkladem této transformace je přesun invariantních výpočtů v cyklu před začátek cyklu.

9.2 Základní typy strojově nezávislých optimalizací

V tomto článku se seznámíme s nejčastěji používanými typy strojově nezávislých optimalizací. Budeme se zabývat otázkou výsledného efektu, nikoliv metodami, kterými je tohoto efektu dosaženo. V rámci rozsáhlejšího příkladu budeme sledovat typické optimalizační transformace. Rovněž získáme představu o tom, které dodatečné informace budou transformační algoritmy požadovat.

Příklad 9.2. Uvažujme úsek programu, který provádí součet vektorů. Výsledný vektor $C = (C_1, \ldots, C_{2N+1})$ je součtem vektoru $A = (A_1, \ldots, A_{2N+1})$ a $B = (B_1, \ldots, B_{2N+1})$, kde N je deklarovaná konstanta:

```
I := 1;
while I <= 2*N+1 do
begin
    C[I] := A[I] + B[I];
    I := I+1
end
```

Podívejme se, jak může vypadat vnitřní tvar tohoto programu v jazyce n-adresových instrukcí. Přitom předpokládáme, že na základě tohoto vnitřního tvaru bude generován cílový program pro počítač se slabikovou organizací vnitřní paměti (1 slovo = 4 slabiky). Operand `adr(M)` reprezentuje adresu začátku pole `M`.

```
(1) T1 := 1                (11) T9 := T7[T8]
(2) I  := T1               (12) T10 := T6 +  T9
(3) T2 :=  2*N             (13) T11 := adr(C) -  4
(4) T3 :=  T2+1            (14) T12 := 4*I
(5) if I > T3 goto 19      (15) T11[T12] := T10
(6) T4 : = adr(A) - 4      (16) T13 := I+1
(7) T5 :=  4*I             (17) I   := T13
(8) T6 := T4[T5]           (18) goto 3
(9) T7 := adr(B) - 4       (19)
(10) T8  := 4*I
```

Obr. 9.3: Vnitřní tvar programu

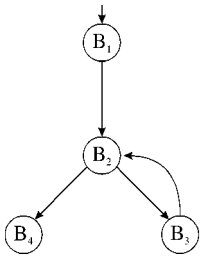
Na první pohled obsahuje generovaný vnitřní tvar programu řadu operací, které vedou k delšímu a pomalejšímu cílovému programu než je nutné. Přesto nelze říci, že příčinou těchto neefektivních operací je programátorova neobratnost při psaní programu nebo triviálně provedený překlad do vnitřního tvaru. Základní bloky tohoto programu a jeho graf toku řízení jsou popsány na obr. 9.4.

9.2.1 Odstranění výpočtů s konstantami

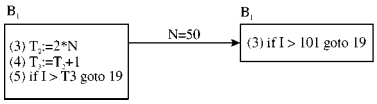
V programu v příkl. 9.2 jsme pro součet dvou vektorů předpokládali, že N je v programu deklarovaná konstanta. Proto může být operace (3) a následně i operace (4) provedena při překladu. Je-li např. `const N = 50`, pak optimalizace odstraňující výpočty s konstantami povede k eliminaci operací (3) a (4) a k modifikaci operace (5) do tvaru:

Rozhodnutí, zda danou operaci můžeme provést v době překladu, nemusí být obecně jednoduché. Je jasné, že operace `A := B op C` dává konstantní výsledek vyčíslitelný v době překladu, pokud `B` i `C` jsou konstanty nebo identifikátory konstant. Abychom však mohli na určité místo `P`, kde se proměnná `A` používá, dosadit hodnotu výrazu `B op C`, musíme vědět, že neexistuje jiné přiřazení hodnoty do proměnné `A`, které definuje rovněž hodnotu proměnné `A` v místě `P`. Prakticky to předpokládá, že pro dané použití proměnné A jako operandu známe všechna taková místa ve vnitřním tvaru programu, kde je proměnné `A` přiřazována hodnota a tato přiřazení mohou ovlivnit hodnotu proměnné `A` v jejím daném použití. Tento problém, označovaný jako hledání platných definic pro dané použití proměnné, je typickou úlohou globální analýzy taku údajů a budeme se jím zabývat v podkapitole 9.4.

Základní blok	Příkazy
B_1	$(1) - -(2)$
B_2	$(3) - -(6)$
B_3	$(7) - -(18)$
B_4	$(19) - -$



Obr. 9.4: Graf toku řízení programu pro výpočet součtu vektorů



Obr. 9.5: Odstranění výpočtů s konstantami

Dosazení konstantních hodnot a provedení operací s nimi v době překladu může vést k zjednodušení grafu toku řízení a k nalezení nedostupných příkazů vnitřního tvaru programu. Uvažujme např. příkaz

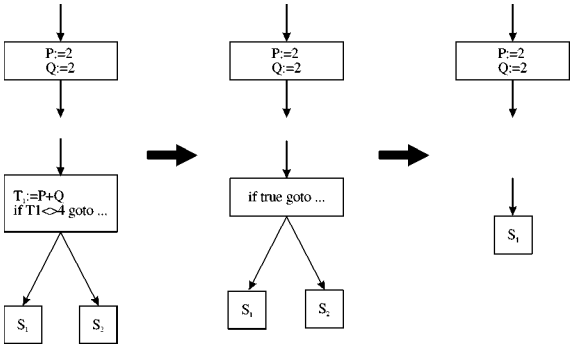
```
if P + Q = 4 then S1 else S2,
```

kde `P` a `Q` jsou konstanty nebo proměnné, které nabývají pro použití v uvedeném příkaze vždy hodnotu 2. Pak lze provést transformaci podle obr. 9.6, která odstraňuje nedostupný blok odpovídající příkazu `S2`.

Dosazení konstantních hodnot rovněž umožní aplikovat některé základní identity jako např:

```
A + 0 = A
A * 1 = A
A or true = true
A and true = A
```

Uvedená optimalizační transformace je důležitá, protože přináší několik zřejmých výhod a nemá žádné nevýhody. Je zvláště efektivní v případě, že podprogramy jsou překládány jako



Obr. 9.6: Odstranění nedostupných operací

otevřené. Skutečnými parametry podprogramů jsou mnohdy konstanty a při jejich dosazení do podprogramů vniká mnoho příležitostí k optimalizaci uvedeného typu.

9.2.2 Odstranění redundantních operací

Operaci nebo posloupnost operací vnitřního tvaru programu nazveme *redundantní*, je-li její výsledná hodnota již přístupná jako výsledek předchozí instrukce nebo posloupnosti instrukcí. Typickým zástupcem redundantních operací jsou výpočty společných podvýrazů — podvýrazů, které dávají stejné hodnoty. Výpočty společných podvýrazů jsou v menší míře důsledkem shodných výrazů objevujících se ve zdrojovém programu pro zlepšení dokumentace nebo čitelnosti programu. Mnoho programátorů dá např. přednost zápisu

```
A[i+1] := B[i+1]  + C[i+1]
```

před zápisem

```
k :=  i+1; A[k] := B[k] + C[k]
```

Hlavním zdrojem výpočtů společných podvýrazů jsou indexované proměnné. Tradičně se uvádí příklad přiřazovacího příkazu (z programu Gauss-Seidlovy iterační metody řešení Laplaceovy rovnice ve Fortranu):

```
A(I,J,K) = (A(I,J,K-1) + A(I,J,K+1) + A(I,J-1,K) - A(I,J+1,K) +
           A(I-1,J,K) + A(I+1,J,K))/6.0
```

Jsou-li `d1` a `d2` velikosti prvních dvou dimenzí pole `A`, pak po překladu tohoto příkazu je sedmkrát generován výraz `I+d1*(J+d2*K)` (v kombinaci s různými konstantami). Šest výskytů (a výpočtů) tohoto výrazu je redundantních.

Obecně vyžaduje problém nalezení a odstranění shodných podvýrazů analýzu toku údajů. Např. při překladu příkazů

```
A := B + C + D
...
E := B + C + F
```

lze generovat posloupnost operací

```
T := B + C
A := T + D
...
E := T + F
```

eliminující společný výraz `B+C` pouze za těchto podmínek:

(1) mezi prvním a druhým výskytem výrazu `B+C` není žádné přiřazení hodnoty proměnné `B` nebo `C`,

(2) neexistuje cesta v grafu toku řízení, která by umožňovala provést příkaz `E:=T+F`, aniž by byl proveden příkaz `T:=B+C`.

Podmínka (2) je automaticky splněna v rámci základního bloku. Existuje metoda eliminace společných podvýrazů v rámci základního bloku, která je relativně velmi efektivní a která umožňuje odstraňovat i společné výrazy, které nejsou textově identické. Např. v posloupnosti

```
A := B + C
R := B
S := C
D := R + S
```

jsou to výrazy `B+C` a `R+S`. Touto metodou se budeme zabývat v podkapitole 9.3.

Za redundantní operace můžeme považovat rovněž některá přiřazení generovaná při zpracování sémantiky. Např. při překladu příkazu `I:=I+1` jsou generovány dvě n-adresové instrukce

```
T := I+1
I := T,
```

z nichž první je generována pro <výraz> a druhá pro <přiřazovací příkaz>. Pokud výraz `I+1` není společným podvýrazem vzhledem k příkazům, které jsou v kontextu operace `T:=I+1`, pak je pomocná proměnná `T` a přiřazení `I:=T` redundantní a uvedená dvojice může být nahrazena jedinou instrukcí `I:=I+1`.

Pro optimalizaci tohoto typu platí v podstatě totéž, co pro odstranění společných výrazů. V kontextu celého programu vyžaduje eliminace příkazů `A:=B` globální analýzu toku údajů. Při lokální optimalizaci v rámci základního bloku je odstranění příkazů typu `A:=B` součástí transformace odstraňující společné výrazy.

Vrátíme-li se k příkl. 9.2 a provedeme-li popsané optimalizace programu z obr. 9.3 včetně odstranění výpočtů s konstantami, dostaneme optimalizovaný vnitřní tvar programu na obr. 9.7.

Redukce počtu instrukcí a počtu pomocných proměnných je důsledkem dosazení konstanty `N`=50, odstranění zbytečných přiřazení (příkazů (1), (2), (16) a (17) na obr. 9.3) a eliminace společného výrazu 4*I (příkazy (10) a (14) na obr. 9.3).

Výhody optimalizačních transformací popsaných v tomto odstavci jsou zjevné. Přinášejí zrychlení i redukci délky programu, poněvadž pro redundantní operace nejsou generovány (a tedy ani prováděny) žádné instrukce. Nevýhodou mohou být větší nároky na využití registrů počítače.

```
1) I := 1                    8) T10 :=  T6+T9
2) if I > 101 goto 13        9) T11 := adr(C)-4
3) T4 := adr(A)-4           10) T11[T5] := T10
4) T5 := 4*I                11) I := I+1
5) T6 : = T4[T5]            12) goto 2
6) T7 := adr(B)-4           13)
7) T9 := T7[T5]
```

Obr. 9.7: Vnitřní tvar programu po optimalizaci

9.2.3 Přesun operací

Přesun operací je optimalizační transformace, která mění pořadí (vybraných) operací s cílem:

a) dosáhnout méně častého provádění přesunutých operací,

b) zkrátit délku programu tím, že operace, společné všem možným cestám programu, se generují pouze jedenkrát.

Nejčastěji se objevuje tato optimalizační transformace v souvislosti s optimalizací cyklů při odstraňování invariantních výpočtů v cyklu — *invariantů cyklu.*

Invariantem cyklu nazýváme takovou posloupnost operací, které dávají při každém průchodu cyklem stejný výsledek. Zdroje invariantů cyklu jsou stejné jako zdroje společných výrazů. Neuvažujeme-li začátečnické programátorské prohřešky, pak invarianty cyklu vznikají tak, že programátor nechce snižovat čitelnost programu (např. rozkládat booleovský výraz řídící cyklus while nebo repeat na podvýrazy, které jsou invariantní v cyklu a ty vyčíslovat před cyklem). Významnějším zdrojem jsou však ty operace generované překladačem, které programátor nemůže ovlivňovat, jako jsou operace generované při překladu indexovaných proměnných.

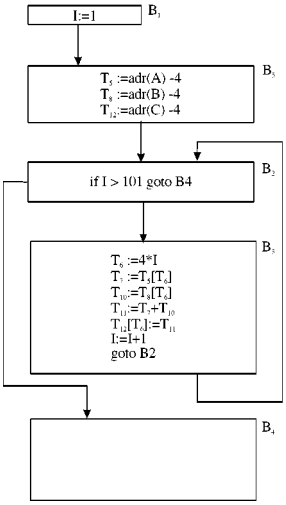
Ilustrujme nejdříve tuto transformaci na příkl. 9.2. Předpokládejme, že paměť pro vektory `A`, `B`, `C` je přidělena dynamicky v zásobníku. Pak výrazy typu `adr(A)-4` tvoří invariant cyklu a mohou byt vyčísleny pouze jednou před vlastním cyklem. Výsledek přesunu invariantu je uveden na obr. 9.8. Přesunutý invariant tvoří základní blok B_5.

Poznamenejme, že pokud by paměťový prostor pro vektory `A`, `B`, `C` byl přidělován staticky, pak lze uvedené výrazy vyčíslit v době překladu. Naopak, při dynamickém přidělování s použitím hromady, nelze tyto výrazy považovat za invarianty cyklu, poněvadž případné čištění paměti (garbage collection) může způsobit změnu uložení některého vektoru v paměti během provádění cyklu.

Přesun invariantu cyklu je obecně velmi účinná, ale také nákladná transformace. Vyžaduje tyto činnosti:

- nalezení základních bloků, které vytvářejí cyklus,
- nalezení invariantu cyklu,
- vlastní přesun invariantu z cyklu.

Nalezení cyklů v programu vyžaduje analýzu grafu toku řízení, při níž se hledají silně souvislé komponenty grafu. Obvykle se však kromě silné souvislosti požaduje, aby existoval pouze jeden "vstupní blok" cyklu, tedy jediný základní blok, kterým vedou všechny možné cesty z vnějšku

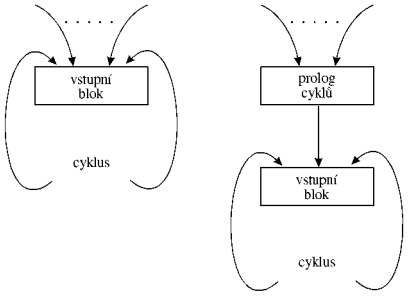


Obr. 9.8: Přesun invariantu cyklu

cyklu do libovolného základního bloku cyklu. Tuto podmínku automaticky splňují cykly, které vzniknou překladem strukturovaných příkazů cyklu (while, repeat, for), ne však cykly, které vzniknou použitím podmíněných a nepodmíněných skoků ve zdrojovém programu. Jediný vstupní blok usnadňuje hledání invariantu cyklu a hlavně jeho přesunutí z cyklu. V případě, že existuje více vstupních bloků cyklu, je nutné předřazovat invariant před každý takový blok, což by mohlo vést k nežádoucímu prodloužení programu.

Nalezení invariantu cyklu vyžaduje informaci o definicích (místech, kde je proměnné přiřazena hodnota) a použitích (místech, kde je hodnota proměnné použita jako operand) všech proměnných cyklu. Operace `A op B` je invariantní, jestliže proměnné `A` a `B` nemají žádnou definici v blocích tvořících cyklus, nebo jejich definice v těchto blocích přiřazují hodnoty v cyklu invariantní (výsledky invariantních operací). Hledání invariantu cyklu tedy vyžaduje globální analýzu toku údajů a je opět procesem, který má iterační charakter.

Vlastní přesun invariantu se zdá být nejsnažším úkonem. Vytvoříme nový základní blok, nazývaný *prolog cyklu*, do něhož přesuneme invariantní operace v pořadí, jak byly detekovány při hledání invariantu. Situaci ilustruje obr. 9.9.



Obr. 9.9: Vytvoření prologu cyklu

Přesun invariantní operace je však komplikován tím, že ne každá operace může být přesunuta do prologu cyklu aniž by nebyl porušen základní předpoklad optimalizace — zachování ekvivalence se zdrojovým programem. Abychom snáze pochopili podmínky, které umožňují přesun, uvažujme příklad nesprávné optimalizace na obr. 9.10. Na obr. 9.10(b) je proveden přesun invariantu `I:=2` před cyklus.

Transformace na obr. 9.10(b) není správná proto, že zatímco v programu na obr. 9.10(a) může být proměnné `J` v bloku B_5 přiřazena jak hodnota 1, tak hodnota 2, v programu na obr. 9.10(b) je proměnné J přiřazena vždy hodnota 2. Příčinou tohoto rozdílu je existence cesty $B_1, B_2, \ldots, B_2, B_4, B_5$ v programu 9.10(a), na které není nikdy provedena invariantní operace `I:=2`. Můžeme tedy zformulovat první podmínku přesunu.

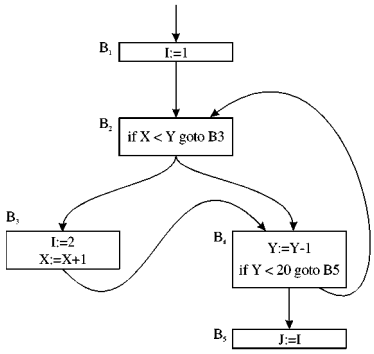
1) Invariantní operace `A := B op C` nemůže být přesunuta do prologu cyklu, není-li blok, v němž se nachází, součástí každé cesty vedoucí ven z cyklu. S rizikem, že odstranění invariantní operace může ve skutečnosti prodloužit výsledný program, lze uvedenou podmínku ignorovat v případě, že se proměnné `A` nepoužívá mimo cyklus, což je častý případ generovaných pomocných proměnných.

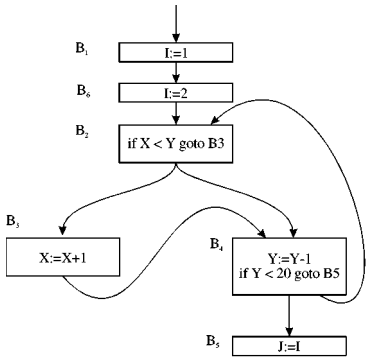
Další dvě podmínky jsou podobného charakteru. Invariantní operace `A := B op C` nemůže být přesunuta do prologu cyklu, pokud:

2) v cyklu existuje další definice proměnné `A`,

3) v cyklu existuje použití proměnné `A`, k němuž se vztahuje jiná definice proměnné `A` (než `A := B op C`).

Chybný přesun invariantu při nesplnění podmínky 2) můžeme ilustrovat na obr. 9.10(a) za předpokladu, že blok B_2 obsahuje příkazy:





Obr. 9.10: Příklad nesprávného přesunu invariantní operace

```
B2:   I := 3
      if X < Y goto B3
```

Ačkoliv je splněna podmínka 1), invariantní příkaz `I:=3` nelze přesunout před cyklus, protože proměnná `J` při provedení posloupnosti $B_1, B_2, B_3, B_4, B_2, B_4, B_5$ základních bloků nabude v bloku B_5 hodnoty 3, avšak po přesunu příkazu `I:=3`, při provedení téže posloupnosti, bude mít `J` hodnotu 2.

Podobně lze ilustrovat význam podmínky 3). Předpokládejme, že blok B_4 na obr. 9.10(a) bude vytvořen příkazy:

```
B4:   K  := I
      Y  := Y-1
      if Y < 20 goto B5
```

K použití proměnné `I` v příkaze `K := I` se vztahuje definice `I` v bloku B_1 i v B_3. Po přesunutí příkazu `I := 2` do bloku B_6 (obr. 9.10(b)) však proměnná `K` již nemůže získat hodnotu 1. Vidíme tedy, že ani vlastní přesun invariantních operací není levnou transformací, protože vyžaduje analýzu toku řízení i toku údajů.

Přesun invariantních operací před cyklus přináší velké úspory strojového času v době výpočtu, avšak pouze v případě že cyklus proběhne mnohokrát. V cyklu, který ošetřuje vyjímečné nebo nestandardní situace, které většinou neprobíhají vůbec, způsobuje přesun invariantů prodloužení výpočtu (invariant se provede, i když se cyklus neprovede).

Optimalizační transformace přesunu operací se uplatňuje rovněž ve spojení s odstraňováním společných výrazů.

Uvažme příklad podmíněného příkazu:

```
if X > 0 then A := B + C + D else A := B + C - D
```

s grafem toku řízení na obr. 9.11(a).

Vidíme, že alternativy tohoto příkazu obsahují stejný podvýraz `B+C`, nikoliv však ve smyslu odst. 9.3.2, poněvadž pro žádný jeho výskyt neplatí, že hodnota výrazu je již k dispozici.

Transformace zobrazená na obr. 9.11(b) redukuje délku programu, poněvadž instrukce pro výpočet výrazu `B+C` budou generovány pouze jedenkrát. Přesun stejných operací před nejbližšího společného předchůdce v grafu toku programu se nazývá *vytažení operací* (code hoisting). Podmínky, za kterých může být tato transformace provedena, není obtížné stanovit:

1) přesouvané operace musí dávat ve všech větvích vždy stejné výsledky,

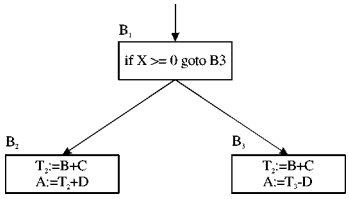
2) neexistuje cesta v grafu toku řízení, která by neobsahovala základní blok, do něhož byly operace přesunuty a zároveň obsahovala základní blok, z něhož byl přesun proveden.

9.2.4 Optimalizace indukčních proměnných

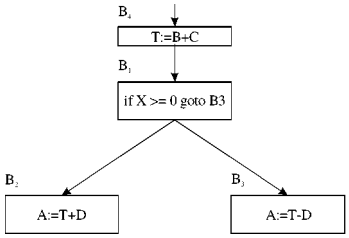
Poslední optimalizační transformací, kterou budeme ilustrovat, je odstraňování redundance způsobené existencí proměnných, jejichž hodnoty jsou vzájemně v průběhu výpočtu vázány jistotou lineární funkcí. Tato transformace se rovněž uplatňuje v cyklech.

Základní indukční proměnnou cyklu nazveme proměnnou `I`, jejíž všechny definice uvnitř cyklu jsou tvaru `I:=I` $\pm$ `C`, kde `C` je konstanta nebo jméno proměnné, jejíž hodnota se v cyklu nemění. *Indukční proměnná* `J` je buď základní indukční proměnná, nebo proměnná, ke které existuje základní indukční proměnná `I` taková, že pro všechny hodnoty i, j proměnných `I`, `J` platí $j = f(i)$ a f je lineární funkce.

(a)



(b)



Obr. 9.11: Vytažení operací

Zdrojem idukčních proměnných jsou obvykle proměnné, jež slouží jako řídící proměnné cyklů nebo proměnné určené k indexování. Jejich hodnoty tvoří velice často aritmetickou posloupnost.

Pokud k dané základní indukční proměnné existuje třída odvozených indukčních proměnných, pak lze výpočty s těmito proměnnými optimalizovat ze dvou hledisek takovým způsobem, že:

a) vybereme jedinou indukční proměnnou, nad kterou budeme realizovat výpočet a ostatní indukční proměnné vyloučíme,

b) realizujeme výpočet hodnot indukční proměnné operacemi, které jsou časově méně náročné.

Podívejme se nyní, jak bude vypadat tato transformace v programu součtu vektorů na obr. 9.8. V bloku B_3 existuje základní indukční proměnná `I` a k ní příslušející indukční proměnná `T5`. Hodnoty těchto proměnných jsou v cyklu vázány vztahem `t5 = 4 * i`. Za předpokladu, že se hodnoty proměnné `I` po ukončení cyklu (`I`=102) v bloku B_4 a jeho následnících nepoužívá, můžeme provést tyto modifikace příkazů vnitřního tvaru programu:

1) před cyklem (blokem B_2) inicializovat proměnnou `T5` (`T5:=0`) a nahradit operaci `T5:=4*I` operací `T5:=T5+4`,

2) upravit podmínku pro test na konec cyklu tak, aby závisela na hodnotě proměnná `T5` a nikoliv na hodnotě proměnné `I`,

3) odstranit příkaz `I:=I+1`.

Výsledný vnitřní tvar programu získaný po této optimalizaci je zobrazen na obr. 9.12.

Nahrazení operace násobení operací sečítání (operace (7) na obr. 9.12) je speciálním případem obecnější optimalizační transformace nazývané *redukce ceny operace* (strength reduction). Na většině počítačů je operace násobení několikanásobně delší než sečítání, proto jejich výměna může při cyklických výpočtech přinést nezanedbatelný zisk.

Výraznějším příkladem redukce ceny operace je případ nahrazení operace výpočtu délky řetězce daného zřetězením řetězců r_1 a r_2

$$LENGTH(r_1.r_2)$$

operací

$$LENGTH(r_1) + LENGTH(r_2).$$

Úplná eliminace proměnné `I` ve vnitřním tvaru programu na obr. 9.12 byla možná pouze za předpokladu, že `I` není po opuštění cyklu "živá," tj. její hodnoty se nepoužije, aniž by předcházela nová definice proměnné `I`. I když je to ve skutečnosti velmi pravděpodobné, je nutné tuto vlastnost ověřit, což vyžaduje analýzu toku údajů.

Optimalizací indukčních proměnných jsme uzavřeli diskusi a ukázky častých optimalizačních transformací. I když uvedený soubor transformací není úplný, můžeme ho pokládat za dostatečně reprezentativní pro většinu vyšších programovacích jazyků. V tab. 9.2 je vyčíslen souhrný přínos uvedených optimalizačních transformací pro demonstrační příklad 9.2.

	Původní program	Optimalizovaný program
délka	19	12
počet proměnných	16	8
počet prováděných operací	1820	813
z toho operací násobení	101	0

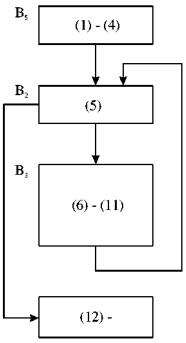
Tab. 9.2: Srovnání neoptimalizovaného a optimalizovaného programu pro součet vektorů

9.3 Optimalizace v základním bloku

V tomto článku se seznámíme s metodou lokálních optimalizací v rámci posloupnosti příkazů tvořících základní blok. Ve srovnání s globálními optimalizacemi nad celým programem nebo souborem základních bloků je tato metoda efektivní, poněvadž nevyžaduje globální analýzu toku údajů. Umožňuje (v rámci základního bloku) eliminovat společné podvýrazy, odstraňovat zbytečná přiřazení typu `A := B` a redukovat počet generovaných pomocných proměnných. Zvláště poslední ze jmenovaných optimalizací se dotýká využití registrů počítače, a proto je tato metoda často implementována jako součást generátoru cílového programu.

Základem popisované metody je reprezentace základního bloku orientovaným acyklickým grafem a zpětná transformace tohoto grafu dávající optimalizovaný základní blok. Dříve, než

```
 1) T4  := adr(A)-4
 2) T7  := adr(B)-4
 3) T11 := adr(C)-4
 4) T5  := 0
 5) if T5 > 400 goto 12
 6) T5  := T5 + 4
 7) T6  := T4[T5]
 8) T9  := T7[T5]
 9) T10 := T6 + T9
10) T11[T5] := T10
11) goto 5
12)
```



Obr. 9.12: Program s optimalizovanými indukčními proměnnými

uvedeme vlastnosti této reprezentace a popíšeme konstrukci příslušného grafu, podívejme se na ilustrační příklad na obr. 9.13, kde je zobrazen

a) zdrojový tvar základního bloku,

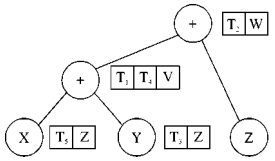
b) jeho vnitřní tvar po překladu,

c) reprezentace tohoto vnitřního tvaru acyklickým orientovaným grafem,

d) zpětně vytvořený optimalizovaný základní blok.

```
(a)                (b)                              (c)
      W:=X+Y+Z        1) T1:=X+Y   6) T4:=X+Z          1) V:=X+Y
      Z:=Y;           2) T2:=T1+Z  7) V:=T4            2) W:=V+Z
      V:=X+Z;         3) W:=T2     8) T5:=X            3) Z:=X
      Z:=X            4) T3:=Y     9) Z:=T5
                      5) Z:=T3
```

(d)



Obr. 9.13: Ilustrační příklad

9.3.1 Definice a vlastnosti grafu reprezentujícího základní blok

Uvažujme základní blok ZB. Graf G_{ZB} je orientovaný acyklický graf (případně multigraf), jehož vrcholy mají dvě ohodnocení h_1, h_2. Popišme nejdříve ohodnocení h_1.

1) Listy grafu, tj. vrcholy, z nichž nevychází žadná hrana, jsou ohodnoceny konstantami nebo jmény proměnných, které reprezentují vstupní údaje pro výpočty realizované základním blokem. Hodnoty těchto proměnných jsou definovány vně bloku ZB.

2) Zbývající vrcholy, tj. vnitřní vrcholy grafu, jsou ohodnoceny operátory jednotlivých operací základního bloku.

3) Hrany vycházející z vnitřního vrcholu v určují operandy (vrcholy reprezentující tyto operandy) operátoru $h_1(v)$.

Ohodnocení h_2 přiřazuje ke každému vrcholu seznam jmen proměnných (může být prázdný), které nesou hodnotu vypočítanou daným vrcholem nebo přiřazenou k danému vrcholu. Přiřazení

těchto proměnných vrcholům grafu je jednoznačné. Poznamenejme, že popsaná reprezentace základního bloku je velmi příbuzná s vnitřním tvarem ve formě trojic. Jestliže se v některé operaci vyskytnou shodné operandy, pak z příslušného vrcholu vycházejí násobné hrany.

Mechanismus vytváření ohodnocení h_2 je určitou formou simulace výpočtu operací základního bloku. Dynamicky zařazuje a vyřazuje prvky seznamů příslušející vrcholům grafu podle definic popsaných jednotlivými operacemi. Tímto mechanismem je realizována analýza toku údajů v rámci základního bloku, při níž jsou nalezeny společné podvýrazy. Na základě výsledného ohodnocení h_2 pak mohou být eliminovány redundantní přiřazení a generované pomocné proměnné.

9.3.2 Konstrukce grafu G_{ZB}

Algoritmus konstrukce grafu G_{ZB} prochází jednotlivé operace tvořící základní blok v daném pořadí a podle dále popsaného typu operace přidává do grafu G_{ZB} nové vrcholy a hrany a vytváří či modifikuje ohodnocení vrcholů. Na počátku je graf G_{ZB} prázdný.

Budeme rozlišovat tři typy operací základního bloku

a) `A := B op C`, `op` je binární operátor,

b) `A := op B`, `op` je unární operátor a

c) `A := B`

a odvolávat se na ně jako na typ a), b), nebo c).

Složky grafu $G_{ZB} = (U, H, \sigma)$ mají obvyklý význam:

U je množina vrcholů,

H je množina hran,

σ je incidenční relace $\sigma : H \to U \times U$.

Hranu h tedy reprezentujeme uspořádanou dvojicí vrcholů $\sigma(h) = (u_1, u_2), u_1, u_2 \in U$.

Při zpracování operací typu a) a b) algoritmus vytváření grafu G_{ZB} nejdříve hledá

1) vrcholy reprezentující operandy operace,

2) případně celý podstrom reprezentující operandy i operátor a identifikuje tak společný podvýraz.

Provedení akce 1) spolu s vytvořením nového vrcholu reprezentujícího hledaný operand v případě neúspěšného hledání popíšeme procedurou *URCI_VRCHOL*.

procedure *URCI_VRCHOL*(*OPERAND*, V);
{procedura v grafu G_{ZB} nalezne nebo vytvoří vrchol V
s ohodnocením $h_1(V) = OPERAND$}
begin
  if $\exists u \in U(OPERAND = h_2(u))$**then** $V := u$
  else
    if $\exists list \in U(OPERAND = h_1(list))$ **then** $V := list$
    else begin {je třeba vytvořit nový vrchol}

$V := NOVY\_VRCHOL$;
$h_1(V) := OPERAND$
end
end

Použitá procedura *NOVY_VRCHOL* vytváří další vrchol grafu G_{ZB} (v tomto případě budoucí list grafu) a zařadí tento vrchol do stávající množiny vrcholů U. Pořadí testů na existenci vrcholu u v grafu G_{ZB} je důležité. Nejprve je třeba ověřit, zda neexistuje vrchol, jehož značení h_2 obsahuje prvek *OPERAND*. Pokud ano, pak při zpracování předcházela operace definující *OPERAND* (tj. *OPERAND*:= ...) a pak takový vrchol, a nikoli list u se značením $h_1(u) = OPERAND$, reprezentuje hodnotu hledaného operandu operace.

Celý základní krok algoritmu vytváření grafu G_{ZB} je popsán na obr. 9.14. Kromě uvedených procedur používá procedury:

VYTVOR_HRANU(U, V), která vytváří hranu (U, V),

ODSTRAN(P, V), která odstraní z $h_2(V)$ prvek P,

DOPLN(P, V), která doplní do $h_2(V)$ prvek P.

Na obr. 9.15 je ilustrováno vytváření grafu GZB pro základní blok z obr. 9.13(b) po jednotlivých krocích.

Uvedený algoritmus je jednoduchý a efektivní, avšak nemusí dát ekvivalentní reprezentaci základního bloku v případě, že základní blok obsahuje operace indexování, výskyty ukazatelů a volání procedur.

Uvažme např. základní blok tvořený příkazy

```
(1) X := A[I]
(2) A[J] := Y
(3) Z := A[I]
```

Popsaný algoritmus reprezentuje tento základní blok grafem na obr. 9.16, v němž je výraz `A[I]` považován za společný podvýraz.

Na základě tohoto grafu bude vytvořen optimalizovaný blok

```
(1) X := A[I]
(2) Z := X
(3) A[J] := Y
```

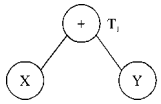
Tento blok však není ekvivalentní výchozí posloupnosti příkazů. V případě, že `I=J` a `Y` $\neq$ `A[J]` budou hodnoty přiřazené proměnné `Z` odlišné. Příčinou uvedené reprezentace základního bloku je jev, který je v angličtině označován termínem "aliasing," což znamená, že tatáž proměnná je reprezentována různými jmény. V našem příkladě, v případě `I=J`, reprezentují indexované proměnné `A[I]` a `A[J]` stejnou proměnnou. Výraz `A[I]` v příkazech (1) a (3) nelze považovat za společný, poněvadž příkazem (2) může být změněna jeho hodnota, a to aniž se změní jeho "operandy" `adr(A)` nebo `I`. Aby uvedený algoritmus konstrukce grafu G_{ZB} dával skutečně ekvivalentní reprezentaci základního bloku, je třeba postupovat takto:

V okamžiku, kdy je prvku pole `A` přiřazena hodnota, musíme dodat ke všem vrcholům, které reprezentují operátor s operandem `adr(A)`, informaci, jež vyloučí tyto vrcholy jako možné reprezentanty společných výrazů. Na základě této informace, bude při zpracování příkazu (3) v diskutovaném příkladě vytvořen nový vrchol $V3$ reprezentující proměnnou `A[I]`, jak ukazuje obr. 9.17.

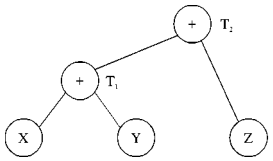
```
begin
  {zpracování jedné operace typu
      (a) A := B op C,
      (b) A := op B,
      (c) A := B
   základního bloku}
  URCI_VRCHOL(B, v1);
  case typ_operace of
    a : begin
      URCI_VRCHOL(C, v2);
      if (∃v ∈ U(h1(v) = op ∧ h1, h2 ∈ H(σ(h1) = (v, v1) ∧ σ(h2) = (v, v2))))
      then {společný podvýraz B op C}
        KOREN := v
      else begin
        KOREN := NOVY_VRCHOL;
        h1(KOREN) := op;
        VYTVOR_HRANU(KOREN, v1);
        VYTVOR_HRANU(KOREN, v2);
      end
    end;
    b : if (∃v ∈ U(h1(v) = op ∧ ∃h ∈ H(σ(h1) = (v, v1))))
      then {společný podvýraz op B}
        KOREN := v
      else begin
        KOREN := NOVY_VRCHOL;
        h1(KOREN) := op;
        VYTVOR_HRANU(KOREN, v1)
    end;
    c : KOREN := v;
  end;
  {aktualizace ohodnocení h2 vrcholů}
  if (∃v ∈ U(A ∈ h2(v)) then
    ODSTRAN(A, v);
  DOPLN(A, KOREN);
end
```

Obr. 9.14: Základní krok vytváření grafu G_{ZB}

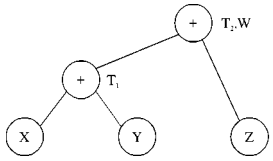
(1)



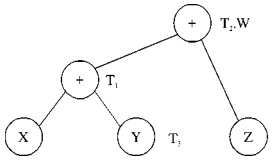
(2)



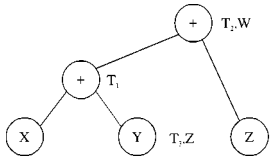
(3)



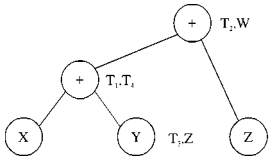
(4)



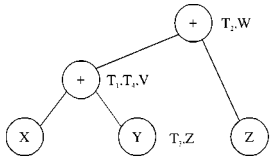
(5)



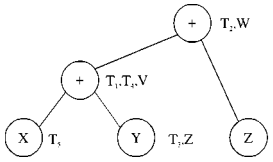
(6)



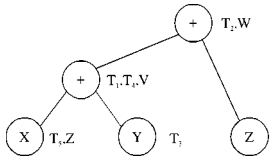
(7)



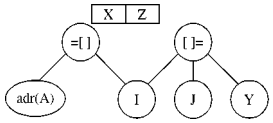
(8)



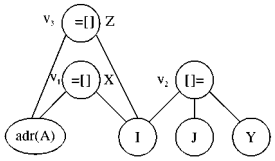
(9)



Obr. 9.15: Vytváření grafu G_{ZP}



Obr. 9.16: Graf G_{ZB} pro operace indexování



Obr. 9.17: Modifikovaný graf G_{ZB} pro operace indexování

Podobná situace nastane v případě, kdy základní blok obsahuje příkaz `A↑:=B`, kde `A` je proměnná typu ukazatel. Pokud nevíme, kam může ukazatel `A` ukazovat, pak musíme v dosud vytvořeném grafu G_{ZB} vyloučit všechny vrcholy jako možné reprezentanty společných výrazů. Je-li takto vyloučen například vrchol n reprezentující proměnnou `P` a následuje-li vzápětí přiřazení do této proměnné, pak algoritmus konstrukce grafu G_{ZB} musí vytvořit nový list, který bude dále reprezentovat proměnnou `P`.

Rovněž v případě operace vyvolání procedury nebo funkce musí být uvedeným způsobem vyloučeny všechny vrcholy grafu kvůli možným vedlejším efektům procedury nebo funkce. Počet vyloučených vrcholů lze omezit, jestliže víme, které proměnné se mohou vyvoláním procedury změnit.

9.3.3 Rekonstrukce optimalizovaného základního bloku z grafu G_{ZB}

Nejdříve uvažujme graf G_{ZB} získaný pro základní blok, v němž se nevyskytovala ani přiřazení indexovaným proměnným nebo proměnným typu ukazatel, ani volání procedur.

Na základě grafu G_{ZB} a ohodnocení h_1 a h_2 můžeme vytvořit redukovanou posloupnost operací základního bloku, ve které budou vynechány operace společných podvýrazů a operace typu $A := B$, pokud nebudou nezbytné. Zatímco vynechání redundantních výpočtů je zcela automatické, poněvadž všechny společné výrazy jsou v grafu G_{ZB} reprezentovány jediným vrcholem, vynechání příkazů $A := B$ vyžaduje, aby algoritmus rekonstrukce základního bloku měl k disposici další informace.

Uvažujme vrchol v a předpokládejme, že $h_2(v) = \{A_1, A_2, \ldots, A_n\}$, tj., že vrchol v reprezentuje hodnotu proměnných $A_1, \ldots, A_n$. Množina $h_2(v)$ může být rozložena do dvou tříd $h_2^g(v)$ a $h_l^l(v)$. Ve třídě $h_2^g(v)$ budou jména těch proměnných, které jsou "výstupními" proměnnými základního bloku. Přesněji $h_2^g(v)$ je množina živých proměnných, jejichž hodnoty vypočítané

v základním bloku budou použity vně bloku. Na druhé straně proměnné z množiny $h_2^l(v)$ jsou lokální proměnné původního základního bloku v tom smyslu, že hodnota těchto proměnných vypočítaná v základním bloku je použita pouze v tomto bloku.

Rozklad množiny $h_2(v)$ na třídy $h_2^o(v)$ a $h_2^l(v)$ je důležitý z toho důvodu, že při rekonstrukci optimalizovaného bloku musíme všem proměnným z množiny $h_2^o(v)$ přiřadit hodnotu, kterou reprezentuje vrchol v, kdežto proměnné třídy $h_2^l(v)$ mohou být spolu s odpovídajícími přiřazeními vypuštěny. Vzniká tedy otázka, jak určit tento rozklad. Obecně nalezení množiny $h_2^o(v)$ vyžaduje znalost živých proměnných základního bloku, což je jedna z charakteristických úloh globální analýzy toku údajů. V případě, že se tato analýza v rámci dalších optimalizací neprovádí, můžeme se spokojit s určitou aproximací množiny $h_2^o(v)$, která samozřejmě pokrývá všechny živé proměnné základního bloku. Tato aproximace se získá jako doplněk k množině proměnných z $h_2(v)$, o nichž určitě víme, že jsou "lokální" v základním bloku. Takovou vlastnost mají obvykle generované pomocné proměnné, např. proměnné `T1`, `T2`, ..., `T5` na obr. 9.13(b). Při určování lokálních proměnných základního bloku je třeba vycházet z metody překladu do vnitřního tvaru, např. pomocné proměnné generované při překladu booleovských výrazů tokem řízení se mohou vyskytnout v několika základních blocích. Za předpokladu, že se proměnné `T1`, ..., `T5` v příkladě na obr. 9.13 nevyskytují v jiném základním bloku, získáme aproximaci množiny $h_2^o(v)$ průnikem množiny $h_2(v)$, s množinou $\{Z, V, W\}$.

Algoritmus rekonstrukce optimalizovaného základního bloku pracuje takto:

1) Vyhodnocuje vrcholy grafu v pořadí, při kterém je zachována podmínka *topologického řazení na orientovaném acyklickém grafu* — vrchol může být vyhodnocen tehdy, jsou-li jeho bezprostředními následníky listy, nebo vnitřní vrcholy, které již byly vyhodnoceny.

2) Vyhodnocení vrcholů:

   a) Pro každý vnitřní vrchol je vytvořena operace $A := B\ op\ C$ nebo $A := op\ B$, kde $op = h_1(v)$, A je vybraná proměnná z množiny $h_2^o(v)$ živých proměnných, které bude reprezentovat vrchol v (hodnotu příslušející vrcholu v), B a C jsou proměnné reprezentující bezprostřední následníky vrcholu v. Je-li následníkem list u, pak je reprezentován proměnnou $h_1(u)$.

   b) Pokud množina $h_2^o(v)$ obsahuje další proměnné, řekněme $P_1, P_2, \ldots, P_k$, pak jsou vygenerovány příkazy `P1:=A; P2:=A; ...; Pk:=A`.

   c) Je-li v list a $h_2^o(v) \neq \emptyset$, pak jsou obdobně generovány příkazy `P:=`$h_1(v)$ pro všechny proměnné P z množiny $h_2^o(v)$. Tyto příkazy však smí být generovány až po tom, kdy byly vyhodnoceny všechny vrcholy, jejichž některý z bezprostředních následníků je list u, takový, že $h_1(u) \in h_2(v)$.

Při provádění kroku (2a) může nastat případ, kdy je množina $h_2^o(v)$ nebo dokonce $h_2(v)$ prázdná, poněvadž jméno proměnné přiřazené algoritmem konstrukce grafu G_{ZB} byla z množiny $h_2(v)$ vyňato a zařazeno do jiné množiny $h_2(v')$. V takovém případě bude pro konstrukci vrcholu v kroku (2a) vybrána pomocná proměnná z $h_2^l(v)$ nebo vytvořena nová pomocná proměnná, je-li $h_2(v) \neq \emptyset$.

Na obr. 9.13(c) je vytvořen optimalizovaný blok ke grafu z obr. 9.13(d). Obsahuje pouze 3 příkazy ve srovnání s 9 příkazy neoptimalizovaného bloku. Povšimněme si rovněž toho, jak byl aplikován bod c) popsaného algoritmu rekonstrukce základního bloku. Příkaz `Z:=X`, příslušející

prvnímu listu, musí být generován až po zpracování kořene grafu, poněvadž jeho pravým operandem je list odpovídající proměnné `Z`. Jinak by rekonstruovaný základní blok nebyl ekvivalentní výchozímu základnímu bloku.

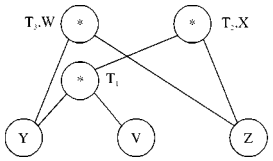
V případě, že základní blok obsahuje přiřazení indexovaným proměnným a proměnným typu ukazatel nebo volání procedur, pak při rekonstrukci optimalizovaného základního bloku nelze obecně aplikovat libovolné topologické řazení vrcholů grafu G_{ZB}. Např. v grafu na obr. 9.17 je nutné vyhodnocení vrcholů v pořadí v_1, v_2, v_3, i když topologické řazení připouští libovolné jiné pořadí.

Posaný princip vylučování některých vrcholů jako možných představitelů společných výrazů a vytváření nových vrcholů vede k tomu, že tyto vrcholy nemohou být považovány za "nezávislé" z hlediska řazení vrcholů při rekonstrukci optimalizovaného základního bloku. Obecně je nutné dodržet u takových nezávislých vrcholů stejné pořadí, kdy může dojít k přiřazení hodnoty a použití této hodnoty, jako ve výchozím základním bloku. Detailnější pravidla pro pořadí vyhodnocování operací indexování, odkazů prostřednictvím ukazatelů a volání procedur jsou uvedena např. v [3].

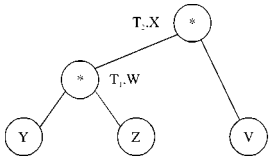
Na závěr dodejme, že uvedené metody reprezentace a rekonstrukce základního bloku lze použít také pro optimalizaci založené na některých algebraických identitách. Je-li možné, vzhledem k vlastnostem aritmetických operací cílového strojového jazyka, aplikovat komutativní a asociativní zákon, pak lze v některých případech transformovat graf základního bloku na optimalizovaný graf, v němž jsou rozpoznány další společné podvýrazy. Tento typ transformace je ilustrován na obr. 9.18 na příkladě optimalizace příkazů

```
X := Y*V*Z
W := Z*Y
```

a) neoptimalizovaný graf



b) optimalizovaný graf



Obr. 9.18: Optimalizace s využitím komutativního i asociativního zákona

Na základě optimalizovaného grafu 9.18(b) bude vytvořen optimalizovaný základní blok obsahující příkazy

```
W := Y*Z
X := W*V
```

Využíváním algebraických zákonů při optimalizaci však nemusí vždy zachovávat požadovanou ekvivalenci výpočtů. Může vést ke ztrátě významných číslic výsledků operací nebo dokonce k chybě. Uvažme např. výraz `(A-B)+C`, jehož vypočet proběhne bezchybně, kdežto v transformovaném výrazu `(A+C)-B` může nastat přetečení při sečítání. Proto je třeba, aby tato optimalizace, pokud je implementována, byla kontrolovatelná programátorem.

9.4 Globální analýza toku údajů

V předchozím výkladu jsme se mnohokrát setkali se situacemi, kdy k provedení určité optimalizační transformace bylo zapotřebí informací, které závisely na celkové struktuře programu, a které vyžadovaly analýzu specifických situací v rámci několika základních bloků programu. Typickými příklady byly transformace spojené s přesunem invariantů cyklu nebo odstraňováním redundantních výpočtů a proměnných, které vyžadovaly informace o proměnných, jejichž hodnota se v určité části programu nemění nebo o proměnných, jejichž hodnota není dále používána. Tyto informace lze získat globální analýzou toku údajů. Nyní se seznámíme se základními principy této analýzy a s některými jejími aplikacemi při optimalizaci programu.

Metody řešení různých problémů globální analýzy toku údajů jsou charakterizovány značnou příbuzností, která je dána shodným přístupem k algoritmizaci procesu šíření údajových vlastností proměnných nebo výrazů programem v závislosti na toku řízení. Proto se podrobněji seznámíme pouze s jednou, snad nejtypičtější úlohou, která se nazývá *výpočet ud-řetězců* (use-definition chains).

9.4.1 ud-řetězce a jejich výpočet

Téměř všechny globální optimalizační algoritmy vyžadují informace, které vyplývají ze vztahu mezi hodnotami proměnných v závislosti na možných posloupnostech definičních a aplikačních výskytů proměnných v programu. *Definičním výskytem* proměnné, krátce *definicí proměnné* P budeme rozumět takový výskyt proměnné P v programu, kdy je při jeho provedení proměnné P přiřazena hodnota. *Aplikačním výskytem* proměnné P, krátce *použitím proměnné* P, rozumíme ten výskyt proměnné P ve vnitřním tvaru programu, kdy je hodnota proměnné P použita jako operand určité operace. Ve vnitřním tvaru, se kterým pracujeme, bude definice proměnné P spojena s výskytem P vlevo od := nebo s voláním podprogramu, kterému předchází instrukce `param` P, kde P je výstupní skutečný parametr podprogramu nebo procedury. Tento případ zahrnuje obvyklou definici proměnné příkazem "čtení."

Např. v příkazech

```
1) A := A+1
2) B := A-C
```

je první výskyt proměnné `A` a výskyt proměnné `B` definicí proměnných `A` a `B`. Ostatní výskyty proměnných jsou použitím proměnných `A` a `C`.

Problém stanovení ud-řetězců programu lze stručně formulovat takto: pro dané použití proměnné v programu chceme znát všechny definice této proměnné, které mohou v průběhu výpočtu programu určovat hodnotu použití proměnné. Přesněji vyjádřeno, definice d se nazývá *platnou*

definicí proměnné P pro dané použití u proměnné P, jestliže v grafu toku řízení programu existuje cesta ze základního bloku obsahujícího definici d do základního bloku, v němž je použití u a současně na cestě počínající definicí d a končící použitím u neleží žádná jiná definice proměnné P, než definice d. Posloupnost tvaru

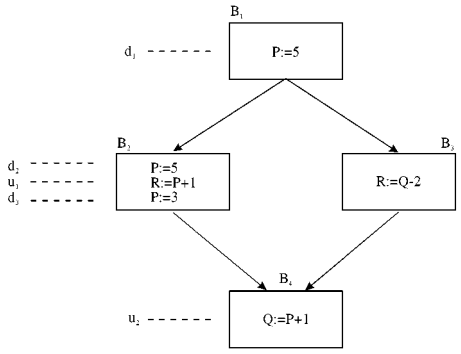
$$u\ d_1\ d_2\ \ldots\ d_n$$

kde u je použití proměnné P a $d_1, d_2, \ldots, d_n$ jsou všechny platné definice pro použití u, nazýváme *ud-řetězcem* pro použití u proměnné P.

Uvažujme program na obr. 9.19, kde d_1, d_2, d_3 značí všechny definice proměnné P a u_1 a u_2 značí použití této proměnné. Pro použití proměnné P mají ud-řetězce tvar

$$u_1\ d_2$$
$$u_2\ d_1\ d_3,$$

protože pro u_1 nejsou platné definice d_1 a d_3 a podobně pro u_2 není platná definice d_2.



Obr. 9.19: Ilustrace ud-řetězce

V případě jednoduchých proměnných můžeme definici, či použití proměnné identifikovat podle umístění jména proměnné v příkazu vnitřního tvaru programu. Složitější situace nastává např. s indexovanými proměnnými, kdy příkaz tvaru `T1[T2] := 1` definuje hodnotu složky pole, přičemž jméno pole se v tomto příkaze vůbec nevyskytuje. Protože zpracování těchto případů obvykle neodpovídá "ceně" získaných informací, výpočet ud-řetězce se provádí pouze pro jednoduché proměnné. Dále tedy nebudeme uvažovat jiné ud-řetězce než pro použití jednoduchých proměnných.

Pro výpočet ud-řetězců programu má zásadní význam skutečnost, že pro každý základní blok B programu můžeme nalézt takovou podmnožinu všech definic programu, které platí pro

použití libovolných proměnných na počátku základního bloku B (před prvním příkazem bloku B). Označme podmnožinu těchto definic symbolem $IN(B)$.

Známe-li pro základní blok B množinu $IN(B)$, můžeme snadno určit platné definice pro všechna použití proměnných, která se nacházejí v bloku B takto:

Nechť u je použití proměnné P v i-tém příkazu bloku B. Pak platí:

a) jsou-li v bloku B definice proměnné P před příkazem i, pak poslední z těchto definic je jedinou platnou definicí pro použití u proměnné P,

b) nejsou-li v bloku B před příkazem i žádné definice proměnné P, pak pro použití u proměnné P jsou platné právě ty definice proměnné P, které obsahuje množina $IN(B)$.

Na obr. 9.19 je např. pro použití u_1 proměnné P podle (a) jedinou platnou definicí definice d_2. Pro použití u_2 téže proměnné jsou podle (b) platné definice d_1 a d_3, poněvadž množina $IN(B4)$ obsahuje d_1, d_3 a definici proměnné R z bloku B_2.

Nyní ukážeme, jakým způsobem lze množinu $IN(B)$ vypočítat.

9.4.2 Rovnice toku údajů a jejich řešení

Označme analogicky k $IN(B)$ symbolem $OUT(B)$ takovou podmnožinu všech definic programu, které jsou platné pro použití libovolných proměnných na konci základního bloku B (za jeho posledním příkazem). Pak zřejmě pro každý základní blok B platí

$$IN(B) = OUT(B_1) \cup OUT(B_2) \cup \cdots \cup OUT(B_m).$$

kde $B_1, B_2, \ldots, B_m$ jsou všechny bloky, které v grafu toku řízení bezprostředně předcházejí bloku B. Je-li B počáteční nebo nedostupný základní blok programu, pak $IN(B)$ je prázdná množina.

Uvažme, které definice programu obsahuje množina $OUT(B)$ příslušející základnímu bloku B, tedy které definice programu mohou určovat hodnoty proměnných na konci základního bloku B.

Můžeme rozlišit dvě skupiny takových definic:

1) Definice, které se nacházejí v bloku B, a které nejsou následujícími definicemi téže proměnné v bloku B zrušeny. Takové definice budeme nazývat *definice generované blokem B* a označovat jako množinu $GEN(B)$.

2) Definice, které platí před prvním příkazem bloku B, a které nejsou zrušeny definicemi bloku B. Tuto skupinu definic můžeme vyjádřit ve tvaru

$$IN(B) - KILL(B),$$

kde $KILL(B)$ označuje množinu definic vně bloku B, které definují proměnnou, jež má rovněž definici v bloku B. Prvky množiny $KILL(B)$ nazýváme *definice rušené blokem B*.

Pro množinu $OUT(B)$ tedy platí

$$OUT(B) = GEN(B) \cup (IN(B) \setminus KILL(B))$$

Pro množiny IN a OUT pro n základních bloků programu platí vztahy:

$$\begin{aligned} OUT(B_i) &= GEN(B_i) \cup (IN(B_i) \setminus KILL(B_i)) \\ IN(B_i) &= \bigcup_{B_p \in \Gamma^{-1}(B)} OUT(B_p) \end{aligned} \tag{9.1}$$

BLOK	GEN	$KILL$
B_1	$\{d_1\}$	$\{d_3, d_5\}$
B_2	$\emptyset$	$\emptyset$
B_3	$\{d_2, d_3\}$	$\{d_1, d_4, d_5\}$
B_4	$\{d_4\}$	$\{d_2\}$
B_5	$\{d_5\}$	$\{d_1, d_3\}$
B_6	$\emptyset$	$\emptyset$

Tab. 9.3: Množiny GEN a $KILL$

kde , $^{-1}(B)$ je množina bezprostředních předchůdců bloku B, $i = 1, 2, \ldots, n$.

Tyto vztahy tvoří soustavu $2n$ množinových rovnic vzhledem k neznámým $IN(B_i)$ a $OUT(B_i)$, které se nazývají *rovnice toku údajů*.

Množiny $GEN(B)$ a $KILL(B)$ lze nalézt poměrně snadno. Procházíme všechny definice základního bloku a rozhodujeme, které z nich budou prvky množiny $GEN(B)$ a které prvky množiny všech definic budou tvořit množinu $KILL(B)$. K tomuto výběru je vhodné mít k dispozici pro každou proměnnou seznam všech jejich definic. Tyto seznamy, stejně jako množiny GEN, $KILL$, IN a OUT jsou podmnožinami množiny všech definic a lze je tedy zobrazovat jako bitové vektory (jako proměnné typu **set of** *MNOZINA_DEFINIC*). To má velký význam nejen z hlediska paměťových požadavků, ale, vzhledem k množinovým operacím v rovnicích toku údajů, také z hlediska rychlosti výpočtů. Připomeňme, že množinový rozdíl $IN(B) \setminus KILL(B)$ lze realizovat jako $IN(B)$ **and not** $KILL(B)$.

Na obr. 9.20 je uveden graf toku řízení, v němž jsou vyznačeny definice proměnných A a B spolu se seznamy definic a bezprostředních předchůdců jednotlivých základních bloků. Tab. 9.3 pak obsahuje množiny $GEN(B)$ a $KILL(B)$.

Nyní se zabývejme algoritmem řešení rovnic toku údajů (9.1), který na základě grafu toku údajů a daných množin GEN a $KILL$ vypočte pro každý základní blok množiny IN a OUT. Tento algoritmus, označený jako Algoritmus 9.2, stejně jako řada dalších algoritmů z oblasti globální analýzy toku údajů popisuje iterační proces. Na základě počátečního odhadu množin IN a OUT modeluje šíření platnosti definic generovaných jednotlivými základními bloky celým programem podle cest stanovených grafem toku řízení. Na počátku předpokládá pouze $IN(B) = \emptyset$ a $OUT(B) = GEN(B)$ pro všechny základní bloky B. V každé iteraci podle rovnic (9.1) začleňuje do množiny $IN(B)$ každého bloku B definice, které "prošly" na výstup vstupních bloků bloku B, aniž byly těmito bloky zrušeny, a aktualizuje množiny $OUT(B)$ na základě přesnější aproximace množin $IN(B)$. Pokud v nové iteraci nedojde ke změně hodnoty žádné z množin IN (a tudíž ani OUT), pak algoritmus končí a získané hodnoty množin $IN(B)$ a $OUT(B)$ představují řešení rovnic (9.1).

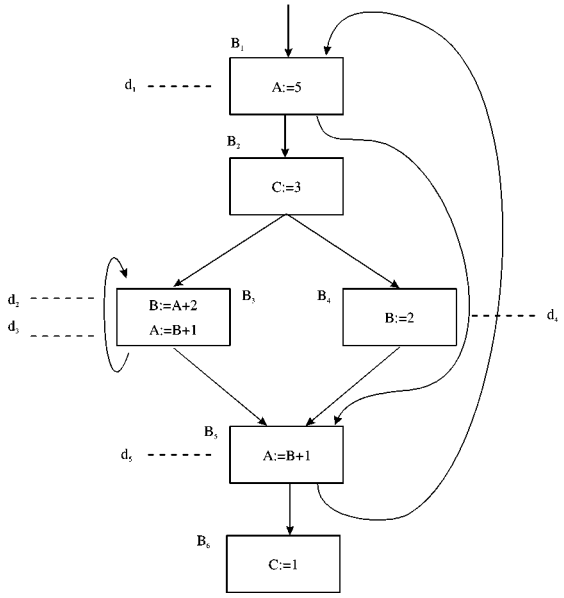
Algoritmus 9.2. (Řešení rovnic toku údajů (9.1))

Vstup: Graf toku řízení a množiny $GEN(B_i)$ a $KILL(B_i)$ pro všechny základní bloky B_i, $i = 1, 2, \ldots, n$.

Výstup: Množiny $IN(B_i)$ a $OUT(B_i)$, $i = 1, 2, \ldots, n$, které jsou řešením rovnic (9.1).

Metoda:

{počáteční aproximace množin IN a OUT}



$$
\begin{aligned}
&, \ ^{-1}(B_1) = \{B_5\} && definice(A) = \{d_1, d_3, d_5\}\\
&, \ ^{-1}(B_2) = \{B_1\} && definice(B) = \{d_2, d_4\}\\
&, \ ^{-1}(B_3) = \{B_2, B_3\}\\
&, \ ^{-1}(B_4) = \{B_2\}\\
&, \ ^{-1}(B_5) = \{B_3, B_4\}\\
&, \ ^{-1}(B_6) = \{B_5\}
\end{aligned}
$$

Obr. 9.20: Graf toku řízení

```
for i := 1 to n do begin
    IN(B_i) := ∅;
    OUT(B_i) := GEN(B_i)
end;
{vlastní iterace s testem, zda dochází ke změně aproximací řešení}
repeat
    DOITEROVANO := true;
    for i := 1 to n do
    begin
        {nová aproximace množiny IN(B_i)}
        nova_IN := ⋃_{B_p ∈ Γ^{-1}(B_i)} OUT(B_p);
        if nova_IN ≠ IN(B_i) then
        begin
            DOITEROVANO := false;
            IN(B_i) := nova_IN;
            OUT(B_i) := GEN(B_i) ∪ (IN(B_i) \ KILL(B_i))
        end;
    end;
until DOITEROVANO
```

Ilustrujme činnost popsaného algoritmu na příkladě grafu toku z obr. 9.20 s množinami GEN a $KILL$ uvedenými v tab.9.3. Na počátku položíme $IN(B_i) = \emptyset$ a $OUT(B_i) = GEN(B_i)$ pro $i = 1, \ldots, 6$. V prvním průchodu cyklem repeat dostaneme tyto hodnoty:

$$
\begin{aligned}
IN(B_1) &= OUT(B_5) = \{d_5\} \\
OUT(B_1) &= GEN(B_1) \cup (IN(B_1) \setminus KILL(B_1)) = \{d_1\} \cup (\{d_5\} \setminus \{d_5\}) = \{d_1\} \\
IN(B_2) &= OUT(B_1) = \{d_1\} \\
OUT(B_2) &= IN(B_2) = \{d_1\} \\
IN(B_3) &= OUT(B_2) \cup OUT(B_3) = \{d_1\} \cup \{d_2, d_3\} = \{d_1, d_2, d_3\} \\
OUT(B_3) &= GEN(B_3) \cup (IN(B_3) \setminus KILL(B_3)) = \{d_2, d_3\} \cup (\{d_1, d_2, d_3\} \setminus \{d_1, d_4, d_5\}) = \{d_2, d_3\} \\
IN(B_4) &= OUT(B_2) = \{d_1\} \\
OUT(B_4) &= GEN(B_4) \cup (IN(B_4) \setminus KILL(B_4)) = \{d_4\} \cup (\{d_1\} \setminus d_2) = \{d_1, d_4\} \\
IN(B_5) &= OUT(B_3) \cup OUT(B_4) = d_2, d_3 \cup \{d_1, d_4\} = \{d_1, d_2, d_3, d_4\} \\
OUT(B_5) &= GEN(B_5) \cup (IN(B_5) \setminus KILL(B_5)) = \{d_5\} \cup (\{d_1, d_2, d_3, d_4\} \setminus \{d_1, d_3\}) = \{d_3, d_4, d_5\} \\
IN(B_6) &= OUT(B_5) = \{d_2, d_4, d_5\} \\
OUT(B_6) &= IN(B_5) = \{d_2, d_4, d_5\}
\end{aligned}
$$

V tab. 9.4 jsou souhrně uvedeny hodnoty počáteční aproximace a hodnoty první a druhé iterace výpočtu množin IN a OUT. Poněvadž další iterace nemění hodnoty těchto množin, je druhá iterace výsledným řešením pomocí algoritmu 9.2.

Řešení rovnic (9.1), získané popsaným algoritmem, má jednu důležitou specifickou vlastnost. Obecně totiž není řešení rovnic toku údajů jednoznačné. Má-li graf toku řízení např. smyčku v bloku B a jsou-li $IN(B)$ a $OUT(B)$ řešením pro blok B, pak je možné nalézt jiné řešení tvaru

	poč. hodnoty		1. iterace		2. iterace	
$BLOK$	$IN[B]$	$OUT[B]$	$IN[B]$	$OUT[B]$	$IN[B]$	$OUT[B]$
B_1	$\emptyset$	$\{d_1\}$	$\{d_5\}$	$\{d_1\}$	$\{d_2, d_4, d_5\}$	$\{d_1, d_2, d_4\}$
B_2	$\emptyset$	$\emptyset$	$\{d_1\}$	$\{d_1\}$	$\{d_1, d_2, d_4\}$	$\{d_1, d_2, d_4\}$
B_3	$\emptyset$	$\{d_2, d_3\}$	$\{d_1, d_2, d_3\}$	$\{d_2, d_3\}$	$\{d_1, d_2, d_3, d_4\}$	$\{d_2, d_3\}$
B_4	$\emptyset$	$\{d_4\}$	$\{d_1\}$	$\{d_1, d_4\}$	$\{d_1, d_2, d_4\}$	$d_1, d_4\}$
B_5	$\emptyset$	$\{d_5\}$	$\{d_1, d_2, d_3, d_4\}$	$\{d_2, d_4, d_5\}$	$\{d_1, d_2, d_3, d_4\}$	$\{d_2, d_4, d_5\}$
B_6	$\emptyset$	$\emptyset$	$\{d_2, d_4, d_5\}$	$\{d_2, d_4, d_5\}$	$\{d_2, d_4, d_5\}$	$\{d_2, d_4, d_5\}$

Tab. 9.4: Výpočet množin IN a OUT z obr. 9.20

$IN'(B) = IN(B) \cup \{d\}$ a $OUT'(B) = OUT(B) \cup \{d\}$, kde d je definice, která není prvkem žádné z množin $IN(B)$, $OUT(B)$ a $KILL(B)$. Podobná situace může nastat v případě, že graf toku řízení obsahuje cyklus. Volbou počáteční aproximace množin IN a OUT v algoritmu 9.2 získáme nejmenší řešení rovnic toku údajů, tedy množiny IN a OUT, pro které platí $IN(B) \subset IN'(B)$ a $OUT(B) \subset OUT'(B)$ pro všechny bloky B v libovolném jiném řešení IN', OUT'. Toto řešení podává také nejpřesnější obraz z hlediska platnosti definic pro použití proměnných.

Druhou poznámku věnujme efektivnosti algoritmu 9.2. Dá se ukázat, že teoretická horní hranice počtu iterací (průchodů cyklem repeat) je rovna mohutnosti množiny vrcholů grafu toku řízení. Počet iterací závisí na pořadí, ve kterém jsou zpracovávány jednotlivé základní bloky. Je-li zvoleno pořadí, které odpovídá uspořádání vrcholů při hledání do hloubky (depth-first ordering), kterého se používá pro identifikaci cyklů v grafu toku řízení, pak prakticky i při aplikacích na reálné programy, je empiricky získaná horní hranice počtu iterací rovna 5 [3]. Spolu se skutečností, že vlastní operace algoritmu mohou být implementovány jako logické operace, dojdeme k závěru, že výpočet množin IN a OUT můžeme realizovat překvapivě rychle.

Známe-li pro všechny základní bloky množiny IN a OUT, pak, jak jsme již uvedli, je velmi jednoduché získat ud-řetězce pro libovolné použití proměnné. Předchází-li danému použití u proměnné základního bloku B definice této proměnné, pak ud-řetězce má tvar ud, kde d je poslední z těchto definic. V opačném případě je ud-řetězec tvaru $u\ d_1\ d_2\ \ldots\ d_m$, kde $d_1, d_2, \ldots, d_m$ jsou všechny definice proměnné z množiny $IN(B)$.

V příkladu grafu toku řízení na obr. 9.20 označme u_1 použití proměnné A v příkazu $B := A+2$ v bloku B_3, u_2 použití proměnné B v následujícím příkazu a u_3 použití proměnné B v bloku B_5. Příslušné ud-řetězce mají tvar:

$$u_1\ d_1\ d_3$$
$$u_2\ d_2$$
$$u_3\ d_2\ d_4$$

ud-řetězce mají široké využití v řadě optimalizačních algoritmů. V odst. 9.4.4 uvedeme jejich využití v algoritmu odstranění výpočtu s konstantami a v algoritmu detekce invariantu cyklu. Vedle optimalizací mají ud-řetězce aplikaci také při vyhledávání chyb. Jestliže danému použití proměnné neodpovídá žádná platná definice, pak zřejmě při výpočtu programu může nastat chyba, kdy se počítá s hodnotami, které nejsou jednoznačně určeny.

9.4.3 Výpočet živých proměnných

V odst. 9.3.3 jsme při optimalizaci v rámci základního bloku potřebovali určit živé proměnné toho bloku, tedy proměnné, jejichž hodnoty jsou použity vně základního bloku. Informaci o živých proměnných využívají i další algoritmy optimalizace (např. odstraňování indukčních proměnných) a rovněž algoritmus generování cílového programu.

Výpočet živých proměnných základního bloku lze provádět podobným způsobem jako výpočet platných definic IN a OUT.

Nejdříve sestavíme příslušné rovnice toku údajů. Označme

- $LIN(B)$ množinu živých proměnných na začátku bloku B,
- $LOUT(B)$ množinu živých proměnných na konci bloku B,
- $DEF(B)$ množinu proměnných, jejichž definice v B předcházejí jejich libovolné použití v B,
- $USE(B)$ množinu proměnných, jejichž použití v B předcházejí jejich libovolnou definici v B.

Rovnice toku údajů pro živé proměnné na začátku a konci bloku B mají tvar analogický rovnicím (9.1):

$$\begin{aligned} LIN(B) &= USE(B) \cup (LOUT(B) \setminus DEF(B)) \\ LOUT(B) &= \bigcup_{B_s \in \Gamma(B)} LIN(B_s) \end{aligned} \tag{9.2}$$

První rovnice stanovuje, které proměnné jsou živé na začátku bloku. Jsou to zřejmě ty proměnné, které jsou v bloku použity, aniž budou jejich hodnoty před prvním použitím předefinovány a dále proměnné, které jsou živé na konci bloku vyjma proměnných, jež byly v bloku předefinovány. Druhá rovnice stanovuje proměnné živé na konci bloku. Jsou to ty proměnné, které jsou živé na počátku alespoň jednoho z následníků bloku B. ($,$ je relace následnosti v grafu toku řízení). Optimalizační algoritmy požadují znalost množiny $LOUT(B)$, poněvadž právě tato množina udává proměnné, jejichž hodnoty budou používány po opuštění bloku B.

Rovněž algoritmus řešení rovnic (9.2) je téměř identický s algoritmem 9.2.

Algoritmus 9.3. (Analýza živých proměnných)

Vstup: Graf toku řízení s vrcholy $B_1, B_2, \ldots, B_n$ a množiny $DEF(B_i)$ a $USE(B_i)$ pro $i = 1, \ldots, n$.

Výstup: Množiny $LOUT(B_i)$ obsahující živé proměnné na konci bloku B_i.

Metoda:

for $i := 1$ **to** n **do** $LOUT(B_i) := \emptyset$;
repeat
  $DOITEROVANO :=$ **true**;
  for $i := 1$ **to** n **do begin**
    $LIN(B_i) := USE(B_i) \cup (LOUT(B_i) \setminus DEF(B_i))$;
    $nova\_LOUT(B_i) := \bigcup_{B_s \in \Gamma(B_i)} LIN(B_s)$;

```
      if nova_LOUT(B_i) = LOUT(B_i) then begin
        DOITEROVANO := false;
        LOUT(B_i) := nova_LOUT(B_i)
      end
    end
  until DOITEROVANO;
```

Výsledkem algoritmu 9.3 je žádané nejmenší řešení rovnic (9.2). Počet iterací může být podstatně redukován, zvolíme-li pořadí vyhodnocování bloků B_i odpovídající tentokrát inverznímu uspořádání vrcholů grafu toku při hledání do hloubky.

Téměř identického postupu jako při analýze živých proměnných, lze použít ke zjišťování du-řetězců (definition-use chains). Tyto řetětce jsou protějškem ud-řetězců a poskytují informace o všech použitích proměnné, pro která je platná daná definice této proměnné. du-řetězce nacházejí uplatnění např. při určování indukčních proměnných.

9.4.4 Příklady optimalizačních algoritmů využívajících informace globální analýzy toku údajů

V tomto odstavci uvedeme příklady dvou algoritmů, které vyžadují znalost ud-řetězce a živých proměnných.

Algoritmus 9.4. (Odstranění výpočtu s konstantami (constant folding))

Vstup: Graf toku řízení programu a ud-řetězce

Výstup: Modifikovaný graf toku řízení

Metoda:

```
repeat
  for i := 1 to n do {pro každý základní blok}
    for j := 1 to m do {pro každý příkaz S_ij bloku B_i}
      for k := 1 to p {pro každý operand O_k příkazu S_ij}
      do begin
        if ud-řetězec pro O_k obsahuje jedinou definici tvaru O_k := C, C je konstanta
          then nahraď operand O_k v příkaze S_ij konstantou C;
        if všechny operandy příkazu S_ij jsou konstanty
        then begin
          if S_ij je příkaz typu ":="
          then begin
            vyčísli pravou stranu příkazu S_ij;
            nahraď příkaz S_ij příkazem A := h, kde A je
              levá strana příkazu S_ij a h je vyčíslená
              hodnota pravé strany příkazu S_ij
              end;
            if S_ij je příkazem podmíněného skoku
            then begin
              příkazy tvaru if false goto l vypusť;
              příkazy tvaru if true goto l
```

nahraď příkazem `goto l`;
modifikuj graf toku řízení programu
end
end
end
until nedošlo ke změně žádného příkazu programu

Algoritmus 9.5. (Detekce příkazů, které provádějí v cyklu invariantní výpočty — invariantů cyklu — a jejich přesun do prologu cyklu)

Vstup: Graf toku řízení $GTR = (\mathcal{B}, H, \sigma)$, množina L základních bloků tvořících cyklus $L \subseteq \mathcal{B}$, ud-řetězce proměnných použitých v blocích cyklu L, množiny živých proměnných pro výstupní bloky cyklu.

Výstup: Cyklus L s prologem, do něhož jsou přesunuty některé příkazy z cyklu L.

Metoda:

1) Označ jako invariantní každý příkaz S bloku B, $B \in L$, jehož všechny operandy O splňují podmínku:

   O je konstanta nebo O má všechny definice v blocích z $B \setminus L$.

repeat

2) označ jako invariantní každý příkaz S bloku B, $B \in L$, který není dosud označen a současně všechny jeho operandy O splňují podmínku:

   a) O je konstanta nebo

   b) pro každou definici d operandu O platí

      (i) d je v bloku z $B \setminus L$ nebo

      (ii) d je příkaz již označený jako invariantní

until

nebyl označen nový invariantní příkaz.

3) Pro každý příkaz S definující hodnotu proměnné A, který byl nalezen v kroku 2 ověř, zda platí buď

   a) podmínky (i) - (iii):

      (i) S je v bloku, který je součástí každé cesty vedoucí ven z cyklu,

      (ii) A nemá v L jinou definici,

      (iii) pro všechna použití proměnné A v L je platná pouze definice příkazem S, nebo

   b) pouze podmínka (iii) a současně A není prvkem $LOUT(B)$ žádného výstupního bloku B cyklu L.

4) Příkazy vyhovující podmínce testované v kroku 2 přesuň do prologu cyklu L v pořadí, ve kterém byly vyhledány v kroku 1.

Kapitola 10

Generování cílového programu

V této kapitole se budeme zabývat poslední částí kompilačního překladače — generátorem cílového programu. Vstupem generátoru cílového programu je posloupnost příkazů ve vnitřním jazyce překladače, výstupem pak konečný produkt překladu — cílový program ekvivalentní zdrojovému programu. Existují kompilační překladače, které vytvářejí cílový program ve vyšším programovacím jazyce. Jsou to například některé překladače simulačních jazyků nebo jinak specializovaných jazyků jako Altran, který slouží pro popis algebrických úprav, a jehož překladač generuje program v jazyce Fortran. V této kapitole se zaměříme na kompilátory, jejichž cílovým jazykem je strojově orientovaný jazyk.

10.1 Specifické problémy generování cílového programu

Generátor cílového programu může být považován za nejjednodušší a současně nejobtížnější část překladače. Tento paradox vyplývá z požadavků, které vzneseme na funkce a vlastnosti generátoru. Požadujeme-li pouze, aby generátor vytvářel jen sémanticky ekvivalentní cílový program, pak, s ohledem na vlastnosti vnitřního jazyka, je tento úkol poměrně snadný. Chceme-li však, aby z mnoha ekvivalentních cílových programů byl nalezen takový program, který splňuje dále diskutované a odůvodněné požadavky, pak se může proces generování stát extrémně složitým, ne-li prakticky i teoreticky neřešitelným.

Základními faktory určujícími algoritmy generování cílového programu jsou vstupní a výstupní jazyk generátoru. Typy vstupního jazyka, vnitřního jazyka překladače, byly diskutovány v kap. 8.

10.1.1 Výstupní jazyk generátoru

Cílový program vytvářený generátorem cílového programu může mít obecně tyto formy:

- Posloupnost strojových instrukcí s absolutními adresami.
- Posloupnost strojových instrukcí s relativními adresami, tj. přemístitelný strojový program.
- Posloupnost příkazů jazyka symbolických instrukcí.

Překlad zdrojového programu do strojových instrukcí s absolutními adresami je nejefektivnějším způsobem kompilace. Vytvořený program lze okamžitě spustit a celý proces překladu a

výpočtu lze provést během krátké doby. Hlavní nevýhodou tohoto typu překladu je skutečnost, že musí být překládán celý program. Není možné použití předem připravené knihovny. Z tohoto důvodu se generování strojových instrukcí s absolutními adresami používá jen pro malé programy a v případě, že se jedná především o ladění programů (zejména při výuce). Překlad do strojových instrukcí s absolutními adresami provádějí inkrementální překladače.

Posloupnost strojových instrukcí s relativními adresami je nejčastějším a nejběžnějším výstupem kompilátoru. Výstupem kompilátoru jsou tzv. přemístitelné moduly, které je ovšem dále nutno zpracovat sestavovacím programem. Generování přemístitelného strojového programu umožňuje samostatnou kompilaci podprogramů nebo procedur. Spojování a zavádění modulů sice představuje časově nezanedbatelnou operaci, získáme však možnost velmi pružné práce s podprogramy při kompletaci úplného proveditelného programu. Můžeme samostatně překládat podprogramy, volat již dříve přeložené podprogramy nebo spojovat podprogramy přeložené z různých programovacích jazyků. Proto většina komerčních překladačů generuje právě přemístitelný strojový program.

Generování programu v jazyce symbolických instrukcí usnadňuje problém generování cílového programu. Kromě případného využití makroinstrukcí tohoto jazyka spočívá usnadnění v tom, že asembler provede výpočet relativních adres včetně adres skoků na později definovaná návěští. Cena, kterou platíme za toto usnadnění, jsou další dva průchody realizované právě asemblerem. Protože však kompilátor obvykle neduplikuje celou funkci asembleru, je i toto řešení přijatelné, zvláště v případě nedostatku vnitřní paměti, který je třeba řešit mnohaprůchodovým kompilátorem.

Z hlediska algoritmů generování cílového programu v kompilátoru je volba úrovně výstupního jazyka méně podstatná. Určujícími faktory procesu generování jsou specifika výstupního jazyka a tedy specifika a detaily konkrétního počítače, na kterém bude přeložený program prováděn. Tato specifika můžeme rozčlenit podle tří základních složek, které jsou dány architekturou počítače a které se nejvýznamněji podílejí na modelu generátoru cílového programu. Jsou to paměti, přístupové cesty k údajům v pamětech a operace počítače.

Pro účely generování mohou být paměti počítače rozděleny do těchto tříd:

- *Hlavní paměť:* pole stejně velikých paměťových míst, které jsou přímo přístupné pro_střed_nictvím adresy.
- *Zásobník:* paměť, která je zpřístupněna podle pravidla LIFO (last in, first out).
- *Celočíselný střadač:* paměť, se kterou pracují operace v celočíselné aritmetice.
- *Střadač se zobrazením v pohyblivé čárce:* paměť, se kterou pracují operace aritmetiky v pohyblivé čárce.
- *Bázový registr:* paměť obsahující adresu, které se používá pro zpřístupnění operandu operace.
- *Indexregistr:* paměť obsahující celočíselný index (offset), jež se používá pro zpřístupnění operandu operace.
- *Programový čítač:* paměť obsahující adresu příští prováděné instrukce.
- *Podmínkový kód* paměť uchovávající výsledek srovnání nebo instrukce testu.
- *Jiné speciální registry* (např. ukazatel na vrchol zásobníku, registr přetečení apod.)

Každý počítač má alespoň hlavní paměť a programový čítač. U většiny počítačů lze mnohé paměťové prostředky řadit do více než jedné z uvedených tříd. Bázové registry jsou obvykle identické třídy. Např. u počítačů IBM 370 slouží víceúčelové registry jako bázové registry, indexregistry i celočíselné střadače.

Přístupové cesty popisují hodnotu nebo adresu operandu, výsledku nebo skoku. Klasifikace instrukcí na 0-, 1-, 2- a 3-adresové instrukce je prováděna právě podle počtu přístupových cest, které instrukce popisují.

Každá přístupová cesta specifikuje počáteční prvek operandu nebo výsledku v dané paměťové třídě. Přístupové cesty k některým paměťovým třídám jako je zásobník, programový čítač, podmínkový kód, či speciální registry, se v instrukci normálně explicitně nespecifikují.

Nejčastější explicitní přístupové cesty zahrnují tyto hodnoty a jejich výpočty:

- *Konstanta.* Hodnota konstanty se objevuje přímo v instrukci.
- *Registr.* Hodnota je získána jako obsah specifikovaného registru.
- *Registr + konstanta.* Hodnota je získána jako součet obsahu registru a konstanty uvedené v instrukci.
- *Registr + registr.* Hodnota je získána jako součet obsahu obou registrů.
- *Registr + registr + konstanta.* Hodnota je získána jako součet konstanty a obsahů obou registrů.

Takto získaná hodnota může být dále chápána již jako operand, nebo jí lze použít jako efektivní adresy operandu, nebo jako nepřímé adresy, která vyžaduje další výběr (výběry) z paměti.

Operace počítače jsou specifikovány instrukcemi, které lze obvykle rozdělit do těchto čtyř tříd:

Výpočty: Realizují funkce přiřazující n-ticím hodnot m-tice reprezentující výsledek. Tyto funkce mohou mít vedlejší účinky (přerušení, nastavení podmínkových kódů).

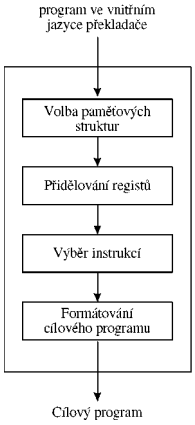
Přesun údajů: Kopírují informace mezi pamětmi stejné třídy nebo z paměti jedné třídy do paměti jiné třídy.

Řízení programu: Mění normální posloupnost provádění instrukcí buď nepodmíněně, nebo podmíněně.

Řízení okolí: Mění okolí, ve kterém je výpočet prováděn, např. instrukce blokující určitá přerušení nebo instrukce skoku do podprogramu, které nastavují adresové registry a mění adresovatelné okolí programu.

Uvedené rozdělení operací není samozřejmě jednoznačné. Určité instrukce mohou zahrnovat, podobně jako při klasifikaci pamětí, více funkcí.

Moderní počítače se vyznačují proměnnou délkou instrukcí, při níž se dosahuje větší efektivity kódování programu. Tato skutečnost má rovněž význam pro generování cílového programu.



Obr. 10.1: Struktura generátoru cílového programu

10.1.2 Struktura generátoru cílového programu

Proces syntézy cílového programu může být, stejně jako proces analýzy zdrojového programu, rozložen na dílčí části. Schématické znázornění tohoto rozkladu je uvedeno na obr. 10.1.

Volba paměťových struktur zahrnuje reprezentaci údajových struktur zdrojového programu údajovými strukturami cílového programu, jejich přidělení prvkům paměťových tříd počítače a stanovení přístupových cest k reprezentovaným údajům. U většiny kompilátorů je tato část syntézy částečně řešena už v rámci generování vnitřního tvaru programu, který používá operace a jim příslušející operandy v souladu s navrženou reprezentací údajových struktur a přístupovými cestami k prvkům těchto struktur. Budeme předpokládat, že v průběhu generování vnitřního tvaru programu byly každému objektu programu přiřazeny: jméno paměťové oblasti a jeho relativní adresa uvnitř této oblasti. V rámci generování cílového programu pak musí být prováděno přiřazení "adres" (absolutních, relativních, implicitních) symbolickým operandům vnitřního tvaru programu a to v závislosti na zobrazení elementárních objektů (čísel v pevné a pohyblivé řádové čárce, zobrazení booleovských hodnot, znaků, ukazatelů), na zobrazení strukturovaných objektů (řetězců, polí, záznamů, množin) v paměťových třídách počítače a na specifických vlastnostech přístupových cest daného počítače (např. problém zarovnávání u počítačů se slabikovou organizací hlavní paměti).

Přidělování registrů je dílčí proces generování cílového programu, v němž se provádí výběr konkrétních registrů pro přechodné nebo trvalé uložení některých objektů cílového programu

a pro operace nad nimi. Tato část syntézy cílového programu podstatně ovlivňuje efektivitu výsledného programu.

Výběr instrukcí zahrnuje volbu strojových instrukcí pro výpočty výrazů, pro řídicí struktury programu a pro zpřístupnění údajů. Vlastní výběr instrukcí probíhá ve velmi úzkém spojení s přidělováním registrů, poněvadž registry jsou používány pro vyčíslení výrazů, výpočet adres i specifikaci přístupových cest. Přidělování registrů a výběr instrukcí jsou vzájemně závislé činnosti, poněvadž volba instrukce může předepisovat přidělení registrů a naopak dostupnost údaje v registru ovlivňuje výběr instrukcí.

Formátování cílového programu: Výstup generátoru cílového programu musí být formátován do tvaru, který je přijatelný (zpracovatelný) složkami operačního systému počítače, na němž má být přeložený program prováděn. V případě cílového programu v jazyce symbolických instrukcí je to tvar, který vyžaduje příslušný asembler. V případě výstupu ve tvaru strojového programu se vytváří lineární soubor slov reprezentujících instrukce a údaje programu. Tento soubor je obvykle členěn do záznamů (link modul records), které navíc obsahují informace pro zavaděč (relokační konstantu, počáteční adresu, informace o vyvážených a dovážených symbolech).

Obr. 10.1 vyjadřuje logické členění generátoru. Pořadí, ve kterém jsou realizovány jednotlivé činnosti generátoru, nelze pevně předepsat, poněvadž mezi výběrem paměťových struktur, přidělováním registrů a výběrem instrukcí existují velmi silné vzájemné závislosti dané vnitřním tvarem programu a architekturou cílového počítače.

10.1.3 Požadavky na generátor cílového programu a faktory ztěžující jeho realizaci

Při návrhu a realizaci generátoru cílového programu se můžeme setkat s řadou požadavků kladených na vlastní generátor i na generovaný cílový program. K nejčastějším z takových požadavků patří:

1. rychlý překlad zahrnující také rychlé algoritmy generování cílového programu,
2. bezpečný výpočet cílového programu, který předpokládá řadu kontrol prováděných v době výpočtu programu,
3. dobrá diagnostika chyb, které se vyskytnou v době výpočtu,
4. přímá korespondence mezi strukturami zdrojového a cílového programu, jež usnadní ladění programu,
5. efektivita generovaného programu, a to jak z hlediska rychlosti výpočtu, tak i z hlediska paměťových požadavků.

Je zřejmé, že mnohé z těchto požadavků jsou protichůdné. Největší důraz je obvykle kladen na efektivitu generovaného cílového programu, které lze dosáhnout generováním pouze nezbytných částí cílového programu a jejich "optimálním" tvarem, oboje ve smyslu ručně psaného cílového programu. Taková podoba cílového programu je ve sporu s požadavky 2. a 4. K tomu, abychom dosáhli bezpečného provádění cílového programu, je nutné do generovaného programu začlenit instrukce, které provádějí např. kontrolu indexování polí (hodnoty indexů jsou v předepsaném intervalu), odkazů prostřednictvím ukazatelů, přetečení zásobníku, v případě některých programovacích jazyků dynamické typové kontroly apod. Současné počítače neumožňují většinu z potřebných kontrol v průběhu výpočtu realizovat automaticky technickými prostředky, a proto generovaný výstup překladače obsahuje "neproduktivní" úseky programu.

Rovněž dobrá diagnostika je drahá a objemná. Objeví-li se při výpočtu chyba, pak uživatel požaduje zprávu o povaze chyby, jak byla zjištěna, kde se nachází zdroj této chyby ve zdrojovém programu a jak bylo v průběhu výpočtu dosaženo bodu programu, ve kterém nastala chyba (zpětné sledování programu). To vše opět přináší neproduktivní úseky generovaného programu a značné paměťové požadavky, v lepším případě na vnější paměti, kde se uchovává informace pro účely diagnostiky.

Požadavek na optimálnost generovaného cílového programu vede především k výběru takové posloupnosti instrukcí pro jednotlivé příkazy vnitřního tvaru programu, která je co nejrychlejší. Tento výběr je závislý na "kontextu" výpočtu a musí využívat všech možností, které nabízí instrukční soubor a architektura cílového počítače (speciálních instrukcí, všech registrů a přístupových cest — adresových módů). Požadavek na efektivitu může vést, buď ve spojení se strojově nezávislou optimalizací, nebo i bez této optimalizace, ke změně pořadí provádění operací vzhledem k pořadí ve zdrojovém programu.

Uvážíme-li nyní, že výběr mezi velkým počtem alternativních posloupností instrukcí vyžaduje složitou analýzu mnoha případů a složitý a rozsáhlý algoritmus generování a že změna pořadí operací porušuje korespondenci mezi strukturami cílového a zdrojového programu, pak vidíme, že požadavek optimálnosti je ve sporu s požadavky 1. a 4.

K řešení těchto vnitřních rozporů se v reálných kompilátorech přistupuje dvěma způsoby. První, méně častý způsob, předpokládá existenci různých verzí plně kompatibilních překladačů daného zdrojového jazyka. Tyto překladače se liší tím, že splňují jen některé z uvedených požadavků na generátor cílového programu. Typickým příkladem takového řešení jsou některé překladače firmy IBM, např. série překladačů jazyka Fortran završená překladačem FORTRAN H, který generuje vysoce optimální cílový program. Jiným příkladem jsou překladače jazyka PL/I ve verzích PL/I – Checkout compiler, PL/I – Full compiler a PL/I – Optimizing compiler, pokrývající široké spektrum požadavků od diagnostických prostředků po efektivitu generovaných programů.

Druhé řešení umožňuje získat požadované vlastnosti překladače včetně vlastností generátoru cílového programu v rámci jediného překladače. V tomto případě je nutné zabudovat různé strategie a algoritmy jednotlivých částí generátoru a jejich výběr předepisovat volbami (přepínači), podle požadovaných vlastností cílového programu. Jako příklad tohoto řešení můžeme uvést opět překladač firmy IBM – Pascal/VS compiler, který umožňuje volby DEBUG/NODEBUG, CHECK/NOCHECK a OPTIMIZE/NOOPTIMIZE.

Existují dva hlavní zdroje obtížnosti návrhu a realizace generátoru cílového programu:

1. odlišnosti zdrojového a cílového programu vyplývající z principiálního rozdílu mezi člověkem a počítačem,

2. nesoulad mezi rozšířenými programovacími jazyky a funkčními charakteristikami a architekturami existujících počítačů.

Přirozené vlastnosti člověka, v souladu s vývojem programovacích metod a jazyků, vedou k programům, jež se vyznačují verbálností, redundancí a odrážejí schopnost člověka postihnout vztahy mezi objekty v širokém kontextu. Tyto vlastnosti programu jsou však v rozporu se schopnostmi počítače a vlastnostmi efektivního cílového programu. I když určitá část verbality programu (dlouhé identifikátory) i redundancí je nebo může být odstraněna v předchozích částech překladače, určitá část vždy zůstává a musí být řešena v rámci generátoru cílového programu. Uvedený zdroj potíží bohužel nebyl a pravděpodobně nikdy nebude vyřešen rostoucí

rychlostí a paměťovými možnostmi počítačů, poněvadž se ukazuje, že nároky úloh, které chceme řešit, rostou přinejmenším stejně rychle jako výkonnost počítačů.

Nesoulad mezi prostředky vyšších programovacích jazyků a architekturami nejrozšířenějších tříd počítačů se projevuje v několika hlediscích. Mnoho základních jazykových prostředků, jako je rekurze, vnořené příkazy, bloková či modulární struktura programu, nelineární údajové struktury, nemá přímý obraz v architektonických strukturách počítačů. Pokud daný počítač obsahuje instrukce pro volání podprogramu, pro cyklus for nebo selektivní příkaz, odpovídající "vyšším" prostředkům jazyka, pak použití těchto instrukcí je vhodné pouze pro překlad velmi omezeného souboru jazyků. To je důsledkem značné sémantické nekompatibility podobných jazykových konstrukcí v existujících programovacích jazycích. Nejvážnější nesoulad v architekturách počítačů z hlediska generování cílového programu nalézáme v nesymetriích, které se projevují tak, že určitou vlastnost nemají všechny prvky jinak homogenní množiny. Tak např. počítače, které mají "obecně použitelné (general purpose)" registry, ve skutečnosti neumožňují, aby byl pro operand vybrán libovolný registr. Registr 0 nemůže být použit jako indexregistr nebo bázový registr. Podobně při násobení nebo dělení smí být použita dvojice sousedících registrů, z nichž první je sudý (problém registrových párů). Nesymetrie se objevuje rovněž v instrukčním souboru. Některé operace smí být provedeny pouze mezi registry, jiné mezi registrem a pamětí. Jiným příkladem je nemožnost použít všech relačních operátorů při větvení programu. Uvedené příklady spolu s dalšími neregularitami komplikují výběr instrukcí jak pro vlastní operace, tak i pro specifikaci přístupových cest k údajům.

Z literatury jsou známy obecné principy návrhu architektury počítače, které usnadňují konstrukci překladačů a zvlášťě generátoru cílového programu. Jsou to tyto principy:

- *Regularita (pravidelnost):* Jestliže je možné něco udělat určitým způsobem na určitém místě, pak má být možné udělat totéž stejným způsobem i na jiném místě.

- *Ortogonalita:* Specifikaci počítače (strojového jazyka) je možné rozdělit na oblasti, které lze definovat izolovaně jednu od druhé. Např. definovat údajové typy, adresovací mechanismy a instrukční soubor vzájemně nezávisle.

- *Komposicionalita:* Je-li dodržen princip regularity a ortogonality, pak je možné skládat základní elementy strojového programu libovolným způsobem. To znamená, že lze použít libovolného adresovacího mechanismu spolu s libovolným operátorem a libovolným typem údajů.

Poněvadž v počítačích, se kterými se běžně setkáváme, nejsou vždy uvedené principy dodržovány, musí generátor efektivního cílového programu provádět nákladnou analýzu mnoha speciálních případů, která je algoritmicky obtížná a časově velmi náročná. Obtížnost návrhu a realizace dobrého generátoru je pak přímo závislá na rozsahu a složitosti instrukčního souboru.

V této souvislosti je pozoruhodná architektura spojená s velmi velkou integrací obvodů (VLSI), která je označována zkratkou RISC (Reduced Instruction Set Computer). RISC je počítač, který se vyznačuje omezeným souborem instrukcí a několika desítkami registrů. Vlastnosti tohoto počítače mají bezprostřední dopad na zjednodušení a zrychlení generátoru cílového programu i zrychlení generovaného programu. Generování cílového programu pro počítače RISC se budeme věnovat v kapitole 11.

Instrukce	Význam
`LOAD M`	$S \leftarrow < M >$
`STORE M`	$M \leftarrow < S >$
`ADD M`	$S \leftarrow < S > + < M >$
`SUB M`	$S \leftarrow < S > - < M >$
`MUL M`	$S \leftarrow < S > * < M >$
`DIV M`	$S \leftarrow < S > / < M >$
`CHS`	$S \leftarrow - < S >$

Tab. 10.1: Symbolické instrukce cílového počítače

10.2 Klasické metody generování cílového programu

V této podkapitole se seznámíme s charakteristickými rysy určité skupiny metod generování, které se nejčastěji objevují v učebnicích o překladačích a které jsou také aplikovány ve většině existujících kompilátorů. Budeme je proto nazývat klasickými nebo konvenčními metodami, na rozdíl od metod založených na aplikaci formálních překladů a atributových gramatik, které popíšeme v podkapitole 10.4.

Základní princip těchto klasických metod vychází z předpokladu, že množina jazykových prostředků určitého programovacího jazyka představuje množinu nezávislých problémů generování. Každý takový problém je řešen metodou, která je nejvhodnější pro určitou jazykovou konstrukci. Celý generátor je pak implementován jako soubor procedur, z nichž každá řeší svůj specifický úzký úkol. Jsou-li např. vnitřním jazykem překladače čtveřice (n-tice), pak generátor cílového programu interpretuje každou čtveřici jako volání procedury, operátor jako jméno procedury, operandy jako její parametry. Problém realizace generátoru se tak rozpadá na řadu téměř triviálních podproblémů — implementovat pro každý typ čtveřice samostatnou proceduru a přidat procedury pro inicializaci, ukončení, přidělování registrů a řízení celého generátoru.

Podobně, je-li vnitřním tvarem programu postfixová notace, pak jsou operátory interpretovány opět jako jména procedur. Jejich skutečné parametry jsou však zpřístupňovány prostřednictvím globálního zásobníku. V případě, že je vyvolána procedura odpovídající n-árnímu operátoru, pak používá n vrcholových položek zásobníku, stejně jako v interpretačním překladači. Tato varianta má výhodu v tom, že výsledná informace může být uložena na vrchol zásobníku a použita v následujícím kroku (po vyvolání další procedury).

10.2.1 Generátor pro jednoduché aritmetické výrazy

Abychom ilustrovali podstatu klasických metod generování cílového programu, uvažujme část generátoru zpracovávající aritmetické výrazy přeložené do posloupnosti čtveřic. Budeme pracovat s jednoduchou architekturou cílového počítače s omezeným počtem instrukcí a s výstupem generátoru na úrovni jazyka symbolických instrukcí. Tab. 10.1 obsahuje popis symbolických jednoadresových instrukcí, kde jediná adresa `M` specifikuje druhý operand binární operace umístěný v hlavní paměti. První operand je implicitní a je reprezentován obsahem jediného střadače `S`. Pro výstup generovaných instrukcí použijeme proceduru `GENERUJ(X,Y)`, kde `X` je operační kód instrukce a `Y` je symbolická adresa.

Generování pro čtveřice

Už v případě uvedených zjednodušujících předpokladů nelze generátor řešit takovým způsobem, že pro každou čtveřici, např. `(-,A,B,C)`, je generována posloupnost instrukcí

```
LOAD  A
SUB   B
STORE C
```

bez ohledu na to, jaká čtveřice předchází před a následuje za zpracovávanou čtveřicí. Kdyby v uvedeném příkladě předcházela čtveřice, specifikující operaci s uložením výsledku do `A` (třeba `(*,D,E,A)`), pak je zbytečné generování instrukce `LOAD A`. Podobně, následuje-li čtveřice, která používá `C` jako svůj operand, můžeme v rámci generování instrukcí pro výraz vypustit instrukci `STORE C`.

Problém potlačení výstupu redundantních instrukcí `LOAD` a `STORE` je v generátorech řešen metodou "simulace" výpočtu v tom smyslu, že je registrován obsah střadače (obecně registrů) tak jak bude definován generovanými instrukcemi v průběhu výpočtu programu. V diskutovaném generátoru instrukcí pro překlad aritmetických výrazů budeme tedy uchovávat informaci o obsahu jediného střadače v globální proměnné $STRDC$. Na základě této informace vygenerujeme instrukci `LOAD` s případným předchozím uložením obsahu střadače pouze v případech, kdy je to nezbytně nutné. Při tomto rozhodování budeme brát v úvahu komutativitu operací `+` a `*` v našem cílovém jazyce, která však obecně nemusí platit. Procedura $ULOZ\_DO\_STRADACE(P, Q)$ má za úkol zajistit nezbytné generování instrukcí `LOAD` a `STORE` tak, aby střadač obsahoval hodnotu operandu P nebo Q (pro komutativní operace) nebo hodnotu P (pro nekomutativní operace, $Q = P$). Činnost procedury závisí na obsahu proměnné $STRDC$. V případě, že je ve střadači druhý operand komutativní operace, provede procedura výměnu operandů příslušné čtveřice. Procedura má tvar:

```
procedure ULOZ_DO_STRADACE(var P, Q : promenna);
var T : promenna;      {pomocná proměnná}
begin
  if STRDC <> P then
  begin
    if STRDC = nedef then   {nedefinovaný obsah}
    begin
      GENERUJ('LOAD', P);
      STRDC := P
    end
    else if STRDC = Q then   {výměna operandů}
    begin
      T := P;
      P := Q;
      Q := T
    end
    else   {úschova střadače}
    begin
      GENERUJ('STORE', STRDC);
      GENERUJ('LOAD', P);
      STRDC := P
```

```
    end
  end
end;
```

Vlastní procedury generátoru mají parametry, které popisují složky čtveřice — operandy a proměnnou, do které je uložen výsledek operace určené danou čtveřicí. Tyto procedury mají tvar uvedený v tab. 10.2.

Operátor čtveřice	Příslušná procedura
+	**procedure** $GADD(OPERAND1, OPERAND2, VYSLEDEK)$; **begin** $ULOZ\_DO\_STRADACE(OPERAND1, OPERAND2)$; $GENERUJ('\texttt{ADD}', OPERAND2)$; $STRDC := VYSLEDEK$ **end**;
-	**procedure** $GSUB(OPERAND1, OPERAND2, VYSLEDEK)$; **begin** $ULOZ\_DO\_STRADACE(OPERAND1, OPERAND1)$; $GENERUJ('\texttt{SUB}', OPERAND2)$; $STRDC := VYSLEDEK$ **end**;
*	**procedure** $GMUL(OPERAND1, OPERAND2, VYSLEDEK)$; {analogicky s procedurou $GADD$}
/	**procedure** $GDIV(OPERAND1, OPERAND2, VYSLEDEK)$; {analogicky s procedurou $GSUB$}
unární minus	**procedure** $GUM(OPERAND1, VYSLEDEK)$; **begin** {příslušná čtveřice má pouze jeden operand} $ULOZ\_DO\_STRADACE(OPERAND1, OPERAND1)$; $GENERUJ('\texttt{CHS}', NIC)$; $STRDC := VYSLEDEK$ **end**;

Tab. 10.2: Procedury generátoru

Příklad 10.1. Uvažujme aritmetický výraz `((A + B * C) - A * B) * C`. Tab. 10.3 ilustruje činnost procedur generátoru aplikovaných na jednotlivé čtveřice vzniklé překladem zadaného výrazu.

Při pozornější analýze generovaného úseku cílového programu pro tento aritmetický výraz zjistíme, že počet generovaných instrukcí přesunu mezi střadačem a pamětí není minimální. Ještě vážnějším nedostatkem této metody je předpoklad, že pomocná proměnná reprezentující operand v určité čtveřici již není použita v žádné následující čtveřici. V opačném případě by totiž nemusela být hodnota tohoto mezivýsledku k dispozici v paměti. To ovšem vylučuje možnost optimalizace na úrovni čtveřic, která eliminuje výpočet společných podvýrazů, kdy všechny společné podvýrazy jsou reprezentovány právě hodnotou takové pomocné proměnné. Abychom tento nedostatek odstranili, museli bychom dodat do vnitřního tvaru programu informaci o tom, zda je proměnná živá, či nikoli a změnit algoritmus procedury $ULOZ\_DO\_STRADACE$.

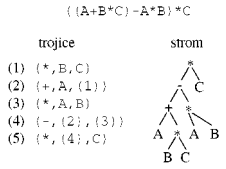
Čtveřice	Generované instrukce	Obsah proměnné $STRDC$
		$nedef$
(*,B,C,T1)	LOAD B	
	MUL C	T1
(+,A,T1,T2)	ADD A	T2
(*,A,B,T3)	STORE T2	
	LOAD A	
	MUL B	T3
(-,T2,T3,T4)	STORE T3	
	LOAD T2	
	SUB T3	T4
(*,T4,C,T5)	MUL C	T5

Tab. 10.3: Generování instrukcí pro čtveřice

Generování pro trojice

Jako další z klasických metod generování instrukcí pro aritmetické výrazy popíšeme nyní metodu, která vychází z vnitřního tvaru programu ve formě trojic. Tato metoda odráží dvě důležité vlastnosti technik generování — rekurzi a rozhodovací tabulky.

Trojice, na rozdíl od čtveřic, nespecifikují explicitně výsledek operace. Pro binární operaci mají tvar $(OPERATOR, OPERAND1, OPERAND2)$, kde na místě operandu stojí proměnná nebo ukazatel na trojici, která určuje příslušný operand. Posloupnost trojic lze chápat jako linearizovaný zápis stromové vnitřní formy programu, jak ilustruje obr. 10.2 na příkladě aritmetického výrazu z příkl. 10.1.



Obr. 10.2: Reprezentace výrazu trojicemi a stromem

Základem této metody generování je rekurzivní procedura, kterou nazveme $KOMP$, jejímž úkolem je zpracovat danou trojici, tj. generovat instrukce pro operaci specifikovanou touto trojicí. Výběr akcí, které procedura $KOMP$ provede, závisí na operátoru trojice a na umístění jejích operandů. Vzhledem k operandům mohou nastat tři případy:

(1) operand je ve střadači,

(2) operand je reprezentován symbolickou proměnnou,

(3) operand je reprezentován jinou trojicí, přesněji ukazatelem na ni.

Trojice, která má být zpracována, je určena parametrem procedury $KOMP$. Nastává-li případ (3), pak procedura $KOMP$ volá rekurzivně samu sebe pro specifikovanou trojici. Řízení

OPE-RÁTOR	OPERAND2 OPERAND1	STŘADAČ	PROMĚNNÁ	TROJICE
+	STŘADAČ		GEN('ADD',OPER2)	T:=NEWTEMPORARY GEN('STORE',T) KOMP(OPER2) GEN('ADD',T)
	PROMĚNNÁ	GEN('ADD',OPER1)	GEN('LOAD',OPER1) GEN('ADD',OPER2)	KOMP(OPER2) GEN('ADD',OPER1)
	TROJICE		KOMP(OPER1) GEN('ADD',OPER2)	KOMP(OPER1) OPER1:='STRADAC' KOMP(SEBE)
-	STŘADAČ		GEN('SUB',OPER2)	
	PROMĚNNÁ	T:=NEWTEMPORARY GEN('STORE',T) OPER2:=T KOMP(SEBE)	GEN('LOAD',OPER1) GEN('SUB',OPER2)	KOMP(OPER2) T:=NEWTEMPORARY GEN('STORE',T) OPER2:=T KOMP(SEBE)
	TROJICE	GEN('STORE',T) KOMP(OPER1) GEN('SUB',T)	KOMP(OPER1) GEN('SUB',OPER2)	KOMP(OPER2) OPER2:='STRADAC' KOMP(SEBE)
CHS		GEN('CHS',' ')	GEN('LOAD',OPER2) GEN('CHS',' ')	KOMP(OPER2) GEN('CHS',' ')

Tab. 10.4: Řízení procedury KOMP

procedury *KOMP* je popsáno rozhodovací tabulkou, v níž nejsou uvedeny případy operátoru * a /, poněvadž se liší od případů + a - pouze v generovaných instrukcích pro aritmetickou operaci. Použitá funkce *NEWTEMPORARY* definuje jméno nové pomocné proměnné pro uložení hodnoty v paměti. Prázdné položky tabulky odpovídají případům, které nemohou nastat. Rozeberme podrobněji některé případy z rozhodovací tabulky 10.4.

Uvažujme část odpovídající operátoru +, která využívá komutativity tohoto operátoru. Jsou-li oběma operandy trojice jména proměnných (případ označený PROMĚNNÁ × PROMĚNNÁ), pak je generována dvojice instrukcí LOAD<jméno prvního operandu> a ADD<jméno druhého operandu>. V případě PROMĚNNÁ × TROJICE, kdy na místě druhého operandu stojí ukazatel na jinou trojici, je nejdříve rekurzivně vyvolána procedura *KOMP* pro druhý operand zpracovávané trojice. Ta zpracuje příslušný "podstrom" tak, že budou vygenerovány instrukce vyčíslující výraz odpovídající podstromu a výsledek bude uložen ve střadači. Po návratu z rekurzivně volané procedury *KOMP* je generována vlastní instrukce ADD<jméno prvního operandu> s využitím komutativity.

Nejsložitější je případ TROJICE × TROJICE, kdy oba operandy jsou podvýrazy. Nejdříve je vyvolána procedura *KOMP*, která vytvoří instrukce pro podvýraz reprezentovaný prvním operandem. Ve druhém kroku je na místo prvního operandu zpracovávané trojice zapsána informace o dostupnosti prvního operandu ve střadači a vzápětí je vyvolána procedura *KOMP* pro zpracovávanou trojici s takto změněným operandem. Její činnost je nyní řízena položkou STŘADAČ × TROJICE a zahrnuje úschovu střadače do nově vytvořené pomocné proměnné,

Index zpracovávané trojice	Trojice	Rekurzivní volání procedury $KOMP$	Generované instrukce	Další akce
(5)	(*,(4),C)	KOMP(4)		
(4)	(-,(2),(3))	KOMP(3)		
(3)	(*,A,B)		LOAD A	
			MUL B	RETURN(3)
				KOMP(4)
(4)	(-,(2),STRADAC)		STORE T1	KOMP(2)
(2)	(+,A,(1))	KOMP(1)		
(1)	(*,B,C)		LOAD B	
			MUL C	RETURN(1)
(2)			ADD A	RETURN(2)
(4)			SUB T1	RETURN(4)
(4)				RETURN(4)
(5)			MUL C	RETURN(5)

Tab. 10.5: Činnost procedury $KOMP$

vytvoření instrukcí pro druhý operand zpracovávané trojice a konečně vygenerování instrukce součtu.

Zbývá dodat, že vygenerování instrukcí pro celý výraz dosáhneme vyvoláním procedury $KOMP(n)$, kde n je ukazatel na poslední trojici v posloupnosti reprezentující celý výraz (trojice odpovídající kořenu příslušného stromu).

V tab. 10.5 je ilustrována celá metoda generování na příkladě výrazu z obr. 10.2. Po vyvolání $KOMP(5)$ jsou zde sledována postupná rekurzivní volání, generované instrukce a ukončení procedury po zpracování k-té trojice ($RETURN(k)$).

Srovnáme-li výslednou posloupnost instrukcí s posloupností generovanou pro čtveřice, vidíme, že procedura KOMP dává kvalitnější výsledek, a to jak z hlediska počtu generovaných instrukcí, tak také z hlediska použitých pomocných proměnných. Tento rozdíl je způsoben pořadím, ve kterém jsou zpracovávány složitější operandy nekomutativních operací.

První metoda zpracovává čtveřice v pořadí jejich výskytu, které odpovídá generování instrukcí nejprve pro levý operand a pak pro pravý operand. Pro nekomutativní operace je v důsledku toho třeba uchovat obsah střadače reprezentujícího hodnotu prvého operandu do pomocné proměnné, přesunout levý operand do střadače a provést operaci.

Metoda, která používá trojice, zpracovává operandy nekomutativních operací v opačném pořadí. Nejdříve jsou generovány instrukce pro pravý operand. Po zpracování levého operandu pak není třeba měnit obsah střadače, ale je možné přímo generovat instrukci nekomutativní operace s vnitřní adresou reprezentující hodnotu pravého operandu.

V případě, že by posloupnost trojic představovala optimalizovaný vnitřní tvar výrazu s vyloučením výpočtu společných podvýrazů, by ovšem ani procedura $KOMP$ nedávalá správné výsledky. Přepsání operandu trojice na "STRADAC" by vedlo ke ztrátě hodnoty operandu v případě, že tento operand reprezentuje společný podvýraz používaný několika trojicemi. Za předpokladu, že by do vnitřního tvaru byla dodána informace o takových "sdílených" operandech, můžeme modifikovat proceduru $KOMP$ tak, aby generovala optimalizovaný cílový program.

Skupina metod, které jsme nazvali klasickými metodami generování cílového programu a

jejichž podstatu jsme ilustrovali zejména na příkladě generování instrukcí pro výraz reprezentovaný čtveřicemi, je na první pohled přímočará, avšak má některé nedostatky. Ve skutečnosti činnosti, které provádějí jednotlivé procedury generátoru, nejsou vzájemně nezávislé, ale závisejí na kontextu, ve kterém se zpracovávané operace vnitřního tvaru programu nacházejí. To obvykle vyžaduje zavedení globálních proměnných, či globálních údajových struktur, jejichž prostřednictvím se přenášejí důležité informace z jedné procedury do druhé.

Absence netriviální globální strategie řízení generátoru je zdrojem určitých nedostatků. Generovaný cílový program není příliš efektivní a vyžaduje obvykle další optimalizaci, která odstraňuje specifické nežádoucí rysy generovaného programu. Při tomto způsobu řízení na základě dodatečných globálních příznaků mohou při vzájemné spolupráci jednotlivých procedur snadno vzniknout situace, kdy je generován chybný výstup. Charakteristickým rysem této třídy metod je skutečnost, že kvalita a spolehlivost generovaného cílového programu je nadmíru závislá na pečlivosti a vtipnosti řešení generátoru a jeho implementaci.

10.3 Přidělování a přiřazování registrů

Efektivní využívání registrů v generovaném cílovém programu je obzvláště důležité, poněvadž instrukce specifikující jako operandy pouze registry jsou rychlejší a kratší než instrukce pracující s operandy v paměti. Efektivní využívání registrů vede k minimalizaci přesunů údajů mezi pamětí a registry instrukcemi typu LOAD, STORE, MOV, což opět výrazně přispívá k rychlosti generovaného programu.

Při návrhu generátoru se obvykle v souvislosti s problémem využívání registrů rozlišují dvě činnosti — přidělování registrů (registr allocation) a přiřazování registrů (registr assignement).

Přidělování registrů zahrnuje rozhodnutí, které objekty programu budou uchovávány trvale nebo přechodně v registrech jako operandy nebo složky specifikací přístupových cest. Přiřazování registrů určuje konkrétní registry, kde budou tyto objekty uloženy. Jeden z přístupů k přidělování a přiřazování registrů, který zjednodušuje návrh generátoru, vychází z pevného přiřazení skupin registrů určitým specifickým typům objektů cílového programu. Např. jednu skupinu tvoří registry zajišťující volání a návrat z podprogramů, další skupinu tvoří bázové registry, jiné registry jsou přidělovány lokálním a pomocným proměnným, jiné registry globálním proměnným. Tento přístup, je-li aplikován příliš striktně, může vést k tomu, že určitá skupina registrů je většinou nevyužitá, kdežto nedostatek registrů v jiné skupině způsobuje nadměrné generování instrukcí přesunu údajů mezi registry nebo mezi registry a hlavní pamětí.

V této podkapitole se zaměříme na některé techniky, které pomáhají řešit problém přidělování a přiřazování registrů. Budeme rozlišovat lokální a globální přidělování registrů podle toho, zda vybíráme registry pro proměnné v rámci základního bloku (viz kap. 9) nebo v rámci několika základních bloků.

10.3.1 Lokální přidělování a přiřazování registrů

Strategie lokálního přidělování a přiřazování registrů závisí na tom, zda optimalizujeme využití registrů v rámci základního bloku (sekvence příkazů vnitřního jazyka) nebo pouze v rámci výpočtu výrazu. V případě přidělování registrů v rámci základního bloku potřebujeme mít zřejmě k dispozici více registrů. Nejmenší uvažovaný počet je roven počtu registrů adresovatelných jedinou instrukcí (např. 4 registry na počítačích IBM/370 vzhledem k instrukci MVCL).

Jako první popíšeme metodu přidělování registrů pro proměnné základního bloku, která využívá informace o použití proměnné v základním bloku. Pod pojmem “použití proměnné”

rozumíme, stejně jako v kap. 9, výskyt proměnné na místě operandu. Toto použití se váže k určité platné definici této proměnné, která může být uvnitř, ale i vně základního bloku. Nejprve charakterizujme algoritmus, který pro každý příkaz `A:=B op C` vnitřního tvaru programu určí všechna další použití proměnných `A`, `B`, `C` v základním bloku. Princip tohoto algoritmu spočívá ve zpětném průchodu příkazy tvořícími základní blok (od posledního k prvnímu) a přenosu informace o dalším použití proměnné a informace, zda proměnná není živá, do předchozích příkazů. K tomu se používá tabulky symbolů, kde se tyto průběžné informace zapisují k proměnným základního bloku.

Algoritmus vyhledávání dalších použití proměnné provádí pro každý i-tý příkaz tvaru

```
(i) A := B op C
```

následující kroky v uvedeném pořadí:

1. Podle momentálního záznamu v tabulce symbolů opatři i-tý příkaz informací o dalším použití proměnných `A`, `B`, `C` a o jejich živosti.

2. K proměnné `A` zapiš do tabulky symbolů informaci "není živá" a "nemá další použití".

3. K proměnným `B` a `C` zapiš informaci "je živá" a "má použití v příkazu (i)".

Pravidla pro příkazy tvaru `A:=B` nebo `A:=op B` jsou stejná jako 1.–3. s tím, že operand `C` není uvažován.

Pokud uvažujeme přísně lokální využívání registrů v základním bloku, pak při přechodu do dalšího základního bloku jsou všechny registry "prázdné" — nemají definované hodnoty. Abychom zjistili, že hodnoty proměnných uchovávaných pouze v registrech a používaných vně základního bloku nebudou ztraceny, je třeba před opuštěním základního bloku generovat instrukce, které takové živé proměnné, jež nemají současné uložení v paměti, uloží do paměti. Proto musí mít popisovaný algoritmus na počátku k dispozici informaci o tom, které proměnné jsou živé po ukončení základního bloku. Tuto informaci získáme algoritmem 9.3 v případě, že je v rámci překladu prováděna analýza toku údajů.

Popisovanou metodu přidělování registrů však můžeme aplikovat i v případě, kdy není analýza toku údajů prováděna. Potom součástí algoritmu vyhledávání dalších použití proměnných je identifikace konce a začátku základního bloku tak, jak byla popsána v kap. 9 a bezpečný odhad těch proměnných, které mohou být živé po ukončení bloku. Takový odhad obvykle nezahrnuje pouze pomocné proměnné, jejichž platnost nemůže překročit hranice základního bloku.

Příklad 10.2. Ilustrujme nyní činnost algoritmu, který je podkladem pro lokální přidělování a přiřazování registrů na příkladě základního bloku s příkazy:

```
(1)  A  := B  + C
(2)  T1 := B  * F
(3)  B  := C  + T1
(4)  T2 := T1 + A
(5)  C  := T2 - F
```

Na počátku jsou jako živé proměnné identifikovány proměnné `A`, `B`, `C` a `F`.

Při zpracovaní příkazu (5) je nejdříve k tomuto příkazu připojena informace "`C`, `F` jsou živé, `T2` není živá" podle počátečního stavu tabulky symbolů a pak je tento stav změněn na "`C` není živá, `T2` a `F` mají použití v příkazu (5)." Při zpracovaní příkazu (4) je nejdříve tento příkaz opatřen informací "`T2` má použití v (5), `A` je živá, `T1` není živá" a změněm záznam v tabulce symbolů na "`T2` není živá a `A`, `T1` mají použití ve (4)." Stejným způsobem jsou zpracovány

příkazy (3), (2) a (1). Výsledek algoritmu uvádí tabulka 10.6, kde je získaná informace zapsána zkráceně.

Informace o použití a živosti proměnných		
(1)	A:=B+C	A má použití v (4), B v (2) a C v (3)
(2)	T1:=B*F	F živá, T1 má použití v (3),(4), F v (5)
(3)	B:=C+T1	B živá, T1 má použití v (4)
(4)	T2:=T1+A	A živá, T2 má použití v (5)
(5)	C:=T2-F	C, F živá

Tab. 10.6: Ilustrační příklad

Jestliže jsme zpětným průchodem základním blokem zjistili informaci o použití a živosti proměnných, můžeme normálním průchodem základním blokem provádět výběr instrukcí pro jednotlivé příkazy vnitřního jazyka spolu s přidělováním a přiřazováním registrů. K tomu je ovšem potřeba uchovávat informaci o momentálním obsahu registrů a informaci o tom, kde všude je proměnná v daném okamžiku dostupná (v registru, v paměti, v registru i paměti). Tuto informaci nám poskytuje *registrový a adresový deskriptor*, jenž je vytvořen pro každý registr a každou proměnnou základního bloku. Adresový deskriptor může být uchováván v tabulce symbolů.

Pro ilustraci nyní popišme jednu z možných jednoduchých strategií výběru registrů pro proměnnou `A` při překladu příkazu `A:=B op C`, kde `op` je nekomutativní operace. Výběr registrů je samozřejmě úzce svázán s možnostmi generovatelných instrukcí. Předpokládejme, že pro příkaz `A := B op C` bude generována instrukce `OP X1,X2`, které provádí akci `X2` ← `<X2> op <X1>` a `X1`, `X2` mohou být adresy registrů nebo paměti v libovolné kombinaci. Pravidla pro přidělení a přiřazení registrů proměnné `A` mohou mít tento tvar:

1. Je-li `B` v registru `R`, který neuchovává hodnoty jiných proměnných (v důsledku předchozích příkazů tvaru `X:=B`) a proměnná `B`, po provedení příkazu `A:=B+C`, nemá další použití a není živou proměnnou, pak proměnné `A` přiřaď registr `R`. Aktualizuj registrový deskriptor pro `R`.

2. Neplatí-li (1), přiděl proměnné `A` volný registr.

3. Pokud není k dispozici volný registr a má-li `A` další použití v základním bloku (nebo `op` je operátor, který vyžaduje registr jako např. indexování), pak vyber pro `A` obsazený registr R'. Generuj instrukci přesunu obsahu R' do paměti, není-li obsah R' současně v paměti, a aktualizuj registrový a adresový deskriptor.

4. Pokud `A` nemá v základním bloku další použití nebo pokud nelze najít vhodný obsazený registr, pak pro `A` vyber paměťové místo a případně inicializuj adresový deskriptor.

V tomto popisu strategie přidělování registrů nejsou některé body specifikovány detailně. Např. v bodě 2. může být přiřazení volného registru vázáno na podmínku, aby nebyly porušovány registrové páry, které jsou požadovány určitými instrukcemi. Podobně v bodě 3. může záviset výběr registru, který se bude uvolňovat pro `A`, na tom, zda existuje kopie tohoto registru nebo na tom, jak je vzdálené další použití uchovávané hodnoty. Předpokládáme, že proměnné, které nemají další použití v základním bloku a nejsou ani živé po opuštění bloku, uvolňují okamžitě příslušný registr.

Příklad 10.3. Pro základní blok z příkladu 10.2 bude generátor cílového programu generovat instrukce tak, jak uvádí obr. 10.7. Přidělování a přiřazování registrů se provádí podle popsané strategie využívající informace z obr. 10.6. Pro proměnné základního bloku jsou rezervovány registry `R1`, `R2` a `R3`. Popsané stavy deskriptorů se vztahují k okamžikům po zpracovaní příkazu vnitřního jazyka a jsou uvedeny pouze změny. Závěrečné instrukce přesunu jsou generovány na konci základního bloku pro ty proměnné, které jsou živé vně bloku a mají uložení pouze v registru.

Příkaz	Generované instrukce	Registrový deskriptor	Adresový deskriptor
		registry `R1`, `R2`, `R3` jsou prázdné	`B`, `C`, `F` je v paměti

(1)	`MOV B R1`
	`MOV C R2`
	`MOV R1 R3`
	`ADD R2 R3`
		`R1` obsahuje `B`
		`R2` obsahuje `C`
		
		`R3` obsahuje `A`
			`B` je v paměti a v `R1`
			`C` je v paměti a v `R2`
			`F` je v paměti
			`A` je v `R3`

(2)	`MUL F R1`	`R1` obsahuje `T1`	`T1` je v `R1`, `B` je v paměti
(3)	`ADD R1 R2`	`R2` obsahuje `B`	`B` je v `R2`
(4)	`ADD R3 R1`	`R1` obsahuje `T2`	`T1` nemá uložení, `T2` je v `R1`
(5)	`SUB F R1`	`R1` obsahuje `C`	`C` je v `R1`

	`MOV R1 C`
	`MOV R2 B`
	`MOV R3 A`
		
		registry `R1`, `R2`, `R3` jsou prázdné
			`A` je v paměti
			`B` je v paměti
			`C` je v paměti

Tab. 10.7: Přidělování registrů při generování instrukcí

10.3.2 Přidělování registrů pro překlad výrazů

Metoda přidělování registrů, se kterou se seznámíme, je lokalizována na posloupnost instrukcí odpovídajících překladu výrazu. Tato metoda umožňuje předem naplánovat minimální počet registrů, které musí být přiděleny, aniž by bylo potřeba ukládat mezivýsledky výpočtu do paměti, a stanovuje optimální pořadí vyhodnocování překládaného výrazu. Optimální pořadí je chápáno ve smyslu takového pořadí, které dává nejkratší posloupnost instrukcí pro daný výraz, nikoliv však nutně časově nejrychlejší posloupnost instrukcí.

Princip metody vychází z reprezentace výrazu stromem. Pro vrcholy stromu mohou být spočítána celočíselná ohodnocení, která reprezentují minimální počet registrů potřebných pro vyhodnocení příslušného podstromu. Ohodnocení kořene pak představuje minimální počet registrů pro vyhodnocení výrazu, při němž není třeba uchovávat mezivýsledky výpočtu v paměti. Překračuje-li ohodnocení některého vrcholu počet dostupných registrů, pak je nutné ukládat mezivýsledky výpočtu do paměti.

Předpokládejme, že se vyčíslení výrazu bude realizovat pouze v registrech, přičemž máme k dispozici maximálně n registrů. Dále pro jednoduchost předpokládejme, že všechny operace ve výrazu jsou binární. Uvažujme nyní operaci, jejíž levý operand vyžaduje k_1 registrů a pravý operand k_2 registrů a nechť $k_1 > k_2$. V tomto případě bude vždy prvním zpracovávaným operandem levý operand. Výsledek tohoto zpracování zůstává v registru jako mezivýsledek, takže pro vyhodnocení druhého operandu potřebujeme $k_2 + 1$ registrů. Poněvadž $k_2 + 1 \leq k_1$, nemůže

počet registrů pro vyčíslení celé operace převýšit k_1. Je-li $k_1 < k_2$, vyčíslujeme nejdříve pravý operand a potřebujeme analogicky nejvýše k_2 registrů.

V případě $k_1 = k_2$ nezáleží na pořadí vyhodnocování operandů. Vyhodnocení prvního operandu vyžaduje k_1 registrů, vyhodnocení druhého operandu $k_1 + 1$ registrů, poněvadž uchováváme výsledek reprezentující první operand. Pro vyhodnocení celé operace je tedy potřeba $k_1 + 1$ registrů.

Je-li $k_1 = k_2 = n$, pak nelze příslušný výraz (podvýraz) vyčíslit v dostupných registrech. V tom případě vyhodnotíme nejdříve jeden z operandů (obvykle pravý) v n registrech a výsledek uložíme do paměti. Tím uvolníme všech n registrů pro vyčíslení druhého operandu a po přesunu hodnoty prvního operandu do některého z $(n-1)$ volných registrů dokončíme celou operaci.

Proces zahrnující ohodnocování vrcholů stromu výrazu počtem potřebných registrů, určování pořadí vyhodnocování operandů a určování, kdy je třeba uložit mezivýsledek do paměti, můžeme popsat atributovou gramatikou.

V gramatice s pravidly

$$E \to a \mid E\ op\ E \mid (E)$$

popisující uvažované výrazy s binárními operátory zavedeme pro nonterminální symbol E tyto atributy:

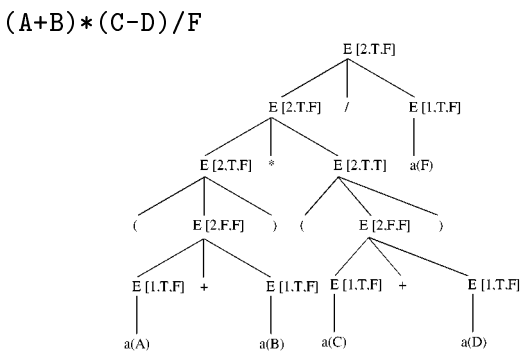
- *E.k* — počet registrů pro vyčíslení výrazu E,
- *E.pořadí* — atribut určující pořadí vyhodnocování operandů výrazů E; je-li *E.pořadí=true*, pak je třeba výraz E vyčíslit v pořadí: levý operand, pravý operand, je-li *E.pořadí=false*, pak je pořadí vyčíslování opačné,
- *E.paměť* — atribut určující uložení hodnoty výrazu E do paměti (v případě *E.paměť=true*), resp. do registru (v případě *E.paměť=false*).

Pravidla atributové gramatiky jsou uvedena v tab. 10.8.

PRAVIDLA	SÉMANTICKÉ AKCE
$E \to a$	$E.k := 1$
	$E.pořadí := true$
$E_0 \to E_1$ **op** E_2	$E_0.pořadí := E_1.k > E_2.k$
	$E_0.k :=$ **if** $E_1.k = E_2.k$ **then** $min(E_1.k + 1, n)$
	else $max(E1.k, E2.k)$
	$E_1.paměť := false$
	$E_2.paměť := E_1.k = n$ **and** $E_2.k = n$
$E_0 \to (E_1)$	$E_0.k := E_1.k$
	$E_1.paměť := false$
	$E_0.pořadí := true$

Tab. 10.8: Pravidla atributové gramatiky

Na obr. 10.3 je proveden výpočet atributů pro výraz `(A+B)*(C-D)/F` pro případ $n = 2$ (n je dostupný počet registrů). Atributy neterminálního symbolu jsou zapsány ve tvaru $[k, pořadí, paměť]$. Poněvadž jsme sémantickým pravidlem `E.k:=1` pro $E \to a$ požadovali, aby




```
MOV D,R1              MOV T,R1
MOV C,R2              MUL R1,R2    R2:=(A+B)*(C-D)
SUB R1,R2             MOV F,R1
MOV R2,T   T:=C-D     DIV R1,R2
MOV B,R1
MOV A,R2
ADD R1,R2  R2:=A+B
```

Obr. 10.3: Atributový strom a odpovídající posloupnost instrukcí

všechny operandy aritmetických instrukcí byly v registrech, nelze uvedený výraz vyčíslit se dvěma registry bez přesunu mezivýsledku do paměti (podvýraz `(A+B)*(C-D)` vyžaduje tři registry).

V případě, že druhý operand aritmetické instrukce může být v registru nebo v paměti, lze snadno modifikovat přidělování registrů tak, že namísto $E.k := 1$ bude $E.k := 0$ v případě, že E je pravým (jednoduchým) operandem. Levý operand vždy požaduje alespoň jeden registr.

10.3.3 Globální přidělování registrů

Při lokálním přidělování registrů jsme uvažovali každý základní blok izolovaně. Museli jsme však zajistit, aby hodnoty živých proměnných byly po opuštění základního bloku dostupné v paměti, což vedlo ke generování instrukcí přesunu údajů mezi registry a pamětí. Jakmile nyní přejdeme do základního bloku, v němž jsou některé z těchto proměnných použity, budou generovány buď pomalejší instrukce s operandem v paměti, nebo instrukce opačného přesunu mezi pamětí a registrem.

Globální přidělování registrů se vyznačuje tím, že hodnoty proměnných, které jsou často používány, zůstávají v registrech i po překročení hranice základního bloku a jsou tedy přiděleny

registrům v oblasti zahrnující několik základních bloků. Největšího zisku globálního přidělení registrů se dosáhne v případě základních bloků tvořících vnitřní cyklus. Předpokládejme tedy, že globální přidělování registrů probíhá po tom, co byl vytvořen graf toku řízení programu (procedur), na základě kterého mohou být nalezeny cykly i jejich vnoření. Rovněž potřebujeme výsledky globální analýzy toku údajů, zvláště pak analýzy živých proměnných.

Určení proměnných, které budou přiděleny registrům globálně v rámci cyklu není jednoduchý problém. Některé programovací jazyky, jako např. jazyk C, dovoluje, aby programátor tuto informaci předal překladači prostřednictvím deklarace proměnných, jež mají být (v rámci procedury) uloženy v registrech (deklarace proměnných typu register). Toto řešení je však výjimečné a obvykle očekáváme, že výběr takových proměnných provede překladač sám, a že tyto proměnné se budou v různých cyklech přirozeně různit. Počet registrů, které lze přidělovat globálně, je omezený, poněvadž je třeba rezervovat určité registry pro lokální přidělování v rámci základních bloků. Je proto potřeba z mnoha kandidátů na globální přidělení vybrat ty, jež přinášejí největší zisk.

Jedna z prakticky používaných metod kvantifikuje tento zisk podle počtu použití proměnné v základních blocích tvořících cyklus. Přitom je brán ohled na lokální přidělování registrů v základním bloku podle počtu dalších použití proměnné, jak je uvedeno v předchozím odstavci. Předpokládejme, že v rámci cyklu L byl proměnné `A` globálně přidělen určitý registr `R`. Pak přínos tohoto přidělení je závislý na dvou faktorech:

1. na počtu použití proměnné `A` v základních blocích tvořících cyklus,

2. na počtu vyloučení přesunů obsahu registru `R` do paměti, je-li `A` živou proměnnou po ukončení základního bloku.

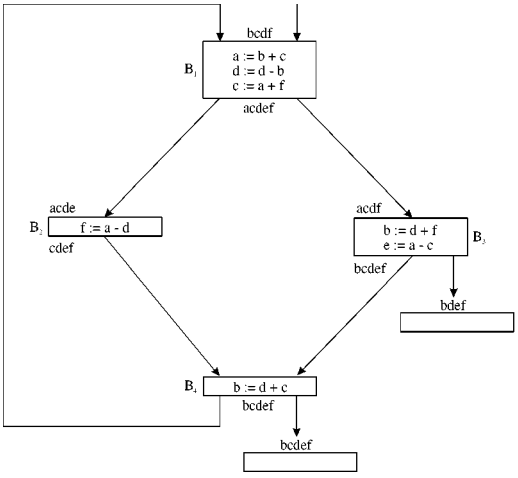
Jestliže použití proměnné `A` v základním bloku předchází v témže bloku definice proměnné `A`, pak existuje reálná možnost, že `A` bude vybrána pro lokálně přidělované registry a nebudeme tudíž přínos takového použití proměnné `A` zahrnovat do zisku globálního přidělení registru `R` proměnné `A`. Celkový přínos pro proměnnou `A` pak může být aproximován vztahem

$$\sum_{B_i \in L} (p.USE(\texttt{A}, B_i) + q.LIVE(\texttt{A}, B_i)), \tag{10.1}$$

kde $USE(\texttt{A}, B_i)$ udává počet použití proměnné `A` v základním bloku B_i, kterým nepředchází definice proměnné `A` v bloku B_i, $LIVE(\texttt{A}, B_i)$ je rovno 1, je-li `A` živá na výstupu z bloku B_i a současně má v B_i definici, jinak je $LIVE(\texttt{A}, B_i) = 0$, p, q jsou váhy, jejichž hodnoty kvantifikují zisk prvního a druhého členu součtu v konkrétním počítači. Součet vážených veličin USE a $LIVE$ probíhá přes všechny základní bloky tvořící cyklus L.

Vztah (10.1) je aproximací skutečného zisku při globálním přidělení registru proměnné `A` v cyklu L. Předpokládá, že počet opakování cyklu L je natolik velký, že akce spojené s přidělením registru, jako je případné definování registru `R` v prologu cyklu nebo přesunu obsahu `R` při opuštění cyklu, jsou zanedbatelné. Dále předpokládá, že všechny bloky cyklu L jsou prováděny stejně často.

Příklad 10.4. Uvažujme graf toku řízení cyklu na obr. 10.4. Jeho základní bloky neobsahují, pro stručnost, příkazy větvení (předpokládáme, že tyto příkazy neovlivní výběr proměnných pro globální přidělení registrů). Graf je doplněn výsledkem analýzy živých proměnných (živé proměnné na vstupu a výstupu každého základního bloku).



Obr. 10.4: Výběr proměnných pro globální přidělení registrů v cyklu L

Vyčíslíme nyní pro jednotlivé proměnné vztah (10.1). Uvažujme hodnoty vah $p = 1$ a $q = 2$. Je vidět, že proměnná `a` je živá na výstupu základního bloku B_1 a současně v něm má definici, ale není již živou proměnnou na výstupu bloků B_2, B_3 a B_4. Proto je

$$\sum_{B \in L} 2 * LIVE(\mathtt{a}, B) = 2.$$

Hodnota $USE(\mathtt{a}, B_1)$ je rovna nule, poněvadž použití proměnné a v tomto bloku nepředchází žádná její definice na rozdíl od bloků B_2 a B_3, kde tedy platí $USE(\mathtt{a}, B_2) = USE(\mathtt{a}, B_3) = 1$. V bloku B_4 není proměnná `a` vůbec použita, takže $USE(\mathtt{a}, B_4) = 0$. Proto

$$\sum_{B \in L} USE(\mathtt{a}, B) = 2$$

a hodnota vztahu (10.1) pro $A =$ `a` je 4. Vybereme-li tedy pro globální přiřazení registru proměnnou `a` je příslušný zisk kvantifikován hodnotou 4. Stejným způsobem můžeme spočítat zisky pro proměnné `b`, `c`, `d`, `e`, `f`, které činí postupně 5, 3, 6, 4, 4. Máme-li např. pro globální přiřazení k dispozici 3 registry, pak vybereme proměnné `b`, `d` a některou z proměnných `a`, `e` nebo `f`.

10.3.4 Globální přidělování s využitím barvení grafu

Metoda, kterou nyní stručně popíšeme, je založena na formulaci problému přidělování registrů v podobě barvení neorientovaného grafu a na rozdíl od jiných metod předpokládá podstatně větší počet registrů.

Informační struktura, nad níž se řeší přidělování registrů, se nazývá *interferenční graf*. Jeho vrcholy tvoří symbolické (abstraktní) registry, odpovídající proměnným vnitřního tvaru programu. Dva vrcholy grafu interferují, jestliže příslušné proměnné jsou současně živé, tj., je-li jedna z nich živá v oblasti platnosti definice druhé proměnné. Interferující vrcholy jsou spojeny hranou. Je zřejmé, že proměnným odpovídajícím takovým vrcholům nemůže být přidělen jediný registr (nemohou ho postupně sdílet), a proto při barvení grafu musí těmto vrcholům příslušet různé barvy (v důsledku jejich incidence). Je-li možné celý interferenční graf obarvit n barvami, kde n je počet skutečných registrů, které jsou k dispozici, pak obarvení grafu dává výsledné přidělení registrů. Proměnné odpovídající stejně obarveným vrcholům budou přidělovány jedinému registru.

Interferenční graf je také podkladem pro optimalizaci, která odstraňuje zbytečné operace kopírování obsahu registrů tak, že vrcholy, které odpovídají kopiím registrů, jsou sloučeny do jediného vrcholu. To je možné udělat pouze tehdy, když slučované vrcholy spolu neinterferují. Po sloučení vrcholů interferenčního grafu nebude třeba generovat instrukce přesunu mezi registry a rovněž velikost grafu se zmenší.

Algoritmus barvení grafu je založen na velmi jednoduchém poznatku: graf G lze obarvit n barvami tehdy, když lze n barvami obarvit redukovaný graf G', který byl získán z grafu G vypuštěním vrcholu stupně menšího než n. Algoritmus postupně odebírá z interferenčního grafu všechny vrcholy mající stupeň menší než n. Tento proces, který v každém kroku snižuje stupně zbývajících vrcholů grafu, často vede až k redukci na prázdný graf, což indikuje, že výchozí interferenční graf je obarvitelný n barvami. V tomto případě získáme hledané obarvení zpětným postupem, který vede k danému interferenčnímu grafu; přidáváním vrcholů v opačném pořadí, než v jakém byly tyto vrcholy odebírány. S přidáním každého vrcholu je nalezena jeho barva, tak aby byla splněna podmínka obarvení (tato barva vždy existuje, poněvadž stupeň každého vrcholu je v okamžiku přidání menší než n). Uvedený algoritmus má lineární výpočtovou složitost a v aplikacích pro přidělování 32 registrů při překladu podmnožiny jazyka PL/I dával výborné výsledky [10]. Připomeňme, že klasická úloha nalezení chromatického čísla grafu je NP-úplný problém.

Popsaný algoritmus nevede k cíli pouze v případě, že v určitém kroku není v redukoveném grafu G' žádný vrchol, který má stupeň menší než n. V tom případě je třeba modifikovat interferenční graf (a s ním i vnitřní tvar programu) takovým způsobem, že se vyjme některý jeho vrchol a výsledek příslušného výpočtu pak nebude uchováván v registru, ale v paměti. V cílovém programu se pak generuje instrukce přesunu mezi pamětí a registrem. Pro výběr takového vrcholu je možné využít ohodnocení "ceny" vyjmutí vrcholu z interferenčního grafu, která je závislá např. na stupni vrcholu nebo lokálním použití odpovídající proměnné v základním bloku. Je možné dokonce namísto uchování údaje v paměti generovat opakovaný výpočet v registrech, pokud je cena tohoto výpočtu menší než cena uchování výsledku v paměti. Vyjmutí určitého vrcholu však mění výchozí podmínky interference, proto je nutné celý postup opakovat od počátku a to tak dlouho, až získáme graf obarvitelný (a obarvený) n barvami. Ve zmíněné aplikaci této metody však již druhá iterace byla obvykle úspěšná.

U architektur, které se vyznačují vysokou regularitou, je možné graf budovat pouze nad symbolickými registry odvozenými z vnitřního tvaru překládaného programu. Metoda barvení grafu

umožňuje začlenit do formulace problému i omezení v používání registrů, která jsou dána nesymetriemi většiny současných architektur. Na rozdíl od regulárního využívání registrů je však třeba interferenční graf vytvářet nejen nad abstraktními registry, ale rovněž každému strojovému registru odpovídá jeden vrchol grafu. Všechny strojové registry spolu vzájemně interferují. Jestliže např. registr 0 nemůže být použit jako bázový registr nebo indexregistr (IBM 370, EC), pak vrchol odpovídající registru 0 interferuje se všemi abstraktními registry, jejichž hodnota představuje hodnotu bázového registru nebo indexregistru. Podobně lze částečně začlenit omezení na použití registrových párů. Např. v případě násobení lze předepsat interferenci abstraktního operandu násobení s každým sudým registrem, čímž docílíme, že mu bude přidělen lichý registr. Zavedením nového abstraktního registru a jeho interferencí se všemi lichými strojovými registry a interferencí s živými abstrakními registry v okamžiku násobení docílíme, že bude tomuto pomocnému abstraktnímu registru přidělen sudý strojový registr. Přidělené registry však nemusí tvořit požadovaný pár a pak je třeba generovat instrukce přesunu mezi registry. S aplikací uvedené metody přidělování registrů se setkáme v kap. 11.

10.4 Využití formálních a atributovaných překladů

Nyní se budeme zabývat třídou metod generování cílového programu, jejichž základním východiskem je překladová gramatika. Tyto metody odrážejí celkem zjevnou skutečnost. Proces syntézy strojového programu z vnitřního tvaru je překladem stejně jako proces syntézy vnitřního tvaru programu ze zdrojového programu. Je proto překvapující, že praktické aplikace rozvinuté teorie překladu se ve fázi generování cílového programu objevily, vzhledem k ostatním částem překladače, velmi pozdě, až koncem 70. let.

Nejdříve ukážeme na dvou příkladech atributových gramatik základní princip řízení generátoru cílového programu, který odstraňuje některé nedostatky konvenčních metod generování, jež byly popsány v podkapitole 10.2. Tento princip můžeme nazvat syntaxí řízené generování cílového programu, poněvadž řídící funkci generátoru přebírá syntaktický analyzátor konstruovaný pro atributovou překladovou gramatiku popisující překlad z vnitřního jazyka překladače do cílového jazyka.

10.4.1 Příklady překladových gramatik pro specifikaci generátoru

Uvažujme, podobně jako v odst. 10.2.1 počítač s jedním střadačem a s příslušným souborem jednoadresových aritmetických instrukcí a instrukcemi přesunu mezi střadačem a hlavní pamětí. Ukážeme možné tvary atributovaných překladových gramatik, které popisují překlad z vnitřního tvaru programu do posloupností instrukcí tohoto počítače. Omezíme se na překlad přiřazovacího příkazu, který je v prvním případě ve vnitřním tvaru reprezentován prefixovým zápisem a ve druhém případě postfixovým zápisem.

Syntax vstupního jazyka (prefixového zápisu) může být popsána touto LL(1) gramatikou:

$$G = (\{P, L, V\}, \{\texttt{:=}, \texttt{+}, \texttt{-}, \texttt{TA}, \texttt{DR}\}, R, P),$$

kde `TR`, resp. `DR` jsou operátory "Transfer Address," resp. "Dereference" a množina R obsahuje pravidla

$$\begin{aligned} P &\rightarrow \texttt{:=}\ L\ V \\ L &\rightarrow \texttt{TA} \end{aligned}$$

$$\begin{array}{rcl} V & \rightarrow & \texttt{+}\ V\ V \\ & | & \texttt{-}\ V\ V \\ & | & \texttt{DR TA} \end{array}$$

Neterminální symboly P, L, V značí postupně přiřazovací příkaz, proměnnou na levé straně přiřazovacího příkazu a výraz. Omezili jsme se pouze na jednu komutativní a jednu nekomutativní operaci. Pokud prefixová operace nepřipouští alternativní struktury operandů, pak popis syntaxe vede přímočaře k LL(1) gramatice. Nyní gramatiku G doplníme o atributy a výstupní symboly tak, aby výsledná atributová gramatika popisovala překlad na ekvivalentní posloupnosti instrukcí. Pro symboly překladové gramatiky zavedeme tyto atributy:

um umístění operandu (STR,PAM),

adr adresa operandu v paměti,

dso, *sso* dědičný a syntetizovaný atribut, udávající obsazení střadače (TRUE,FALSE),

dpa, *spa* dědičný a syntetizovaný atribut, udávající adresu pro uložení mezivýsledku.

Gramatiku G rozšíříme o čtyři výstupní symboly `ST`, `US`, `PLUS`, `MINUS`, které budou v závislosti na atributech reprezentovat akce generování instrukcí podle tabulky 10.9.

Atributová překladová gramatika bude mít pravidla uvedená v tab. 10.10.

Do pravidel $V \rightarrow$ `+`VV a $V \rightarrow$ `-`VV je vložen výstupní symbol `US`, který zajistí vytvoření instrukce pro uložení obsahu střadače v případě, že je střadač obsazen. Ukládání střadače je vynuceno tím, že při provádění intrukcí `ADD` a `SUB` je vždy střadač použit pro uložení jednoho z operandů. Podívejme se, jak bude přeložen přiřazovací příkaz

```
X:=(A+B)-(C+(D-E)),
```

kterému odpovídá prefixový vnitřní tvar:

(1)	`:=`	(9)	`+`
(2)	`TA adrX`	(10)	`TA adrC`
(3)	`-`	(11)	`DR`
(4)	`+`	(12)	`-`
(5)	`TA adrA`	(13)	`TA adrD`
(6)	`DR`	(14)	`DR`
(7)	`TA adrB`	(15)	`TA adrE`
(8)	`DR`	(16)	`DR`

Překladový strom pro tento prefixový zápis výrazu je uveden na obr. 10.5.

Po vyčíslení příslušných atributů podle sémantických pravidel generují výstupní symboly následující posloupnost instrukcí (předpokládáme, že počáteční adresa pro uložení mezivýsledku $V.dpa = 50$):

```
LOAD  adrA              ADD   adrC
ADD   adrB              STORE 51
STORE 50                LOAD  50
LOAD  adrD              SUB   51
SUB   adrE              STORE adrX
```

Ve druhém příkladu ukážeme atributovou gramatiku, která popisuje generování cílového programu pro stejný počítač, ale z postfixové vnitřní reprezentace přiřazovacího příkazu. Zápis

`ST(adrL,umV,adrV)`

`umV = STR`	`umV = PAM`
`STORE adrL`	`LOAD adrV` `STORE adrL`

(a) Přiřazení

`US(dso,dpa,spa)`

`dso = TRUE`	`dso = FALSE`
`STORE dpa` `spa:=dpa+1`	`spa:=dpa`

(b) Úschova střadače

`PLUS(umV1,adrV1,umV2,adrV2)`

	`umV1=STR`	`umV1=PAM`
`umV2=STR`		`ADD adrV2`
`umV2=PAM`	`ADD adrV1`	`LOAD adrV1` `ADD adrV2`

(c) Sečítání

`MINUS(umV1,adrV1,umV2,adrV2,dpa)`

	`umV1=STR`	`umV1=PAM`
`umV2=STR`		`SUB adrV2`

`umV2=PAM`	`STORE dpa`
	`LOAD adrV1`
	`SUB dpa`
		`LOAD adrV1`
		`SUB adrV2`

(d) Odečítání

Tab. 10.9: Význam výstupních symbolů `ST`, `US`, `PLUS` a `MINUS`

přiřazovacího příkazu v postfixové formě popisuje gramatika

$$G = (\{P, L, V\}, \{\texttt{:=}, \texttt{TA}, \texttt{+}, \texttt{-}, \texttt{DR}\}, R, P),$$

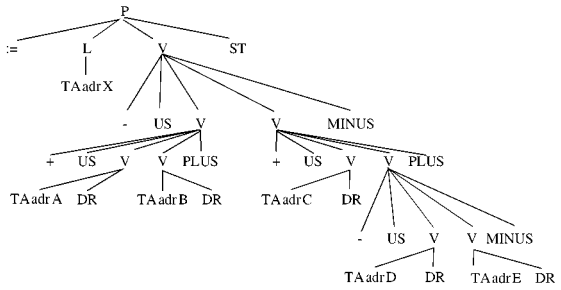
kde R obsahuje pravidla:

$$\begin{aligned} P &\rightarrow L\ V\ \texttt{:=} \\ L &\rightarrow \texttt{TA} \\ V &\rightarrow V\ V\ \texttt{+} \\ &\mid V\ V\ \texttt{-} \\ &\mid \texttt{TA DR} \end{aligned}$$

Mohli bychom se přesvědčit, že tato gramatika je silná LR(0) gramatika. Rozšíření gramatiky G na požadovanou atributovou překladovou gramatiku lze udělat podobně jako v předchozím

PRAVIDLA	SÉMANTICKÁ PRAVIDLA
$P \rightarrow := L\ V\ ST$	$V.dso := FALSE$
	$V.dpa := pocadresa$
	$ST.adr := L.adr$
	$ST.umV := V.um$
	$ST.adrV := V.adr$
$L \rightarrow TA$	$L.adr := TA.adr$
$V_0 \rightarrow + US\ V_1\ V_2\ PLUS$	$US.dso := V_0.dso$
	$US.dpa := V_0.dpa$
	$V_1.dso := FALSE$
	$V_1.dpa := US.spa$
	$V_2.dso := V_1.sso$
	$V_2.dpa := V_1.spa$
	$PLUS.umV_1 := V_1.um$
	$PLUS.adrV_1 := V_1.adr$
	$PLUS.umV_2 := V_2.um$
	$PLUS.adrV_2 := V_2.adr$
	$V_0.sso := TRUE$
	$V_0.spa := V_2.spa$
	$V_0.um := STR$
	$V_0.adr := 0$
$V_0 \rightarrow - US\ V_1\ V_2\ MINUS$	$US.dso := V_0.dso$
	$US.dpa := V_0.dpa$
	$V_1.dso := FALSE$
	$V_1.dpa := US.spa$
	$V_2.dso := V_1.sso$
	$V_2.dpa := V_1.spa$
	$MINUS.umV_1 := V_1.um$
	$MINUS.adrV_1 := V_1.adr$
	$MINUS.umV_2 := V_2.um$
	$MINUS.adrV_2 := V_2.adr$
	$MINUS.dpa := V_2.spa$
	$V_0.sso := TRUE$
	$V_0.spa := V_2.spa$
	$V_0.um := STR$
	$V_0.adr := 0$
$V \rightarrow DR\ TA$	$V.adr := TA.adr$
	$V.um := PAM$
	$V.sso := V.dso$
	$V.spa := V.dpa$

Tab. 10.10: Atributová překladová gramatika pro generování z prefixového zápisu



Obr. 10.5: Překladový strom pro překlad prefixového zápisu příkazu `X:=(A+B)-((C+(D-E))`

případě. Ukažme však jinou variantu, kdy rozhodnutí o vytváření instrukce `STORE` pro uložení střadače provedeme na základě syntaktické struktury vstupní věty.

Stejně jako v předchozím příkladu použijeme atributy *adr* a *dpa*. Výstupní symboly `STORE`, `LOAD`, `ADD` a `SUB` odpovídají generovaným instrukcím. Jejich atributy reprezentují adresovou část instrukce. Syntaktická a sémantická pravidla jsou uvedena v tab. 10.11.

Pravidla překladové gramatiky jsou volena tak, že umožňují rozlišit situace, kdy operandy v operacích sčítání, odčítání a přiřazení jsou výrazy nebo proměnné. V případě, že oba operandy v operaci sčítání nebo odčítání jsou výrazy, vytváří se instrukce pro uložení střadače po překladu prvního operandu, protože střadač bude použit pro výpočet hodnoty druhého operandu. Dále se vytváří instrukce `STORE` v případě, kdy druhý operand operace odčítání je výraz, protože instrukce `SUB` předpokládá, že ve střadači je uložena hodnota prvního operandu.

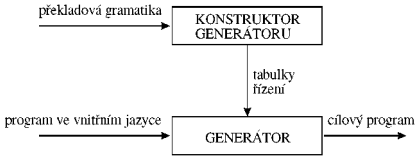
10.4.2 Graham-Glanvillovy metody generování cílového programu

Tímto názvem je v odborné literatuře označována třída moderních metod konstrukce generátoru cílového programu, která se vyznačuje obecností a strojovou nezávislostí algoritmů generování. Seznámíme se nyní se základními principy této metody, jejímž základem je modifikovaná překladová gramatika a dále s Ganapathiho rozšířením na atributovaný překlad.

Jádrem Glanvillovy metody, zobecňující proces konstrukce generátoru, je konstruktor, který na základě popisu instrukcí cílového počítače formou, která je velmi blízká bezkontextové překladové gramatice, vytváří tabulky pro řízení vlastního algoritmu výběru instrukcí při generování cílového programu. Schematicky je tento proces znázorněn na obr. 10.11. Popišme nyní jednotlivé složky, se kterými uvedená metoda pracuje.

PRAVIDLA	SÉMANTICKÁ PRAVIDLA
$P \rightarrow L$ `TA DR` $:= LOAD\ STORE$	$LOAD.adr :=$ `TA`$.adr$ $STORE.adr := L.adr$
$P \rightarrow L\ V\ :=\ STORE$	$STORE.adr := L.adr$ $V.dpa := P.dpa$
$L \rightarrow$ `TA`	$L.adr :=$ `TA`$.adr$
$V_0 \rightarrow V_1\ STORE\ V_2\ +\ ADD$	$V_1.dpa := V_0.dpa$ $STORE.adr := V_1.dpa$ $V2.dpa := V_0.dpa + 1$ $ADD.adr := V_0.dpa$
$V_0 \rightarrow$ `TA DR` $V_1\ +\ ADD$	$V_1.dpa := V_0.dpa$ $ADD.adr :=$ `TA`$.adr$
$V_0 \rightarrow V_1$ `TA DR` $+\ ADD$	$V_1.dpa := V_0.dpa$ $ADD.adr :=$ `TA`$.adr$
$V \rightarrow$ `TA`$_1$ `DR TA`$_2$ `DR` $+\ LOAD\ ADD$	$LOAD.adr :=$ `TA`$_1.adr$ $ADD.adr :=$ `TA`$_2.adr$
$V_0 \rightarrow V_1\ STORE_1\ V_2\ -\ STORE_2\ LOAD\ SUB$	$V_1.dpa := V_0.dpa$ $STORE_1.adr := V_0.dpa$ $V_2.dpa := V_0.dpa + 1$ $STORE_2.adr := V_0.dpa + 1$ $LOAD.adr := V_0.dpa$ $SUB.adr := V_1.dpa + 1$
$V_0 \rightarrow$ `TA DR` $V_1\ -\ STORE\ LOAD\ SUB$	$V_1.dpa := V_0.dpa$ $STORE.adr := V_0.dpa$ $LOAD.adr :=$ `TA`$.adr$ $SUB.adr := V_0.dpa$
$V_0 \rightarrow V_1$ `TA DR` $-\ SUB$	$V_1.dpa := V_0.dpa$ $SUB.adr :=$ `TA`$.adr$
$V0 \rightarrow$ `TA`$_1$ `DR TA`$_2$ `DR` $-\ LOAD\ SUB$	$LOAD.adr :=$ `TA`$_1.adr$ $SUB.adr :=$ `TA`$_2.adr$

Tab. 10.11: Atributová překladová gramatika pro generování z postfixového zápisu



Obr. 10.6: Zobecnění konstrukce generátoru

Vnitřní jazyk

Vnitřní jazyk IR (Internal Representation) je nižší úrovně než obvyklé jazyky symbolických instrukcí a podobá se spíše jazykům pro popis meziregistrových přenosů (např. ISP). Výrazy v jazyce IR jsou prefixové ekvivalenty překladových stromů, ve kterých je přístup k proměnným vyjádřen prefixovými zápisy výběru hodnot nebo adres. Řídící struktury jsou v IR vyjádřeny operacemi podmíněných a nepodmíněných skoků.

Na úrovni programu v IR nejsou ani hlavičky funkcí a procedur, ani deklarace objektů. Svůj obraz mají pouze příkazové části. Zápis $m.n$ označuje symbol syntaktické kategorie m, jehož hodnota je n. Např. $l.2$ značí návěští 2, $r.5$ značí registr 5, $k.-1$ značí konstantu -1. Zápis $k.CH$ reprezentuje adresu proměnné `CH`, nikoliv její hodnotu.

Na obr. 10.7 je uvedena deklarace funkce v jazyce PASCAL a jí odpovídající vnitřní tvar v jazyce IR pro překlad do strojového jazyka počítače PDP. Tento program odráží některé strojové a implementačně závislé vlastnosti vnitřního tvaru. Lokální proměnné jsou zpřístupňovány prostřednictvím bázové adresy. Např. hodnota proměnné `LVAL` je v jazyce IR pro PDP zpřístupněna výrazem

```
↑ + k.LVAL r.5.
```

Registr $r5$ obsahuje bázovou adresu a `k.LVAL` je posunutí. Sečtením dostaneme efektivní adresu a operátorem ↑ vybereme uloženou hodnotu. Dále předpokládáme, že globální proměnné jsou dostupné přímo bez bázového registru. Složené booleovské výrazy jsou vyhodnocovány tokem řízení (jako jednoduché výrazy a skoky). Operátor ? je operátorem srovnání.

Specifikace generátoru (překladová gramatika)

Na vstup konstruktoru generátoru cílového programu přichází informace o cílovém počítači, která je zapsána jazyce TMDL (Target Machine Description Language). Svou strukturovaností a čitelností patří TMDL k nejlepším specifikačním jazykům pro specifikaci generátorů.

Specifikace v TMDL obsahuje čtyři sekce, sekci *popisu voleb*, *registrů*, *symbolů* a *instrukcí*. Na obr. 10.8 je uvedena část specifikace generátoru cíloveho programu pro počítač PDP.

Sekce popisu voleb slouží k nastavení požadovaných tisků pro účely ladění. Sekce registrů obsahuje jména všech registrů cílového počítače a jejich rozčlenění na registry přidělovatelné (allocatable) v rámci generování a registry se speciálním určením (dedicated). Na obr. 10.8 je v první skupině rovněž registr pro nastavení podmínkového kódu (*cc*), ve druhé skupině je registr $r5$, jenž bude používán jako bázový registr, dále ukazatel na vrchol zásobníku (*sp*) a čítač instrukcí (*pc*).

```
{program  v PASCALU}
var ch:char;
function readn:integer;
var lval,base:integer;
begin
    while ch=' ' do read(ch);
    if (ch <= '9') and  (ch >= '0') then begin
        if  ch = '0'  then
            base:=8
              else
                  base:=10;
              lval:=0;
              repeat
                  lval:=lval*base+ord(ch)-ord('0');
            read(ch);
              until (ch < '0') or ((ord(ch)-ord('0') >= base);
              readn:=lval;
        end
        else
            readn:=-1;
  end {readn};
```

(a) zdrojový program

```
:l.1     <> l.2 ? ↑ k.CH k.' '
         jl.1
:l.2     > l.3 ? ↑ k.CH k.'9'
         < l.3 ? ↑ k.CH k.'0'
         := + k.BASE r.5 k.8
         j.l6
:l.5     := + k.BASE r.5 k.10
:l.6     := + k.LVAL r.5 k.0
:l.7     := + k.LVAL r.5 - + * ↑ + k.LVAL r.5 ↑ + k.BASE r.5 ↑ k.CH k.'0'
         < l.8 ? ↑ k.CH k.'0'
         < l.7 ? - ↑ k.CH k.'0' ↑ + k.BASE r.5
:l.8     := + k.READN r.5 ↑ +  k.LVAL r.5
         j l.4
:l.3     := + k.READN r.5 k.-1
:l.4
```

(b) vnitřní tvar v jazyce IR

Obr. 10.7: Příklad reprezentace v jazyce IR

Třetí sekce uvádí symboly, které jsou používány ve specikaci instrukcí v TMDL nebo ve vnitřním jazyce IR. Tyto symboly jsou rozděleny na proměnné (neterminální symboly) a terminální symboly. Jména registrových proměnných mohou být používána také v IR programu. V podsekci proměnných je provedena další strukturalizace množiny registrů. V případě počítače PDP jsou zavedeny sudé a liché registry, registrové páry a registr podmínkového kódu. V podsekci terminálních symbolů jsou stanoveny rozsahy elementárních operandů a počet operandů jednotlivých operátorů.

Nejobsáhlejší částí popisu je sekce instrukcí. Má tvar seznamu pravidel překladové gramatiky. V jednotlivých pravidlech jsou tyto částí:

a) levá strana pravidla, která určuje umístění výsledku,

b) pravá strana pravidla, která popisuje konstrukci IR jazyka a jí odpovídající instrukci cílového počítače.

Např. první pravidlo v sekci instrukcí na obr. 10.8 obsahuje frázi `(k.1)`, umístění v registru `r.1` a instrukci `mov #k.1, r.1`. Sémantická interpretace tohoto pravidla říká, že konstanta (literál) `k.1` může být do registru `r.1` přesunuta instrukcí `mov #k.1,r1` (`#` je označení literálu). Některá pravidla mají na levé straně symbol označující "prázné" umístění výsledku, což se uplatňuje v případech, kdy operátor `:=` je součástí IR konstrukce.

Zápis `m.n` má různý význam v IR programu a TMDL programu. V IR programu `n` značí hodnotu symbolu `m`. V TMDL je `n` sémantický kvalifikátor symbolu `m`, který identifikuje různé výskyty téhož symbolu `m`. Např. v posledním pravidle pro instrukce `mov` značí `r.1` a `r.2` libovolný z registrů $r0, r1, \ldots, r4$. Kvalifikátory 1 a 2 slouží k popisu korespondence registrů v části vzoru a v části instrukce. Podobně je tomu v případě konstant `k.1` a `k.2` v témže pravidle.

Konstruktor tabulek generátoru

Funkce konstruktoru, který na základě specifikace cílového počítače vytváří tabulky, jež slouží k řízení obecného algoritmu generování, je analogická funkci konstruktoru tabulek pro LR analyzátory. Výstupní tabulky akcí a přechodů odpovídají pravidlům, jež jsou zapsány v sekci instrukcí TMDL programu a představuje řídící informaci překladového zásobníkového automatu. I když sekce instrukcí nemá přesný tvar bezkontextové překladové gramatiky, algoritmus konstruktoru dovede uvedené modifikace akceptovat, poněvadž nejsou principiálně odlišné.

Uvedená gramatika nemá např. jediný počáteční neterminál, ale každý symbol λ je považován za počáteční symbol. Program v jazyku IR tak může být zredukován na řetězec symbolů λ. Stačí však nahradit symbol λ nonterminálním symbolem L a přidat pravidla $S \rightarrow L \mid S\ L$ a celý IR program lze generovat z jediného počátečního symbolu S.

Závažným problémem, který musí konstruktor řešit, je nejednoznačnost příslušné gramatiky, která se projevuje řadou vznikajících konfliktů ve vytvářené tabulce akcí. V popisované metodě se tento problém neřeší transformací gramatiky, což by bylo vzhledem k povaze problému velmi obtížné, ne-li nemožné, ale stanovením určité strategie podle níž se z konfliktních akcí provádí jednoznačný výběr preferované akce. Při konfliktu typu *redukce/přesun* se dává přednost akci *přesun*, při konfliktu typu *redukce/redukce* pak preferujeme tu redukci, které přísluší delší redukční část. Tato jednoduchá heuristika zabraňuje generování jednoduchých instrukcí v případě, že mohou být vybrány instrukce, které mají komplexnější efekt. Obsahuje-li např. zásobník symboly

```
#options  statesets,tables,loops,items;
#registers i
    #allocatable r0,r1,r2,r3,r4,cc
    #dedicated  r5,sp,pc
#symbols
    #variables
          r=r0,r1,r2,r3,r4,r5,sp,pc;
          d=<r0,r1>,<r2,r3>;
          o=r1,r3;
          e=r0,r2;
          c=cc;
    #terminals
          k:0,32767;
          l:0,1023;
          + binary;  - binary;   * binary; / binary;
          ↑ unary;  := binary;  j unary;  : unary;
          ? binary;
          < binary;  >  binary; <= binary;
          >= binary; =  binary; <> binary;
          & binary;  ↑  binary;
          ! unary;   m  unary;
#instructions
    r.1 ::= (k.1)                       "mov #k.1,r.1";
    r.1 ::= (↑k.1)                      "mov *k.1,r.1";
    λ ::= (:=k.1 r.1)                   "mov r.1,*k.1";
    λ ::= (:=k.1↑k.2)                   "mov *k.2,*k.1";
    r.2 ::= (↑+k.1 r.1)                 "mov k.1(r.1),r.2";
    λ ::=(:=+k.1 r.1 r.2)               "mov r.2,k.1(r.1";
    r.2 ::= (↑r.1)                      "mov (r.1),r.2";
    λ ::=(:=r.1 r.2)                    "mov r.2,(r.1)";
    λ ::=(:=r.1↑r.2)                    "mov (r.2),(r.1)";
    r.2 ::= (↑↑+k.1 r.1)                "mov *k.1(r.1), r.2";
    λ ::=(:=k.1  k.2)                   "mov #k.2,*k.1";
    λ ::=(:=+k.1 r.1 k.2)               "mov #k.2,k.1(r.1)";
    λ ::=(:=↑+k.1 r.1  k.2)             "mov #k.2,*k.1(r.1)";
    λ ::=(:=r.1  k.1)                   "mov #k.1,(r.1)";
    λ ::=(:=r.1↑+k.2  r.2)              "mov k.2(r.2), (r.1)";
    λ ::=(:=r.1↑k.1)                    "mov *k.1,(r.1)";
    λ ::=(:=+k.1  r.1 +k.2 r.2)         "mov k.2(r.2),k.1(r.1)";
    r.1 ::= (+r.1 k.1)                  "add #k.1,r.1";
    r.1 ::= (+↑r.1 k.1)                 "add *k.1,r.1";
    r.1 ::= (-r.1 r.2)                  "sub r.2,r.1";
    r.1 ::= (-r.1 k.1)                  "sub #k.1,r.1";
    r.1 ::= (k=0)                       "clr r.1";
    λ ::= (:=r.1 k=0)                   "clr (r.1)";
    λ ::= (:=k.1 k=0)                   "clr (r.1)";
    λ ::= (:=+k.1 r.1 k=0)              "clr k.1(r.1)";
    r.1 ::= (+r.1 k=1)                  "inc r.1";
    e.1 ::= (r.1)                       "mov r.1,e.1";
    o.1 ::= (r.1)                       "mov r.1,o.1";
    d.1 ::= (r.1)                       "mov r.1,d.2;sxt d.1.1";
    d.1 ::= (*e.1↑+k.1  r.1)            "mul  k.1(r.1,e.1";
    d.1 ::= (*↑+k.1 r.1 e.1)            "mul  k.1(r.1),e.1";
    o.1 ::= (*o.1↑+k.1 r.1)             "mul k.1(r.1),o.1";
    o.1 ::= (*↑+k.1 r.1 o.1)            "mul k.1(r.1),o.1";
    λ ::= (:l.1)                        "L/.1:";
    c.1 ::= (?r.1 r.2)                  "cmp r.1,r.2";
    c.1 ::= (?k.1↑k.2)                  "cmp #k.1,*k.2";
    c.1 ::= (?↑k.1 k.2)                 "cmp *k.1, #k.2";
    λ ::= (<> l.1 c.1)                  "jne L/.1";
#end
```

```
+ r.0  ↑ k.CH,
```

pak je možné použít k redukci jedno z pravidel

```
r1 ::= (k.1)                    "mov  *k.1,r.1";
r.1 ::= (+r.1 ↑ k.1)            "add *k.1,r.1";
```

Podle uvedené strategie bude vybráno druhé pravidlo, což je v tomto případě pravděpodobně lepší. Existují však případy, kdy výběr prvního pravidla povede k efektivnějšímu cílovému programu.

Další nejednoznačnost při výběru pravidla pro redukci vzniká tehdy, když k jednomu syntaktickému vzoru (redukční části) existuje více pravidel. Např. pravidlům

```
r.1 ::= (+r.1 k.1)                "add #k.1,r.1";
r.1 ::= (+r.1 k=1)                "inc r.1"
```

přísluší stejná redukční část popisující součet obsahu registru a konstanty. Tento typ víceznačnosti (kolizí) je řešen vytvořením uspořádaných množin pravidel, příslušejících jednomu syntaktickému vzoru, kde uspořádání odráží vrůstající obecnost generovaných instrukcí. Při výběru pravidla jsou tak nejdříve zkoumány specifické případy, které vedou k efektivnějším instrukcím. V příkladě pro součet je tedy nejdříve zkoumána možnost k=1 a v kladném případě je vybrána instrukce inkrementující registr.

Jednou z důležitých vlastností metod generování založených na aplikaci překladových gramatik je vyšší stupeň spolehlivosti generátoru ve srovnání s klasickými metodami generování. Tato vlastnost se týká jak výstupu generátoru (bezchybnosti generovaného programu), tak i samotného procesu generování. Konstruktor generátoru provádí kromě syntézy výstupních tabulek také analýzu vstupní překladové gramatiky s cílem vyloučit ty stavy generátoru, které by vedly k jeho zablokování nebo k neustálému cyklickému provádění redukcí. Testování takových cyklů je usnadněno tím, že příslušná překladová gramatika neobsahuje ϵ-pravidla (pravidla s pravou stranou tvořenou prázdným řetězcem).

Prováděcí program generátoru (exekutor)

Algoritmus programu, který provádí výběr generovaných instrukcí je klasický algoritmus LR analýzy, který přesouvá do zásobníku vstupní symboly a redukuje obsah zásobníku pomocí vybraného pravidla. Po redukci pak realizuje výstup příslušné instrukce, která je součástí pravidla, podle něhož byla redukce provedena.

Na obr. 10.10 je ilustrována činnost algoritmu generování pro příkaz

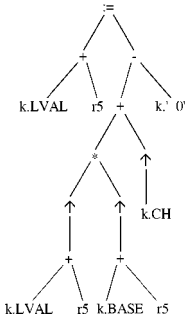
```
lval := lval*base+ord(ch)-ord('0'),
```

který se vyskytuje v těle funkce `readn` (viz obr. 10.7).

Překladový strom tohoto příkazu a odpovídající reprezentace v jazyce IR je uvedena na obr. 10.9. Činnost algoritmu je dokumentována výpisem vstupu, obsahu zásobníku a akcí, kterou algoritmus provádí. Při redukci uvádíme aplikované pravidlo i instrukci, která je přidávána do výstupu. Tento příklad ilustruje rovněž prioritu redukcí, které vedou k výstupu složitějších instrukcí (zpožďování redukcí).

10.4.3 Ganapathiho rozšíření o atributy

Ganapathiho metoda představuje přirozené rozšíření předchozího přístupu k návrhu a implementaci generátoru cílového programu. Toto rozšíření se vztahuje zejména na popis generátoru prostřednictvím atributové překladové gramatiky. Základní schéma konstrukce generátoru z obr.




```
:= + k.LVAL r.5 - + * ↑ + k.LVAL r.5 ↑ k.BASE r.5 ↑ k.CH k.'0'
```

Obr. 10.9: Překladový strom pro `lval:=lval*base+ord(ch)-ord('0')` a jeho reprezentace v IR

10.4 zůstává v podstatě zachováno. Na rozdíl od předchozí metody je položen důraz na strojovou nezávislost vnitřního jazyka a na začlenění strojově závislych optimalizací cílového programu. Charakterizujme nyní ty složky, u nichž dochází k významnějším rozdílům.

Vnitřní jazyk

Navržený vnitřní jazyk je založen na stejných principech jako jazyk IR. Zdrojový program je vyjádřen v prefixovém zápisu na nižší (detailnější) úrovni popisu operátorů i operandů. Symboly tvořící program ve vnitřním tvaru jsou opatřeny atributy. Reprezentace zdrojového programu ve vnitřním jazyce je strojově nezávislá. To např. znamená, že proměnné a parametry jsou popsány svými atributy a nikoli paměťovými místy či registry počítače. Zobrazení jazykových objektů na strojové objekty je řešeno až ve fázi generování stejně jako rozhodnutí o tom, jak adresovat lokální a globální objekty. Strojová nezávislost vnitřního jazyka, která zvyšuje přenositelnost překladačů, je zde dosažena určitým "sjednocujícím" přístupem, který je schopen odrazit rozdíly v populárních zdrojových jazycích (typu Pascal, Modula, C) i v cílových jazycích. Jedním z příkladů tohoto přístupu je pojem adresy obsahující pět složek

(1) bázový registr,

(2) posunutí vzhledem k bázi (offset),

(3) volitelná úroveň nepřímosti,

(4) indexregistr,

(5) jméno proměnné (je-li globální),

VSTUP	ZÁSOBNÍK	AKCE
:= + k.LVAL r.5 - + * ^+ k.LVAL r.5 ^+ k.BASE r.5 ^k.CH k.'0'		
#	přesun	
+ k.LVAL r.5 - + * ^+ k.LVAL ^+ k.BASE r.5 ^k.CH k.'0'		
:= +	přesun	
r.5 - + * ^+ k.LVAL r.5 ^+ k.BASE r.5 ^k.CH k.'0'		
:= + k.LVAL	přesun	
- + * ^+k.LVAL r.5 ^+ k.BASE r.5 ^k.CH k.'0'		
:= + k.LVAL r.5	přesun	
+ * ^+ k.LVAL r.5 ^+ k.BASE r.5 ^k.CH k.'0'		
:= + k.LVAL r.5 -	přesun	
* ^+ k.LVAL r.5^+ k.BASE r.5^k.CH k.'0'		
:= + k.LVAL r.5 - +	přesun	
^+ k.LVAL r.5 ^+ k.BASE r.5^k.CH k.'0'		
:= + k.LVAL r.5 - + *	přesun	
+ k.LVAL r.5 ^+ k.BASE r.5 ^k.CH k.'0'		
:= + k.LVAL r.5 - + *^	přesun	
k.LVAL r.5 ^+ k.BASE r.5 ^k.CH k.'0'		
:= + k.LVAL r.5 - + * ^+	přesun	
r.5 ^+ k.BASE r.5 ^k.CH k.'0' := + k.LVAL		
r.5 - + * ^+ k.LVAL	přesun	
^+ k.BASE r.5 ^k.CH k.'0'		
:= + k.LVAL r.5 - + * ^+ k.LVAL r.5	redukce	
'r.2 ::= (^+ k.1 r.1)'	"mov k.1(r.1),r.2"	
	mov lval(r5),r0	
^ +k.BASE r.5 ^ k.CH k.'0'		
:= + k.LVAL r.5 - + * r.0	přesun	
+ k.BASE r.5 ^ k.CH k.'0'		
:= + k.LVAL r.5 - + * r.0 ^	přesun	
k.BASE r.5 ^ k.CH k.'0'		
:= + k.LVAL r.5 - + * r.0 ^ +	přesun	
r.5 ^ k.CH k.'0'		
:= + k.LVAL r.5 - + * r.0 ^ + k.BASE	přesun	
^k.CH k.'0'		
:= + k.LVAL r.5 - + * r.0 ^ + k.BASE r.5	redukce	
'o.1 ::= (*o.1 ^ + k.1 r.1)'	"mul k.1(r.1),o.1"	
	mul base(r5),r0	
^ k.CH k.'0'		
:= + k.LVAL r.5 - + r.0	přesun	
k.CH k.'0'		
:= + k.LVAL r.5 - + r.0 ^	přesun	
k.'0'		
:= + k.LVAL r.5 - + r.0 ^ k.CH	redukce	
'r.1 ::= (+ r.1 ^ k.1)'	"add *k.1,r.1"	
	add ch,r0	
k.'0'		
:= + k.LVAL r.5-r.0	přesun	
:= + k.LVAL r.5-r.0 k.'0'	redukce	
'r.1 ::= (- r.1 k.1)'	"sub #k.1,r.1"	
	sub #60,r0	
:= + k.LVAL r.5 r.0	redukce	
'L ::= (:= + k.1 r.1 r.2)'	"mov r.2,k.1(r.1)"	
	mov r0,lval(r5)	

Obr. 10.10: Zpracování příkazu

které postačují pro zobrazení do adresovacích mechanismů širokého spektra počítačů.

Na obr. 10.11 je pro ilustraci uveden zdrojový program v jazyce C a jeho vnitřní tvar.

V uvedeném příkladě má symbol funkce `strncmp` tři atributy — počet parametrů, druh symbolu (*Function*) a typ funkce (*Integer*). Symboly `str1` a `str2` mají atributy *Parametr*, *pointer* a *Registr*. Další atributy značí `L`=lokální a `c`=znak. Symbol ↑ značí syntetizovaný atribut, symbol ↓ dědičný atribut, (použitý v dalším textu). Symbol `@` odpovídá operátoru `*` ve zdrojovém programu (ukazatel).

Specifikace generátoru

Specifikace generátoru cílového programu má tvar atributové překladové gramatiky, v níž jsou začleněny *akční symboly* a *predikáty* pro rozhodování nejednoznačností. Jednotlivá pravidla gramatiky mají spíše podobu popisu akcí, které se pro určitý syntaktický vzor musí provést, méně už podobu popisu instrukcí cílového jazyka, jak tomu bylo v předchozím případě. Uvažujme např. pravidlo:

```
Word ↑r -> + Word ↑a Word ↑b GETTEMP(↓'word'↑r)
                             EMIT(↓'mov'↓b↓r)
                             EMIT(↓'add'↓a↓r)
```

jež je součástí popisu generátoru pro počítače řady PDP-11/70. Toto pravidlo říká: mají-li být sečteny obsahy dvou slov `a`, `b`, pak je třeba nejprve přidělit pomocnou proměnnou (nejraději registr), generovat instrukci přesunu na toto přidělené místo a generovat instrukci pro součet. Tyto akce jsou popsány akčními symboly `GETTEMP` a `EMIT`.

Jestliže určitému syntaktickému vzoru přísluší několik alternativních instrukcí nebo posloupností instrukcí, pak je možné popisovat tyto alternativní případy pravidly, jež obsahují predikáty závislé na hodnotách atributů. V procesu generování se uplatní to pravidlo, jehož rozhodovací predikát je pro dané atributy pravdivý. S využitím rozhodovacích predikátů je řešen také problém generování specifických instrukcí cílového počítače i některé případy strojově závislé optimalizace. Uvažujme např. pravidla.

```
Word ↑r  -> + Word↑a Word↑b IsCons(↓a ↓b) KFOLD(↓+↓a↓b↓r)
          -> + Word↑r Word↑a IsOne(↓a) IsTemp(↓r) AUTOINC(↓'inc'↓r)
```

První pravidlo popisuje provedení výpočtu s konstantami (constant folding). Jsou-li `a` i `b` konstanty (`IsCons(a,b)=true`), pak se "provede" akční symbol `KFOLD`. Sečtou se hodnoty `a`, `b` a výsledek, syntetizovaný atribut `r`, je předán jako odpovídající atribut levé strany pravidla.

Druhé pravidlo popisuje častý speciální případ typu `i:=i+1`. Je-li `a`=1 (`IsOne(a)=true`) a `r` pomocná proměnná-registr (`IsTemp(r)=true`), pak je volán akční symbol `AUTOINC`. V rámci odpovídající akce, jež je variantou akce `EMIT`, se aplikuje, je-li to možné, autoinkrementační adresování namísto generování instrukce inkrementu. Výsledkem je pak "sloučení" dvou instrukcí do jediné instrukce.

Na obr. 10.12 je uvedena část atributové gramatiky pro specifikaci generátoru cílového programu pro počítače PDP-11/70. Celá gramatika je rozčleněna na osm skupin pravidel, které popisují adresování, zobrazení údajových typů, přesuny, speciální instrukce, aritmetické a logické operace, relační operace a větvení programu a volání podprogramu.

Úplný popis přesahuje sedm stran a příslušná gramatika obsahuje okolo deseti akčních symbolů a dvaceti rozhodovacích predikátů. Na obr. 10.12 jsou zastoupena pravidla z každé skupiny s výjimkou speciálních instrukcí. Akční symboly a rozhodovací predikáty jsou rozlišeny zápisem tak, že jméno akčního symbolu obsahuje pouze velká písmena.

```
#define SAME  0
#define DIFF  1
strncmp(str1,str2,len);
register char *str1,*str2;
register int len;
{
    /* Funkce  strncmp porovnává řetězce  str1 a str2; jsou-li shodné má
     * hodnotu 1; v  opačném případě má hodnotu 0
     */
    register int i;
    for(i=0; i < len; i++)
        if(*str1++ != *str2++) return(DIFF);
    return(SAME);
}
```

(a) Zdrojový program v jazyce C

```
 1. :strncmp ↑3  ↑F ↑I {
 2.       :str1  ↑P ↑1 ↑p ↑R
 3.       :str2 ↑P ↑1 ↑p ↑R
 4.       :len ↑P ↑1 ↑I ↑R
 5.       :i ↑L  ↑1 ↑I ↑R
 6.       :temp ↑L ↑1 ↑c ↑R
 7.       ;
 8.       := i 0
 9.       goto  L25
10. L2001
11.       := temp = @↑c str1 @↑c str2
12.       := str1 + str1 SIZE↑c
13.       := str2 + str2 SIZE↑c
14.       0 <> temp L23
15.       := strncmp 1
16.       goto L13
17. L23
18.       := i + i1
19. L25
20.       < i  len L2001
21.       := strncmp 0
22. L13
23. }
```

(b) Vnitřní tvar programu

Obr. 10.11: Zdrojový a vnitřní tvar programu

(a) Instrukce adresování

```
Address↑a -> DirectModes↑a
 -> IndirectModes↑a
IndirectModes↑a -> @ DirectModes↑b NotIndirect(↑b) ADDR(↑@ ↑b ↑a)
 -> AnotherLevel↑a
DirectModes↑a -> Datum↑a
 -> #Datum↑b ADDR(↑# ↑b ↑a)
 -> Disp↑b Base↑c ADDR(↑b ↑c ↑a)
 -> Register
 -> Subsumptions
Base↑a  -> DirectModes↑a IsReg(↑a)
 -> DirectModes↑b GETREG(↑'word' ↑a) EMIT(↑'mov' ↑b ↑a)
AnotherLevel↑ -> @IndirectModes↑b GETREG(↑'word' ↑r) EMIT(↑'mov' ↑b ↑r) ADDR(↑@ ↑r ↑a)
Subsumptions↑a
 -> @ + Word↑b Word↑c Iscons(↑c) IsReg(↑b) ADDR(↑+ ↑b ↑c ↑a)
 -> @ + Word↑b Word↑c Iscons(↑b) IsReg(↑c) ADDR(↑+ ↑b ↑c ↑a)
```

(b) Instrukce zobrazení údajových typů

```
Byte↑a   -> Address↑a IsByte(↑a)
Word↑a   -> Address↑a IsWord(↑a)
Float↑a  -> Address↑a IsFloat(↑a)
Double↑a -> Address↑a IsDouble(↑a)
```

(c) Instrukce přesunutí

```
Assignment
 -> := Word↑a Word↑b IsZero(↑b) EMIT(↑'clr' ↑a)
 -> := Word↑a Word↑b DELAY(↑'mov' ↑b ↑a)
```

(e) Aritmetické a logické instrukce

```
Word↑r
 -> + Word↑a Word↑b IsCons(↑a ↑b) KFOLD(↑+ ↑a↑b ↑r)
 -> + Word↑a Word↑r IsOne(↑a) IsTemp(↑r) AUTOINC (↑'inc'↑r)
 -> + Word↑r Word↑a IsOne(↑a) IsTemp(↑r) AUTOINC (↑'inc'↑r)
 -> + Word↑r TwoFour(↑a) IsTemp(↑r) AUTOINC (↑'add'↑a ↑r)
 -> + Word↑a Word↑r IsTemp(↑r) EMIT(↑'add' ↑a ↑r)
 -> + Word↑r Word↑a IsTemp(↑r) EMIT(↑'add' ↑a ↑r)
 -> + Word↑a Word↑b GETTEMP(↑'word' ↑r) EMIT (↑'mov' ↑b ↑r) EMIT (↑'add' ↑a ↑r)
```

(f) Řídicí instrukce

```
Control-> Cc↑br Label↑n EMIT(↑br ↑n)
 -> goto Label↑n EMIT(↑'br jmp' ↑n)
Cc↑br
 -> Orelop↑br Word↑a EMIT(↑'tst' ↑a)
 -> Relop↑br Word↑a Word↑b EMIT (↑'cmp' ↑a ↑b)
 -> And↑br Word↑a Word↑b EMIT(↑'bit' ↑a ↑b)
 -> Or↑br Word↑a Word↑b GETTEMP(↑'word' ↑r) EMIT (↑'mov' ↑b ↑r) EMIT (↑'bis' ↑a ↑r)
Orelop↑'beq bne' -> 0=
Orelop↑'bne beq' -> 0<>
Relop↑'beq bne'  -> =
Relop↑'bne beq'  -> <>
And↑'bne beq'    -> &
```

(g) Instrukce volání podprogramu

```
Pcall -> CALL Name↑a EMIT(↑'jsr' ↑'FrameReg' ↑a)
```

Obr. 10.12: Část atributové gramatiky pro PDP

Implementace

Konstrukce tabulek řízení algoritmu generování i vlastní algoritmus generování cílového programu vychází, jako v předchozí metodě, ze syntaktické analýzy LR jazyků. Tyto algoritmy však bylo třeba modifikovat vzhledem k existenci akčních symbolů a rozhodovacích predikátů. Implementace akčních symbolů překladové gramatiky jako volání procedur zapsaných ve vyšším programovacím jazyce je velmi přirozená. Podobně lze implementovat také rozhodovací predikáty. Rozhodovací predikáty umožňují dosáhnout deterministickou syntaktickou analýzou programu ve vnitřním jazyce, ač je příslušná gramatika nejednoznačná. V určité konfiguraci algoritmu generování je možné provést obecně řadu akcí (redukcí). Je vybrána ta akce (v pořadí daném uspořádáním pravidel gramatiky), jejíž všechny rozhodovací predikáty jsou pravdivé.

Pro implementaci Ganapathiho atributové gramatiky popisující generátor cílového programu je rovněž vhodná metoda založená na syntaktické analýze rekurzivním sestupem. Jak jsme již podotkli, prefixový charakter vnitřního jazyka zaručuje, v případě pevné struktury operandů, základní předpoklad aplikace rekurzivního sestupu — popis jazyka LL(1) gramatikou. Výběr alternativy pro expanzi neterminálu je podmíněn vstupním symbolem, ale také hodnotou rozhodovacího predikátu. Na obr. 10.13 je ukázka implementace generátoru pro část aritmetických instrukcí v rámci rekurzivní procedury Word pro PDP.

10.5 Strojově závislé optimalizace

Strojově závislé optimalizace jsou důležitou součástí procesu generování cílového programu, poněvadž mohou odstranit řadu neefektivností cílového programu. Častým řešení těchto optimalizací je samostatný průchod (průchody) generovaným cílovým programem, jehož cílem je nahradit určité krátké posloupnosti instrukcí takovými instrukcemi, které zvyšují efektivnost celého programu (peephole optimization). Ganapathiho metoda specifikace generátoru je pozoruhodná tím, že umožňuje začlenit typické strojově závislé optimalizace již do procesu generování a to velmi pružným způsobem. S využitím rozhodovacích predikátů lze postihnout takové optimalizace jako využití speciálních instrukcí (*the machine idiom problem*), odstranění redundantních instrukcí přesunu údajů, optimalizací skokových instrukcí (odstranění skokových řetězců) slučování instrukcí při kombinaci adresovacích mechanismů (módů) a zpoždění výstupu instrukce v rámci základního bloku. Rozhodovací predikáty umožňují testováním atributů rozpoznávat speciální případy a začlenit do rozhodování kontext (začleněním atributů, jež nesou kontextovou informaci). Přidání nové optimalizace lze relativně snadno provést zavedením nového rozhodovacího predikátu a nového akčního symbolu.

```
procedure Word(var r:atribut);
begin
   .   .   .
   if vstupní_symbol ='+' then begin
      cti_dalsi_symbol;
      Word(a);
      Word(b);
      if IsCons(a,b) then
         KFOLD('+',a,b,r)
      else if IsOne(a) and IsTemp(b) then begin
         AUTOINC('inc',b);
         r:=b
         end
      else if IsOne(b) and IsTemp(a) then begin
         AUTOINC('inc',a);
         r:=a
         end
      else if IsTemp(b) then begin
         EMIT('add',a,b);
         r:=b
         end
      else if IsTemp(a) then begin
         EMIT('add',b,a);
         r:=a
         end
      else begin
         GETTEMP('word',r);
         EMIT('mov',b,r);
         EMIT('add',a,r)
         end
      end;
      . . .
  end;
```

Obr. 10.13: Implementace metodou rekurzivního sestupu

Kapitola 11

Překladače pro počítače s architekturou RISC

Dříve, nežli se budeme zabývat metodami výstavby překladačů pro počítače se zřetězeným zpracováním instrukcí, jejichž speciálním případem jsou počítače architektury RISC, pokusme se shrnout jaké prostředky a možnosti nabízí počítače tohoto typu programátorům. Vzhledem k široké škále uvažovavých počítačů není určení společných rysů snadné. Různé počítače této třídy mohou z naší specifikace více či méně vybočovat. Za společné lze z hlediska programátorského považovat zejména vlastnosti:

- *Zřetězené zpracování*
  Instrukce jsou prováděny v několika fázích (typicky pěti — viz obr. 11.1). Fáze několika instrukcí mohou probíhat současně. Neplatí tudíž dosud základní programátorské pravidlo určující, že účinek (výsledek) každé instrukce je dostupný před tím, nežli je započato s prováděním další instrukce. Tím, že se fáze jednotlivých instrukcí překrývají, vznikají *datové konflikty* a *konflikty instrukcí zpožděného skoku.*

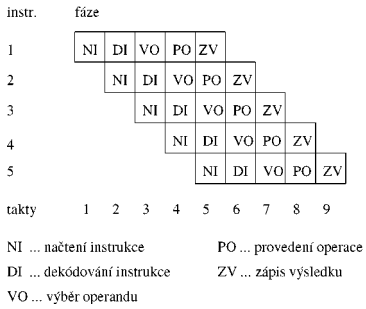
- *Omezení počtu způsobů adresace*
  Užití instrukcí pracujících výhradně s registry (typ registr – registr) spolu s dostatečným počtem registrů dovoluje rozsáhlejší přechovávání proměnných v registrech. Přístup do paměti je prováděn *výhradně* instrukcemi načtení hodnoty do registru (LOAD) a uložení hodnoty do paměti (STORE).

- *Jednoduchý formát instrukcí*
  Instrukce mají shodnou délku (typicky jedno slovo). Jejich vzájemné výměny (např. v reorganizační fázi překladu) jsou snadnější.

- *Významově jednoduché instrukce*
  Obvykle je k dispozici ne příliš rozsáhlá množina instrukcí. Sémantika instrukcí není bohatá, instrukce jsou na úrovni mikroinstrukcí počítačů architektury CISC.

11.1 Jednoduchý model počítače architektury RISC

Pro demonstraci shora specifikovaného programátorského rozhraní počítačů architektury RISC uveďme nyní detailně jedno takové rozhraní. Vzhledem k tomu, že reálné počítače mají většinou



Obr. 11.1: Fáze provedení instrukce

mimo typických vlastností architektury RISC i různé speciální vlastnosti, vytvoříme si abstraktní model počítače architektury RISC (dále aRISC). Tohoto modelu budeme užívat i v příkladech.

aRISC používá zřetězené zpracování instrukcí. Každá instrukce má 6 fází, přičemž v jednom taktu hodin jsou provedeny vždy dvě fáze instrukce. V každém taktu hodin je započato provádění další instrukce. aRISC má 8 instrukcí (viz tab. 11.1) a 8 registrů pro uchování hodnot celočíselného typu (označujeme je R0 až R7).

Instrukce lze rozdělit do čtyř skupin. První skupinou jsou instrukce ADD, SUB, a MOV, které jsou typu registr-registr. Jejich rozložení na fáze je vidět v tab. 11.1. Hodnota je v registru dostupná po provedení 6. fáze. Druhou skupinu tvoří instrukce přístupu do paměti. Pro výběr hodnoty z paměti slouží instrukce LD. Využívá bázované adresace displ(registr), musí proto provádět výpočet skutečné adresy. Fáze 4 a 5 slouží k přístupu do paměti. Hodnota je v registru dostupná po provedení fáze 6. Instrukce ST slouží k uložení hodnoty z registru do paměti. Adresování je totožné s instrukcí LD. Fáze 5 a 6 slouží k přístupu do paměti. Výsledek je v paměti uložen až po provedení fáze 6. Třetí skupinu tvoří instrukce skoku JMP a BRA. Ve fázích 4, 5 a 6 (resp. 5 a 6) se neprovádí žádná činnost. Adresa je vypočtena ve druhém taktu. Proto je zpoždění skoků rovno 1. Poslední skupinu tvoří prázdná instrukce NOP. Ve fázích 3, 4, 5 a 6 se neprovádí žádná činnost. Pro spuštění programu je nutné, aby neobsahoval žádné datové ani skokové konflikty.

Mějme nyní funkci v jazyce C, která provádí mimo jiné kopii položek globálního pole $\underline{a}$ s indexy $0, 2, \ldots, 98$ do položek lokálního pole $\underline{b}$ s indexy $0, 1, \ldots, 49$. Každá kopírovaná položka je zvětšena o 1.

KÓD	ZÁPIS INSTRUKCE PRO ASEMBLER	VÝZNAM
NOP	`NOP`	prázdná operace
ADD	`ADD Z1,Z2,V`	sčítání (V:=Z1+Z2)
SUB	`SUB Z1,Z2,V`	odečítání (V:=Z1-Z2)
MOV	`MOV Z,V`	přesun (V:=Z)
LD	`LD displ(R),V`	natažení z paměti (V:=paměť displ(R))
ST	`ST Z,displ(R)`	uložení do paměti (paměť displ(R):=Z)
JMP	`JMP adresa`	skok na adresu
BRA	`BRA podm,Z1,Z2,adresa`	**if** Z1 podm Z2 **then** skok na adresu

Z zdrojový operand:
Rn označuje registr s číslem *n*
číslo označuje přímý celočíselný operand
V výsledek (vždy *Rn* — registr pro uložení výsledku)
displ relativní vzdálenost přičítaná k obsahu registru *R* pro získání adresy
adresa absolutní adresa — návěští pro skoky
podm relační operátor < > = <= >= <>

1	2	3	4	5	6
1.takt		2.takt		3.takt	

ADD, **SUB**, **MOV**

NI	DI	VO	PO	xx	ZV

LD

NI	DI	VA	xx	xx	ZV

ST

NI	DI	VA	ZV	xx	xx

JMP

NI	DI	VA	xx	xx	xx

BRA

NI	DI	VA	VP	xx	xx

NOP

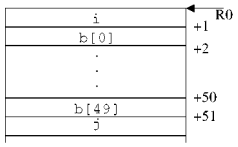
NI	DI	xx	xx	xx	xx

NI načtení instrukce
DI dekódování instrukce
VO výběr operandu
PO provedení operace
ZV zápis výsledku
VA výpočet adresy

Tab. 11.1: Instrukční repertoár aRISCu

```
int a[100];
kopie ()
   int i,b[50],j;
   :
   j=5;
   for (i=0; i<50; i++)
      b[i]=a[i*2]+1;
   i=0;
```

Tohoto příkladu budeme dále využívat pro demonstraci některých technik překladu. Cílový kód překladače je organizován tak, že lokální proměnné (zde `i`, `j` a `b`) jsou bázovány na zásobníku registrem $R0$ a každá celočíselná hodnota zaujímá jedno slovo (aRISC adresuje po slovech). Zásobník bude mít při aktivaci funkce kopie tvar z obr. 11.2.



Obr. 11.2: Zásobník

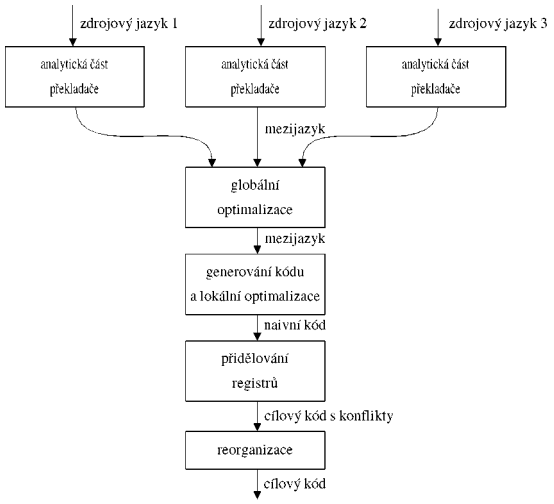
Nechť je globální proměnná `a` uložena na adrese 100. Produkt překladače jazyka C pro náš zdrojový text by pak mohl mít následující tvar

```
č.   INSTRUKCE      KOMENTÁŘ   KONFLIKTY SKUTEČNĚ GENEROVANÉ
 1     MOV #5,R1      j=5          x  IF ID EX MA WB
 2     ST  R1,(R0+1)                  IF ID EX MA WB
 3     MOV #1,R1      i=0             IF ID EX MA WB
 4     CMP R1
 5 LBL: ADD R1,R1,R2   i*2          x  IF ID EX MA WB
 6     LD  100(R2),R3 a[i*2]          x     IF ID EX MA WB
 7     ADD R3,#1,R4   a[i*2]+1           IF ID EX MA WB
 8     ADD R1,R0,R5                      IF ID EX MA WB
 9     ST  R4,1 R5    b[i]=a[i*2]+1                 IF ID EX MA WB
10     ADD R1,#1,R1   i++             x                 IF ID EX MA WB
11 LBL: BGE R1,#50,LBL i<50  ?                               IF ID EX MA WB

12     MOV #1,R1      i=0          x  IF ID EX MA WB
13     ST  R1,(R0)                    IF ID EX MA WB
```

Nejde dosud o program, který lze spustit. Jednak není provedeno přiřazení skutečných registrů (překladač generoval do neomezené množiny registrů r1, r2, r3, ...) a dále nejsou řešeny datové ani skokové konflikty (datové konflikty jsou označeny x a znázorněny šipkami). Jde vlastně o mezijazyk překladače před úpravou pro zřetězené zpracování.

11.2 Překladač

U počítačů architektury RISC hrají překladače vyšších programovacích jazyků velmi důležitou úlohu při zajišťování výkonu počítačů. U tradičních počítačů docházelo obvykle k vytváření překladačů odděleně od návrhu architektury, tj. až po dokončení technického vybavení. Tyto hranice u architektury RISC nejsou. Vzhledem k novým požadavkům technického vybavení na tvar cílového programu (náhrada komplexních operací jazyka elementárními instrukcemi na úrovni mikroprogramů, řešení konfliktů a pod.) dochází k návrhu architektury RISC i překladače současně. Architektura a překladač jsou na sebe úzce vázány. Vazba překladače k počítači a jeho odpovědnost za mnohé vlastnosti cílového kódu dříve ošetřované technickým vybavením si vyžaduje aplikaci některých nových technik překladu.



Obr. 11.3: Schema typického překladače

Počítače architektury RISC poskytují uživateli jednoduché technické prostředky, což dovoluje provádět optimalizaci cílového kódu na mnohem nižší úrovni než dříve. Generování kódu je vzhledem k jednoduchým technickým prostředkům jednodušší, neboť existuje málo alternativ.

Do popředí se dostávají různé **metody optimalizace**.

Vzhledem k tomu, že u počítačů architektury RISC neexistují mikroinstrukce a instrukce jsou na velmi nízké sémantické úrovni, musí překladače v čase překladu transformovat příkazy vyššího programovacího jazyka na úroveň mikroinstrukcí. Tvar uživatelského rozhraní určuje, že překladače preferují v cílovém kódu uchování operandů v registrech, užívání instrukcí typu registr-registr a minimální přístup do operační paměti.

Schéma typického překladače pro počítače architektury RISC je na obr. 11.3. Používáme-li k programování tradičních jazyků (C, Pascalu, Fortranu), budou i jazykově závislé (tj. analytické) části překladače tradiční. Rozdíl se projeví v částech strojově závislých, tj. ve fázi optimalizační a generační.

Klasická část překladače (lexikální, syntaktická a sémantická analýza) generuje většinou na výstupu mezijazyk. Bývá to *zásobníkově orientovaný virtuální strojový kód*. Sémantická mocnost mezijazyka je obvykle dostatečná pro všechny překládané jazyky, což umožňuje vytvářet jedinou společnou optimalizační a generační část pro všechny překladače.

První fází překladače zpracovávající mezijazyk je globální optimalizace. Kromě tradičních optimalizačních technik jako jsou např. optimalizace cyklů a odstraňování společných podvýrazů provádí pro architekturu RISC typický rozvoj krátkých procedur do tvaru otevřeného podprogramu. Další fází strojově závislé části překladače je generátor kódu, který převádí mezijazyk na posloupnost strojových instrukcí. Výstupem generátoru kódu se říká *naivní kód*, neboť obvykle předpokládá *neomezenou* množinu obecně využitelných registrů a je vytvořen bez ohledu na konflikty zřetězeného zpracování.

Mezijazyk obsahuje pseudo-strojové instrukce pro vyjádření i složitějších operací, které se ve skutečném instrukčním repertoáru nevyskytují (jako je násobení, dělení, operace s pohyblivou řádovou čárkou a pod.). Úkolem generátoru kódu je *rozvoj* těchto *makroinstrukcí* do skutečných strojových instrukcí počítače. V této fázi se provádí i lokální optimalizace jako např. odstranění dvojic LOAD a STORE, eliminace nedostupného kódu a pod.

Pro *přidělování registrů* je nejčastěji využívána technika *barvení grafu*, která je výhodná pro větší počty registrů a princip uchovávání proměnných v registrech.

Koncová část překladače se liší podle toho, zda je počítač schopen řešit datové konflikty zřetězeného zpracování technickými prostředky (interlock hardware), či ne. Pokud je toho schopen, bývá výstupem fáze přidělování registrů přímo strojový kód. V tom případě řeší fáze přidělování registrů navíc i skokové konflikty.

Vyžaduje-li počítač cílový kód bez konfliktů, bývá poslední fází překladu tzv. *reorganizátor kódu*. Ten je schopen přeskupit instrukce tak, aby datové konflikty byly odstraněny a řeší i přesuny nutné pro optimální využití zpožděných skoků.

Spojení reorganizace kódu a přidělování registrů je výhodné tím, že přidělování registrů a přemisťování instrukcí se mohou navzájem ovlivňovat, což vede často ke kvalitnějšímu kódu. Značně však narůstá složitost této části překladače. Oddělení reorganizace kódu do zvláštního průchodu vede ke strukturalizaci a zjednodušení překladače. Navíc lze reorganizačního průchodu případně užít pro zpracování ručně psaných programů v jazyce asembleru. Ručně psané programy tvoří vlastně cílový kód s konflikty.

Rozeberme nyní podrobněji některé z technik překladu, které se objevily v souvislosti s překladem programů pro počítače architektury RISC. Některé z nich jsou obecně použitelné (přidělování registrů), jiné jsou typické pro počítače využívající zřetězené zpracování.

11.3 Přidělování registrů metodou barvení grafu

Metoda přidělování registrů barvením grafu byla publikována firmou IBM [1] a použita poprvé pro překladač jazyka PL.8 u počítače IBM 801. Jejím hlavním účelem je uchování co největšího počtu proměnných v registrech po co největší dobu, přičemž se předpokládá větší počet registrů. Řešení spočívá v obvyklém rozdělení programu na základní bloky. *Základním blokem* je nejdelší posloupnost instrukcí s jedním vstupním bodem. Instrukce skoku základní blok ukončuje. Pokud je k dispozici naivní kód pro neomezenou množinu registrů, který je rozdělen na základní bloky, je možné v každém bloku vybudovat graf interference.

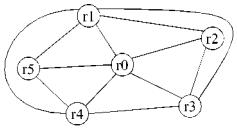
Graf interference obsahuje uzel pro každý registr naivního kódu, který je živý (tj. obsahuje aktuální informaci) v základním bloku. Současná doba života dvou registrů je vyjádřena hranou (sousedností) mezi uzly reprezentujícími registry. Předpokládejme, že máme k dispozici tolik barev, kolik existuje skutečných registrů počítače. Pokud se podaří graf interference obarvit daným množstvím barev, je tím dáno zobrazení registrů naivního kódu na skutečné registry.

Pokud není možné graf interference obarvit, je nutné provést uložení některého z registrů do paměti. Tím je možno snižovat počet živých registrů tak dlouho, dokud nelze graf interference daným počtem barev obarvit.

Mějme například základní blok instrukcí 5–11 z našeho příkladu. Využití (doba života) registrů je viditelná z tabulky.

	REGISTRY					
INSTR	r0	r1	r2	r3	r4	r5
5	x	x	x			
6	x	x	x	x		
7	x	x		x	x	
8	x	x			x	x
9	x	x			x	x
10	x	x				
11	x	x				

Tabulce využití registrů odpovídá graf interference na obr. 11.4)



Obr. 11.4: Graf interference

Pro barvení grafu se užívá nového efektivního algoritmu přizpůsobeného řešené problematice. Vychází se z předpokladu, že většina uzlů v grafu interference G_i má stupeň menší než N, kde N je počet barev (skutečných registrů), které jsou k dispozici. Vytváříme postupně grafy G_{i+1} tak, že G_{i+1} vznikne z G_i vynecháním všech uzlů stupně menšího než N, včetně všech hran z nich vycházejících.

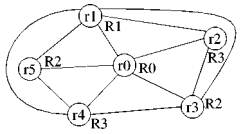
Algoritmus postupně redukuje grafy G_i na G_{i+1}, což se opakuje tak dlouho, dokud výsledný graf G_{i+1} není prázdný. Druhou možností ukončení postupné redukce je graf G_{i+1}, jehož všechny uzly jsou stupně alespoň N. Odstraňované uzly jsou ukládány do zásobníku.

Prázdný graf G_{i+1} se považuje za úspěšný konec redukční fáze algoritmu. V takovém případě se vybírají uzly ze zásobníku (v opačném pořadí, nežli byly odstraňovány) a jsou jim postupně přidělovány barvy (registry). Barvíme-li uzly tímto způsobem, je vždy k dispozici barva pro obarvení uzlu a algoritmus skončí očekávaným přidělením registrů.

Pokud redukce skončí druhou možností (všechny uzly stupně alespoň N), je nutné vložit do vstupního kódu instrukce pro uložení hodnoty uchovávané v registru do paměti, vytvořit znovu graf interference a algoritmus opakovat. Máme-li k dispozici 4 registry-barvy, může posloupnost odstraňovaných uzlů pro náš graf interference být

$$r2\ r5\ r4\ r3\ r1\ r0$$

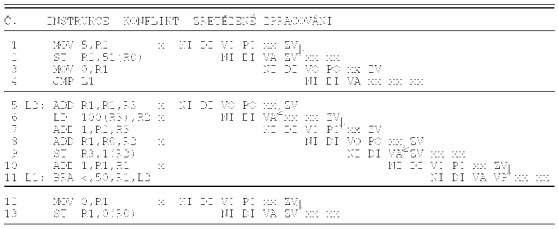
Barvy (R0 až R3) přidělujeme v opačném pořadí posloupnosti při odstraňování. Výsledný obarvený graf je na obr. 11.5



Obr. 11.5: Obarvený graf

V případě, že bychom přidělovali 8 skutečných registrů (R0 až R7) aRISCu, je obarvení triviální (ri = Ri).

Pokud není redukční fáze ukončena dosažením prázdného grafu G_{i+1}, je nutné zmenšit náročnost kódu na živé registry tím, že využijeme paměť spolu s instrukcemi LOAD a STORE. Dodatečné instrukce ovšem zpomalují výpočet a prodlužují program, proto je nutné provést úpravu efektivně. Uložení hodnoty v paměti znamená obvykle její opětné získání před každým užitím a uložení nové hodnoty do paměti v každém dalším definičním bodě hodnoty. Proto je důležitá volba registru, který bude uvolněn, neboť pro jednotlivé uvolněné registry je ovlivnění výsledného kódu různé. Provádí se proto odhad *ceny* za uvolnění daného registru a to tak,že dodaná instrukce LOAD nebo STORE má cenu 1. Cena instrukce uvnitř cyklu roste například desetinásobně. Volí se pak ten uzel (registr), jehož cena dělená jeho stupněm je nejmenší. Je-li volba provedena, je pak modifikován vstupní naivní kód instrukcemi LOAD a STORE a modifikován graf interference. Pokud redukční krok opět není řádně ukončen, je nutné odstranit další uzel atd. V [1] se uvádí, že jak algoritmus vlastní redukce, tak i odstraňování uzlů pro 32 registrů jsou pro většinu případů velmi efektivní. Nakonec uveďme mezijazyk překladu pro náš příklad po přidělení 4 skutečných registrů (jen pro ukázku, neboť aRISC má registrů 8). Přidělením skutečného počtu osmi registrů aRISCu se původní program až na záměnu R za r nezmění.



11.4 Příprava kódu pro zřetězené zpracování

Naivní kód nemusí být po fázi generování kódu zbaven konfliktů zřetězeného zpracování a nemá obvykle ošetřeny zpožděné skoky. Nelze jej proto v tomto tvaru spustit. Existuje triviální úprava, která umožní program vždy spustit. Mezi vzájemně konfliktní instrukce lze vložit prázdné (NOP) instrukce. Rovněž za každý zpožděný skok lze vložit posloupnost prázdných operací v délce zpoždění skoku. Touto úpravou konflikty zmizí. Uvedené triviální řešení však značně prodlužuje program a zpomaluje výpočet.

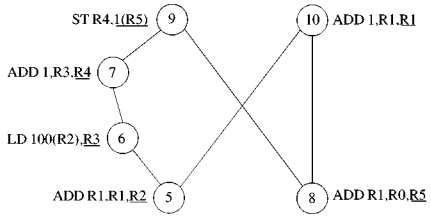
Existují proto metody úpravy, které řeší konflikty a zpožděné skoky přesunem instrukcí v programu při zachování významu programu. Vychází se z předpokladu, že instrukce, které nejsou cílem skoku, ani samy nejsou skokem je možné v programu přemisťovat. Sémantika programu však musí být zachována. Zachováním sémantiky rozumíme zachování stejného obsahu zdrojových registrů i zdrojové paměti každé instrukce před i po přemístění. Vhodným úsekem programu pro přemísťování je základní blok. Neobsahuje totiž ve svém těle skoky ani cíle skoků, což jsou instrukce, jejichž umístění v programu je pevné.

11.5 Odstranění datových konfliktů

Nutná návaznost výsledků a zdrojových operandů jednotlivých instrukcí základního bloku definuje nad instrukcemi částečné uspořádání. Větší instrukce v tomto uspořádání je ta, která užívá jako zdrojový operand *výsledek* jiných instrukcí bloku. Ty jsou naopak v uspořádání menší. Hasseův diagram částečného uspořádání pro základní blok 5–11 našeho příkladu je na obr. 11.6 (uzly jsou označeny čísly instrukcí).

Částečně uspořádanou množinu instrukcí lze vždy lineárně uspořádat tak, aby lineární uspořádání bylo konzistentní s původním částečným uspořádáním. Těchto lineárních uspořádání instrukcí může být i více, přičemž některé z nich mohou způsobovat datové konflikty (označujeme je x) a jiné nemusí. Například pro naše částečné uspořádání lze vytvořit následující lineární uspořádání

5 x 6 x 7 8 x 9 10	původní posloupnost (3 konflikty)
5 8 10 6 x 7 x 9	2 konflikty
5 8 6 10 7 x 9	1 konflikt



Obr. 11.6: Hasseův diagram

Úkolem algoritmu odstranění datových konfliktů je nalezení takového lineárního uspořádání instrukcí základního bloku, které by bylo konzistentní s původním částečným uspořádáním a zároveň by obsahovalo co nejmenší množství datových konfliktů.

Lineární uspořádání bez konfliktů tedy nemusí existovat (což je i případ našeho příkladu). Potom je nutné hledat posloupnost s minimálním množstvím konfliktů a tyto pak ošetřit vložením prázdných operací. V [5] se uvádí, že obecný algoritmus pro hledání minimálního počtu prázdných operací je NP-úplný. Proto byl vyvinut [5] heuristický reorganizační algoritmus pracující v polynomiálním čase. Algoritmus vychází z Hasseova diagramu částečného uspořádání, přičemž postupně pokrývá uzly směrem od nejmenších. Je-li uzel *pokryt*, je pro něj vygenerována instrukce. Uzel je *připraven*, jsou-li všichni jeho následníci pokryti. Podstatou algoritmu je hledání nejvhodnějšího připraveného uzlu v situaci, kdy je diagram částečně pokryt.

Každý uzel grafu je charakteristický svým výsledkem (na obr. 11.6 jsou výsledky podtrženy). Pokud je jeho výsledek zpracováván nějakou větší instrukcí, musí být zajištěno, že mezi uložením výsledku a jeho využitím nebude vložena žádná instrukce se stejnou adresou výsledku (ta by očekávaný výsledek přepsala). *Bezpečný průchod* je pro daný připravený uzel U množina všech uzlů (včetně uzlu U), které musí být pokryty spolu s uzlem U tak, aby již žádný další uzel v diagramu nepotřeboval tentýž výsledek instrukce uzlu U.

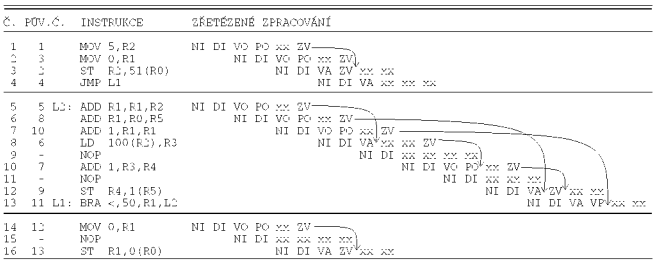
V našem příkladu pro zcela nepokrytý Hasseův diagram jsou připravenými uzly 5 a 8 s výsledky R2 a R5. Pro uzel 5 je bezpečným průchodem množina {5,6} a pro uzel 8 množina $\{8, 9, 7, 6, 5\}$. Výsledek R5 uzlu je spotřebován uzlem 9. K tomu, aby mohl být uzel 9 pokryt, musí být pokryty také uzly 7, 6 a 5.

Algoritmus začíná u zcela nepokrytého diagramu. Vždy hledá bezpečné průchody pro všechny připravené uzly, u nichž při generaci nevznikne datový konflikt. Z těchto uzlů vybere ten s nejmenší mohutností bezpečného průchodu. Je-li více minimálních bezpečných průchodů, provede se náhodný výběr. Neexistuje-li bezkonfliktní uzel, je nutné vložit prázdnou instrukci (tím se konflikt odstraní alespoň pro jeden případ) a opakovat výběr připravených uzlů.

Připravený uzel s minimální mohutností bezpečného průchodu je pak pokryt a krok algoritmu se opakuje až do pokrytí celého grafu. Existují-li v daném kroku částečného pokrytí dva bezpečné průchody pro stejný výsledek a jeden z nich je vybrán, musí být pokrývání druhého *blokováno*. Blokování trvá tak dlouho, dokud není vybraný bezpečný průchod zcela pokryt. Demonstrujme algoritmus na našem příkladu pro základní blok instrukcí 5–11.

KROK	LINEÁRNÍ USPOŘÁDÁNÍ	PŘIPRAVENÉ UZLY		BEZPEČNÉ PRŮCHODY	
1.	prázdné	5	R2	$\{5,6\}$	← výběr
		8	R5	$\{8,9,7,6,5\}$	
2.	5	6	R3	$\{6,7\}$	← konflikt
		8	R5	$\{8,9,7,6\}$	← výběr
3.	5,8	6	R3	$\{6,7\}$	
		10	R1	$\{10\}$	← výběr
4.	5,8,10	6	R3	$\{6,7\}$	← výběr
5.	5,8,10,6	7	R4	$\{7,9\}$	← konflikt, vložit NOP
6.	5,8,10,6x7	9	1(R5)	$\{9\}$	← konflikt, vložit NOP
7.	5,8,10,6x7x9				

V našem případě heuristický algoritmus nenalezl optimální řešení. Lepší je například 5, 8, 6, 10, 7x9 s jediným konfliktem. Program pro aRISC s odstraněnými datovými konflikty má potom tvar



11.6 Zpožděné skoky

Nechť i je číslo skokové instrukce s cílem L. Skok nazveme *zpožděným skokem s délkou* n, je-li posloupnost provádění instrukcí

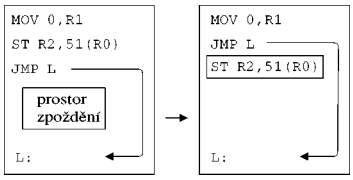
$$i, i+1, i+2, \ldots, i+n, L$$

Instrukcím $i+1, i+2, \ldots, i+n$ říkáme *prostor zpoždění*.

Zpožděné skoky byly úspěšně použity u mnoha počítačů architektury RISC (IBM 801, RISC II, MIPS, HP-Spectrum, GaAs). Triviální ošetření kódu pro zpožděné skoky vložením prázdných instrukcí zcela eliminuje jejich výhody. Proto je žádoucí vkládat do prostoru zpoždění významové instrukce tak, aby nedošlo ke změně významu programu. Existují tři možné transformace kódu pro řešení skokových konfliktů:

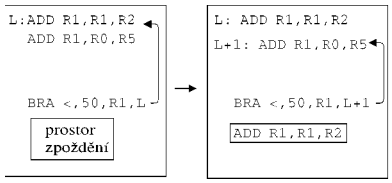
- Přesun n instrukcí nacházejících se před instrukcí zpožděného skoku za ni.

- Kopie prvních n instrukcí z cíle skoku za instrukci skoku. Cíl skoku se posune o n instrukcí dál, nežli byl původní cíl.
- Přesun n instrukcí, nacházejících se za prostorem zpoždění, bezprostředně za instrukci skoku.



Obr. 11.7: Transformace 1

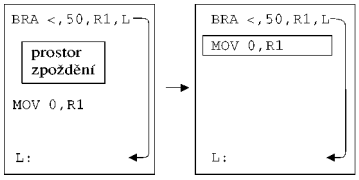
První způsob (obr. 11.7) je přesun instrukcí nacházejících se před zpožděným skokem za něj. Provedení skoku zde nesmí být závislé na výsledku přesunovaných instrukcí. To bývá splněno nejčastěji u nepodmíněného skoku. Výhodou této transformace je to, že není nutné uvažovat instrukce v cíli skoku. Úspora času se projeví při provedení i při neprovedení skoku. Transformace šetří čas výpočtu a do programu nejsou vloženy žádná dodatečné instrukce a jeho délka se nemění.



Obr. 11.8: Transformace 2

Druhým způsobem (obr. 11.8) je kopie prvních n instrukcí z cíle skoku do prostoru zpoždění. Cíl skoku je posunut za blok instrukcí, které byly zdvojeny. Tato transformace šetří čas výpočtu. Délka programu se zvětší o n-instrukcí. Navíc úspora času se projeví pouze tehdy, *je-li skok proveden*. V opačném případě se instrukce v prostoru zpoždění provádějí zbytečně. Proto musí být zajištěno, že tyto instrukce nebudou modifikovat registry nebo paměť, která je aktivní, není-li skok proveden. Vlastnosti transformace 2 ji předurčují pro skoky, které jsou *většinou provedeny*. Takové skokové instrukce lze nejčastěji nalézt u překladu cyklů.

Třetí transformace přesouvá instrukce zpoza prostoru zpoždění bezprostředně za skokovou instrukci (obr. 11.9). Tato transformace šetří čas výpočtu a délka programu se nemění. Úspora



Obr. 11.9: Transformace 3

se však projeví pouze tehdy, *není-li skok proveden*. V opačném případě se blok n-instrukcí za zpožděným skokem provádí zbytečně. Je proto nutné, aby tyto instrukce nemodifikovaly registry a paměť využívané, je-li skok proveden. První transformace je vždy výhodná, proto ji volíme, je-li to možné. Druhá a třetí transformace jsou výhodné pouze při provedení resp. neprovedení skoku. To vyžaduje od překladače odhad, kterou z transformací 2 resp. 3 volit podle převládajícího uplatnění skoku. U návratových skoků v cyklech převládá počet provedení skoku, proto se zde obvykle volí transformace 2 (nelze-li užít 1). Jednoduchý algoritmus pro volbu transformace má pak tento tvar :

Nechť n je zpoždění zpracovávaného skoku.
Pokus se přesunout n instrukcí transformací 1.
if počet přesunutých instrukcí $k < n$ **then**
  if skok je směrem nazpět **then**
    pokus se provést transformaci 2 pro $n - k$ instrukcí
  else
    proveď transformaci 3 pro $n - k$ instrukcí

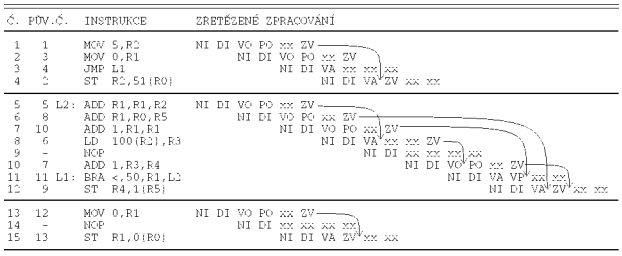
Pro MIPS [5] s délkou zpoždění 1 resp. 2 se uvádí, že lze touto metodou zaplnit až 85% prostorů zpoždění. Zbytek je nutno vyplnit prázdnými instrukcemi. Řešení zpožděných skoků uvedenou metodou šetří okolo 20% času výpočtu oproti situaci, kdy jsou prostory zpoždění vyplněny prázdnými instrukcemi. Přes nesporné výhody této jednoduché varianty zpožděných skoků dochází k jejím dalším modifikacím, které výkonnost dále zvyšují.

Zpožděné skoky s potlačením účinku instrukcí v prostoru zpoždění (squashing, nullification) snižují obtížnost při zaplňování prostoru zpoždění bezpečnými instrukcemi. Technické vybavení počítače totiž umožňuje potlačit účinek provedených instrukcí, které se nacházejí v prostoru zpoždění tak, že tyto instrukce nezmění stav výpočtu. Tím klesají nároky na vlastnosti instrukcí vkládaných do prostoru zpoždění a prostor lze snadněji zaplnit významovými instrukcemi.

Každá instrukce zpožděného skoku může zde být navíc doplněna až dvěma bity nastavovanými v době překladu. *Bit směru skoku* indikuje, zda překladač pro tento skok předpokládal, že skok bude častěji proveden (tj. potlačení účinku instrukcí v prostoru zpoždění se nebude provádět při provedení skoku) nebo opačnou možnost. *Bit řízení* indikuje, zda instrukce v prostoru zpoždění vůbec mění stav výpočtu při nevýhodné variantě a zda je vůbec nutné jejich účinek potlačovat. Nastavení bitu směru skoku je bez dodatečné analýzy programu obtížné a řeší se tak, že je předpokládáno provedení skoku vždy. Toto implicitní řešení však nemusí být pravdivé.

Program pro aRISC, který má jednoduchou variantu řešení zpožděných skoků, je po trans-

formacích řešících zpožděné skoky následující. Je to jeho konečný a spustitelný tvar.



Další zvyšování výkonu se provádí *vyhodnocováním statistiky* provedení (resp. neprovedení) skoku za *běhu programu*. Získané hodnoty mohou pak zpětně ovlivnit provedenou transformaci instrukcí nebo nastavení bitu směru skoku. Problémem je však vhodná volba testovacích vstupních údajů, které by měly dávat průkazné výsledky pro širokou škálu předpokládaných skutečných vstupních údajů.

V [6] je uvedena řada měření zpožděných skoků. Měření byla prováděna na počítači MIPS-X. První měření se provádělo pro určení průměrného počtu hodinových taktů na jeden skok. Druhé určuje rychlost počítače relativně vzhledem k hypotetickému počítači s jedním taktem na jeden skok. Následující tabulka ukazuje výsledky měření.

	POČET TAKTŮ NA SKOK	RELATIVNÍ RYCHLOST
jednoduchý zpožděný skok	2,21	1,130
+ potlačení účinku	1,77	1,083
+ údaje z běhu programu	1,43	1,045

Literatura

[1] M. Ackerman, G. Baum. The Fairchild Clipper. *BYTE*, 12(4):161–174, 1987.

[2] Adobe Systems, Inc.: *PostScript language reference manual.* Addison-Wesley, sixteenth edition, 1990.

[3] A. V. Aho, R. Sethi, J. D. Ullman: *Compilers. Principles, Techniques, and Tools.* Addison-Wesley, 1987.

[4] J. P. Bennett: *Introduction to compiling techniques: a first course using ANSI C, LEX and YACC.* McGraw-Hill, 1990.

[5] Z. Blaˇ zek, P. Kroha: Design of a reconfigurable parallel risc machine. In *Sborník Euromicro'87*, 1987.

[6] T. R. Gross: Code optimization techniques for pipelined architectures. In *Proceedings of Compcon*, strany 278–285, Spring 1983.

[7] A. I. Holub: *Compiler Design in C.* Prentice Hall, 1990.

[8] J. M. Honzí k: *Programovací techniky.* Učební texty vysokých škol. Ediční středisko VUT Brno, 1985.

[9] T. Hruˇ ska. *Modelování sémantiky programovacích jazyků a využití modelů při implementaci překladačů.* Disertační práce, VUT Brno, 1983.

[10] G. J. Chaitin. Register allocation and spilling via graph coloring. *SIGPLAN Notices*, 17:98–105, 1982.

[11] D. E. Knuth: *The METAFONTbook.* Addison-Wesley, 1986.

[12] D. E. Knuth: *The TEXbook.* Addison-Wesley, 1987.

[13] L. Lamport: *LATEX: A Document Preparation System.* Addison-Wesley, 1986.

[14] B. Melichar: Bezkontextové, překladové a atributové gramatiky. In *Sborník konference MOP'85, 1. díl*, strany 23–116, 1985.

[15] Jean-Paul Tremblay, P. G. Sorenson: *The Theory and Practice of Compiler Writing.* Computer Science Series. McGraw-Hill, 1985.

[16] M. Češka a kol.: *Gramatiky a jazyky.* Skriptum VUT Brno. Ediční středisko VUT Brno, 1985.

[17] M. Češka, T. Hruška: *Gramatiky a jazyky — cvičení.* Skriptum VUT Brno. Ediční středisko VUT Brno, 1986.

[18] W. M. Waite, G. Goos: *Compiler construction.* Text and monographs in computer science. Springer-Verlag, 1984.