Semestrální práce z KIV/FJP

Tvorba překladače zvoleného jazyka

Zdeněk Častorál
A19N0026P
zcastora@students.zcu.cz

Jitka Poubová
A19N0043P
jpoubova@students.zcu.cz

https://github.com/JitkaP/FJP-project

4. 1. 2020

Obsah

1 Zadání **1**

2 Návrh gramatiky **3**
2.1 Jazykové konstrukce 3
2.1.1 Datové typy 3
2.1.2 Deklarace konstant a proměnných 4
2.1.3 Podmínky 4
2.1.4 Cykly 4
2.1.5 Procedury 5
2.2 Vlastnosti jazyka 6

3 Návrh překladače **7**

4 Implementace **8**
4.1 Použité technologie 8
4.2 Struktura aplikace 8
4.3 Implementace složitějších rozšíření 9
4.3.1 Pole 9
4.3.2 Porovnávání polí 9
4.3.3 Cykly a proměnná v podmínce 9
4.3.4 Ukládání hodnot proměnných 9

5 Uživatelský manuál **10**
5.1 Překlad aplikace 10
5.2 Spuštění aplikace 10
5.3 Příklady programů 10

6 Závěr **11**

Přílohy **12**
A Gramatika 12

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 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
- ?

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

2 Návrh gramatiky

Při návrhu gramatiky jsme vycházeli z gramatiky jazyka PL/0. Gramatiku jsme rozšířili o další jazykové konstrukce, které jsou popsány dále v textu. Kompletní gramatika je uvedena v příloze A.

2.1 Jazykové konstrukce

Kromě minimálních jazykových konstrukcí, které musí navrhovaný jazyk obsahovat, jsme zvolili následující rozšíření:

- tři typy cyklu – for, while, do-while
- else větev,
- datový typ boolean a logické operace s ním,
- datový typ string s operátory pro spojování řetězců,
- násobné přiřazení,
- ternární operátor,
- paralelní přiřazení,
- pole a práce s jeho prvky,
- operátor pro porovnání řetězců.

2.1.1 Datové typy

Gramatika našeho jazyka umožňuje deklaraci tří datových typů:

- `int` (integer – celé číslo),
- `bool` (boolean – logická hodnota),
- `char` (character – znak).

Z každého z výše uvedených datových typů je možné vytvořit pole – `int[]`, `bool[]` a `char[]`. Je možné pracovat s jednotlivými prvky pole – indexace prostřednictvím závorek `[<index>]`.

Datový typ `boolean` umožňuje použití logických operací nad ním. Pole znaků (`char[]`) reprezentuje řetězec (`string`), nad kterým jsou zavedeny operátory pro spojování porovnání řetězců.

2.1.2 Deklarace konstant a proměnných

Gramatika našeho jazyka vynucuje deklaraci proměnných a konstant vždy na začátku bloku. Při deklaraci musí být uvedený jeden z možných datových typů – viz kapitola 2.1.1.

Konstanty jsou deklarovány použitím klíčového slova `const` – pokud se jedná o pole, neuvádí se zde jeho délka, ale je nutné do něj rovnou přiřadit hodnoty prostřednictvím závorek `{<prvky_pole>}`. Proměnné jsou deklarovány použitím klíčového slova `var` – pokud se jedná o pole, je v tomto případě nutné uvést jeho délku, a není možné do něj rovnou přiřadit hodnoty.

Ukázka deklarace

```
const bool A := true;
var char b;
const int C[] := {1, 2, 3};
var char a[3], b[3];
```

2.1.3 Podmínky

Podmínky jsou v našem jazyce velmi podobné podmínkám v jazyce PL/0, oproti němu však umožňují použití `else` větve.

Ukázka podmínky

```
if (a != b) then
begin
    i := 5;
end

else
begin
    i := 10;
end
```

2.1.4 Cykly

V našem jazyce existují tři typy cyklů:

- `for`,
- `while`,

- `do-while`.

Řídící proměnná cyklu `for` musí být deklarována před použitím v tomto typu cyklu.

Ukázka použití cyklů

```
begin
    for a := 1 to 5 do
    begin
        //telo cyklu
    end;

    while (a != 3) do
    begin
        //telo cyklu
    end;

    do
    begin
        //telo cyklu
    end
    while (a != 1);
end
```

2.1.5 Procedury

Procedury jsou v naší gramatice navrženy stejně, jako v gramatice jazyka PL/0.

Ukázka deklarace a volání procedury

```
var int a;

//deklarace procedury
procedure add;
begin
    a := a + 5;
end;

begin
  a := 1;
  call add; //volani procedury
end.
```

2.2 Vlastnosti jazyka

Námi navržený jazyk má následující vlastnosti:

- proměnné a konstanty mohou být definovány pouze na začátku bloku,
- při deklaraci proměnných a konstant musí být uveden datový typ,
- při deklaraci lze hodnota přiřadit pouze konstantám,
- procedury neobsahují vstupní parametry ani návratovou hodnotu,
- řídící proměnná `for` cyklu musí být deklarována také na začátku bloku.

3 Návrh překladače

Jazyk, který je popsán gramatikou v kapitole 2, bude překládán pro architekturu PL/0, konkrétně do základní instrukční sady PL/0. Instrukce základní instrukční sady PL/0 jsou uvedeny v tabulce 3.1.

Tabulka 3.1: Instrukce PL/0

Instrukce	**Popis**
LIT 0, A	ulož konstantu A do zásobníku
OPR 0, A	proveď operaci A
OPR 0, 1	unární mínus
OPR 0, 2	sčítání
OPR 0, 3	odečítání
OPR 0, 4	násobení
OPR 0, 5	celočíselné dělení
OPR 0, 6	dělení modulo
OPR 0, 7	test, zda je číslo liché
OPR 0, 8	test rovnosti
OPR 0, 9	test nerovnosti
OPR 0, 10	menší než
OPR 0, 11	větší nebo rovno než
OPR 0, 12	větší než
OPR 0, 13	menší nebo rovno než
LOD L, A	ulož hodnotu proměnné z adresy L, A na vrchol zásobníku
STO L, A	zapiš do proměnné z adresy L, A hodnotu z vrcholu zásobníku
CAL L, A	volej proceduru A z úrovně L
INT 0, A	zvyš obsah top–registru zásobníku o hodnotu A
JMP 0, A	proveď skok na adresu A
JMC 0, A	proveď skok na adresu A, je-li hodnota na vrcholu zásobníku 0
RET 0, 0	návrat z procedury (return)

4 Implementace

4.1 Použité technologie

K implementaci navrženého řešení jsme použili programovací jazyk `Java` ve verzi `11.0.4`. Lexikální a syntaktickou analýzu jsme realizovali rekurzivním sestupem s použitím `ANTLR` ve verzi `4.7.2`.

4.2 Struktura aplikace

```
src
    |_ antlr
        |_ gen
        |_ Lang.g4
    |_ main
        |_ compiler
            |_ entity
            |_ enums
            |_ generator
            |_ visitor
            |_ Compiler.java
            |_ InstructionGenerator.java
            |_ ThrowingErrorListener.java
        |_ MainClass.java
```

V balíku `src.antlr` se nachází gramatika a pak adresář se soubory, vygenerovanými pomocí nástroje ANTLR.

V balíku `src.main` se nachází přímo náš kód aplikace. Tento balík se skládá se čtyř hlavních podbalíků:

- `entity` – v tomto adresáři se nacházejí veškeré objekty, které znázorňují jednotlivé části gramatiky. Například `Program`, `Procedure` nebo `Condition`,
- `enums` – zde se nacházejí všechny výčtové typy. Například v `ENumberOp` jsou všechny operátory pro aritmetické výrazy (PLUS, MINUS, MUL, DIV).
- `generator` – tento adresář obsahuje třídy, kde každá z nich má metodu `generate`, ve které vygeneruje dané instrukce,
- `visitor` – v tomto adresáři jsou třídy, které překrývají metodu `visit`.

4.3 Implementace složitějších rozšíření

4.3.1 Pole

Pole se ukládají na zásobník stejně jako v Pascalu. Například když máme pole uložené na adrese 0 s prvky 7,8,9, tak v paměti je na adrese 0 uložená délka pole (3), na adrese 1 je uložen první prvek (7), na adrese 2 je číslo 8 a na adrese 3 je číslo 9.

4.3.2 Porovnávání polí

Náš jazyk umožňuje porovnávání jakýkoliv polí, tedy včetně řetězců (reprezentovaných jako `char[]`). Každé porovnání dvou polí začíná nejprve porovnáním jejich délek, poté následuje porovnávání prvek po prvku.

4.3.3 Cykly a proměnná v podmínce

Pokud v ukončovací podmínce cyklu je použitá nějaká proměnná, musí tato proměnná být deklarovaná již na začátku bloku. Dále je možné měnit její hodnotu i uvnitř daného cyklu.

4.3.4 Ukládání hodnot proměnných

Nejprve jsme implementovali práci s jednoduchými datovými typy, přičemž jsme rovnou generovali instrukce a hodnotu proměnných jsme si nikam neukládali. Ovšem poté jsme začali implementovat pole a práci s jeho prvky a zjistili jsme, že například pokud chceme ukládat hodnotu do `array[i]`, musíme znát hodnotu proměnné `i`, abychom mohli správně vypočítat adresu pro instrukci `STO`. Tudíž u všech proměnných (včetně polí) si ukládáme jejich aktuální hodnoty při každém přiřazení.

5 Uživatelský manuál

5.1 Překlad aplikace

K překladu aplikace je nutné mít nainstalované a správně nakonfigurované následující nástroje:

- `JDK` – verze alespoň 11,
- `Maven`.

Překlad aplikace se provede zadáním následujícího příkazu do příkazového řádku (uvnitř adresáře s projektem, tam, kde se nachází soubor `pom.xml`):

```
mvn clean install
```

`Maven` vytvoří spustitelný `JAR` soubor v novém podadresáři `target`.

5.2 Spuštění aplikace

Po vytvoření `JAR` souboru je možné aplikaci spustit zadáním příkazu (uvnitř adresáře `target`):

```
java -jar FJP-project-jar-with-dependencies.jar <vstupni_soubor>
```

Aplikace vytvoří výstupní soubor, jehož název odpovídá názvu vstupního souboru, se suffixem `_output`.

5.3 Příklady programů

V adresáři `examples` jsou uvedené tři příklady programů v našem jazyce. Pro každý z nich se tam také nachází naším překladačem vygenerované instrukce.

6 Závěr

Cílem semestrální práce bylo vytvořit překladač pro již existující, nebo námi navržený programovací jazyk, který musel obsahovat minimální jazykové konstrukce a některé z rozšiřujících jazykových konstrukcí (viz kapitola 1).

Rozhodli jsme se realizovat vlastní programovací jazyk, který vychází z jazyku PL/0. Nejprve jsme vybrali rozšiřující jazykové konstrukce, které by měl náš jazyk obsahovat, a navrhli gramatiku našeho jazyka. Poté jsme pro náš jazyk vytvořili překladač, který generuje instrukce pro architekturu PL/0, konkrétně pro základní instrukční sadu PL/0.

Naše implementace proběhla z našeho pohledu úspěšně. Největším problémem při implementaci překladače byla realizace polí. V ostatních případech jsme se nesetkali s většími problémy.

Semestrální práce z našeho pohledu splňuje zadání.

Přílohy

A Gramatika

```
grammar Lang;

program
    : block '.'
    ;

block
    : (declaration)* (procedure)* statement
    ;

declaration
    : (consts | constarrays | vars)
    ;

procedure
    : PROCEDURE ident ';' block ';'
    ;

consts
    : CONST TYPE ident ':=' value ';'
    ;

constarrays
    : CONST TYPE ident '[]' ':='
    '{' value (',' value)* '}' ';'
    ;

vars
    : VAR TYPE (ident | ident_arr )
    ( ',' (ident | ident_arr))* ';'
    ;

statement
    : (assignstmt | parallelstmt | callstmt
    | beginstmt | ifstmt | whilestmt
    | dowhilestmt | forstmt | ternarstmt)?
    ;

assignstmt
    : (ident | ident_arr) (':=' (ident | ident_arr))*
    ':=' expression
    ;
```

```
parallelstmt
    : '{' (ident | ident_arr)
    (',' (ident | ident_arr))* '}' ':='
    '{' (expression) (',' (expression))* '}'
    ;

callstmt
    : CALL ident
    ;

beginstmt
    : BEGIN statement (';' statement)* END
    ;

ifstmt
    : IF condition THEN statement (ELSE statement)?
    ;

whilestmt
    : WHILE condition DO statement
    ;

dowhilestmt
    : DO statement WHILE condition
    ;

forstmt
    : FOR ident ':=' number_expression
    TO number_expression
    DO statement
    ;

ternarstmt
    : (ident | ident_arr) ':=' condition '?'
    expression ':' expression
    ;

condition
    : '('
        (
        (number_expression
            ( LESS | LESS_EQ | GREATER | GREATER_EQ )
            number_expression)
        | (expression
            ( EQ | NOT_EQ )
            expression)
        | (bool_expression)
        )
```

```
    ')'
    ;

expression
    : (bool_expression
    | string_expression
    | number_expression)
    ;

bool_expression
    : (NEG)? (BOOLEAN | (ident | ident_arr))
    ((AND | OR)
    (NEG)? (BOOLEAN | (ident | ident_arr)))*
    ;

string_expression
    : (ident | STRING_VALUE)
    (PLUS (ident | STRING_VALUE))*
    ;

number_expression
    : (PLUS | MINUS)? term ((PLUS | MINUS) term)*
    ;

term
    : factor ((MUL | DIV) factor)*
    ;

factor
    : NUMBER
    | ident
    | ident_arr
    | ('(' number_expression ')')
    ;

ident_arr
    : ident '[' (NUMBER | ident) ']'
    ;

ident
    : LETTER (LETTER | NUMBER)*
    ;

value
    : (NUMBER | BOOLEAN | STRING_VALUE)
    ;

CALL
    : 'call'
```

```
    ;

PROCEDURE
    : 'procedure'
    ;

VAR
    : 'var'
    ;

THEN
    : 'then'
    ;

BEGIN
    : 'begin'
    ;

END
    : 'end'
    ;

ELSE
    : 'else'
    ;

WHILE
    : 'while'
    ;

DO
    : 'do'
    ;

FOR
    : 'for'
    ;

TO
    : 'to'
    ;

IF
    : 'if'
    ;

CONST
    : 'const'
    ;
```

```
TYPE
    : ('int' | 'bool' | 'char')
    ;

STRING_VALUE
    : '"' (LETTER | NUMBER)* '"'
    ;

BOOLEAN
    : ('true' | 'false')
    ;

LETTER
    : [a-zA-Z]
    ;

NUMBER
    : [0-9]+
    ;

LESS
    : '<'
    ;

LESS_EQ
    : '<='
    ;

GREATER
    : '>'
    ;

GREATER_EQ
    : '>='
    ;

EQ
    : '='
    ;

NOT_EQ
    : '!='
    ;

NEG
    : '!'
    ;
```

```
AND
     : '&&'
     ;

OR
     : '||'
     ;

MUL
     : '*'
     ;

DIV
     : '/'
     ;

PLUS
     : '+'
     ;

MINUS
     : '-'
     ;

WS
    : [ \t\r\n] -> channel(HIDDEN)
    ;

LINE_COMMENT
     :    '///' ~[\r\n]* -> channel(HIDDEN)
     ;
```