PB111 Nízkoúrovňové programování

Petr Ročkai

Část A: Organizace

Vítejte v předmětu PB111, kde se budeme zabývat modelem výpočtu, který popisuje fungování běžných počítačů. Tato skripta podávají **teoretický** pohled na věc a slouží jako předloha pro přednášky.

A.1: Informace o kurzu

A.1.1 Prerekvizity Tento kurz předpokládá jen minimum znalostí – je ale velmi důležité, abyste měli zvládnutou látku předmětu IB111. Znalosti z předmětů PB150 nebo PB151 o základech fungování počítačů budou jistě výhodou. Překvapivě důležitým požadavkem tohoto předmětu je schopnost soustředit se na psaný text (to platí jak pro teoretickou, tak praktickou část předmětu).

- princip fungování počítače (PB150, PB151)
- základy programování (IB111)
- porozumění psanému textu

A.1.2 Studijní materiály Studijní materiály tohoto předmětu jsou rozděleny do dvou částí – tyto poznámky jsou ta více teoretická, která se váže k přednášce. Praktická část předmětu pak používá sbírku příkladů, která obsahuje krom samotných úloh také řadu prakticky zaměřených ukázek a další materiál spíše referenčního charakteru.
XXX Literatura

- tyto poznámky
- sbírka úloh
- literatura
- příklady na internetu

A.1.3 Ukončení Tento předmět je ukončen **zkouškou**. Je-li pro Vás předmět nepovinný, můžete si jej zapsat také na zápočet – v takovém případě budete hodnoceni pouze ze semestrální práce. Přesný popis hodnocení předmětu a zejména bodování naleznete v kapitole A sbírky.

- hodnocena pouze praktická část
- zejména práce během semestru
- programovací test ve zkouškovém
- podrobněji ve sbírce

A.1.4 Seminář Formát cvičení je stejný, jako v IB111 – v první části budete analyzovat zajímavá řešení příprav, v té druhé pak společně programovat příklady ze sbírky. Detailněji je průběh cvičení popsán v kapitole A sbírky.

- praktické programování
- předpokládá znalosti z této přednášky
- předpokládá znalosti z IB111

A.1.5 Přehled semestru Kurz je rozdělen na 3 velké tematické celky. První měsíc se budeme zabývat převážně tím, jak probíhá výpočet na úrovni procesoru – popíšeme si jaké prostředky a operace nám procesory nabízí a jak nám tento velmi jednoduchý model umožňuje spouštět libovolně složité programy. Naučíme se zejména jak se na úrovni stroje realizují standardní prvky jazyků vyšší úrovně.[1]
Druhý blok se bude zabývat organizací paměti. V kapitolách 5–8 probereme základní stavební prvky datových struktur – pole a ukazatele – a základy dynamické alokace paměti.
V posledním bloku se pak budeme blíže zabývat konkrétními datovými strukturami: dynamické pole, binární halda, hašovací tabulka a vyhledávací strom.[2] Zejména se zaměříme na jejich nízkoúrovňovou realizaci za pomoci znalostí a dovedností nabytých v prvních dvou blocích.

1. model výpočtu
2. organizace paměti
3. datové struktury a algoritmy

Část 1: Výpočetní stroj

V této kapitole si představíme základní výpočetní model, který pak budeme používat celý semestr. Realizovat jej bude výpočetní stroj tiny, který je sice velmi jednoduchý (minimální implementace v jazyce Python má přibližně 200 řádků kódu), ale umožní nám spouštět libovolné programy napsané v jazyce C.[3]

[1] Nebudeme se zde ovšem zabývat interakcí s vnějším světem ani souběžností – tato témata spadají do kurzu PB152 Operační systémy. Pro účely tohoto kurzu pracujeme s izolovaným sekvenčním počítačem který má pouze výpočetní jednotku (procesor) a paměť.

[2] Tou dobou je budete již dobře znát z předmětu IB002 Algoritmy a datové struktury.

[3] Přesto, že v tomto předmětu budeme používat pouze podmnožinu jazyka C, není to na úkor obecnosti: tato podmnožina je dostatečně silná na to, aby do ní bylo možné přepsat libovolný program napsaný v plném ISO C99, a to včetně standardní knihovny. Také by bylo možné přímo jazyk ISO C99 překládat do strojového kódu stroje tiny.

1.1: Model výpočtu

Abychom se mohli o výpočtech přesně vyjadřovat (a také přesně uvažovat), musíme si zavést vhodný **model** výpočtu – efektivně matematický aparát, který nám umožní výpočet uchopit jako objekt. Standardní metodou, jak výpočet popsat, je jako sled **stavů** (konfigurací) nějakého **abstraktního stroje** a přechodů – **akcí** – mezi nimi.

- zjednodušený model počítače
- bez virtualizace, souběžnosti
- 16bitové registry a adresy
- jednoduchá instrukční sada

1.1.1 Motivace Protože tento předmět je zaměřený spíše prakticky (a pragmaticky), chceme aby náš abstraktní výpočetní stroj co nejpřesněji odpovídal skutečným počítačům. Zároveň se ale nechceme takříkajíc „utopit v detailech“ – proto bude náš konkrétní výpočetní model určitým kompromisem.
Většinu modelu a terminologie si půjčíme ze světa skutečných počítačů, v několika ohledech si ale situaci zjednodušíme:

1. nebudeme se zabývat virtualizací a souběžností – náš „počítač“ bude najednou vykonávat jediný sekvenční program,
2. omezíme se na výpočty se 16bitovými hodnotami a na 16bitovou adresaci – to nám umožní mít lepší přehled o obsahu paměti, a zároveň nám velmi usnadní čtení adres a čísel obecně,
3. instrukční sada tiny je oproti těm reálným velmi malá (obsahuje mnohem méně operací), jednodušší (jednotlivé operace toho dělají méně najednou) a pravidelnější.

- hodnoty uložené v
  - registrech
  - paměti
- konečný počet možností

1.1.2 Stav Jak jsme zmínili výše, důležitým aspektem výpočtu jsou **stavy** výpočetního stroje. Stav stroje tiny má tři složky:[4]

1. obecné registry,
2. programový čítač,
3. paměť.

Přesněji řečeno, stavem myslíme konkrétní hodnoty „uložené“ v těchto složkách. Pro ilustraci si na chvíli stroj ještě zmenšíme (na velikost, která už není pro výpočty prakticky použitelná, ale která se nám vejde na stránku). Přisoudíme mu pouze 2 obecné registry (prozatím je nazveme r_1 a r_2), programový čítač pc a 12 bajtů (slabik) paměti. Stavy takovéhoto vskutku mikroskopického výpočetního stroje vypadají jako řádky této tabulky:

pc	r_1	r_2	paměť
0000	0000	0000	00 00 00 00 00 00 00 00 00 00 00 00
0004	0007	03ff	de ad de ad de ad de ad de ad de ad
0000	1007	7fff	00 00 00 00 00 00 00 de ad 00 co de

Samozřejmě zatím nevíme, jaký mají takovéto stavy význam – tím se budeme zabývat za chvíli. Můžeme udělat ale jiné zajímavé pozorování – totiž kolik různých stavů takový výpočetní stroj má. Protože máme k dispozici pevný počet bitů, dva stavy se od sebe budou lišit v případě, že se liší alespoň v jednom bitu. To zejména znamená, že různých stavů bude **konečný počet**; velmi jednoduchá kombinatorická úvaha nás pak dovede k výsledku, že různých stavů je 2^{144} – i pro takto malinký stroj se jedná o velmi úctyhodný počet.
„Skutečný“ stroj tiny (ten, který budeme používat, resp. programovat) má 16bitový programový čítač, 16 16bitových registrů a 64 KiB paměti, tzn. jeho stav obsahuje 65570 bajtů resp. 524560 bitů informace. Různých stavů tedy existuje 2^{524560} což je přibližně 10^{157908}. Pro srovnání, celkový počet baryonů (hlavně protonů a neutronů) ve vesmíru se odhaduje na 10^{80}.

- počáteční stav = program
- akce = přechod mezi stavy
- stav přesně udává akci
  - determinismus
  - 1 program = 1 výpočet

1.1.3 Akce Velmi důležitou vlastností takto definovaného stroje je, že libovolný stav **přesně určuje** celý následující výpočet – jinými slovy, náš výpočetní model je **deterministický**. Chování tohoto typu stroje je velmi jednoduché – je řízen sadou pravidel, podle které určíme jak z daného stavu odvodíme ten následovný.
Výpočet budeme obvykle začínat ve stavu, kdy jsou všechny registry nulové, ale obsah paměti nikoliv – počáteční obsah paměti budeme nazývat **programem**. S každým programem se tak váže právě jeden výpočet.[5]

- výpočet může být konečný
- rozlišení důvodu ukončení
  - konec programu
  - chybná instrukce
  - selhání tvrzení

1.1.4 Speciální stavy Výpočet může mít jeden ze dvou základních tvarů:

1. konečný výpočet, tvořen konečnou posloupností po dvou různých stavů, je obvyklá a žádoucí situace; může mít dvě varianty:
   a. úspěšný výpočet který byl ukončen instrukcí halt, přitom **výsledek** výpočtu jsme obvykle schopni odečíst z posledního stavu,
   b. neúspěšný výpočet, který byl ukončen chybou (viz také níže),

[4] Viz také sekce B.2 sbírky.

[5] To může vypadat na pohled velmi nerealisticky – všichni zřejmě máme zkušenost, kdy spuštění téhož programu (v neformálním smyslu) vede k různým výsledkům. To je způsobeno jednak tím, že reálné počítače nejsou deterministické (reagují na vnější události) a také tím, že v reálných situacích často nemáme počáteční stav zcela pod kontrolou a to co se nám jeví jako tentýž stav je ve skutečnosti mnoho různých, ale těžko odlišitelných, stavů.

2. nekonečný výpočet, ve kterém se nějaká posloupnost stavů nekonečněkrát opakuje – je složen z konečného prefixu (kde se stavy neopakují) a nekonečné smyčky, celkově má tvar tzv. lasa.

Chyby, které mohou výpočet ukončit mohou být různých typů:

- nepodařilo se dekódovat instrukci, tzn. na adrese uložené v registru `pc` nezačíná platné kódování žádné instrukce,
- při vykonání instrukce došlo k fatální chybě (typickým příkladem je zde dělení nulou, případně neplatný přístup do paměti),
- selhalo **tvrzení** (angl. **assertion**) realizované instrukcí `asrt` – došlo k porušení programátorem určené vstupní nebo výstupní podmínky nebo invariantu,
- byla detekována sémanticky chybná operace, obvykle podle anotací vložených překladačem – z pohledu stroje by výpočet mohl bez problémů pokračovat, ale pravděpodobně by vedl k nesprávnému výsledku – situace, která se podobá na selhané tvrzení z předchozího bodu.

1.2: Instrukční sada

V dalším se budeme zabývat akcemi, které má stroj k dispozici. Abychom mohli popsat jejich **efekt**, musíme si ale nejprve upřesnit jak vypadají jednotlivé složky stavu a jaký mají význam.

1.2.1 Registry Nyní se na jednotlivé složky stavu podíváme blíže. Stroj `tiny` má 16 „obecných“ a jeden „speciální“ registr.
Obecné registry mají mnemonické názvy, které ale na výpočet nemají žádný vliv. Naznačují pouze typické použití (programy, které vzniknou překladem z jazyka C se budou tohoto konvenčního použití držet):

- `rv` od **return value** je registr určený pro předávání návratové hodnoty z podprogramu (více si o podprogramech povíme ve třetí kapitole),
- `l1` až `l7` jsou registry pro lokální proměnné a pro předávání parametrů podprogramům (jak později uvidíme, mezi formálním parametrem a lokální proměnnou není v jazyce C příliš velký rozdíl),
- `t1` až `t6` slouží pro dočasné hodnot – mezivýsledky, např. při výpočtu složených výrazů – při výpočtu `a + b + c` budeme potřebovat dočasně někam uložit výsledek `a + b`,
- `bp` a `sp` opět souvisí s podprogramy, konkrétněji se správou zásobníku.

Krom těchto 16 obecných registrů, které mohou být operandy libovolné aritmetické instrukce (a zároveň také jejich cílovým registrem), má stroj `tiny` ještě speciální registr

- `pc` (program counter, programový čítač), který obsahuje adresu ze které bude načtena, dekódována a provedena další instrukce.

- 16 obecných registrů
  - `rv`, `l1` … `l7`, `t1` … `t6`, `bp`, `sp`
  - výpočetně zcela rovnocenné
  - `sp` má navíc speciální využití
  - jména jinak pouze pro lidi
- programový čítač `pc`

1.2.2 Paměť Stroj `tiny` disponuje 64 KiB paměti. Tím se myslí, že tato paměť je složená z 2^{16} buněk, přitom každá buňka je schopna uchovávat číslo od 0 do 255. Tyto buňky jsou navíc očíslované (mají adresy).
S paměťovými buňkami lze pracovat použitím instrukcí `ld` a `st` (více o nich ve čtvrté kapitole). Se kterou buňkou si přejeme pracovat určuje **hodnota uložená v registru**, tzn. adresa zejména může být výsledkem nějakého výpočtu. Všimněte si, že se jedná o zcela zásadní rozdíl oproti registrům – se kterými registry daná instrukce pracuje je její **pevnou součástí**.

- 64 KiB (65536 jednoslabičných buněk)
- adresace 16bitovým číslem
- každá adresa je platná
- program začíná adresou 0

1.2.3 Sémantika Krom **syntaxe** – toho, jak instrukce zapisujeme, potažmo kódujeme, nás bude zajímat jejich **sémantika** – význam. Co instrukce znamená budeme zejména popisovat tím, jaký má efekt na stav. Stav stroje jednoznačně určuje, jaká instrukce bude spuštěna – je to ta, které kódování je uloženo na adrese `pc` (každá instrukce je kódovaná do 4 bajtů, tzn. ve skutečnosti je uložena na adresách `pc`, `pc + 1`, `pc + 2` a `pc + 3`).
Řada instrukcí bude mít podobné efekty. Zejména každá instrukce, která není instrukcí řízení toku, zvýší registr `pc` o 4, čím způsobí, že jako další bude spuštěna instrukce na nejbližší vyšší adrese.

- popis za pomoci mikroinstrukcí
- efekt instrukce na stav
- instrukce určena stavem
  - 4 bajty od adresy dané `pc`

1.2.4 Operandy Podobně obsahují téměř všechny instrukce nějaké **operandy**, které určují, s jakými daty bude operace pracovat. Typicky budou operandy **identifikátory registrů**. Počet a forma operandů se pro různé operace bude lišit, ale ve všech případech spadají do těchto mezí:

- součástí instrukce můžou být až 2 **vstupní registry** – operandy, které určí, ze kterých registrů se mají načíst vstupní hodnoty,
- instrukce může mít nejvýše jeden **výstupní registr** – operand, který určí, do kterého registru bude zapsán výsledek,
- instrukce může obsahovat nejvýše jeden **přímý operand** – 16bitové slovo, které je přímo součástí instrukce, a které se použije jako jedna ze vstupních hodnot (která to bude závisí od operace).

- registrové operandy
  - 0–2 vstupní registry
  - 0 nebo 1 výstupní registr
- přímý operand
  - hodnota je součást instrukce

1.2.5 Kódování Instrukce stroje `tiny` mají velice jednoduché a pravidelné kódování – je to zejména proto, abychom byli v případě potřeby schopni instrukce dekódovat (alespoň přibližně) i „ručně“, z číselného

- dvě slova (2·16 = 32b)
- horní slovo
  - kód operace
  - výstupní registr
  - vstupní registr 1
- spodní slovo
  - vstupní registr 2 nebo

výpisu obsahu paměti. Také se tím značně zjednodušuje implementace dekodéru (který máte k dispozici jako ukázku ve sbírce).
Instrukce jsou kódovány do dvou 16bitových slov, přitom:

1. horní slovo obsahuje:
   a. 8bitový kód operace v horní slabice,
   b. dva registrové operandy ve spodní slabice – výstupní registr v horní půlslabice a první vstupní registr v té spodní,
2. spodní slovo kóduje zbývající vstupní operand:
   - druhý vstupní registr (v nejvyšší půlslabice), nebo
   - přímý operand (využije celé spodní slovo).

Jak se dekódují operandy je určeno kódem operace, který je proto potřeba dekódovat jako první.

- 1× vstup + 1× výstup
- nastaví hodnotu registru
- copy $reg_1 \rightarrow reg_2$
- put $imm \rightarrow reg_1$

1.2.6 Kopírování hodnot Nejzákladnější operací, kterou můžeme v programu potřebovat, je nastavení registru, a to buď na předem známou konstantu, nebo na hodnotu aktuálně uloženou v některém jiném registru. K tomuto účelu můžeme použít operace copy (kopie mezi registry) a put (nastavení registru na konstantu).

- aritmetika, srovnání
- bitové operace (logické, posuvy)
- 2× vstup + 1× výstup
- vstup/výstup se mohou překrýt

1.2.7 Binární operace Binárních operací je k dispozici celá řada – odpovídají běžným aritmetickým a bitovým operacím, které z velké části již znáte. Patří sem:[6]

- aritmetika: sčítání add, odečítání sub, násobení mul, dělení (sdiv, udiv, smod, umod – rozdíly mezi verzemi s___ a u___ vysvětlíme později),
- srovnání – výsledkem je hodnota 0 nebo 1: rovnost eq, nerovnost ne, znaménkové porovnání slt, sgt, sle, sge, neznaménkové porovnání ult, ugt, ule, uge,
- bitové operace: logické and, or, xor a posuvy shl, shr, sar (aritmetický).

- efekt na hodnotu pc
- nepodmíněný přímý skok jmp
- podmíněný přímý skok jz, jnz
- volání podprogramu později

1.2.8 Operace řízení toku Do registru pc není možné běžnými instrukcemi přímo zapisovat ani z něj číst; efekt jmp addr je velmi podobný hypotetické instrukci put addr → pc, ale protože je tento efekt velmi odlišný od všech ostatních použití operace put, má svůj vlastní zápis.[7]
Naproti tomu operace jz a jnz se na žádnou další známou instrukci nepodobají – jsou totiž **podmíněné** – jejich efekt se bude lišit podle hodnoty operandu. Uvažme instrukci jz reg, addr – tato instrukce se bude chovat jako jmp addr v případě, že je v registru reg uložena nula. V případě opačném bude výpočet pokračovat následující instrukcí. Jinými slovy:

- jz reg, addr nastaví pc na hodnotu přímého operandu addr, je-li v registru reg uložena nula,
- jinak zvýší hodnotu pc o 4.

Operace jnz je pak analogická, liší se pouze inverzí podmínky (skok se provede je-li registr nenulový).

- zastavení halt
  - bez podmínky
  - indikuje úspěch
- tvrzení asrt
  - podmíněno vstupním registrem
  - nula = neúspěch → chyba

1.2.9 Řízení stroje Jak jsme již zmiňovali dříve, za výstup programu budeme brát pouze formu jeho ukončení:

- výpočet může být úspěšně dokončen, což program signalizuje provedením instrukce halt, která nemá žádné operandy,
- výpočet může skončit chybou – tuto lze signalizovat instrukcí asrt reg, přitom chyba nastane pouze v případě, že v registru reg je uložena nula (v opačném případě výpočet pokračuje další instrukcí, tzn. registr pc se zvýší o 4) – jedná se o analogii tvrzení assert jak ho znáte z jazyka Python,
- výpočet může skončit jinou chybou (špatná instrukce, atp.) nebo se může „zacyklit“ – neskončí nikdy (v praxi budeme vždy délku výpočtu nějak uměle omezovat,[8] „nikdy“ je totiž velmi dlouhá doba).

1.3: Řízení toku

V poslední části první kapitoly si ukážeme, jak ve strojovém kódu zapsat standardní konstrukce řízení toku z vyšších programovacích jazyků – podmíněný příkaz a cyklus. Protože jediný vyšší programovací

[6] Mnohem podrobněji je význam jednotlivých operací popsán ve sbírce.
[7] Tento přístup – striktně oddělovat instrukce řízení toku – je zcela běžný i u reálných procesorů.
[8] Jak již bylo zmíněno dříve, stroj tiny může v principu zacyklení spolehlivě detekovat (je to díky tomu, že konfigurace stroje má pevnou velikost – existuje tak pouze konečně mnoho různých konfigurací). V praxi může být takový cyklus velmi dlouhý a tedy těžce odhalitelný.

jazyk, který v tuto chvíli známe, je Python, budeme jej prozatím využívat jako pseudokód. V pozdějších kapitolách pak budeme používat jazyk C.

1.3.1 Konstrukce strojového kódu Jak již bylo zmíněno v kapitole B, překladem rozumíme zobrazení, kterého vstupem je nějaký program P, zatímco výstupem je nový program Q (typicky v jiném jazyce), přitom ale platí, že výpočet P končí úspěchem **právě když** výpočet Q končí úspěchem. Zobrazení je přitom definováno pouze pro **platné** vstupní programy.

Nyní jsme připraveni provést první náčrt toho, jak toto zobrazení **spočítáme** (jinými slovy, jak funguje překladač). Z předchozího studia[9] víte, že **výrazy** (zejména aritmetické, ale i libovolné jiné) můžeme reprezentovat pomocí stromů (ve smyslu datové struktury). Stejný přístup lze použít i pro příkazy a další prvky programovacích jazyků.

Jinak řečeno, zdrojové jazyky překladu mají typicky **rekurzivní strukturu**, kterou lze zachytit (abstraktním) **syntaktickým stromem**. Tento strom (angl. AST, z **abstract syntax tree**) je tak přesnou reprezentací **vstupního programu** a budeme jej považovat za startovní bod překladu.[10]

Jednoduchý překladač sestavuje strojový kód **rekurzivně**, podle struktury vstupního stromu. To zejména znamená, že překlad nějakého uzlu obdržíme vhodnou kombinací překladů jeho potomků.

V této kapitole si ukážeme pouze překlad uzlů, které odpovídají příkazům řízení toku – `if` a `while` – abychom získali představu, jaký je vztah mezi vstupním programem (tím, který typicky píšeme) a strukturou výstupního strojového kódu. Kompletní algoritmus překladu jazyka C[11] vybudujeme postupně (naleznete jej vždy ve sbírce, ve skriptech a přednášce vyzvedneme jen ty nejzajímavější části).

- rekurzivní algoritmus
- postupujeme po struktuře programu
  - blok složíme z příkazů
  - výraz složíme z podvýrazů
- zatím pouze intuitivně

1.3.2 Podmíněný příkaz Použité instrukce:

- `jmp`, `jz`, `jnz`
- podmíněný skok realizuje **rozhodování**

- nejjednodušší forma: `if b:` $\mathtt{stmt}_1$
  - realizace jedním podmíněným skokem
  - není-li `b` splněno, přeskoč tělo
- rozšíření: `else` větev
  - podmíněný + nepodmíněný skok
  - na konci „then“ přeskočíme za blok `else`

1.3.3 Nekonečný cyklus

- jak zapsat `while True`?
- realizace jedním skokem
  - nepodmíněným
  - na nižší adresu
- časem na něj opět narazíme

1.3.4 Cyklus `while`

- nekonečný cyklus + podmíněný `break`
- `break` je jeden podmíněný skok
  - pro `while` na začátku těla
  - skok za konec těla
- jednodušší: `do-while`

Část 2: Základní prvky jazyka C

Tato kapitola se bude zabývat základními stavebními kameny jazyka C – hodnotami, proměnnými, výrazy a příkazy. U každé výpočetní konstrukce si zejména ukážeme, jak se abstraktní zápis na úrovni jazyka C realizuje sekvencemi instrukcí konkrétního výpočetního stroje.

2.1: Hodnoty, objekty, proměnné

V této části se budeme zabývat základními **datovými** prvky jazyka C – připomeneme si pojmy jako hodnota, objekt, proměnná nebo typ, které již znáte z jazyka Python, a zasadíme je do nového kontextu.

2.1.1 Hodnota Stejně jako v jazyce Python, fundamentálním předmětem výpočtu je v jazyce C **hodnota** – pro tuto chvíli se bude jednat o celé číslo v nějakém pevném rozsahu, nicméně v pozdějších kapitolách se setkáme i se složitějšími hodnotami.

- význam podobný Pythou
- celé číslo
- později složitější
- 12 ~ XII ~ $(1100)_2$ ~ `0xc`

[9] Např. příklady z kapitoly 9 sbírky předmětu IB111, nebo mnohem podrobněji přednášky 11 a 12 téhož.

[10] Skutečné překladače tento strom konstruují ze vstupního textu. Touto částí překladu – syntaktickou analýzou – se zde zabývat nebudeme. Předpokládejme, že naším vstupem je již hotový abstraktní syntaktický strom.

[11] Resp. jeho podmnožiny, kterou budeme v tomto předmětu používat.

Pozor, hodnota je **abstraktní** – hodnota **není** totéž co **reprezentace**. Zejména nesmíme hodnotám přisuzovat vlastnosti použité reprezentace. Zápisy `0x10`, 16, šestnáct, XVI všechny reprezentují XXX tutéž XXX hodnotu.
[12]

Cf. Platón.

- pracuje s hodnotami
- hodnoty je potřeba si pamatovat
- realizace výpočetním strojem
- příklad: součet dvou hodnot

2.1.2 Operace Konceptuálně není výpočet ničím jiným, než mechanickou manipulací hodnot použitím kompatibilních **operací**. Klasickým příkladem operace je např. sčítání – vstupem jsou dvě celočíselné hodnoty a výstupem je nová hodnota. Aby bylo možné výpočet provést, je nutné si potřebné hodnoty **pamatovat** – při výpočtech ve středoškolské matematice k tomu používáme typicky papír a tužku. Počítač k tomu bude samozřejmě využívat nějaký typ elektronické **paměti**.

- abstrakce (zobecnění) paměťové buňky
- pamatuje si hodnotu
- má identitu
  - různost při stejné hodnotě
  - stejnost během výpočtu

2.1.3 Objekt Přímá práce s pamětí a registry je pro větší programy značně nepohodlná – musíme při programování neustále pamatovat, kde máme uloženy které hodnoty (ve kterém registru nebo na jaké adrese), navíc jména registrů nejsou příliš popisná a je jich omezený počet.
Z jazyka Python jsme zvyklí hodnoty uchovávat v **proměnných**. Vazba mezi proměnnou a hodnotou ale není přímá – ani v Pythonu, ani v C. Hodnoty jsou totiž invariantní, anonymní entity – nemají identitu, ani schopnost se vnitřně měnit. Abychom obdrželi sémantiku, na kterou jsme při programování intuitivně zvyklí, musí do hry vstoupit ještě jeden prvek – **objekt**.
Hlavními vlastnostmi objektu jsou:

- schopnost uchovávat, číst a měnit **hodnotu** (abstraktní entitu programovacího jazyka),
- identita:
  - můžeme od sebe rozlišit různé objekty i v případě, že zrovna obsahují stejnou hodnotu,
  - jsme schopni určit, že se jedná o tentýž objekt, i když se hodnota v něm uložená během výpočtu změnila.

Objekt je tak zobecněním paměťové buňky – má operace „přečti“ a „ulož“ a jeho identita je obdobou adresy.

- vazba jména na objekt
- syntaktický rozsah platnosti
- vazba je neměnná (pevná)
- platnost jména ~ život objektu

2.1.4 Proměnná Objekty mají sice identitu, ale nemají **jména** – jejich identita je více nebo méně abstraktní.[1] Abychom tedy mohli s objektem pracovat, potřebujeme mu přiřadit jméno – a to je přesně úloha **proměnné**.
Proměnná je (v jazyce C) pojmenovaný objekt, přičemž její jméno (identifikátor) má **syntakticky** omezený **rozsah platnosti**[13] – přímo ze zdrojového kódu umíme lehce identifikovat, ve kterých příkazech a výrazech je použití tohoto jména přípustné, a případně ke které deklaraci se váže. Vazba mezi jménem (proměnnou) a objektem je **pevná** – vznikne při deklaraci proměnné a až do jejího zániku tuto vazbu není lze měnit.[14]

- je vlastnost hodnoty
- určuje přípustné operace
- určuje chování operací
- pouze v době překladu

2.1.5 Typ Různé hodnoty mají různé vlastnosti a různé operace. Uvažme 16bitové hodnoty bez znaménka $u_1 = 3, u_2 = 5$ a podobné (ale ne tytéž!) 16bitové hodnoty se znaménkem $s_1 = 3, s_2 = 5$. Zřejmě:

- $s_1 + s_2 = 8$, podobně $u_1 + u_2 = 8$,
- $s_2 - s_1 = -2$ ale $u_2 - u_1 = 65534$.

Je tedy potřeba podobné hodnoty rozlišovat – k tomu slouží **typy**. Typy mají v programovacím jazyce dvě funkce:

1. určí, jaké operace jsou pro dané hodnoty přípustné,
2. je-li operace přípustná, typy mohou ovlivnit její **význam** – např. existují dvě různé operace odečítání (viz výše) pro 16bitová čísla: znaménkové a neznaménkové.

Typ je **vlastnost hodnoty**, ale to neznamená, že je ke každé hodnotě „fyzicky“ připojen její typ – řada programovacích jazyků, a C mezi nimi, typovou informaci uchovává pouze v **době překladu** – ve strojovém kódu bychom informaci o typech hledali marně.[15] To samozřejmě neznamená, že typy nemají pro výsledný strojový kód důsledky – bude na nich třeba záviset, jestli se pro srovnání použije operace `slt` nebo `ult`, atp.
Typy můžeme přisuzovat krom hodnot také objektům a skrze objekty také proměnným. Je-li objekt nějakého typu, znamená to, že je schopen uchovávat hodnoty pouze tohoto typu. Je-li proměnná nějakého typu, znamená to, že je svázána s objektem tohoto typu.[16]

---

[12] V standardní implementaci jazyka Python je každý objekt identifikovatelný adresou, na které je v paměti uložen. V jazyce C to ale neplatí, protože objekt nemusí mít adresu žádnou, nebo se jeho adresa může během výpočtu měnit.

[13] Známý též jako **lexikální** nebo **statický**, v kontrastu s **dynamickým**.

[14] Zde se objevuje důležitý rozdíl mezi C a Pythonem. Přiřazení v C, jak za chvíli uvidíme, značí **zápis do objektu**, kdežto v jazyce Python značí změnu vazby na **jiný objekt**.

[15] Tomuto konceptu se říká „vymazání typů“, angl. „type erasure“ – udržování informací o typech za běhu programu představuje dodatečnou režii a je-li to možné, překladače se tomu vyhýbají.

[16] Polymorfismus – schopnost objektu uchovávat různé typy hodnot – lze chápat např. tak, že takovému objektu přisoudíme součtový typ. V jazycích, kde je možné měnit vazbu mezi proměnnou a objektem může polymorfismus existovat jak na úrovni objektu (lze uložit různé typy hodnot) tak na úrovni proměnné (k jednomu jménu lze vázat různé typy objektů). To se ale nacházíme už mimo hranice tohoto předmětu.

2.1.6 Deklarace (jak vznikne proměnná, objekt, …)

- proti Pythonu nový prvek
- `typ jméno = výraz;`
- vytvoří zároveň objekt i vazbu
- bez inicializace = zapovězená hodnota

2.2: Výrazy

Nyní známe základní datové (pasivní) prvky jazyka a můžeme se začít zabývat výpočetními (aktivními) – těmi nejzákladnějšími jsou **výrazy**.
(XXX denotační + operační sémantika)

2.2.1 Elementární výrazy (jméno proměnné, konstanta)
Pozor – číselný literál musí být platnou hodnotou příslušného typu: je-li typ `int` 16bitový, literál `0xffff` je chybný (tak velké číslo není možné reprezentovat). Naproti tomu, protože `0xffffu` je typu `unsigned`, je zde vše v pořádku.

- název proměnné: `x` (typ dle proměnné)
- číselný literál:
  - typu `int`: `3`, `-1`, `0x1f`
  - typu `unsigned`: `3u`, `0x1fu`

2.2.2 Aritmetické a logické operace (operátory bez vedlejších efektů)

- popisují hodnotu, žádný vedlejší efekt
- e_1 + e_2, …, e_1 % e_2
- e_1 << e_2, e_1 >> e_2
- e_1 & e_2, e_1 | e_2, e_1 ^ e_2
- -e_1, ~e_1, !e_1

2.2.3 Výpočet hodnoty výrazu (strojový kód; „vyhodnocení do registru“)

- vyhodnocení do registru R
- var ~ copy A → R
- e_1 + e_2, …, e_1 ^ e_2
  - vyhodnoť e_1 do t_1
  - vyhodnoť e_2 do t_2
  - add t_1, t_2 → R

2.2.4 Kontrola typů (je výraz typově správný?)

- ověří spávnost operace × hodnoty
- vkládá implicitní konverze
  - přesná pravidla jsou složitá
- špatně utvořený program zamítne

2.2.5 Implicitní konverze Aritmetické, bitové, atp. operace vyžadují, aby byly operandy stejného typu. Jazyk C zároveň nepodporuje operace na typech menších než `int` resp. `unsigned` (pro nás to znamená, že „jednobajtová“ aritmetika neexistuje – všimněte si, že podobně neexistuje ani ve výpočetním stroji).
Povýšení: typy s rozsahem, který je menší než rozsah typu `int`, se nejprve zvětší na `int`.[17] Po povýšení se pak najde společný typ – je-li to možné, preferuje se znaménkový typ.[18]
Pozor: povýšení se dotkne i unárních operátorů. Máme-li `signed char x = 5;`, bude hodnota výrazu `-x` typu `int`, nikoliv typu `signed char`!
V naší omezené verzi jazyka to dopadne takto (všechny případy jsou symetrické):

1. povýšení – vše menší než `int`
   - vejde se do `int` → `int`
   - jinak `unsigned`
2. stejná znaménkovost → pouze zvětšení
3. různá vede na:
   a. znaménkový je-li striktně větší
   b. jinak na neznaménkový

operand	operand	společný typ
`signed char`	`signed char`	`int`
	`unsigned char`	`int`
	`int`	`int`
	`unsigned`	`unsigned` !
`unsigned char`	`unsigned char`	`int`
	`int`	`int`
	`unsigned`	`unsigned`
`int`	`int`	`int`
	`unsigned`	`unsigned`
`unsigned`	`unsigned`	`unsigned`

Pozor: na řádcích označených ! dochází ke konverzi operandu s menším rozsahem ve dvou krocích. Nejprve je rozšířen na typ `int` a poté až na společný `unsigned`. Zejména to znamená, že jednobajtové hodnoty jsou převedeny **znaménkovým rozšířením**. Např.:

[17] Rozsahy všech jednobajtových typů (`char`, `signed char`, `unsigned char`) jsou menší než rozsah typu `int`. Rozsah typu `int` není větší než rozsah typu `unsigned`, protože např. `40000` není přípustná hodnota typu `int` ale je to přípustná hodnota typu `unsigned`.
[18] V našem jazyce žádné znaménkové typy větší než `int` nejsou, proto se toto pravidlo neuplatní – společný typ je `unsigned` je-li alespoň jeden operand `unsigned`, jinak je to vždy `int`.

```
signed char x = -3;         /*   0xfd */
unsigned y = 1;             /* 0x0001 */
assert( x + y == 65534u ); /* 0xfffe */
```

Součet je 65534 (`0xfffe`), nikoliv 254 (`0x00fe`) jak by mohl někdo čekat.

- `var` = e_1 (pozor na levou stranu)
- proběhne typová konverze pravé strany
- výraz s vedlejším efektem
- provede zápis do objektu
- hodnota je to, co bylo zapsáno

2.2.6 Přiřazení Prozatím budeme uvažovat pouze přiřazení tvaru `var` = e_1 – levá strana může být tedy pouze název proměnné. Pozor, vyhodnocení se bude pro složitější formy lišit!
Vyhodnocení tohoto typu přiřazení probíhá takto:

1. vyhodnotí se pravá strana,
2. výsledek se převede (konvertuje) na typ levé strany,
   - má-li typ levé strany větší nebo stejný rozsah, probíhá stejně jako konverze operandů aritmetických operátorů,
   - je-li levý operand menší a je neznaménkového typu, vyšší bity se implicitně oříznou,
   - je-li levý operand menší a je znaménkového typu, výsledek závisí na implementaci – program **může** spadnout, ale **nemá** nedefinované chování,
3. převedená hodnota se zapíše do objektu určeného levou stranou,
4. výsledná hodnota (přiřazení je výraz) je ta, která byla zapsaná (tzn. hodnota po konverzi z druhého bodu).

- binární e_1 `&&` e_2, e_1 `||` e_2
- ternární e_1 `?` e_2 `:` e_3

2.2.7 Booleovské operace (řízení toku)

- vyhodnocení e_1 `&&` e_2 do R:
  - vyhodnoť e_1 do R
  - je-li R true, vyhodnoť e_2 do R
- vyhodnocení e_1 `||` e_2 do R:
  - vyhodnoť e_1 do R
  - je-li R false, vyhodnoť e_2 do R

2.2.8 Vyhodnocení booleovských operací (strojový kód)

2.3: Příkazy

Výrazy nám poskytují mocný výpočetní aparát, ale díky své deklarativní struktuře nejsou ideální pro popis sledu výpočetních kroků. Je-li potřeba provést nějakou sekvenci operací v daném pořadí, budou se mnohem lépe k zápisu hodit **příkazy**.

- e_1`;` – jakýkoliv výraz
- provede vedlejší efekty
- hodnota je zapomenuta

2.3.1 Výrazový příkaz (např. `a = b;`)

- sekvence příkazů
- uzavřena do složených závorek
- připouští navíc deklarace
  - rozsah platnosti jmen

2.3.2 Složený příkaz (ve složených závorkách)

- `if ( expr ) stmt`
- `if ( expr ) stmt`$_1$ `else stmt`$_2$

2.3.3 Podmíněný příkaz (`if`, `else`)

- `do stmt while ( expr );`

2.3.4 Cyklus `do … while` (jump na konci)

- `break;` – ukončí cyklus
- `continue;` – ukončí aktuální iteraci

2.3.5 Řízení iterace (`break`, `continue`)

- `while ( expr ) stmt`

2.3.6 Cyklus `while` (while true + break)

- `for ( decl; expr`$_1$`; expr`$_2$ `) stmt`

2.3.7 Cyklus `for` (deklarace)