PB173 – Binární programování Linux

II. Parsery

Jiri Slaby

ITI, Fakulta informatiky

24. 9. 2013

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

Část I

Parsery

Parsery

- Analyzátor jazyka
  - Přirozeného, *programovacího*, . . .
- Výstupem je nějaký lepší popis jazyka
  - Např. informace v grafu nebo stromu
  - Několik mezifází, jak toho dosáhnout
  - Projdeme si na dalších slajdech
- 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

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

ANTLR

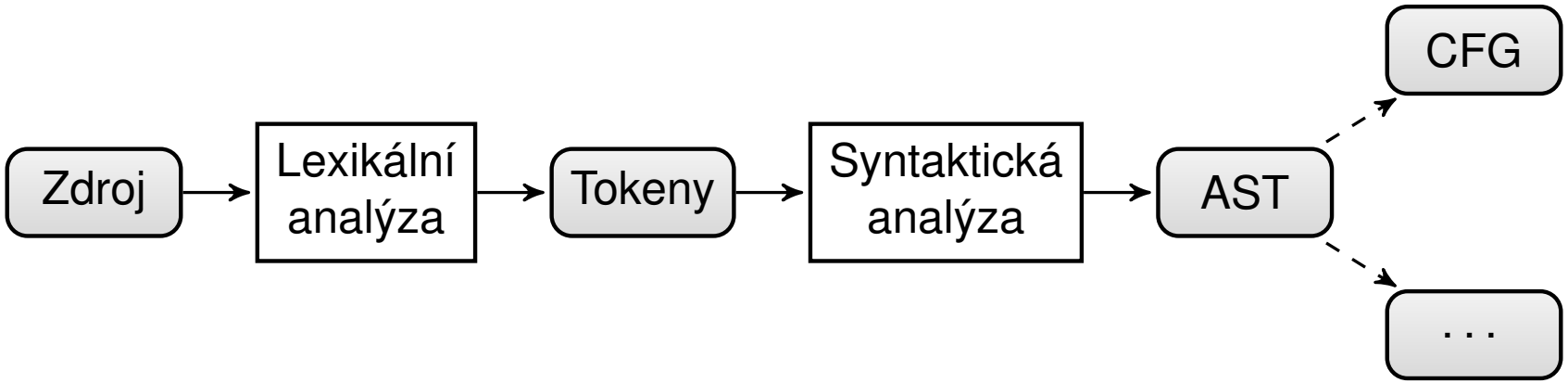
- Nástroj ke generování parserů
- Psaný v Javě
- Vstup je LL($*$) gramatika (shora-dolů, čitelná)
- Generuje parsery do různých jazyků
  - Java, C, C++, C#, . . .
  - Podpora pro C jen ve verzi 3 (pro 4 se připravuje)

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 Java binaries jar”
2. Knihovna pro C (`libantlr3c` a `libantlr3c-devel`)
   - RPM: `http://software.opensuse.org`
   - Zdroje: `http://www.antlr3.org/download/C/`
3. Vyzkoušejte generovat `Parser.g` z `pb173-bin/02/test/`
   - `make`
4. Pusťte jej
   - `./parser test_input`
   - Změnte `test_input` a zkuste znovu

Struktura parseru



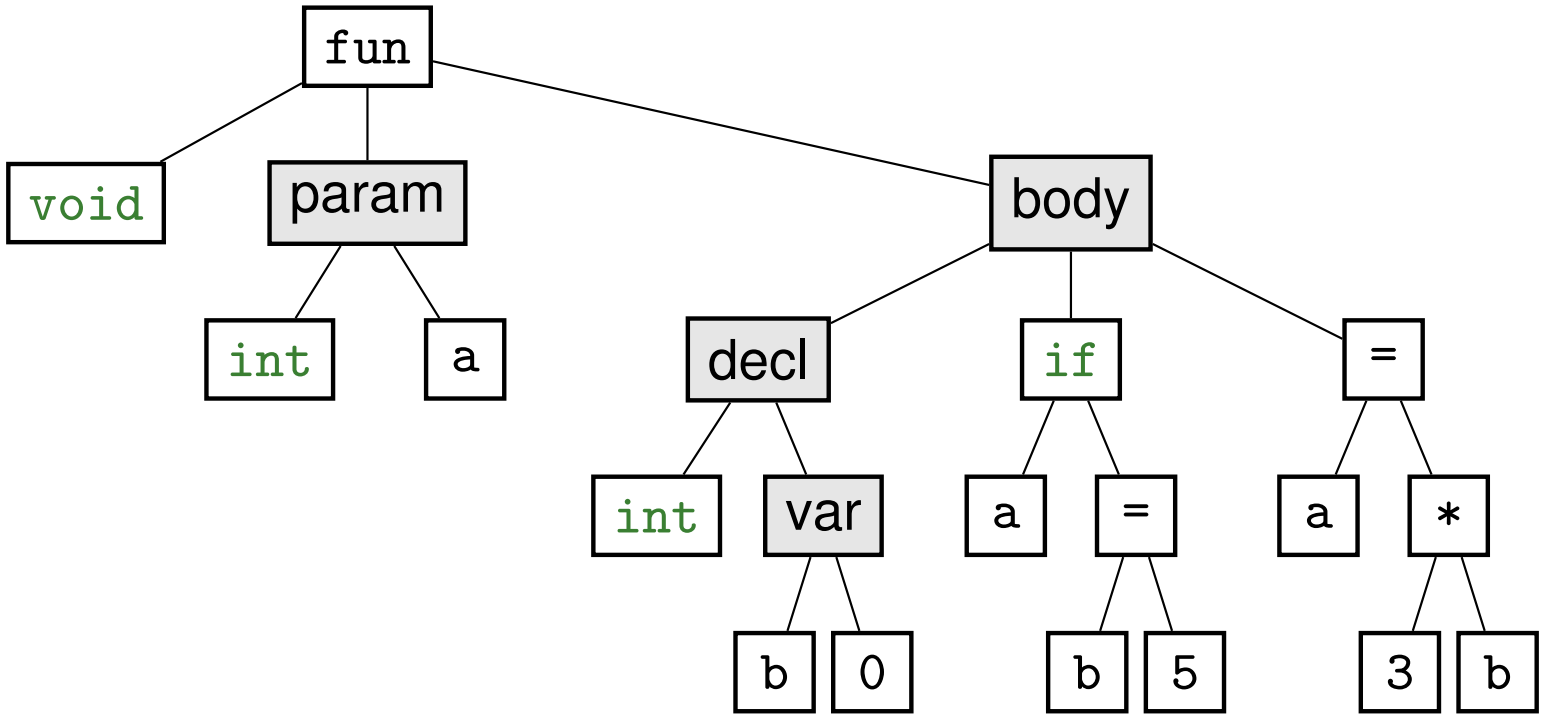
1. Lexikální analýza
   - Převod sekvence znaků na sekvenci „tokenů”
     - Např. „`int a`” převede na „`KEYWORD:int ID:a`”
2. Syntaktická analýza
   - Převod sekvence tokenů na strom. . .

Abstraktní syntaktický strom (AST)

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

$\Rightarrow$

```
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
```



ANTLR obsahuje obě části v jednom souboru

- 2 typy pravidel
  - Počáteční velké písmeno – lexikální
  - Počáteční malé písmeno – syntaktická

```
/* syntakticka analyza */
helloWorld
   : AHOJ SVETE // 2 tokeny v jednom syntaktickem pravidle
   | SVETE AHOJ { puts("Nepise se to obracene?"); }
   ;

/* lexikalni analyza */
AHOJ
   : 'Ahoj' // vytvori token Ahoj z retezce "Ahoj"
   ;

SVETE
   : 'Svete'
   ;
```

Pravá strana pravidel

- `|` odděluje možnosti
  - Kombinace podčásti: `zvirata : ('pe'|'ko')'s';`
- Opakování pravidla: `zvukHada : 's'+ ;`
  - `pravidlo?` – 0–1krát
  - `pravidlo*` – 0–∞krát
  - `pravidlo+` – 1–∞krát
- Speciální pravidlo pro konec souboru: `EOF`
  - Donutí číst celý soubor
- Může obsahovat akce
  - `p : ID { x++; } ID { x++; } ID { puts("neco"); } ;`
  - Akce můžou referencovat předcházející části
    - Implicitně (`p : ID { $ID... } ;`)
    - Nebo pojmenováním (`p : id1=ID { $id1... } ;`)

Formát celého souboru pro ANTLR

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

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

Úkol

Rozšíření gramatiky

1. Prozkoumejte obsah `02/test/Parser.g`
2. Přidejte do `helloWorld` nová pravidla
   - Pro anglický pozdrav
   - Možnost opakovaně zdravit „Hi” pomocí nového pravidla `hi*`
3. Dopište akce pro všechna pravidla `helloWorld`
   - Výpis nějakého textu
4. Vyzkoušejte nově podporované vstupy

Tvorba AST

- Do `options`
  - `ASTLabelType = pANTLR3_BASE_TREE;`
  - `output = AST;`
- Kořenem je uzel tvořený počátečním symbolem gramatiky
- Podstromy se připojují ve vnořených pravidlech
  - Pomocí speciálních přepisovacích pravidel
  - Explicitně: `pravaStrana -> podstromAST`
    - `^(KOREN pravaStrana)`
    - `KOREN` musí být v token nebo v `tokens`
  - Implicitně: `pravaStrana` (vytvoří podstrom `pravaStrana`)
    - `^` – kořen (pod)stromu
    - `!` – nevkládat do stromu

```
statement
   : expression^ ';'! // implicitni strom
   | e1=expression '+' e2=expression ';' -> ^(BINARY_OP '+' e1 e2)
```

Úkol

Dopište tvorbu stromu do gramatiky `test/Parser.g`

1. Nejprve přesuňte `test/Parser.g` do `parser/Parser.g`
   - Je zde již podpora pro AST
2. Definujte token `KOREN`
   - V sekci `tokens`
3. Explicitně vytvořte AST podstromy v `helloWorld`
   - Např. `AHOJ SVETE` přepište na `^(KOREN AHOJ SVETE)`

Co s AST?

- Vypsat (viz `main.c`)
  - `puts((char *)parseTree.tree->`
    `toStringTree(parseTree.tree)->chars);`
- Vytvořit novou stromovou gramatiku
  - Čte AST
  - Nemusí se starat o původní jazyk
  - Provádí akce na podstromech
    - Interpretuje je
    - Překládá
    - Vytváří graf toku řízení
    - ...
  - `tree grammar JmenoAST;` v hlavičce
  - `tokenVocab = JmenoGramatiky;` v `options`
  - Pravidla podle generovaných: `^(BINARY_OP '+'e1 e2)`
  - Na pravé straně může být akce:
    `{ printf("\%s + \%s\n", $e1.text->chars,`
    `$e2.text->chars); }`

Úkol

Napište stromovou gramatiku pro `Parser.g`

1. Vytvořte nový soubor ASTParser.g (nebo použijte ukázkový)
2. Zapište všechny možné AST stromy, které generujete v `Parser.g`
   - Můžete strukturovat za použití pravidel a rekurze
3. A vypište si ke každému pravidlu jeho jednotlivé části
   - `p : X { puts($X.text->chars); } ;`