

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ FAKULTA INFORMAČNÍCH TECHNOLOGIÍ



Formální jazyky a překladače

Implementace překladače imperativního jazyka IFJ19

Tým 046, varianta II

11. prosince 2019

Rozšíření:

Vojtěch Mimochodek	(xmimoc01)	25 %
Dávid Špavor	(xspavo00)	30 %
Vojtěch Jurka	(xjurka08)	25 %
Martina Tučková	(xtucko00)	20 %

Obsah

1	Úvod	2
2	Návrh a implementace	2
2.1	Lexikální analýza	2
2.1.1	LEXICAL_ANALYSIS.H	2
2.1.2	LEXICAL_ANALYSIS.C	2
2.2	Syntaktický a sémantický analyzátor	2
2.2.1	Syntaktický analyzátor	2
2.2.2	Syntaktický analyzátor pro výrazy	3
2.2.3	Sémantický analyzátor	4
2.3	Generátor kódu	4
2.3.1	Generování instrukcí	4
2.3.2	Výpis kódu	4
2.4	Ostatní části překladače	4
2.4.1	Tabulka symbolů	4
2.4.2	Zásobník	4
3	Spolupráce v týmu	5
3.1	Rozdělení práce	5
3.2	Rozdělení bodů	5
4	Závěr	5
5	Přílohy	5

1 Úvod

Cílem projektu je tvorba programu v jazyce C, který je podmnožinou jazyka Python 3. Načítá zdrojový kód zapsaný ve zdrojovém jazyce IFJ19 a překládá jej do cílového jazyka IFJcode19 (mezikód).

2 Návrh a implementace

Projekt se skládá ze čtyř hlavních částí, které spolu navzájem spolupracují.

2.1 Lexikální analýza

Lexikální analýza provádí skenování vstupního kódu Python 3, který uživatel vkládá při spouštění programu. Jedná se o první část programu provádějícího překlad abstraktního programovacího jazyku Python 3 do IFJcode19 a zastává práci deterministického automatu, který parsuje jednotlivé lexémy, provádí jejich kontrolu a vyhodnocuje jejich význam. Výstupem lexikální analýzy je typ a hodnota každého lexému.

Implementace:

Lexikální analyzátor je implementován podle zadání projektu v jazyce C a nachází se v souborech *lexical_analysis.h* a *lexical_analysis.c*.

2.1.1 LEXICAL_ANALYSIS.H

V hlavičkovém souboru nalezneme definici struktury `Symbol`, která je výstupem analyzátoru a obsahuje v sobě hodnoty `Type` a `Data`. `Type` udává výčet typů, kterých parsovaný lexém může nabývat. Příkladem může být: *integer*, *string*, *function*, *double*, atd.. `Data` jsou typu `union`, jež obsahuje jeden z datových typů `integer`, `double` nebo `char*`. Tato hodnota je závislá na typu lexému. Pokud tedy načteme například číslo 5, tak jeho typem bude `integer` a hodnota bude číslo 5 uložena v unionu jako `integer`. Pro případ řetězce, by to pak byl typ `string` a hodnota v unionu uložena jako `char*`. Díky tomuto zápisu můžeme vracet různé hodnoty podle nahráného datového typu.

Dále zde nalezneme výčet všech možných stavů, do kterých se lexikální analyzátor může dostat. Tento seznam je shodný s konečným automatem, který je umístěn v kapitole Přílohy.

2.1.2 LEXICAL_ANALYSIS.C

Obsahuje implementaci funkcí pro práci s lexémy. Hlavní funkce, která parsuje lexémy se nazývá `getNextSymbol` a její předpis je následující: `struct Symbol getNextSymbol(FILE*);` Můžeme si tedy všimnout, že parametrem je datový typ `FILE*`, neboli ukazatel na soubor, ze kterého máme číst vstupní kód (nebo hodnota `stdin`). Parsování probíhá v neustále se opakujícím `while` cyklu, který se pomocí přepínače posouvá ze stavu do stavu podle nahrávaných hodnot. Čtení se provádí po jednotlivých znacích a je neustále vyhodnocována správnost nahrávaného lexému. Ten je obvykle ukončen mezerou, znakem konce řádku, nebo znakem konce souboru. V takovém případě pak vrátíme zjištěná data pomocí struktury `Symbol`.

2.2 Syntaktický a sémantický analyzátor

Syntaktický analyzátor a sémantický analyzátor dohromady tvoří celek nazývaný též parser. Jedná se o nejsložitější a nejdůležitější část celého překladače.

2.2.1 Syntaktický analyzátor

Syntaktický analyzátor je implementovaný metodou rekurzivního zostupu. Syntaktická analýza je říadená pomocí pravidel LL-gramatiky (viz obr. 2.2.1.1). Na základě pravidel byla navrhnutá LL tabulka (viz obr

2.2.1.2), vďaka ktorej bolo možné použiť dané pravidlá pre rôzne situácie. Problém pri našej navrhnutej gramatike je, že to nie je LL(1) gramatika. Je to kvôli pravidlám 15, 16, 17, ktoré reprezentujú priradenie hodnoty do premennej. My nevieme či priradzujeme výraz, voláme používateľom definovanú funkciu alebo voláme vstavanú funkciu. Tento problém sme vyriešili tak, že sme si pred-načítali jeden Token, na základe ktorého sme sa rozhodli, ktoré pravidlo použijeme. Syntaktický analyzátor je implementovaný ako jednopriechodový. V jednom priechode vo vstupnom programe sa vykoná kontrola syntaxe a sémantiky, plnenie tabuľky symbolov a plnenie zoznamu inštrukcií. Pravidlá sme implementovali pomocou funkcií deklarovaných v súbore *syntax_analysis.h* a definovaných v súbore *syntax_analysis.c*. Na základe LL-tabuľky sa funkcie medzi sebou volajú (môžu aj rekurzívne) a tým simulujú vytváranie abstraktného syntaktického stromu pre každé pravidlo čo vedie k syntaktickej analýze celého vstupného súboru.

2.2.2 Syntaktický analyzátor pro výrazy

Pro zpracování výrazů jsme zvolili metodu precedenční syntaktické analýzy. Tato metoda využívá model zásobníkového automatu. Její součástí je tzv. precedenční tabulka. V té je v podstatě zapsáno, kdy použít jaké pravidlo. Volba padla právě na tuto metodu, neboť je jednoduše implementovatelná a pro naše účely dostačující. Vstupem je řetězec, u kterého se kontroluje syntaktická správnost. Výstupem je pravý rozbor, což je posloupnost použitých pravidel v nejpravější derivaci, ale v opačném pořadí. Základem pro tvorbu precedenční tabulky je priorita a asociativita. Podmínkou je také to, aby žádné ze zadaných gramatických pravidel neobsahovalo ϵ -pravidla a žádné dvě pravidla neměla stejnou pravou stranu pravidla.

Implementace:

Precedenční tabulku jsme zmenšili z větších rozměrů na rozměry 7x7. To protože operátory + a – nebo *,/,// mají stejnou jak prioritu, tak asociativitu. Z toho důvodu pro ně platí stejná pravidla a tak jsme je mohli umístit do stejného řádku, případně sloupce. To samé platí pro všechny relační operátory. Implementace je ve zdrojových souborech *expression.h* a *expression.c*. Tento analyzátor je také úzce provázán se strukturou zásobník, která je naimplementována v souborech *stack.h* a *stack.c*. Hlavní část precedenční analýzy je naimplementována ve funkci `Expression()`. Tato funkce je volána syntaktickým analyzátozem tehdy, když narazí na výraz. Ta vyhodnotí pomocí precedenční tabulky, jestli se provede shift, reduce, chyba ve výrazu nebo konec výrazu. Implementována je pomocí do-while cyklu. Pro získání symbolu z tabulky je používána další funkce `get_precedence_table_symbol()`. V případě stavu shift, který je označen jako stav P (posun), funkce provede push symbolu shift (i, v kódu je označen jako `SYMBOL_SHIFT`) za nejvrchnější terminál na zásobníku a push symbolu ze vstupu na zásobník. V případě stavu reduce, označen jako R, funkce zavolá funkci `reduction()`. Ta má za úkol nalézt nejvrchnější symbol shift. Mezi tímto symbolem a vrcholem nalezne pravou stranu nějakého pravidla. To provede pomocí funkce `get_precedence_table_rule()`. Dále odstraní tuto část na zásobníku včetně symbolu shift, což zajistí funkce `Item_to_pop()`. Nakonec nahradí levou stranou pravidla. Při stavu equal, označen jako E, se provede push symbolu ze vstupu na zásobník. Stav K značí konec vyhodnocování výrazu a stav X značí chybu. Celý cyklus do-while probíhá do doby, než na nejvrchnějším terminálu zásobníku a na vstupním symbolu nejsou dva symboly \$ (v kódu jako `SYMBOL_DOLLAR`). Vše bylo implementováno pomocí algoritmu pro precedenční syntaktickou analýzu.

```

//|+- /* /| r | ( | ) | i | $ |
{ R, P, R, P, R, P, R }, // "+, -"
{ R, R, R, P, R, P, R }, // "*, /, //"
{ P, P, X, P, R, P, R }, // r -> "<, <=, >, >=, ==, !="
{ P, P, P, P, E, P, X }, // "("
{ R, R, R, X, R, X, R }, // ")"
{ R, R, R, X, R, X, R }, // i -> "ID, INT, FLOAT, STRING"
{ P, P, P, P, X, P, K } // "$"

```

Obrázek 1: Precedenční tabulka

2.2.3 Sémantický analyzátor

Sémantický analyzátor zabezpečuje sémantické kontroly vstupného programu. Je využívaná tabuľka symbolov, konkrétne funkcie `SymTableSearch()` a `SymTableInsert()`, vďaka ktorým sme boli schopný vykonať sémantické kontroly (redefinícia, počet parametrov vo funkcii, typová kontrola...). Sémantický analyzátor je implementovaný súčasne so syntaktickým analyzátorom vo funkciách, ktoré simulujú pravidlá gramatiky.

2.3 Generátor kódu

Cílový kód `ifjcode19` je generovaný na standardní výstup až po úspešnej lexikálnej, syntaktickej a sémantickej analýze prekladača. Funkcie generátoru je implementovaná v najmä v súbore `generator.c` a `generator.h` spoločne s jednou funkciou v súbore `syntax_analysis.c`.

2.3.1 Generovanie instrukcií

Generovanie jednotlivých instrukcií je riešené volaním funkcie `generateInstruction()`, ktorá do ADT jednosmerne väzanej tabuľky vkladá položky s informáciami o type instrukcie a jej operandoch. Toto volanie je vykonávané na príslušných miestach priamo v syntaktickom analyzátor.

2.3.2 Výpis kódu

Výpis cílového kódu na standardní výstup je riešený funkciou pre tlač instrukcií v súbore `generator.c`. Táto funkcia postupne prechádza celú tabuľku instrukcií a na základe atribútov jednotlivých položiek každú vytiskne v príslušnom formáte.

2.4 Ostatné časti prekladača

2.4.1 Tabuľka symbolov

Tabuľka symbolov je implementovaná ako tabuľka s rozptýlenými položkami (hashovaná tabuľka). Znalosti o jej implementácii sme využili z predmetov IAL a IJC. Je implementovaná v súboroch `symtable.h` a `symtable.c`. Vytvárame dve inštancie tabuľky symbolov

Globálna tabuľka symbolov – veľkosť 7901 položiek

Lokálna tabuľka symbolov – veľkosť 313 položiek

Vďaka dvom inštanciám tabuľky, dokážeme vykonávať sémantické kontroly v globálnom a aj lokálnom kontexte vstupného programu. Veľkosti tabuľiek sme sa rozhodli stanoviť na základe efektivity rozptyľovacej funkcie tak, aby sme zabránili sekvencnému vyhľadávaniu a tým dosiahli vyššiu rýchlosť vyhľadávania.

Dôležitou časťou tabuľky symbolov je jej položka `SymTableItem`, ktorá obsahuje:

`Type` – typ identifikátora (premenná, parameter, funkcia)

`NumberOfParameters` – počet parametrov funkcie

`ActualParameter` – aktuálny parameter funkcie

`SymData` – Token, ktorý obsahuje typ a dáta identifikátora

`SymItemNext` – ukazovateľ na ďalšiu položku v tabuľke

Vďaka položke dátum v položke tabuľky symbolov si dokážeme uchovať dôležité dáta o identifikátoroch, ktoré následne slúžia ako základ pre sémantickú kontrolu.

2.4.2 Zásobník

Struktúra zásobníka je naimplementovaná v už zmiňovaných súboroch `stack.h` a `stack.c`. Naimplementované má ako štandardnú funkciu zásobníka, tak funkcie pre doplnenie chodu ostatných častí. Jsou jimi:

`sInit()` – funkce pro inicializaci zásobníku.
`sPush()` – funkce pro push prvku na zásobník.
`*sTop()` – funkce pro zjištění nejvrchnějšího prvku na zásobníku.
`sLexTop()` – funkce pro zjištění nejvrchnějšího prvku na zásobníku pre skener.
`sLexPop()` – funkce pro popnutí prvku na zásobníku pre skener.
`*sTopPop()` – funkce pro popnutí nejvrchnějšího prvku na zásobníku.
`sDelete()` – funkce pro smazání zásobníku.
`sDispose()` – funkce pro smazání všech prvků zásobníku.
`sEmpty()` – funkce pro zjištění, zdali je zásobník prázdný.
`TopTerminal()` – funkce pro získání nejvrchnějšího terminálu na zásobníku. Je využívána v precedenční syntaktické analýze.
`Push_TopTerminal()` – funkce pro push nejvrchnějšího terminálu na zásobníku. Je rovněž využívána v precedenční syntaktické analýze.

3 Spolupráce v týmu

V týmu se někteří z nás znali již před projektem. Se zbylými členy týmu jsme se dali dohromady skrz platformu Discord. Problémy a komunikaci jsme řešili především pomocí Facebooku, Discordu a osobními setkáními. Ke sdílení zdrojových souborů pro práci jsme využili známý Github.

3.1 Rozdělení práce

Vojtěch Mimochodek – Precedenční syntaktický analyzátor pro výrazy, zásobník, testování a dokumentace.
Dávid Špavor – Syntaktický analyzátor, sémantický analyzátor, lexikální analyzátor, tabulka symbolů, testování a dokumentace.
Vojtěch Jurka – Generování kódu, testování a dokumentace.
Martina Tučková – Lexikální analyzátor, testování a dokumentace.

3.2 Rozdělení bodů

Body jsme se snažili rozdělit rovnoměrně mezi všechny týmu, s výjimkou Dávida Špavora, který pracoval také na lexikálním analyzátoru a pomohl tak Martině dokončit podstatnou část tohoto dílu projektu. Z tohoto důvodu má 30

4 Závěr

Ze začátku projektu jsme dané problematice příliš nerozuměli. Postupem času, kdy v předmětu IFJ a IAL byla látka postupně probírána, vše dávalo větší smysl. Byli jsme rádi, že jsme stihli obě pokusné odevzdání, což nám dalo dobrou zpětnou vazbu, a tudíž jsme věděli, na čem je potřeba ještě zapracovat. Naprosto přesně zde funguje to, že je důležité nejprve velmi dobře porozumět dané problematice, vědět co se požaduje a pak až začít programovat. Projekt bychom celkově zhodnotili jako výborný úvod do odvětví překladačů. Dále také jako skvělou zkušenost, která nás v umění programování posunula zase o kousek dál. V neposlední řadě bylo fajn poznat a pracovat s novými lidmi a nalézt si tak nové přátele.

5 Přílohy



```

(1)  PROG → STAT PROG
(2)  PROG → FUNC PROG
(3)  PROG → ε
(4)  STAT → ID
(5)  STAT → KEYWORDS
(6)  STAT → BUILTIN
(7)  STAT → eol
(8)  KEYWORDS → IF
(9)  KEYWORDS → WHILE
(10) KEYWORDS → PASS
(11) KEYWORDS → RETURN
(12) KEYWORDS → NONE
(13) FUNC → def id "(" PARAMS ")" ":" eol INDENT STAT DEDENT
(14) FUNCCALL → id "(" PARAMS ")" eol
(15) ID → id "=" expr
(16) ID → id "=" FUNCCALL
(17) ID → id "=" BUILTIN
(18) IF → if expr ":" eol INDENT STAT DEDENT ELSE
(19) WHILE → expr ":" eol INDENT STAT DEDENT
(20) PARAMS → ε
(21) PARAMS → params "," PARAMS
(22) INDENT → indent
(23) DEDENT → dedent
(24) ELSE → ε
(25) ELSE → else ":" eol INDENT STAT DEDENT
(26) INPUTS → inputs "(" ")" eol
(27) INPUTI → inputi "(" ")" eol
(28) INPUTF → inputf "(" ")" eol
(29) PRINT → print "(" PARAMS ")" eol
(30) SUBSTR → substr "(" s " " i " " n ")" eol
(31) LEN → len "(" s ")" eol
(32) ORD → ord "(" s " " i ")" eol
(33) CHR → chr "(" i ")" eol
(34) RETURN → return
(35) BUILTIN → INPUTS
(36) BUILTIN → INPUTI
(37) BUILTIN → INPUTF
(38) BUILTIN → PRINT
(39) BUILTIN → LEN
(40) BUILTIN → SUBSTR
(41) BUILTIN → ORD
(42) BUILTIN → CHR
(43) PASS → pass eol
(44) NONE → none eol

```

Obrázek 3: Obr 2.2.1.1 LL-gramatika

Sheet	0	ed	def	id	"e"	"e"	"e"	"e"	exr	if	params	"e"	indent	dedent	else	inputs	input1	input2	input3	print	subst	s	i	n	len	ord	dr	return	\$
T0		11.10.1	12.10.2	11.10.1					11.10.1	11.10.1						11.10.1	11.10.1	11.10.1	11.10.1	11.10.1	11.10.1				11.10.1	11.10.1	11.10.1	11.10.1	€ 3
T1		ed, 7		13.4					14.5	14.5						15.6	15.6	15.6	15.6	15.6	15.6				15.6	15.6	15.6	14.5	
T2			16.17.18.19.13	15.11.16.11.17																									
T3									112.9	111.8																		114.11	
T4																													
T5																													
T6						€ 20					16.21																		
T7													22																
T8		ed, 7		124.4					125.5	125.5																		125.5	
T9														23															
T10				16.14																									
T11										17.18.12.17.28.18																			
T12									17.18.19.19																				
T13																													
T14																												34	
T15																													
T16																													
T17																													
T18																													
T19																													
T20																													
T21																													
T22																													
T23																													
T24				15.11.16.11.17																								33	
T25									131.9	130.8																		133.11	
T26																													
T27														23															
T28		€ 24	€ 24	€ 24					€ 24	€ 24					17.18.19.25	€ 24	€ 24	€ 24	€ 24	€ 24	€ 24				€ 24	€ 24	€ 24	€ 24	
T29				16.14																									
T30																													
T31									17.18.14.5.19	17.18.14.3.14.18																			
T32																													
T33																													
T34																												34	
T35																													
T36																													
T37																													
T38																													
T39																													
T40																													
T41																													
T42																													
T43																													
T44																													
T45																													

Obrázek 4: Obr 2.2.1.2 LL-tabul'ka