Západočeská univerzita v Plzni
Fakulta aplikovaných věd
Katedra informatiky a výpočetní techniky

Semestrální práci

Překladač jazyka do PL/0

Martin Kantořík kantorik@students.zcu.cz
Michal Tušl tuslm@students.zcu.cz

7. ledna 2018

Obsah

1 Zadání

Cílem práce je vytvoření vlastního jazyka a překladače pro tento jazyk. Překládat jsme se rozhodli do instrukční sady PL/0. Při vytváření jazyka jsme se snažili napodobit syntaxi jazyků java a C. Od vytvořeného jazyka jsme požadovali, aby uměl následující základní elementy:

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

Dále jsme se rozhodli jazyk rožšířit o další složitější konstrukce. Mezi složitější konstrukce, které jazyk umí patří:

- další typy cyklů (for, do...while, until, repeat...until)
- else větev
- datový typ boolean a logické operace s ním
- rozvětvená podmínka (switch, case)
- vícenásobné přiřazení (a = b = c = 3;)
- podmíněné přiřazení / ternární operátor (min = (a < b) ? a : b;)
- pole a práce s jeho prvky
- parametry předávané hodnotou
- návratová hodnota podprogramu
- komentáře

- řídící příkazy (break, continue)

Za plně funkční překladač pro jazyk, které umí tyto konstrukce by mělo být uděleno 25 bodů. Další bonusový bod by mohl být za realizaci komentářů, které nebyly uvedeny v seznamu možných konstrukcí. Případně další bod za příkazy break a continue.

2 Popis řešení

2.1 Překlad

Překlad programu se skládá ze tří fází. V první fázi se najdou a zaznamenají všechny hlavičky funkcí a zároveň se zjistí potřebná velikost v paměti na parametry a návratovou hodnotu. Návratová hodnota může zabírat místo pro jednu proměnnou nebo žádnou proměnnou, protože funkce mohou vracet jen jednoduché datové typy. Místo pro parametry má velikost rovnu největšímu počtu parametrů, které mohou do některé z funkcí vcházet. Dělání zaznamenávání funkcí jako první krok, je důležité proto, aby se funkce mohli volat z jakéhokoli místa v programu. Navíc slouží k rychlé kontrole ve správnosti vytvořených funkcích.

V druhé fázi se překládá hlavní kód programu, což znamená překládání kódu mimo funkce. V této fázi se nejdříve vytvoří místo na parametry a návratovou hodnotu a posléze se již překládají jednotlivé příkazy. Pokud se narazí na volání funkce, překladač se ji pokusí najít v registrovaných funkcích. Pokud je funkce nalezena, vytvoří se kód pro volání a zároveň se volání přidá do pole určeného pro čekání přiřazení počáteční adresy funkce.

Ve třetí fázi se již překládají pouze funkce. Překládání těl funkcí probíhá stejně jako překlad hlavního kódu. Rozdíl je při překládání hlavičky funkce, kdy se funkce přidá do tabulky symbolů a automaticky se doplňují adresy ke všem jejím voláním. Pokud se objeví volání v těle funkce, tak je první krok stejný jako v předchozí fázi, ale navíc se udělá ještě průzkum, zda již funkce nebyla přiřazena a není tak možné adresu přiřadit k volání rovnou. Jako poslední krok při překládání funkce se pak validuje, zda opravdu existuje návratová hodnota, pokud ji funkce má.

Výhodou tohoto překladu je, že hlavní kód je vždy hned na začátku a následují ho definice funkcí. Nemusíme tak řešit různé odskoky uprostřed kódu kvůli přeskočení funkce.

Pokud během překladu bylo naraženo na nějakou chybu, kód pro PL0 nebude vytvořen a místo toho se vytvoří soubor s chybovým hlášením.

2.2 Nápomocné tabulky

V řešení bylo použito několik tabulek (polí) pro snadnější překlad. Jsou jimi pole pro symboly, hotový kód, volání funkcí, registrované funkce, chyby a

pole pro odskoky a zvýšení místa v zásobníku.

V tabulce symbolů jsou ukládány všechny proměnné a funkce. Jsou zde informace především o jejich typech, jménech, adresách a číslech objektu, ve kterém se nachází.

V tabulce s hotovým kódem se nachází již vygenerovaný kód v PL/0. S tímto kódem se manipuluje, jen když se potřebují doplnit adresy odskoků či volání funkcí. Nachází se zde informace o instrukci, úrovni, adrese a indexu v poli.

V tabulce pro volání funkcí jsou obsaženy všechny volání, které potřebují doplnit adresu. Volání jsou sem přidána, pokud u přidávání do tabulky s hotovým kódem mají hodnotu -1. Jsou uloženy s indexem kódu v tabulce s kódem a zároveň se k nim přidá hodnota *objectID*, podle které je volání posléze nalezeno a doplněno.

Další tabulkou jsou registrované funkce, jejich funkčnost již byla popsána v kapitole 2.1.

Předposlední tabulkou je pole s chybami. Tato tabulka slouží, jak již název napovídá, pro uchovávání chyb zachycených během překladu. Je zde popis chyby, číslo řádku a číslo znaku.

Poslední tabulka obsahuje odskoky a instrukci pro zvýšení místa v zásobníku. Tato tabulka slouží pro zachycení příkazů, které potřebují doplnit adresu nebo je její hodnota měněna. Každý záznam má informace o indexu v poli kódu a *objectID*, podle kterého jsou nalezeny a doplňovány. V případě odskoků se po doplnění adresy odskok z tabulky smaže.

2.3 Viditelnost proměnných

Viditelnost proměnných je zajištěna pomocí *objectID* informace. Objekt je u nás vše z následujících případů: cyklus, if, else, switch, funkce a ternární if. Jakmile překladač vstoupí do nového objektu je číslo *objectID* zvýšeno a následně zase zpátky sníženo, když se daný objekt opustí. Žádný objekt nemůže mít stejné číslo jako už některý objekt před ním a počáteční hodnota *objektID* je nula. Tato hodnota posléze slouží k nacházení správné proměnné či pro identifikaci správného příkazu v některé tabulce.

Tento mechanismus navíc zajišťuje jakýsi strom rodičovství u jednotlivých objektů, a proto proměnná vytvořená v rodičovském objektu je následně viděna všemi dětmi. Tato proměnná však může být v některém dítěti přetížena tím, že se v něm deklaruje nová proměnná stejného jména. Poté se již v dalších dětech přistoupí jen k této nové proměnné.

2.4 Defaultní hodnoty proměnných

Pokud u deklarace proměnných není určena počáteční hodnota, jsou všechny hodnoty nastavené na nulu. V případě typu boolean to znamená false. Nově vytvořené pole se těmito pravidly řídí také, to znamená, že všechny indexy pole jsou nulové.

2.5 Struktura překladače

Překladač obsahuje šest balíčků s třídami.

2.5.1 Convertor package

Tento balík obsahuje dvě statické třídy. První je TypeConvertor, která slouží pro změnu typu uvnitř překladače a druhá je Validators, ve které se nachází validační metody pro zjištění různých věcí z textu. Například napomáhá při zjišťování, zda se má získat jméno proměnné nebo hodnota.

2.5.2 CreateFilePL0

Zde je jen jedna třída a ta slouží pro zapsání hotového kódu do souboru, či vytvoření chybového souboru.

2.5.3 Elements

V tomto balíčku se nachází třídy pro ošetření jednotlivých řádků programu. Například třida *DeclarationTranslate* se stará o správně přeloženou deklaraci nebo třida ForTranslate překládá cyklus for. Třída, která stojí za zmínku je SolveProblem, ve které se generuje vyhodnocování výrazů, proto tuto třídu dělí každá třida v balíčku elements.

2.5.4 Enums

Tento balíček obsahuje enum třídy s chybovými hláškami, instrukcemi a definovanými operacemi.

2.5.5 GeneratedParser

V tomto balíčku je vygenerovaný parser pomocí ANTLR a dva naše vlastní listenery – *SLLanguageRegisterFunction* a *SLLanguageMainListener*.

2.5.6 TableClasses

V posledním balíčku jsou třídy s jednotlivými tabulkami a práce s nimi.

2.6 Postup překladu

Překlad samotný je realizován pomocí listenerů, které dědí od vygenerovaného listeneru pomocí ANTLR. Pomocí listenerů poté procházíme celý strom programu. Pro překlad používáme dva listenery. *SLLanguageRegisterFunction* listener slouží jen pro registraci funkcí a druhý listener *SLLanguageMainListener* dělá zbytek.

Z těchto listenerů jsou poté volány funkce z balíčku *elements* pro přeložení jednotlivých částí. Při překládání částí kódu se generují instrukce, které jsou ukládány do ArrayListu *tableOfMainCode* ve třídě *TableOfCodes*. V tomto poli jsou uloženy jen instance třídy Code. Případně pokud se musí doplnit ještě adresa, tak jsou přidány ještě do ArrayListu *tableOfCalls* nebo *tableOfIntsJump* do kterých se ukládá instance třídy *ExpectingAddress*.

Zároveň pokud se vytváří v některé části kódu nová proměnná, uloží se v ArrayListu *tableOfSymbols* ve třídě *TableOfSymbols*. Do tohoto pole lze vložit jen instance třídy *Symbol*. Toto pole zároveň zajišťuje, že se nevloží stejná proměnná znovu, jelikož se při každém vložení nejdřív prohledá, zda proměnná již neexistuje. Zároveň jsou zde uloženy i funkce.

Pro doplnění adresy slouží update metody v třídě *TableOfCodes*, které stačí jen zavolat s aktuálním *objectID* a novou adresou a metody tyto hodnoty doplní na správná místa.

3 Syntaxe

V této kapitole bude krátce okomentována a ukázána syntaxe jazyka.

3.1 Definice proměnných a přiřazení

Deklarace proměnných:

```
int cislo;
 boolean logika;
```

Deklarace konstant:

```
const int CISLO = 5;
const boolean LOGIKA = true;
```

Konstanty musí mít vždy přiřazenou hodnotu již při deklaraci.
Deklarace s přiřazením:

```
int cislo = 5;
boolean logika = true;
```

Přiřazení:

```
cislo = 5 + 3;
cislo = cislo + 1;
```

Vícenásobné přiřazení:

```
int a, b ,c;
a = b = c = 5;
```

3.2 Podmínky

3.2.1 Podmínka if

Část programu ve větvi `if` se provede, pokud je splněná podmínka. V podmínce se může využívat všech logických operátorů, viz příklady. Zároveň je možné doplnit na konci větve `if` větev `else`, která se provede v případě, že není splněná podmínka.
Ukázka podmínky:

```
if(!(2 < 3 && 1 > 0) || 1 != 0)
{...}
else{...}
```

Podmínka musí být v závorkách následována ihned po příkazu `if`. Za podmínkou ve složených závorkách se pak nachází část kódu, který se má vykonat v případě splněné podmínky.

3.2.2 Switch

V podmínce `switch` se musí nacházet pouze celé číslo nebo proměnná `int`. Podle dané hodnoty se provede určitý `case` uvnitř `switch`. Zároveň lze na konci `switch` udělat větev `default`, která se provede v případě, že žádný `case` neodpovídal hodnotě v podmínce. Narozdíl od jazyků C a java, se vždy provede pouze jeden `case`.
Ukázka podmínky:

```
switch(2){
 case 1:
 case 2: int a = 2;
 default: int b = 0;}
```

3.2.3 Ternární operátor

Jazyk umožňuje i zkrácený zápis podmínky `if`, případně podmíněného přiřazení.
Ukázka ternární podmínky

```
(cislo < 2) ? cislo = 2 : cislo = 3;
```

Ukázka podmíněného přiřazení

```
cislo = (cislo < 2) ? 2 : 3;
```

3.3 Cykly

Cykly slouží k určitému opakování stejného kódu. Cyklus se dá ukončit příkazem `break`. Případně skočit znovu na začátek cyklu příkazem `continue`.

3.3.1 While

Cyklus, který se provádí dokud je splněná podmínka. Platí zde stejná pravidla jako v podmínce `if`.
Ukázka cyklu:

```
while(cislo < 3){
 cislo = cislo + 1;
}
```

3.3.2 Do...while

Podmínka se ověřuje až na konci cyklu, tedy program se vykoná vždy alespoň jednou.
Ukázka cyklu:

```
do{
 cislo = cislo + 1;
}while(cislo < 3);
```

3.3.3 Until

Podobný cyklus jako `while`, akorát se provádí pokud podmínka je nesplněná. Jakmile se podmínka splní, cyklus končí.
Ukázka cyklu:

```
until(cislo > 3){
 cislo = cislo + 1;
}
```

3.3.4 Repeat...until

Podobný cyklus jako `until`, akorát podmínka se ověřuje až na konci cyklu. Program se tedy vykoná alespoň jedenkrát.

3.3.5 For

Cyklus s určitým počtem opakování. Podmínka se skládá ze tří částí. V první části musí být nastavení počáteční hodnoty proměnné. V druhé části musí být podmínka, při její splnění se bude cyklus provádět. V poslední části je pak operace, která se provede na konci cyklu.
Ukázka cyklu:

```
for(int i = 0; i < 3; i = i + 1){
 ...
}
```

3.4 Pole

Pokud se používá v programu pole je potřeba výsledných program spouštět v interpretu PL0 s rozšířenou instrukční sadou, jelikož pro fungování polí jsou potřeba instrukce `PLD` a `PST`.

Pole jsou vytvářená kódem:

```
<type> [] <name> = new <type>[<size>]
```

Hodnota size musí být číslo. Následně se již může k poli přistupovat kódem:

```
int a = pole[<index>]
```

Hodnota index může být libovolný výraz, který má celočíselný datový typ. Pokud se pole nachází na pravé straně přiřazení, může se na jeho první hodnotu přistupovat jen pomocí uvedení jména proměnné bez hranatých závorek. Na ostatní indexy se již musí použít hranaté závorky.

Přiřazování do pole může být dvěma způsoby. Při uvedení indexu u pole se dá přiřadit hodnota na danou pozici v poli. Pokud žádný index dán není, očekává se na druhé straně pole, které má být překopírováno do daného pole. Kopírování hodnot vždy přenese maximální možnou velikost přiřazovaného pole omezenou velikosti pole, kam se data přenášejí.

3.5 Funkce

Program lze členit do podprogramů pomocí funkcí. Funkce musí být definovány na začátku programu, při definici je důležité klíčové slovo `function`. Funkcím lze předávat parametry a zároveň funkce můžeš vracet jednu hodnotu, viz příklad.

Ukázka funkce:

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

Volání funkce:

```
int c = soucet(1, 2);
```

3.6 Komentáře

Komentáře slouží k označení části kódu, která se nebude překládat do instrukcí. Realizovány byly blokové komentáře, které jsou označeny sekvencí `/*` na začátku bloku a `*/` na konci bloku.

Ukázka komentářů:

```
/* tohle je komentar */
```

4 Testovací příklady

Pro testování našeho překladače jsme si vytvořili vlastní testovací mechanismus, který testuje překlad souborů a následně i správný výstup. Kontrolujeme tak možné chyby v gramatice, a zároveň i správné generování.

Testy fungují tak, že si berou všechny soubory s koncovkou *.sll* nebo *–errors.log* ze složky *tests/testFiles*, které jsou jsou následně přeloženy do složky *tests/testOutput*. Odtud jsou posléze porovnávány se správnými soubory uloženými v *tests/testValidation*. Výstup testů je následně uložen do souboru *testsResult.txt* v adresáři *tests/testValidation*. Odtud lze poté jednoduše zjistit, kde se stala chyba.

Testy se spouští pomocí třídy *RunTests*, která je uložená ve složce *java/src*.

Pomocí těchto testů lze validovat i nekorektní soubory a může se tak snadno udělat otestované prostředí. Tato testovací možnost se nám líbila nejvíc, protože je jednoduchá a otestujeme tím, jak gramatiku, tak výstupní soubory, což u krátkého vývoje má asi největší využití.

Některé kratší ukázky a výstupní instrukce jsou přiloženy zde.

4.1 Test přiřazení

Program:

```
int a = 5;
int mn, ob = 5 + a, i = 3, or;
boolean c = true;
a = 3;
const int TEST = 4;
int b = TEST;
int d = b;
int e = b;
if (a < 5) {
 b = 3;
} else {
 b = 8;
}
c = a == b;
```

Instrukce:

```
0 JMP 0 1
```

```
1 INT 0 13
2 LIT 0 5
3 STO 0 3
4 LIT 0 0
5 STO 0 4
6 LIT 0 5
7 LOD 0 3
8 OPR 0 2
9 STO 0 5
10 STO 0 6
11 LIT 0 0
12 STO 0 7
13 LIT 0 1
14 STO 0 8
15 LIT 0 3
16 STO 0 3
17 LIT 0 4
18 STO 0 9
19 LOD 0 9
20 STO 0 10
21 LOD 0 10
22 STO 0 11
23 LOD 0 10
24 STO 0 12
25 LOD 0 3
26 LIT 0 5
27 OPR 0 10
28 JMC 0 32
29 LIT 0 3
30 STO 0 10
31 JMP 0 34
32 LIT 0 8
33 STO 0 10
34 LOD 0 3
35 LOD 0 10
36 OPR 0 8
37 STO 0 8
38 RET 0 0
```

4.2 Test cyklu a podmínek

Zde je otestovaný pouze cyklus `for` a podmínka `if`. Všechny cykly jsou testovány v souboru *tests/testFiles/cykly/testCycles.sll*.
Program:

```
int i;
for(int k = 0; k < 3; k = k + 1){
 if(i < 3){
 i = i - 1;
 }
 else{
 i = i * k;
 }
}
```

Instrukce:

```
0 JMP 0 1
1 INT 0 5
2 LIT 0 0
3 STO 0 3
4 LIT 0 0
5 STO 0 4
6 LOD 0 4
7 LIT 0 3
8 OPR 0 10
9 JMC 0 28
10 LOD 0 3
11 LIT 0 3
12 OPR 0 10
13 JMC 0 19
14 LOD 0 3
15 LIT 0 1
16 OPR 0 3
17 STO 0 3
18 JMP 0 23
19 LOD 0 3
20 LOD 0 4
21 OPR 0 4
22 STO 0 3
23 LOD 0 4
24 LIT 0 1
```

```
25 OPR 0 2
26 STO 0 4
27 JMP 0 6
28 RET 0 0
```

4.3 Testování funkcí

Ukázka funkce pro součet dvou čísel.
Program:

```
int function soucet (int a, int b) {
 return a + b;
}
int c = soucet(1, 2);
```

Instrukce:

```
0 JMP 0 1
1 INT 0 8
2 LIT 0 0
3 STO 0 3
4 LIT 0 0
5 STO 0 4
6 LIT 0 1
7 STO 0 4
8 LIT 0 2
9 STO 0 5
10 CAL 0 14
11 LOD 0 3
12 STO 0 6
13 RET 0 0
14 INT 0 5
15 LOD 1 4
16 STO 0 3
17 LOD 1 5
18 STO 0 4
19 LOD 0 3
20 LOD 0 4
21 OPR 0 2
22 STO 1 3
23 RET 0 0
```

5 Spuštění překladu

Spustitelný jar, lze nalézt ve složce */compiled*.
Překlad souboru se pak provede příkazem:

```
java -jar FJP_super_language.jar program.sll
```

Tímto příkazem se provede překlad a vytvoří se nový soubor se jménem stejným jako měl překládaný program a příponou *.pl*.
Nepovinným parametrem lze specifikovat cíl a jméno výstupního souboru. Stačí jako druhý argument uvést cestu se jménem požadovaného výstupního souboru.

```
java -jar FJP_super_language.jar program.sll compile/vystup.pl
```

6 Závěr

Semestrální práci se podařilo úspěšně dokončit. Nicméně během tvorby jazyka a překladače jsme narazili na řadu problémů a obtíží. Nejhorší byl začátek, kdy bylo potřeba sestavit gramatiku jazyka a zprovoznit nástroj ANTLR pro parsování programu. Dále se pak naučit jak funguje PL/0, a co znamenají jednotlivé instrukce.

Práce byla poměrně rozsáhlá a bylo potřeba mnoho úsilí, aby byly splněny minimální požadavky na rozsah. Na druhou stranu byla práce originální a poskytla nám pohled do fungování překladačů, jak se z nám známeho programovacího jazyka stane posloupnost strojových instrukcí použitelných pro procesor.

Vzhledem k tomu, že práce byla dělána ve dvojicích, se bylo vždy možno poradit, když jsme si nevěděli rady. Zároveň si myslíme, že i komunikace byla lepší, než kdyby jsme tuto práci dělali v početnějším týmu. Ačkoliv si nedovedeme představit, kolik času by bylo potřeba nad touto prací strávit, abychom dosáhli maximálního počtu bodů.

Celý projekt byl veden na githubu na adrese FJP_super_language[1]. Zde jsme dělali často commity a zakládali issue, abychom měli přehled o stavu projektu. Za semestrální práci, tak jak bylo řečeno v kapitole 1, by jsme očekávali 25 bodů. Případně bonusové body za implementaci komentářů, příkazu break a continue.

[1] Adresa v případě nefunkčnosti odkazu: https://github.com/tuslm/FJP_super_language