



Tvorba překladače zvoleného jazyka

Semestrální práce z předmětu KIV/FJP

Michal Linha
mlinha@students.zcu.cz

4. ledna 2020

Obsah

1	Zadání	1
2	Jazyk	4
2.1	Zvolené vlastnosti jazyka	4
2.2	Gramatika	4
2.3	Omezení jazyka	7
2.4	Konstrukce jazyka	7
2.4.1	Deklarace a inicializace proměnných a konstant	7
2.4.2	Podmínky	7
2.4.3	Deklarace funkcí	8
2.4.4	Deklarace procedur	8
2.4.5	Volání funkcí a procedur	8
2.4.6	Cyklus	9
3	Implementace	10
3.1	Projekt	10
3.2	Lexikální analyzátor	10
3.3	Rekurzivní sestup	11
3.3.1	Syntaktická analýza	11
3.3.2	Sémantická analýza	11
3.3.3	Generování kódu	12
4	Uživatelská dokumentace	13
5	Závěr	14

Kapitola 1

Zadání

Cílem práce bude vytvoření překladače zvoleného jazyka. Je možné inspirovat se jazykem PL/0, vybrat si podmnožinu nějakého existujícího jazyka nebo si navrhnout jazyk zcela vlastní. Dále je také potřeba zvolit si pro jakou architekturu bude jazyk překládán (doporučeny jsou instrukce PL/0, ale je možné zvolit jakoukoliv instrukční sadu pro kterou budete mít interpret).

Jazyk musí mít minimálně následující konstrukce:

- definice celočíselných proměnných
- definice celočíselných konstant
- přiřazení
- základní aritmetiku a logiku (+, -, *, /, AND, OR, negace a závorky, operátory pro porovnání čísel)
- cyklus (libovolný)
- jednoduchou podmínku (if bez else)
- definice podprogramu (procedura, funkce, metoda) a jeho volání

Překladač který bude umět tyto základní věci bude hodnocen deseti body. Další body (alespoň do minimálních 20) je možné získat na základě rozšíření,

jsou rozděleny do dvou skupin, jednodušší za jeden bod a složitější za dva až tři body. Další rozšíření je možno doplnit po konzultaci, s ohodnocením podle odhadnuté náročnosti.

Jednoduchá rozšíření (1 bod):

- každý další typ cyklu (for, do .. while, while .. do, repeat .. until, foreach pro pole)
- else větev
- datový typ boolean a logické operace s ním
- datový typ real (s celočíselnými instrukcemi)
- datový typ string (s operátory pro spojování řetězců)
- rozvětvená podmínka (switch, case)
- násobné přiřazení ($a = b = c = d = 3;$)
- podmíněné přiřazení / ternární operátor ($min = (a < b) ? a : b;$)
- paralelní přiřazení ($a, b, c, d = 1, 2, 3, 4;$)
- příkazy pro vstup a výstup (read, write - potřebuje vhodné instrukce které bude možné využít)

složitější rozšíření (2 body):

- příkaz GOTO (pozor na vzdálené skoky)
- datový typ ratio (s celočíselnými instrukcemi)
- složený datový typ (Record)
- pole a práce s jeho prvky
- operátor pro porovnání řetězců
- parametry předávané hodnotou
- návratová hodnota podprogramu

- objekty bez polymorfismu
- anonymní vnitřní funkce (lambda výrazy)

Rozšíření vyžadující složitější instrukční sadu než má PL/0 (3 body):

- dynamicky přiřazovaná paměť - práce s ukazateli
- parametry předávané odkazem
- objektové konstrukce s polymorfním chováním
- instanceof operátor
- anonymní vnitřní funkce (lambda výrazy) které lze předat jako parametr
- mechanismus zpracování výjimek

Kapitola 2

Jazyk

Zvolený jazyk byl nazván SchoolLanguage a jeho struktura je velmi podobná programovacímu jazyku C. Všechna klíčová slova jsou oproti jazyku C přeložena do češtiny. Používá pouze dva typy a to int a boolean, tedy číslo a logicky.

2.1 Zvolené vlastnosti jazyka

Kromě základních vlastností ze zadání, obsahuje jazyk následující rozšíření:

- parametry předávané hodnotou
- else větev
- datový typ boolean a logické operace s ním
- návratová hodnota podprogramu

2.2 Gramatika

Na obrázku 2.1 a 2.2 je zobrazena gramatika jazyka.

```

program >>      globPromenne funkceAProcedury
globPromenne >> globPromenne modifikator typ IDENTIFIKATOR = matVyr az; |
globPromenna modifikator typ IDENTIFIKATOR = logVyr az; | zadne
lokPromenne >> lokPromenne typ IDENTIFIKATOR = matVyr az; |
lokPromenne typ IDENTIFIKATOR = logVyr az; | zadne
modifikator >> konst | zadne
funkce >>      funkce funkce typ IDENTIFIKATOR(parametry) {
vnitrekFunkce } | zadne
procedury >>   procedury procedura IDENTIFIKATOR(parametry) {
vnitrekProcedury } | zadne
parametry >> parametry, parametr | parametr | zadne
parametr >>   typ IDENTIFIKATOR
vnitrekFunkce >> lokPromenna viceAkci vracHodnoty
vnitrekProcedury >> lokPromenna viceAkci
akce >>       cyklus | rozhodnuti | vyraz | volaniFunkce | vraceniHodnoty | zastaveni |
IDENTIFIKATOR = logVyr az; | IDENTIFIKATOR = matVyr az;
viceAkci >>   viceAkci akce | zadne
cyklus >>    zatimco(sloz_podminka) { viceAkci }
rozhodnuti >> pokud(sloz_podminka) { vice_akci } | pokud(sloz_podminka) { vice_akci }
               pokud ne { vice_akci }
volaniFunkce >> IDENTIFIKATOR(vstupni_hodnoty);
vstupHodnoty >> vstupni_hodnoty, IDENTIFIKATOR | vstupni_hodnoty, "hodnota" |
IDENTIFIKATOR | "hodnota" | zadne
vracHodnoty >> vrat "hodnota"; | vrat IDENTIFIKATOR;
zastaveni >>  zastav;

```

Obrázek 2.1: Gramatika část 1.

```

typ >>      cislo | logicky
podmOperator >> > | < | <= | >= | == | !=
matVyras >> term | matVyras2
matVyras2 >> + term matVyras2 | - term matVyras2 | zadne
term >>      faktor term2
term2 >>     * faktor term2 | / faktor term2 | zadne
faktor >>    (matVyras) | IDENTIFIKATOR | volaniFunkce | "hodnota"
logVyras >>  "hodnota" | volaniFunkce | IDENTIFIKATOR
slozPodm >>  podmTerm slozPodm2
slozPodm2 >> || podmTerm slozPodm2 | zadne
podmTerm >>  podmFaktor podmTerm2
podmTerm2 >> && podmFaktor term2
podmFaktor >> (slozPodm) | !(podmFaktor) | IDENTIFIKATOR podmOperator
IDENTIFIKATOR | IDENTIFIKATOR podmOperator "hodnota" |
"hodnota" | "hodnota" podmOperator "hodnota" |
"hodnota" podmOperator IDENTIFIKATOR

```

Obrázek 2.2: Gramatika část 2.

2.3 Omezení jazyka

Jazyk obsahuje pouze jeden typ cyklu a to cyklus while. Návrat z funkce je možné uskutečnit pouze na konci těla funkce, tedy jako poslední příkaz, nelze ho například vložit do podmínky. Jako návratovou hodnotu lze použít pouze hodnotu nebo proměnnou. Jazyk obsahuje jak funkce, tak procedury. Při deklaraci proměnné je nutné jí rovnou inicializovat na základní hodnotu. Všechny deklarace je možné provádět jen na začátku programu, jako globální, nebo na začátku funkce či procedury, jako lokální. Deklarace proměnných na jiných místech není možná. Všechny proměnné, procedury a funkce musí být před použitím deklarovány. Tělo programu, které je na konci všech deklarací, umožňuje pouze volání procedury, kterou lze považovat za ekvivalent funkce main jazyka C. Tato procedura by měla být deklarována jako poslední, aby měla přístup ke všem ostatním funkcím či procedurám. V podmínkách nelze volat funkce.

2.4 Kontrukce jazyka

2.4.1 Deklarace a inicializace proměnných a konstant

```
cislo a = 5 * (8 - 6);  
konst cislo b = 5;  
logicky l = pravda;
```

2.4.2 Podmínky

```
pokud(o == 10 || o == 15) {  
    .    // telo  
    .  
    o = 15 + 15;    // telo  
    .    // telo  
    .  
}
```

2.4.3 Deklarace funkcí

```
funkce cislo a(cislo o, cislo p) {  
    cislo a = 0;  // lok promenne  
    cislo b = 5;  // lok promenne  
    .  // telo  
    .  
    b = a * b * p * o; // telo  
    .  // telo  
    .  
  
    vrat b; // navratov hodnota  
}
```

2.4.4 Deklarace procedur

```
procedura b() {  
    cislo a = 0;  // lok promenne  
    cislo b = 5;  // lok promenne  
    .  // telo  
    .  
    b = a * b; // telo  
    .  // telo  
    .  
}
```

2.4.5 Volání funkcí a procedur

```
b();  
a = a(5, 6);
```

2.4.6 Cyklus

```
zatimco(o < 10) {  
    .    // telo  
    .  
    o = o + 1;    // telo  
    .    // telo  
    .  
}
```

Kapitola 3

Implementace

3.1 Projekt

Zdrojové kódy projektu jsou strukturovány do několika balíků. Balík `app` obsahuje třídu `Main`, která se stará o spuštění překladače. V balíku `analyzer` jsou pak uloženy analyzátory.

3.2 Lexikální analyzátor

Lexikární analyzátor je uložen v balíku `lex`. Jeho hlavní implementace je provedena ve třídě `Lexer`, kde dochází k načítání znaků a jejich zpracovávání konečnými automaty. Automaty jsou implementovány v balíku `fsm`. `Lexer` také zjišťuje, jestli načtený token nereprezentuje identifikátor, což zjišťuje pomocí porovnání načteného stringu s klíčovými slovy. Pro každé klíčové slovo je také vytvořen automat. Třída `LexicalAnalyzer` spouští lexikální analýzu. Výstupem syntaktického analyzátoru jsou tokeny, které zároveň obsahují hodnoty pokud se jedná hodnoty, nebo názvy které jednotlivé identifikátory reprezentují.

3.3 Rekurzivní sestup

Po lexikální analýze se používá rekurzivní sestup a to tak, že v jednom průchodu rekurzivního sestupu dochází k syntaktické i sémantické analýze a zároveň také ke generování kódu. Rekurzivní sestup je implementován v balíku synsemgen ve třídě RecursiveDescentParser. Pro každou levou stranu gramatiky byla vytvořena jedna metoda, která jí zpracovává. A metoda getNextSymbol(), která načte další token ze vstupu.

3.3.1 Syntaktická analýza

Pro potřeby syntaktické analýzy je ve třídě RecursiveDescentParser implementována metoda verify(), která zkontroluje, zda vstupní token skutečně odpovídá očekávané hodnotě. Samotná syntaktická analýza je prováděna standardně pomocí rekurzivního sestupu, kdy se podle pravých stran kontroluje správnost posloupnosti tokenů. Při nalezení chyby se vypíše chybová hláška a analýza pokračuje.

3.3.2 Sémantická analýza

Jak již bylo řečeno výše, sémantická analýza probíhá souběžně s analýzou lexikální v jednom průchodu rekurzivního sestupu. Během sémantické analýzy je jsou do zásobníku ukládány tabulky symbolů. Globální proměnné jsou uloženy v první tabulce symbolů, druhá tabulka symbolů je do zásobníku přidána při vstupu do funkce nebo procedury. Po opuštění dané funkce se druhá tabulka symbolů vyhodí ze zásobníku. Tabulka symbolů je vytvořena pomocí třídy SymbolTable a obsahuje seznam záznamů. Záznam do tabulky symbolů je reprezentován třídou SymbolTableEntry, která uchovává pozici záznamu v paměti, jméno, tedy identifikátor, typ záznamu, např. funkce, datový typ záznamu, úroveň záznamu, informaci, zda se jedná o konstantu a seznam parametrů, pokud se jedná o funkci či proceduru.

Na konci třídy RecursiveDescentParser jsou implementovány metody pro prohledávání tabulek symbolů a pro kontrolu správnosti typů.

3.3.3 Generování kódu

Generování kódu je opět prováděno během rekurzivního sestupu. Do pomocného souboru se ukládají jednotlivé příkazy pro PL/0 a poté jsou data v souboru seřazena podle čísel příkazů a vypsána do výstupního souboru. Důvod řazení je ten, že u některých pravidel dochází k zapsání až po dokončení průchodu nadřazeného pravidla. Například u při procházení cyklu a nalezení akce zastavení cyklu ještě není známo, jakým příkazem cyklus skončí. Číslo řádky příkazu se tedy vyhradí příkazu pro zastavení a generování příkazů pokračuje. Po dokončení procházení cyklu dojde k vytvoření příkazu pro zastavení s vyhrazeným číslem.

Kapitola 4

Uživatelská dokumentace

Spustitelný soubor se ze zdrojových souborů vytvoří příkazem `ant distjar` spuštěným z příkazové řádky otevřené ve složce se složkou `src` a souborem `build.xml`. Překladač se spouští příkazem `java -jar sl.jar "název-souboru"`. Program poté provede lexikální analýzu a všechny tokeny vypíše na obrazovku, poté provede syntaktickou a sémantickou analýzu a generování kódu a případné chyby vypíše na obrazovku. Výsledný soubor se bude jmenovat jako název, který je před první tečkou v názvu souboru + přípona `.sl`, nebo celý název, pokud tečku neobsahuje + přípona `.sl`.

Kapitola 5

Závěr

Pro ukázkové programy funguje překladač správně avšak při překladu generuje velké množství instrukcí, což je částečně dáno jeho jednoprůchodovou implementací. Jako příklad uvedu příkaz `INT 0 1`, který se použije při každé definici proměnné. Pokud je tedy proměnných velké množství dojde k vygenerování velkého množství těchto příkazů. V tomto případě by ale bylo možné použít stejný způsob, který byl popsán pro příkaz `zastav` v cyklu. Tedy došlo by k vyhrazení čísla příkazu a spočtení všech proměnných a poté až k vytvoření příkazu `INT`.