

KIV/FJP

Semestrální práce

Překladač jazyka Not so Swift

Daniel Schnurpfeil, Jiří Trefil

1. ledna 2023

Obsah

1	Zadání	2
2	Navržený jazyk Not so Swift	3
2.1	Gramatika	3
3	Architektura překladače	6
4	Popis implementace	7
4.1	Lexikální analýza	7
4.2	Syntaktická analýza	7
4.3	Derivační syntaktický strom a tabulka symbolů	7
4.4	Sémantická analýza	7
4.5	Generování instrukcí PL/0	8
4.6	Testování	9
5	Závěr	10
6	Uživatelská příručka	11

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).

Cílová architektura:





- instrukce PL/0 

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

Kromě základní funkcionality byla zvolena následující rozšíření.

Jednoduchá rozšíření:

- else větev 
- repeat while 
- while 
- datový typ boolean a logické operace s ním 

Více složité rozšíření:

- parametry předávané hodnotou 
- návratová hodnota podprogramu 
- anonymní vnitřní funkce (lambda výrazy) 
- pole a práce s jeho prvky

2 Navržený jazyk Not so Swift

V rámci této semestrální práce byl navrhnout programovací jazyk Not so Swift. Jak už z názvu vyplývá, jedná se o zjednodušenou verzi jazyka Swift, kde je mírný rozdíl v syntaxi o proti jazykům, jako je například C++. Nicméně je ale zachován zápis například zápis for cyklu jako je v programovacím jazyce C. V adresáři `sample_input` lze naléznout příklady jednoduchých programů v jazyce Not so Swift.

```
.....  
  
var a: Int = 52;  
func someOtherFunction(c: Int) -> Int {  
    for(var j: Int = 0; j < 10; j+= 1;) {  
        c += 1;  
    }  
    return c;  
}  
a = someOtherFunction(a);  
if( 100 < a) {  
    repeat {  
        a = a - 1;  
    } while a > 50;  
}  
  
.....
```

Obrázek 1: Příklad vstupního programu

2.1 Gramatika

```
S -> program | S  
program -> dekl_list  
dekl_list -> dekl  
dekl_list -> expression semicolon  
dekl_list -> var_modification semicolon  
dekl_list -> var_modification semicolon dekl_list  
dekl_list -> dekl dekl_list  
dekl_list -> block  
var_modification -> id sub expression  
var_modification -> id add expression  
var_modification -> id mulby expression  
var_modification -> id divby expression  
var_modification -> id equals expression  
var_modification -> id lparent int rparent equals expression  
var_modification -> var_modification  
dekl -> var var_dekl  
dekl -> let var_dekl  
dekl -> fun_dekl  
var_dekl -> id ddot dtype semicolon  
var_dekl -> id ddot dtype equals expression semicolon
```

```

dtype -> int_type
dtype -> boolean_type
dtype -> array_dekl
dtype -> string_type
array_dekl -> array lparent int rparent
expression -> expression minus term
expression -> expression plus term
expression -> term
expression -> ternary
ternary -> condition question_mark expression ddot expression
term -> term multiply factor
term -> term divide factor
term -> factor
factor -> lparent expression rparent
factor -> minus expression
factor -> val
factor -> call
empty -> <empty>
call -> id lparent arguments rparent
val -> int
val -> bool
val -> id
val -> quote id quote
val -> lparent integer_list rparent
integer_list -> int comma integer_list
integer_list -> int
fun_dekl -> func id lparent params rparent arrow dtype comp_block
fun_dekl -> func id lparent params rparent arrow Void comp_block
params -> params_var
params -> empty
params_var -> id ddot dtype comma params_var
params_var -> id ddot dtype
arguments -> val comma arguments
arguments -> val
arguments -> empty
comp_block -> lparent block rparent
block -> comp_block dekl_list
block -> loop_block dekl_list
block -> cond_block dekl_list
block -> let var_dekl dekl_list
block -> var var_dekl dekl_list
block -> var_modification semicolon dekl_list
block -> expression semicolon dekl_list
block -> return expression semicolon
block -> loop_block
block -> cond_block
block -> let var_dekl
block -> var var_dekl
block -> expression semicolon
block -> var_modification semicolon
loop_block ->

```

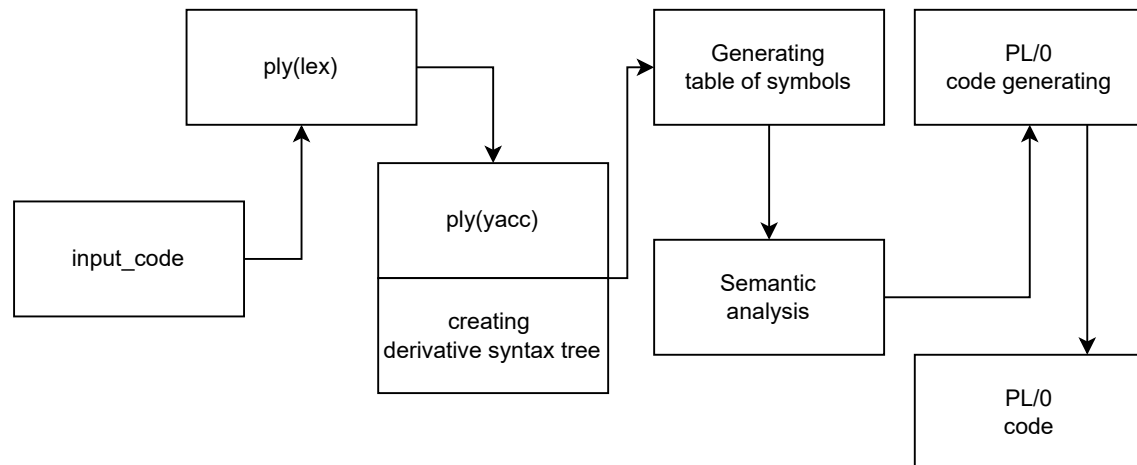
```

-> for lparent loop_var condition semicolon step semicolon rparent comp_block
loop_block -> while condition comp_block
loop_block -> repeat comp_block while condition semicolon
cond_block -> if lparent condition rparent comp_block
cond_block -> if lparent condition rparent comp_block else comp_block
loop_var -> var var_dekl
loop_var -> id semicolon
step -> id add int
step -> id sub int
condition -> expression relation_operator expression and condition
condition -> expression relation_operator expression or condition
condition -> exclamation_mark lparent condition rparent
condition -> expression relation_operator expression
condition -> val
condition -> expression and condition
condition -> expression or condition
condition -> exclamation_mark lparent condition rparent and condition
condition -> exclamation_mark lparent condition rparent or condition
relation_operator -> equals_equals
relation_operator -> lt
relation_operator -> gt
relation_operator -> le
relation_operator -> ge
relation_operator -> not_equal

```

3 Architektura překladače

Architektura se skládá z pěti částí. Všechny části jsou psány v jazyku `Python`. První částí je lexikální analýza vstupního kódu. Druhou částí je syntaktická analýza a generování derivačního syntaktického stromu. Dále následuje sémantická analýza. Nakonec jsou generovány instrukce pro PL/0.



Obrázek 2: Znázornění architektury překladače jazyka Not so Swift

Pro lexikální a syntaktickou analýzu je použita knihovna *PLY*¹. Sestavení derivačního syntaktického stromu, sémantická analýza a generování instrukcí používá stromovou strukturu knihovny *ETE3*².

¹<https://pypi.org/project/ply/>

²<https://pypi.org/project/ete3/>

4 Popis implementace

Zdrojový kód je rozčleněn do podadresářů. Každá část architektury má svůj vlastní adresář kromě generování tabulky symbolů a derivačního syntaktického stromu, které jsou části adresáře syntaktické analýzy.

4.1 Lexikální analýza

Lexikální analyzátor kontroluje, zda vstupní soubor obsahuje validní **tokeny**³. Tyto **tokeny** jsou popsány regulárními výrazy:

- klíčová slova
 - `let, var, func, for, return, if, else, and, or, while, repeat`
- ostatní tokeny
 - `equals, equals_equals, plus, minus, divide, multiply, int_type, int, bool, id, semicolon, rparent, lparent, lt, le, gt, ge, arrow, rcparent, lcparent, ddot, comma, add, sub, not_equal, divby, mulby, question_mark`

Přesné znění regulárních výrazů lze najít v `/src/lex_analyzer/lexer.py`.

4.2 Syntaktická analýza

V rámci syntaktické analýzy se zkoumá zda proud **tokenů** (vytvořen lexikální analýzou), odpovídá pravidlům gramatiky jazyka **Not so Swift**. Implementace syntaktické analýzy se nachází v `/src/syntax_analyzer/parser.py`.

4.3 Derivační syntaktický strom a tabulka symbolů

Během syntaktické analýzy se tvoří derivační syntaktický strom pomocí zmíněné knihovny *ETE3*. Příklad vizualizace je vidět na obrázku 3. Na rozdíl od derivačního syntaktického stromu se tabulka symbolů generuje až po vykonání syntaktické analýzy. Implementace derivačního syntaktického stromu a tabulky symbolů se nachází v `/src/syntax_analyzer/`.

4.4 Sémantická analýza

Sémantická analýza má za úkol zjistit, zda je vstupní kód smysluplný. Překladač prochází syntaktický strom strategií **preorder** a vyhodnocuje zda datový typ proměnných odpovídá jejich hodnotám, tedy silnou typovou kontrolu. Dále vyhodnocuje správnost aritmetiky, funkcí, implementovaných cyklů, podmínek a v neposlední řadě polí.

³Identifikátory, klíčová slova,...

-----record-----		-----record-----	
2031406785728	id	2029810020016	id
a	name	someOtherFunction	name
0	real_level	0	real_level
Int	symbol_type	func	symbol_type
False	const	False	const
0	level	0	level
3	address	4	address
0	size	0	size
-----		Int	return_type

-----params-----		-----locals-----	
-----record-----		-----record-----	
2031406993520	id	2029810888960	id
c	name	j	name
0	real_level	1	real_level
Int	symbol_type	Int	symbol_type
False	const	False	const
0	level	someOtherFunction	level
3	address	4	address
0	size	0	size
True	param	-----	

Tabulka 1: Příklad tabulky symbolů pro vstup z obrázku 1.

4.5 Generování instrukcí PL/0

Cílem překladače je vygenerovat posloupnost instrukcí v jazyce PL/0. Překladač používá základní sadu instrukcí PL/0:

- lit 0,A – ulož konstantu A do zásobníku
- opr 0,A – proved' instrukci A
 - 1 – unární minus
 - 2 – +
 - 3 – -
 - 4 – *
 - 5 – div - celočíselné dělení (znak /)
 - 6 – mod - dělení modulo (znak %)
 - 7 – odd - test, zda je číslo liché
 - 8 – test rovnosti (znak =)
 - 9 – test nerovnosti (znaky <>)

- 10 – <
- 11 – >=
- 12 – >
- 13 – <=
- lod L,A – uložit hodnotu proměnné z adr. L,A na vrchol zásobníku
- sto L,A – zapsat do proměnné z adr. L,A hodnotu z vrcholu zásobníku
- cal L,A – volej proceduru A z úrovně L
- int 0,A – zvýš obsah top-registru zásobníku o hodnotu A
- jmp 0,A – proved skok na adresu A
- jmc 0,A – proved skok na adresu A, je-li hodnota na vrcholu zásobníku 0
- ret 0,0 – návrat z procedury (return)

Při generování instrukcí je hojně používán derivační syntaktický strom (dále jen syntaktický strom), tabulka symbolů a rekurze. Tudíž tato fáze překladač probíhá bezprostředně jako poslední. Procházení syntaktického stromu (velice zjednodušeně popsáno) funguje následujícím způsobem. Nejprve se projde syntaktický strom strategií **preorder**. Každý procházený uzel a jeho potomci jsou vkládány do seznamu. Výsledný seznam se dále prochází v cyklu. Pokud je nalezeno primitivum, ze kterého lze vygenerovat instrukce, tak se zavolá příslušná metoda, která vygeneruje instrukce pro dané primitivum. Implementaci lze najít v `/src/pl0_code_generator/`.

4.6 Testování

Pro řádné otestování funkčnosti zdrojového kódu překladače, byly vytvořeny jednotkové testy obsažené v rámci standardní knihovny jazyka *Python verze 3.9*. Správnost vygenerovaných instrukcí byla ověřována debuggerem PL/0. Celkem bylo napsáno 18 testů ověřujících jednotlivé konstrukce jazyka **Not so Swift**, což jsou například: přiřazení, definice proměnných, podmínek, cyklů nebo funkcí. Dále byly napsány 3 komplexnější testy ověřující robustnost překladače. Jednodušší z nich je zde v dokumentaci uveden jako příklad.

V rámci testování je též implementován i virtuální stroj PL/0. Jeho výstup vypadá pro vstup z obrázku 1 například takto:

-----PL/0 start-----	
0	0
1	0
2	0
3	1737
4	0
5	1
6	1
7	1737
8	9
9	54
10	2
11	1
12	1
13	0
14	1
15	54

Tabulka 2: Obsah zásobníku po provedení instrukcí

5 Závěr

Byl navržen programovací jazyk **Not so Swift**, ke kterému byl vymyšlen a implementován jeho překladač. Tento překladač, generuje instrukce pro jazyk PL/0, na základě vstupního souboru se zdrojovým kódem psaným v jazyce **Not so Swift**. Projekt byl veden pomocí GITu a lze ho najít na

<https://github.com/dartix-45/kiv-fjp>.

Zdrojový kód překladače obsahuje (bez tetovacích scénářů) přes 2500 řádků psaných v jazyce Python. Časová náročnost byla následující:

- Jiří Trefil
 - navrhnutí gramatiky jazyka (4h)
 - implementace tabulky symbolů (10h)
 - implementace lexikální analýzy (2h)
 - implementace syntaktické analýzy (35h)
 - implementace sémantické analýzy (45h)
- Daniel Schnurpfeil
 - založení a vedení projektu na gitu (7h)
 - sestavení architektury překladače, integrace knihoven PLY a ETE3 (6h)
 - implementace tabulky symbolů (7h)
 - implementace generování instrukcí pro PL/0 (47h)
 - testování, virtuální stroj pro PL/0 (16h)
 - napsání dokumentace, přepsání gramatiky do dokumentace (20h)

6 Uživatelská příručka

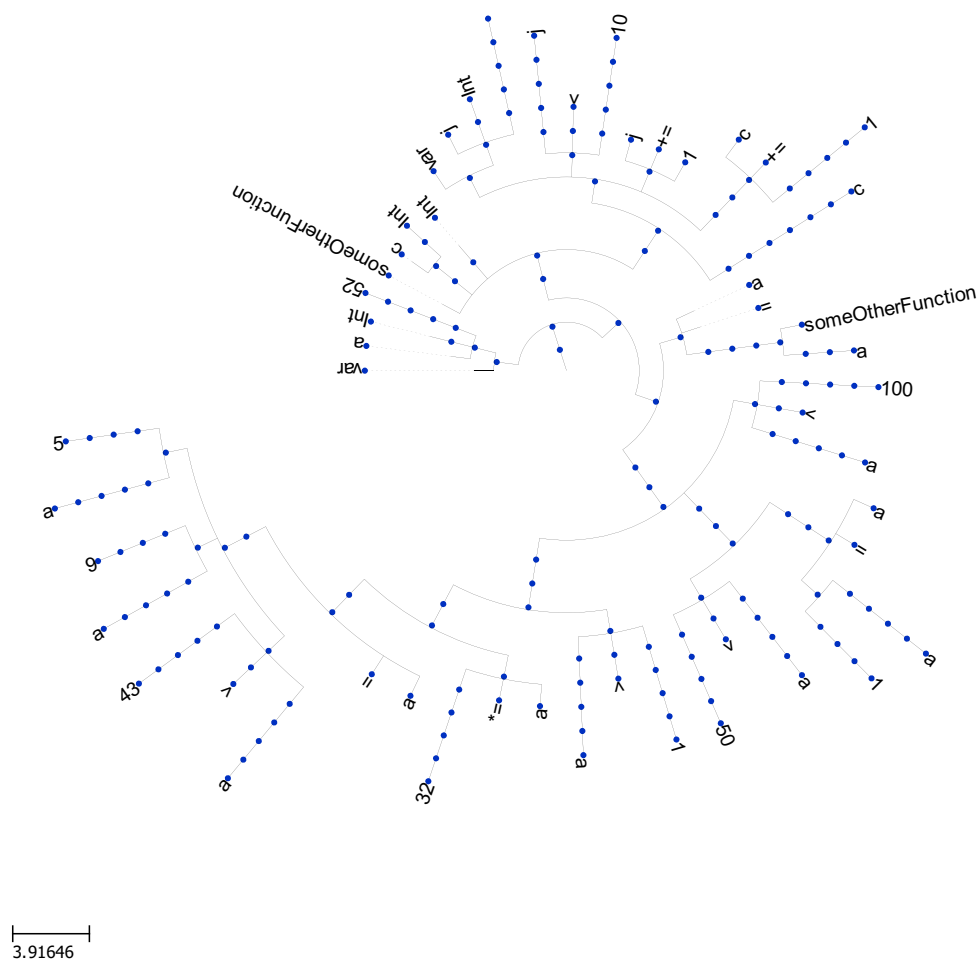
Překladač je možné spouštět pomocí Pythonu:

```
pip install -r requirements.txt
python not_so_swift_compiler.py --f_input <CESTA ke vstupnímu souboru>
```

Pro pohodlí uživatele je možné si stáhnout distribuci binárního souboru na stránkách projektu [Github](#). Pak stačí spustit:

```
not_so_swift_compiler.exe --f_input <CESTA ke vstupnímu souboru>
```

Program vygeneruje složku `output` s patřičným obsahem. Její umístění lze změnit parametrem `--out`.



Obrázek 3: Příklad vizualizace derivačního syntaktického stromu pro vstupní kód z obrázku 1.