FAKULTA APLIKOVANÝCH VĚD
ZÁPADOČESKÉ UNIVERZITY
V PLZNI

KATEDRA INFORMATIKY
A VÝPOČETNÍ TECHNIKY

Semestrální práce

Překladač jazyka Ligma

Milan Janoch – janochmi@students.zcu.cz
Jakub Pavlíček – jpvlck@students.zcu.cz

PLZEŇ

26. 12. 2024

Obsah

Zadání 1

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 (+, -, *, /, &&, ||, !, (), ==, !=, <, >, <=, >=)
- 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)
- 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;
- ternární operátor: min = (a < b) ? a : b;
- paralelní přiřazení {a, b, c, d} = {1, 2, 3, 4};
- příkazy pro vstup a výstup (read a write)

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

Vlastní interpret (řádkový, složitý alespoň jako rozšířená PL/0) je za 6 bodů.

Zohledňují se i další věci, které mohou pozitivně nebo negativně ovlivnit bodování:

- testování: tvorba rozumné automatické testovací sady +3 body
- kvalita dokumentace: -x bodů až +2 body podle kvality a prohřešků
- využití GITu: -x bodů až +2 body podle důslednosti a struktury příspěvků
- kvalita zdrojového textu: -x bodů až +2 body podle obecně známých pravidel

Návrh jazyka 2

2.1 Zvolená rozšíření jazyka Ligma

- cyklus for
- cyklus do-while
- cyklus repeat-until
- datový typ boolean a logické operace s ním
- else větev
- násobné přiřazení: a = b = c = d = 3;
- parametry předávané hodnotou
- návratová hodnota podprogramu

2.2 Omezení jazyka

- Při deklaraci proměnné je třeba vždy nastavit hodnotu
- V hlavičce for cyklu se musí deklarovat proměnná
- Podporované datové typy: int, boolean
- Identifikátor nesmí obsahovat speciální znaky (kromě '_') a začínat číslem
- Funkce musí být definovány až pod samotnými příkazy
- Funkce lze definovat pouze v globálním rozsahu
- Funkce musí vždy vracet hodnotu (return je vždy poslední příkaz funkce)

2.3 Konstrukce jazyka

2.3.1 Povinné

Definice celočíselných proměnných

```
int a = 5;
```

Definice celočíselných konstant

```
const int a = 5;
```

Přiřazení

```
int a = 1;
a = 5;
```

Základní aritmetika a logika

Aritmetika:

```
int a = 5 + 1;
int b = 5 - 1;
int c = 5 / 1;
int d = 5 * 1;
int e = -1;
int f = +1;
int g = 5 % 5;
```

Logika:

```
boolean a = 1 < 5;
boolean b = 1 <= 5;
boolean c = 1 >= 5;
boolean d = 1 > 5;
boolean e = 1 == 5;
boolean f = 1 != 5;
boolean g = true && true;
boolean h = true || false;
```

While cyklus

```
int a = 0;

while (a < 5) {
    a = a + 1;
}
```

Podmínka if (bez else)

```
if (true) {
    ...
}
```

Definice funkce a její volání

```
int res = foo();

func int foo() {
    return 5;
}
```

2.3.2 Rozšiřující

Cyklus do-while

```
int a = 1;

do {
    a = a + 1;
} while (a < 5);
```

Cyklus repeat-until

```
int a = 1;

repeat {
    a = a + 1;
} until (a >= 5);
```

Cyklus for

Automaticky inkrementuje definovanou proměnnou v hlavičce (zde „a“) o 1. Vždy se porovnává, zda proměnná „a“ je menší než hodnota výrazu za tokenem „to“.

```
for (int a = 0 to 10) {
    ...
}
```

Else větev

```
if (false) {
    ...
} else {
    ...
}
```

Datový typ boolean a logické operace s ním

```
boolean a = true;
boolean b = false;
boolean c = a && b;
boolean d = a || b;
boolean e = !a;
boolean f = true == false;
boolean g = true != false;
```

Násobné přiřazení

Nejdříve se vyhodnocuje výraz na pravé straně.

```
int a = 1;
int b = 2;

a = b = 5;
```

Parametry předávané hodnotou

```
int res = foo(3, 4);

func int foo(int a, int b) {
    return a * b;
}
```

Návratová hodnota podprogramu

```
int res = foo(1);

func int foo(int a) {
    return a - 1;
}
```

2.3.3 Vlastní

Mocnina

```
int a = 2 ^ 5;
```

Komentáře

```
// Line comment

/*
    Multiline comment
*/
```

Implementace 3

K vytvoření překladače jazyka Ligma jsme použili nástroj ANTLR a programovací jazyk Java. Zdrojové soubory se nachází v adresáři `/ligma/src/`. Při implementaci byl používán nástroj Git pro verzování kódu a správu změn. Odkaz na GitHub repozitář: `https://github.com/M1LNES/FJP-semester-work`.

3.1 Struktura projektu

```
src
├── main
│   ├── java
│   │   └── ligma
│   │       ├── enums ..... výčtové typy (PL/0 instrukce, datové typy, ...)
│   │       ├── exception ..... vlastní definované výjimky
│   │       ├── generated ..... soubory vygenerované ANTLR za překladu
│   │       ├── generator ..... generátory PL/0 instrukcí
│   │       ├── ir ..... vnitřní reprezentace jazyka
│   │       ├── listener ..... listenery pro lexikální/syntaktickou analýzu
│   │       ├── table ..... tabulka symbolů
│   │       ├── visitor ..... třídy pro průchod derivačního stromu
│   │       └── App.java ..... vstupní bod programu
│   └── resources
│       ├── output. ..... výstupy programů (PL/0 instrukce)
│       └── programs. ..... ukázkové programy
└── test
    ├── java. ..... testy
    └── resources. ..... testovací soubory
```

3.2 Lexikální a syntaktická analýza

Gramatika jazyka, definovaná v souboru `Ligma.g4`, popisuje lexikální a syntaktická pravidla pro překladač. Obsahuje pravidla pro klíčová slova, datové typy, literály, operátory a struktury, jako jsou cykly, podmínky, funkce a přiřazení hodnot. Zmíněný soubor tvoří základ pro generování derivačního stromu, který je dále využíván v překladači.

Lexikální část gramatiky (`lexer rules`) rozpoznává jednotlivé tokeny, zatímco syntaktická část (`parser rules`) určuje hierarchii a strukturu příkazů. Gramatika je navržena tak, aby umožnila parsování komplexních výrazů včetně operátorů s různou prioritou.

Pro kontrolu chyb během lexikální a syntaktické analýzy jsme vytvořili vlastní listenery v balíčku `listener`. Ty zachycují chyby a vytváří výjimky, které obsahují informace o chybě v kódu.

3.3 Sémantická analýza

Sémantická analýza je prováděna v průběhu průchodu derivačního stromu. V rámci sémantické analýzy jsou kontrolována pravidla jazyka, jako je např. kontrola deklarace proměnných, kontrola typů operandů, atd. V případě nalezení chyby je vyhozena výjimka, která obsahuje informace o chybě v kódu. Během sémantické analýzy je vytvářena vnitřní reprezentace jazyka, která je následně použita pro generování PL/0 instrukcí.

Pro sémantickou analýzu slouží třídy v balíčku `visitor`. Tyto třídy implementují rozhraní `LigmaBaseVisitor`, které obsahuje metody pro každý typ uzlu v derivačním stromu. Uvnitř těchto metod je prováděna sémantická analýza.

3.3.1 Tabulka symbolů

Během sémantické analýzy se také vytváří tabulka symbolů, která obsahuje informace o deklarovaných proměnných a funkcích. Tabulka symbolů je implementována jako zásobník rozsahů, kde každý blok (např. funkce, cyklus) má svůj vlastní rozsah (scope). Každý rozsah pak obsahuje mapu identifikátorů a jejich příslušných informací (datový typ, adresa, ...). Díky tomu je možné jednoduše získat informace o proměnných a funkcích v rámci daného bloku.

3.4 Generování PL/0 instrukcí

Generování PL/0 instrukcí je prováděno pomocí vnitřní reprezentace jazyka, která byla dříve vytvářena sémantickou analýzou. Vnitřní reprezentace obsahuje informace o jednotlivých částech programu, které jsou následně převedeny do instrukcí jazyka PL/0. Generování instrukcí je prováděno v balíčku `generator`, přičemž výsledné instrukce jsou ukládány do souboru.

3.4.1 Generování funkcí

Jednotlivé funkce jsou generovány jen v případě, že je daná funkce volána. Instrukce funkce se tak po jejím zavolání nachází „uvnitř" samotného volání, což lze chápat jako rozvinutí makra v daném místě. Přičemž pomocí pomocné mapy funkcí je zařízeno, že během generování volání funkce se nastaví správná adresa začátku funkce. Pokud je ale stejná funkce volána vícekrát, je generována pouze jednou.

3.4.2 Generování mocniny

Generování mocniny jsme vyřešili pomocí „pomocné" funkce, která v sobě obsahuje cyklus, který násobí hodnotu základu tolikrát, kolik je hodnota exponentu. Díky tomu byla zajištěna izolace pomocných proměnných a zároveň bylo dosaženo správného výsledku. Ve výrazu mocniny se tak můžou nacházet jak hodnoty, proměnné, tak různě složité výrazy.

3.4.3 Výsledná implementace

Výslednou implementaci lze popsat následujícím pseudokódem:

Algorithm 1 Překladač jazyka Ligma

Fáze 1: Lexikální analýza
Vstup: Zdrojový kód v jazyce Ligma
Výstup: Seznam tokenů
for každý znak v kódu **do**
    Pokud neplatný znak, **pak** generuj chybu, **jinak** vytvoř token
end for

Fáze 2: Syntaktická analýza
Vstup: Seznam tokenů
Výstup: Derivační strom
for každý token **do**
    Pokud token neodpovídá syntaxi, **pak** generuj chybu, **jinak** vytvoř uzel
end for

Fáze 3: Sémantická analýza
Vstup: Derivační strom
Výstup: Vnitřní reprezentace jazyka a kontrola sémantiky/typů
for každý uzel ve stromě **do**
    Pokud chyba sémantiky/typu, **pak** generuj chybu, **jinak** vytvoř vnitřní reprezentaci
end for

Fáze 4: Generování PL/0 instrukcí
Vstup: Vnitřní reprezentace jazyka
Výstup: Soubor s PL/0 instrukcemi
for každý objekt **do**
    Pokud chyba při generování, **pak** generuj chybu, **jinak** vygeneruj instrukce
end for

Testování 4

Projekt obsahuje sadu automatických testů, které testují lexikální, syntaktickou a sémantickou analýzu. Všechny testy (celkem **242**) jsou automaticky spouštěné při vytváření výsledného `.jar` souboru pomocí skriptu `run.sh` (případně `run.bat` na Windows). Testování bylo prováděno na operačních systémech Windows a macOS. Testy se nachází v adresáři `/src/test` (dále uvažujme tento soubor).

Kód překladač zahrnuje logování důležitých informací o průběhu sémantické analýzy a generování instrukcí. Díky důkladnému logování jsme bylo schopni snadno identifikovat chyby a problémy v kódu.

4.1 Lexikální analýza

Testy pro lexikální analýzu spouští třída `ExpressionLexicalTest`. Obsahuje celkem **12** negativních testů, které validují správnou funkčnost lexikální analýzy. Testovací soubory se nachází v adresáři `resources/lexical`.

4.2 Syntaktická analýza

Testy pro syntaktickou analýzu spouští třída `ExpressionSyntaxTest`. Obsahuje celkem **148** testů (pozitivních + negativních). Testovací soubory se nachází v adresáři `resources/syntax`. Důraz u negativních testů byl kladen zejména na provádění neplatných operací (např. chybějící operandy/operátory) či používání neplatných instrukcí.

4.3 Sémantická analýza

Testy pro sémantickou analýzu spouští třídy `ExpressionSemanticTest` a `FunctionSemanticTest`. Tyto třídy testují sémantiku a zároveň kontrolují typové chyby výrazů a funkcí. Obsahují celkem **72** testů (pozitivních + negativních). Testovací soubory se nachází v adresáři `resources/semantic`.

4.4 Generování PL/0 instrukcí

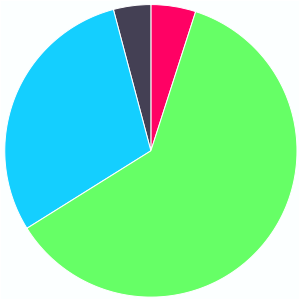
Testy pro generování PL/0 instrukcí spouští třída `ExpressionGeneratorTest`. Tato třída vezme ukázkové soubory (celkem **10**) uvedené v adresáři `/programs` a vygeneruje z nich PL/0 instrukce do stejnojmenných souborů v adresáři `/output`. Tímto je zaručeno, že při spuštění skriptu `run.sh` (případně `run.bat` na Windows) budou opětovně vygenerovány PL/0 instrukce pro všechny ukázkové programy. Zároveň je zaručeno, že nedojde k žádné chybě při generování instrukcí z ukázkových souborů.

Ukázkové zdrojové kódy přeložené do instrukční sady PL/0 se nachází v adresáři `/src/main/resources` – adresář `/programs` obsahuje zdrojové kódy jazyka Ligma a adresář `/output` vygenerované `PL/0` instrukce.

4.5 Zhodnocení

Výsledná implementace byla otestována celkem **242** testy, které pokrývají všechny části jazyka Ligma. Díky rozsáhlé sadě testů bylo možné identifikovat a odstranit chyby v kódu. Výsledky testování jsou zobrazeny v tabulce a grafu na obr. 4.1.

TESTŮ CELKEM	242
LEXIKÁLNÍ TESTY	12
SYNTAKTICKÉ TESTY	148
SÉMANTICKÉ TESTY	72
GENERÁTOR TESTY	10



Obrázek 4.1: Výsledný počet testů

Uživatelská dokumentace

5

5.1 Prerekvizity

Pro úspěšné přeložení je vyžadováno:

- Java verze 23 (kvůli podpoře Markdown komentářů)
- Maven verze 3.9.9

5.2 Instalace a spouštění

1. **Otevření adresáře s aplikací**
    - Otevřete adresář `ligma`: `cd ligma`
2. **Instalace a spuštění**
    - Pro **Windows**: `run.bat`
    - Pro **Linux** a **macOS**: `sh run.sh`

Skript spustí příkaz `mvn clean install`, který stáhne veškeré potřebné knihovny, přeloží projekt (vytvoří adresář `/target` s přeloženými soubory), následně spustí testy a vytvoří finální spustitelný soubor `ligma.jar` jak v aktuálním adresáři, tak v adresáři `/target`.

Nakonec se spustí ukázkový program a vygeneruje se výstupní soubor s PL/0 instrukcemi. Ve skriptu je možné upravit cestu k ukázkovému programu, který se má přeložit. Díky spuštěným testům je však zaručeno, že se opětovně vygenerují PL/0 instrukce pro všechny ukázkové programy.

Program lze spustit i bez použití skriptu, a to příkazem:

```
java -jar ligma.jar <input-file> <output-file>
```

kde `<input-file>` je cesta k souboru se zdrojovým kódem jazyka Ligma a `<output-file>` je výsledný soubor s PL/0 instrukcemi.

Závěr 6

V rámci semestrální práce byl úspěšně vytvořen funkční překladač jazyka Ligma do instrukcí PL/0.

Navrhli jsme si vlastní jazyk Ligma, který je popsán gramatikou `Ligma.g4`. Z gramatiky je pak nástrojem ANTLR vygenerován derivační strom, který je možné dále procházet či zpracovávat. Při průchodu derivačního stromu jsou kontrolována pravidla jazyka a zároveň je vytvářena vnitřní reprezentace jazyka v podobě objektů. Z vnitřní reprezentace jsou následně generovány instrukce jazyka PL/0. Celý průběh překladu je podrobně logován do konzole.

Pro testování vygenerovaných instrukcí ukázkových programů byl využit online interpret `https://home.zcu.cz/~lipka/fjp/pl0/`. Výsledný překladač byl otestován velkou sadou testů, které pokrývají všechny části jazyka Ligma.

Seznam obrázků