PB173 – Binární programování Linux

II. Parsery

Jiri Slaby

Fakulta informatiky
Masarykova univerzita

27. 9. 2016

Důležité informace

- Kolokvium za DÚ
  - DÚ do příštího cvičení
- Login/heslo
  - vyvoj/vyvoj
- GIT: `http://github.com/jirislaby/pb173-bin`
  - `git pull --rebase`
- Studijní materiály v `ISu`

Sekce 1

Parsery

Parsery

- Analyzátor jazyka
  - Přirozeného, *programovacího*, ...
- Výstupem je nějaký lépe zpracovatelný popis jazyka
  - Např. informace v grafu nebo stromu
  - Několik mezifází, jak toho dosáhnout
  - Projdeme si na dalších slidech
- Získané informace se dále využijí
  - K překladu do jiného jazyka (např. strojového)
  - K interpretaci
  - ...
- Detailněji o jazycích a parserech např. v „Dragon Book"

Výroba parseru

1. Ručně psané
   - Rychlost parseru a explicita
   - GCC (**Demo**)
2. Generované nástroji
   - Rychlost napsání parseru a přehlednost kódu
   - LEX+YACC
   - FLEX+BISON
   - ANTLR
   - A další

ANTLR

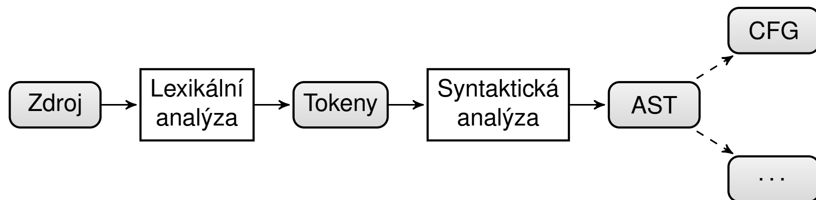
- Nástroj ke generování parserů
- Psaný v Javě
- Vstup je LL($*$) gramatika (shora-dolů, čitelná)
- Generuje parsery do různých jazyků
  - Java, C, C++, C#, ...
  - Podpora pro C jen ve verzi 3 (pro 4 se připravuje)

Úkol

Stáhněte a zprovozněte si ANTLR 3

1. `http://www.antlr3.org/download.html`
   - Ukládejte do `pb173-bin/02/`
   - ANTLRWORKS: „Version 1.5"
   - ANTLR: „Complete ANTLR 3.5.2 Java binaries jar"
2. Knihovna pro C (`libantlr3c` a `antlr3c-devel`)
   - RPM: `http://software.opensuse.org`
   - Zdroje: `http://www.antlr3.org/download/C/`
3. Vyzkoušejte generovat `Parser.g` z `pb173-bin/02/test/`
   - `make`
   - Spousty varování ignorujte
4. Pusťte jej
   - `./parser test_input`
   - Změňte `test_input` a zkuste znovu

Části parseru



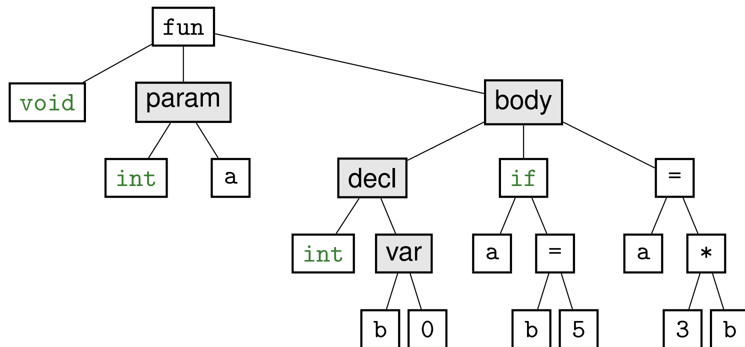
1. Lexikální analýza
   - Převod sekvence znaků na sekvenci „tokenů”
     - Např. „`int a`” převede na „`KEYWORD:int ID:a`”
2. Syntaktická analýza
   - Dává sekvence tokenů smysl?
   - Ano $\Rightarrow$ sémantická analýza
     - Přímá interpretace, tvorba AST, . . .

Vnitřní struktury parseru

```
void fun(int a)
{ /* comment */
  int b = 0;
  if (a)
    b = 5;
  a = 3 * b;
}
```

⇒

```
key:void id:fun LPAR key:int id:a RPAR
LCUR
  key:int id:b EQ num:0 SEMIC
  key:if LPAR id:a RPAR
    id:b EQ num:5 SEMIC
  id:a EQ num:3 MULT id:b SEMIC
RCUR
```



Formát celého souboru pro ANTLR

```
grammar Jmeno; // shoda s nazvem souboru
options {
   language = C; // vystupni jazyk parseru
}
@header {
#define X 10
   // vlozi se na zacatek generovaneho parseru
}
@members {
   static void moje_funkce() { ... }
   // vlozi se za hlavicku parseru
}

pravidlo1
   : pravidlo2
   | pravidlo3;
...
```

Části parseru v ANTLR

ANTLR obsahuje obě části v jednom souboru

- 2 typy pravidel
  - Počáteční *velké* písmeno – lexikální
  - Počáteční *malé* písmeno – syntaktická
- ; odděluje pravidla
- : odděluje levou a pravou stranu pravidla

Příklad

```
/* syntakticka analyza */
helloWorld
   : AHOJ SVETE
   | SVETE AHOJ { puts("Nepise se to obracene?"); }
   ;

/* lexikalni analyza */
AHOJ : 'Ahoj' ;
SVETE : 'Svete' ;
```

Pravá strana pravidel

- | odděluje možnosti
  - Více možností: `zvirata : savci | obojzivelnici ;`
  - Také kombinace podčástí: `strunatci : ('pe'|'ko') 's' ;`
- Opakování pravidla
  - `pravidlo?` – 0–1krát, např. `nicNeboX : 'x'? ;`
  - `pravidlo*` – 0–∞krát, např. `kolonaAut : 'auto'* ;`
  - `pravidlo+` – 1–∞krát, např. `zvukHada : 's'+ ;`
- Rozsahy
  - `'A'..'Z'` – celá velká ASCII abeceda
- Speciální pravidlo pro konec souboru: `EOF`
  - Donutí číst celý soubor

Úkol

Rozšíření gramatiky

1. Prozkoumejte obsah `02/test/Parser.g`
2. Vytvořte si token `NUMBER` pro čísla
   - `('0'..'9')+`
3. Vytvořte si tokeny pro `+` a `-`
4. Vytvořte si nové pravidlo `expr` pro tyto 2 operace
   - Bude obsahovat 2 možnosti (oddělené `|`)
   - Pro sčítání např. `NUMBER PLUS NUMBER`
5. Přidejte do `translationUnit` novou možnost
   - Bude odpovídat 1–∞ opakováním `expr`
6. Vyzkoušejte nově podporované vstupy
   - `1+2`, `100-10`, apod.

Sekce 2

Akce pravidel

Akce pravidel

Pravidla mohou obsahovat akce

- Kód spouštěný při určitých událostech
- `{ C code }` za pravostrannými tokeny
- Vykoná se ihned, když token odpovídá vstupu

Příklad

```
p : ID {
      puts("found an ID");
  } ID {
      puts("found second ID");
  } ID {
      puts("wow, 3 IDs");
  }
  ;
```

Úkol

Akce pravidel

1. Dopište akce na konec obou možností `expr`
2. Vypište nějaký text
   - Ale různý pro každou možnost
3. Vyzkoušejte

Vlastnosti tokenů

Každý token má vlastnosti

- `line` – zdrojový řádek
- `pos` – zdrojový sloupec
- `text` – tokenizovaný řetězec („Ahoj” pro AHOJ)
  - Jak pro tokeny (pravé strany), tak pro pravidla (levé strany)
  - Typ: `pANTLR3_STRING`, obsahuje `char *chars`
- Akce můžou odkazovat předcházející tokeny
  - Implicitně jménem tokenu/pravidla: `p : NUMBER { $NUMBER.line; } ;`
  - Nebo pojmenováním: `p : n1=NUMBER PLUS n2=NUMBER { $n2.pos; } ;`

Příklad

```
helloWorld : x=AHOJ y=SVETE {
        printf("line=\%d col=\%d\n", $x.line, $x.pos);
        printf("x=\%s y=\%s whole=\%s\n", $x.text−>chars, $y.text−>chars, $helloWorld.text−>chars);
  } ;
```

Úkol

Vlastnosti pravidel

1. Pro každou operaci sčítání/odčítání
   - Vypište číslo řádku a sloupce pro obě čísla (`$NUMBER.line` a `$NUMBER.pos`)
   - Vypište obě čísla jako řetězec (`$NUMBER.text->chars`)
2. Vyzkoušejte roztroušením textu po testovacím vstupu

Předávání hodnot

Každé pravidlo muže akceptovat/vracet hodnotu

- Stejně jako funkce
- `levaStrana[type1 param1, ...] returns [typeR ret] : pravaStrana;`
  - Pak lze v akcích odkazovat na `$param1` typu `type1`
  - Akce *musí* nastavovat `$ret` typu `typeR`

Příklad

```
translationUnit
        : expr[5] EOF { printf("I'm done \%d\n", $expr.ret); }
;

expr[int par] returns [int ret]
  : n1=NUMBER PLUS n2=NUMBER {
    $ret = $par * atoi($n1.text->chars) + atoi($n2.text->chars);
  }
  | SVETE AHOJ { $ret = 0; }
  ;
```

Úkol

Vlastnosti pravidel

1. Vytvořte si pravidlo pro čísla
   - `number : NUMBER ;`
2. Z něj a z `expr` si vracejte (smysluplnou) hodnotu
   - Tj. počítejte
3. V `translationUnit` vypište výsledek
   - Pro každé `expr`

Inicializace/ukončení

Každé pravidlo muže mít inicializace/ukončení

- Na konci levé strany pravidla (před dvojtečkou)
- Před aplikováním pravidla: `@init{ ... }`
  - Např. inicializovat proměnné
- Po aplikování pravidla: `@after{ ... }`
  - Např. počítání

Příklad

```
expr[int par] returns [int ret]
@init{ $ret = 0; }
@after{ $ret--; }
  : n1=NUMBER PLUS n2=NUMBER {
    $ret = $par * atoi($n1.text->chars) + atoi($n2.text->chars);
  }
  ;
```