Západočeská univerzita v Plzni

Formální jazyky a překladače

KIV/FJP

„Java“ do PL/0

Čarnogurský Jan, Vojtěch Danišík
A19N0025P, A19N0028P
cagy@students.zcu.cz, danisik@students.zcu.cz

3. ledna 2020

Obsah

1 Zadání

1.1 Tvorba překladače zvoleného jazyka

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.

Jednoduché rozšíření (1 bod):

- každý další typ cyklu (for, do .. while, while .. do, repeat .. unitl, 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 řětezců)

- 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

1.2 Bodované náležitosti

Kromě toho že by program měl fungovat se zohledňují i další věci, které mohou pozitivně nebo negativně ovlivnit bodování:

- testování - tvorba rozumné automatické testovací sady +3 body (pro inspiraci hledejte test suit pro LLVM nebo se podívejte na Plum Hall testy, ale jde skutečně jen o inspiraci, stačí výrazně jednodušší řešení). Užitečné a s tručné povídání na dané téma najdete také tady .
- Kvalita dokumentace -x bodů až +2 body podle kvality a prohřešků (vynechaná gramatika, nesrozumitelné věty, příliš chyb a překlepů, bitmapové obrázky pro diagramy s kompresními artefakty, ...).
- Vedení projektu v GITu -x bodů až +2 body podle důslednosti a struktury příspěvků.
- Kvalita zdrojového textu -x bodů až +2 body podle obecně známých pravidel ze ZSWI, PPA a podobně (magická čísla, struktura programu a dekompozice problému, božské třídy a metody, ...)

2 Návrh jazyka

2.1 Zvolené konstrukce

Pro naší semestrální práci jsme zvolili následující konstrukce:

- minimální konstrukce
- zbylé cykly (bez foreach)
- else větev
- datový typ boolean a logické operace s ním
- rozvětvená podmínka (switch, case)
- násobné přiřazení (a = b = c = d = 3;)
- parametry předávané hodnotou
- návratová hodnota podprogramu

2.2 Struktura jazyka

Náš jazyk připomíná Javu, z které vychází. Popis námi zvolené gramatiky je v souboru `SimpleJava.g4`, který se nachází ve složce se zdrojovými soubory. Pro demonstraci funkčnosti gramatiky jsou v sekci 2.2.2 předvedeny všechny možné konstrukce. Tělo programu musí být umístěno ve složených závorkách.

2.2.1 Omezení jazyka

- jazyk používá striktní typování
- při deklaraci proměnné je vždy nutné nastavit její hodnotu
- z metody nelze vyskočit dřív než před koncem jejího bloku
- void metody neobsahují return
- iterační proměnná `for` cyklu nesmí být deklarována. Její deklarace je provedena v hlavičce cyklu
- zkrácené konstrukce (cislo++, ..) nejsou podporovány

2.2.2 Ukázky jednotlivých konstrukcí

Detailnější ukázky programů jsou uloženy ve složce *testFiles*. Blok programu musí být vždy ve složených závorkách.

Deklarace proměnných

```
{
   const int a = 10;
   boolean b = true;
   int c = d = a;
   int e = volaniMetodyNavratovaHodnota(1, false);
   int f = -a;
   int g = -(a + b);
}
```

Deklarace metod

```
{
   int function metoda1(int a, boolean b)
   {
      return a + b;
   }

   // pokud je metoda typu void neobsahuje return
   void function metoda2(){}
}
```

Cykly

```
{
   for(a = 0 ... 10) // odpovida 0 <= 10
   {
      // telo
   }

   while(true)
   {
      // telo
   }

   do
   {
      // telo
   }while(true)

   repeat
   {
      // telo
   }until(false)
}
```

If else

```
{
   if (a < b)
   {
      // telo
   }
   else
   {
      // telo
   }
}
```

Rozvětvená podmínka

```
{
   switch(cislo) {
      case 1 :
      {
         // telo
      }
      case 2 :
      {
         // telo
      }
      default :
      {
         // telo
      }
   }
}
```

Volání metod

```
{
   volaniMetody(1, true);

   int a = volaniMetodyNavratovaHodnota(1, false);
}
```

3 Implementace

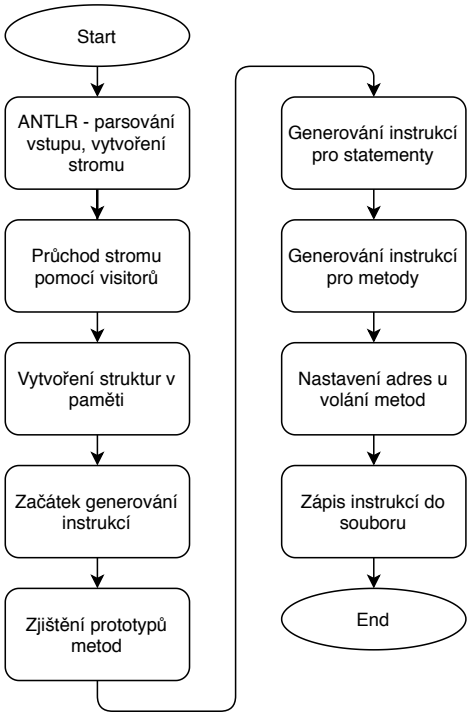
3.1 Struktura projektu

Projekt obsahuje dva hlavní balíky `generate` a `compilator`. Balík `generate` obsahuje třídy vygenerované pomocí ANTLR. Balík `compilator` obsahuje následující strukturu:

- `compilerPart` - balík, který obsahuje třídy pro generování instrukcí
- `enums` - balík, obsahuje všechny výčtové typy
- `error` - balík s error třídami
- `object` - balík se společnými objekty
- `visitor` - balík, který obsahuje visitory pro průchod derivačního stromu
- `Compilator` - třída, která obsluhuje proces aplikace
- `ErrorHandler` - třída pro obsluhu chyb při generování instrukcí
- `InstructionGenerator` - třída pro spuštění generování instrukcí
- `LexerParserErrorListener` - třída pro obsluhu chyb při parsování vstupního souboru

3.2 Princip funkčnosti

Zpracování je rozděleno do několika kroků. Průběh zpracování je znázorněn na obrázku 1. O průběh zpracování se stará třída `Compilator`, která jednotlivé bloky spouští. Třída `Compilator` je volána z třídy `Main`, a jako parametry jsou předány cesty k vstupnímu a výstupnímu souboru. Kód je generován na dva průchody. Prvním průchodem se uloží derivační strom do paměti a druhým průchodem se z uložených struktur v paměti vygenerují instrukce.



Obrázek 1: Průběh zpracování

3.2.1 Parsování vstupního souboru

Parsování vstupu, vytvoření derivačního stromu, a jeho zpracování je řešeno pomocí ANTLR. Gramatika je popsána v souboru `SimpleJava.g4`. Soubory vytvořené pomocí ANTLR jsou vygenerovány do balíku `generate`.

3.2.2 Průchod stromu

Průchod stromu je řešen pomocí `visitorů`. Jednotlivé `visitory` jsou uloženy v balíku `compilator/visitor`. `Visitory` jsou rozděleny do tříd tak, aby odpovídaly hlavním uzlům v derivačním stromu. `Visitory` slouží k průchodu stromu, a k jeho následnému uložení do paměti.

3.2.3 Struktury v paměti

Struktury v paměti slouží pro uložení derivačního stromu do paměti, aby s ním bylo možné dále pracovat. Názvy struktur odpovídají hlavním položkám v gramatice. Výsledkem průchodu je objekt `Program`, který představuje kořen derivačního stromu a obsahuje odkazy na další objekty (uzly), které obsahují další odkazy.

3.2.4 Začátek generování instrukcí

V této částí je program přepnut do zpracování uložených struktur v paměti. Je zavolána třída `InstructionGenerator` a je jí předán objekt `Program`. Výsledkem generování instrukcí je list vygenerovaných instrukcí.

3.2.5 Prototypy metod

Před samotným generováním instrukcí je nejprve nutné zjistit prototypy metod (hlavičky metod), aby při generování volání metod bylo možné určit, jestli metoda má návratovou hodnotu, popřípadě jakého je typu. Prototypy metod nepracují s parametry. Typy parametrů jsou zkontrolovány při nastavení adres u volání metod.

3.2.6 Generování instrukcí

Pro generování instrukcí je připraveno několik tříd, kde každá zpracovává různě námi uložené struktury:

- `BaseCompiler` - abstraktní třída od které jsou další třídy odděděny. Obsahuje společné proměnné
- `ProgramCompiler` - zpracovává objekt `Program`
- `BlockCompiler` - zpracovává objekt `Block`
- `BlockStatementCompiler` - zpracovává objekt `BlockStatement`, ve kterém jsou uložena veškerá těla metod. Uvnitř tohoto kompileru jsou volány subpřekladače (`ExpressionCompiler, MethodCallCompiler, Methodompiler`), dle potřeby
- `ExpressionCompiler` - zpracovává objekt `Expression`, který představuje veškeré výrazy
- `MethodCallCompiler` - zpracovává objekt `MethodCall`, který představuje volání metod
- `MethodCompiler` - zpracovává objekt `Metod`, který představuje deklaraci metod

Generování je rozděleno do dvou kroků. Prvním krokem je vygenerování instrukcí pro hlavní tělo programu, druhým krokem je vygenerování instrukcí pro metody. To způsobí, že začátek programu je vždy ve výsledných instrukcích uložen na adrese jedna a metody jsou uloženy na konci. Tato implementace umožní nadefinování metod v kódu až po jejich zavolání.

3.2.7 Nastavení adres u volání metod

Tím, že je umožněno, aby metody byly vytvořeny až po jejich zavolání, je nutné u instrukcí pro volání metod zpětně nastavit adresu, na které se volaná metoda nachází. Tyto instrukce jsou uloženy s adresou „-1“. Aby bylo možné zpětně upravit adresu, je do objektu instrukce uložen mimo jiné objekt `MethodCall`.

V cyklu se projdou veškeré instrukce a pokud se jedná o instrukci `CALL`, je z objektu instrukce získán objekt `MethodCall`. Zkontroluje se, zda název metody uložený v `MethodCall` existuje v tabulce symbolů, dále se zkontrolují zda sedí počet a typ parametrů. Pokud není cokoliv splněno je vyhozena chyba a program je ukončen. Pokud je vše splněno, je instrukci `CALL` změněna adresa podle tabulky symbolů.

3.2.8 Zápis instrukcí do souboru

Instrukce se zapíšou do souboru, který byl zadán jako vstupní parametr. V cyklu se projde list instrukcí, který vrátil objekt `InstructionGenerator`. Pokud výstupní soubor již existuje, je vždy přepsán novými instrukcemi.

3.3 Viditelnost proměnných

Proměnné nadefinované v hlavním těle programu jsou viditelné v celém kódu (jsou celou dobu držené v tabulce symbolů) a je pro ně zvýšen obsah zásobníku. Pro proměnné nadefinované mimo hlavní blok programu (těla metod, těla smyček, ...), je v momentě vykonávání bloku zvýšen obsah zásobníku, a jsou zahrnuty do tabulky symbolů. Po skončení bloku jsou proměnné odstraněny z tabulky symbolů, a je snížen vrchol zásobníku o jejich počet. Tím je zajištěno, že nejsou dále viditelné.

3.4 Volání metod s parametry

Volání metod je řešeno v `MethodCallCompiler`. Pokud má metoda parametry, které je nutné předat, jsou před instrukcí `CALL` na vrchol zásobníku přidány hodnoty parametrů. V metodě se zvýší vrchol zásobníku o počet parametrů +3 co představuje defaultní velikost metody. Parametry metody jsou získány pomocí instrukce `LOD` se zápornou hodnotou. Záporná hodnota odpovídá počtu parametrů. Následně se instrukce `LOD` volá znovu se zápornou hodnotou změnšenou o jedna. To se opakuje dokud se nenačtou všechny parametry. Načtené hodnoty se průběžně ukládají do tabulky symbolů, aby s nimi bylo možné v metodě dále pracovat. Po návratu z volání metody, je

ještě nutné předané parametry odstranit z vrcholu zásobníku. Provede se instrukce `STO` s hodnotou -1, tolikrát, kolik bylo předáno parametrů.

3.5 Metoda s návratovou hodnotou

Volání metody je řešeno v `MethodCallCompiler`. Pokud má metoda návratovou hodnotu, je před instrukcí `CALL` zvýšen obsah zásobníku o jedna. Výsledek, který metoda vrací je umístěn za klíčovým slovem „return“. Po vyhodnocení výsledku je výsledek uložen na adresu počet proměnných +1. Tato adresa odpovídá místu, které se vytvořilo na vrcholu zásobníku jeho zvětšením.

3.6 Error kódy

Program pracuje se svými chybovými kódy. Chyby jsou spravovány pomocí třídy `ErrorHandler`, která se stará o vypsání chyby a ukončení programu s příslušným kódem. Níže jsou popsány chybové kódy s jejich popisem.

Kód	Popis
1	Proměnná, kterou se snažíte přiřadit neexistuje.
2	Pokoušíte se přepsat hodnotu u konstanty.
3	Při volání metody nesedí počet parametrů.
4	Metoda s tímto jménem již existuje.
5	Metoda, kterou se snažíte zavolat neexistuje.
6	Výsledek výrazu nesedí s očekávaným výsledkem (např. přiřazení matematického výrazu do logického).
7	Návratový typ metody nesedí s proměnnou, do které se přiřazuje.
8	Není možné kombinovat logické a matematické výrazy dohromady (např. true + 10).
9	Snaha přiřadit proměnné typu INT hodnotu BOOLEAN a naopak.
10	Switch nemůže obsahovat více defaultních bloků.
11	Proměnná s tímto názvem již je deklarována.
12	Snaha pracovat s proměnnou, která neexistuje.
13	Při zavolání metody typu void ve výrazu.
14	Typy parametrů metody a typy parametrů ve volání se neshodují.
15	Vstupní soubor neexistuje.
16	Chyba při parsování souboru ANTLR.
17	Neznámá chyba (i když je program otestovaný, může se stát, že se objeví neočekávaná chyba).
18	Chybné spuštění aplikace, nesedí parametry.
19	Chybná cesta k výstupnímu souboru.
20	Neplatné použití operátorů. Způsobeno pokud jsou v programu špatně použity operátory (a ++ +a)

4 Testování

Pro překlad nebyly vytvořeny žádné automatické testy. Testy, které jsme prováděli byly všechny manuální. Všechny námi vytvořené testovací scénáře jsou uloženy ve složce *testFiles*. Pro každý testovací scénář existuje i soubor s výslednými PL/0 instrukcemi.

5 Uživatelská dokumentace

Pro překlad je nutné mít nainstalovaný Maven. Překlad projektu se provede spuštěním příkazu:

- mvn clean install

Pozor, Maven vytvoří dva jar soubory. Aby bylo možné program spustit je nutné pracovat se souborem, který má v názvu „-jar-with-dependencies“.

Pro spuštění je nutné mít nainstalovanou Javu, program se spouští příkazem:

- java -jar <nazev JAR >.jar <vstupní soubor >␣<výstupní soubor >

Pro demonstraci je vytvořen soubor, který provede ukázkové spuštění.

- Linux : ./run.sh
- Windows : run.bat

V připadě, že nepůjde skript na linuxu spustit je nutné nastavit práva:

- chmod +x run.bat

Při špatném spuštění, je uživateli vypsána hláška s postupem, jak se program spouští.

6 Závěr

Podařilo se nám vytvořit překladač, který dokáže náš vytvořený jazyk převést do instrukční sady PL/0. Pro testování bylo vytvořeno několik scénářů, které demonstrují, že si kompilátor dokáže poradit s jednoduššími, ale i se složitějšími konstrukcemi jako je rekurzivní volání.

Semetrální práce byla vedena na adrese https://github.com/danisik/FJP.