Dokumentácia k projektu z predmetov IFJ a IAL

Implementace překladače imperativního jazyka IFJ22

Tým xbucka00, varianta TRP

Timotej Bučka	`xbucka00`	25%
Daniel Mačura	`xmacur09`	25%
Tobiáš Štec	`xstect00`	25%
Patrik Früštök	`xfrust00`	25%

1.december.2022

Obsah

Spôsoby riešenia jednotlivých častí projektu

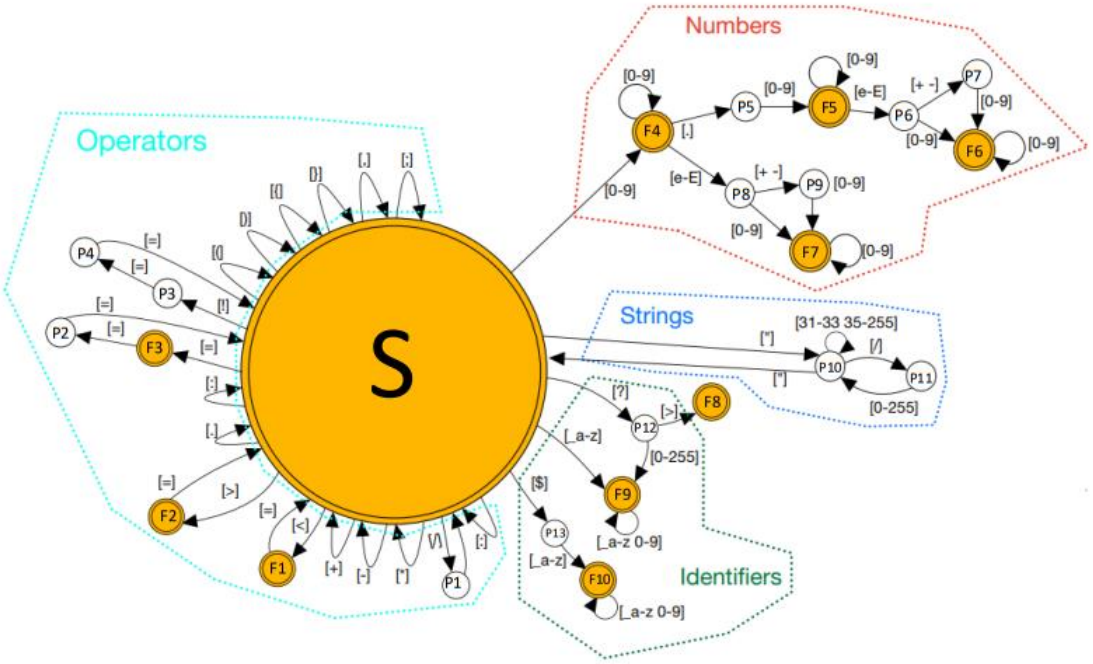
1. Lexikálna analýza

Lexikálnu analýzu sme implementovali s pomocou podporných štruktúr a funkcií zo súborov ***lexer.c*** a ***lexer.h*.** V súbore ***token.h*** máme definovanú dátovú štruktúru `token`, ktorá sa skladá z typu (`ID`) a atributu hodnoty (`VAL`), ktorá uchováva reťazcovú reprezentáciu daného tokenu, resp. identifikátor kľúčového slova.

Hlavnou funkciou našej lexikálnej analýzy je `lexer_next_token`, ktorá je volaná z makra `next_tok`. V tomto makre sa získaný token pridá do nášho dvojito viazaného zoznamu `dll`. Funkcia `lexer_next_token`, načíta zo súboru token po tokene, a postupne každému pridá jeho typ (`ID`), na základe deterministického konečného automatu (**viz. 1.1**) a jeho stavov, implementovaného priamo vo funkcii ako switch, ktorý sa opakuje pokiaľ zo vstupu nie je načítaný znak `EOF`. Jednotlivé prípady `case` predstavujú už samotné stavy automatu. Ak jeden z načítaných znakov nie je validný, program vráti hodnotu `LEXICAL_ERR` (==1) a ukončí proces. Inak sa prechádza do ďalších stavov a načítajú sa ďalšie znaky, až do chvíle kedy už máme hotový ďalší token, ktorý vraciame a ukončujeme túto funkciu.

Po načítaní tokenu sa prevedie kontrola, či sa nejedná o kľúčové slovo alebo identifikátor už predom známy v jazyku IFJ22.

1.1 Konečný stavový automat lexikálnej analýzy



Legend:

S – STATE_START

F1 - STATE_ST_E
F2 - STATE_GT_E
F3 - STATE_EQ_E
F4 - STATE_INTEGER_E
F5 - STATE_DOUBLE_E
F6 - STATE_DOUBLE_EXPONENT_E
F7 - STATE_INTEGER_EXPONENT_E
F8 - STATE_CLOSING_TAG
F9 - STATE_IDENTIFIER_OR_KEYWORD_E
F10 - STATE_VARIABLE_E

P1 - STATE_SLASH
P2 - STATE_EQEQ
P3 - STATE_NOT
P4 - STATE_NOT_EQ
P5 - STATE_INTEGER_AND_SEPARATOR
P6 - STATE_DOUBLE_EXPONENT_START
P7 - STATE_DOUBLE_EXPONENT_SIGN
P8 - STATE_INTEGER_EXPONENT_START
P9 - STATE_INTEGER_EXPONENT_SIGN
P10 - STATE_QUOTATION_CENTER_E
P11 – STATE_QUOTATION_ESCAPE_CHAR
P12 - STATE_QUESTIONMARK
P13 - STATE_VARIABLE_START

2.Syntaktická a sémantická analýza

Najdôležitejšiou časťou celého programu je syntaktická analýza.

2.1 Implementačné detaily

Hlavnú kostru syntaktickej a sémantickej analýzy tvoria funkcie definivané v súbore ***analysis.c***.

Vo funckii `run_analysis` dochádza k inicializácii globálnej a lokálnej tabuľky symbolov implementovanými ako TRP. Do globálnej tabuľky sú vo funkcii `preload_hash_table` postupne vkladané preddefinované funkcie.

Zvolili sme metódu rekurzívneho zostupu. Derivačné kroky simulujeme pomocou rekurzívnho volania funkcií na základe jednotlivých gramatických pravidiel. Za pomoci backtrackingu, ktorý je možné realizovať vďaka uloženiu tokenov v dvojito spojenom zozname `dll`, rozhodneme, ktoré gramatické pravidlo bude použité. Každá funkcia volá makro `next_tok`, ktoré sa rozšíri do volania funkcie `lexer_next_token` v situácii, kedy je aktívny element rovný poslednému elementu `dll`, inak je aktivita posunutá na ďalší element v `dll` a jeho hodnota je navrátená. Na základe získaného typu tokenu, daná funkcia rozhoduje o syntaktickej validite danej postupnosti tokenov. V prípade, že je očakávaný výraz, je zavolaná funkcia `parse_expression`, ktorá má na starosti syntaktickú kontrolu výrazov. Ak sa analýza nachádza v stave, kde je iný rámec (scope) ako rámec hlavnej funkcie, je inkrementovaná globálna premenná `BODYRECURSIONCOUNT`, ktorá zaistí, že sa bude pracovať s lokálnou tabuľkou symbolov miesto globálnej.

Pri deklarácii funkcie je do globálnej tabuľky symbolov pridaný prvok reprezentujúci danú funkciu s názvom, parametrami a

návratovým typom funkcie. To isté platí aj pre premenné, kde je daná premenná pridávaná vždy do správnej tabuľky symbolov (globálna alebo lokálna) určenej stavom premennej `BODYRECURSIONCOUNT`.

Za priebehu syntaktickej analýzy kontrolujeme jednotlivé sémantické pravidlá a predpoklady. Funkcia `parse_expression` prijíma ako jeden z parametrov ukazateľ na `data_type`, ktorého hodnotu mení na výsledný typ výrazu. Vo vnútri funkcie je kontrolovaná typová kompatibilita operátorov a pri výskyte premennej vo výraze dochádza k vyhľadaniu v tabuľke symbolov pomocou `hash_table_lookup`. Pri syntaktickej kontrole volania funkcie je zavolaná funkcia `hash_table_lookup` nad globálnou tabuľkou symbolov, a je skontrolovaný správny počet a typ argumentov.

Syntaktická analýza končí neúspechom, ak je niektoré z pravidiel gramatiky porušené, a správna chybová hodnota je zapísaná do globálnej premennej `ERROR`. Hodnota `ERROR` je vrátená hlavnou funkciou `main` a v prípade že analýza prebehne bez vzniku syntaktických alebo sémantických chýb je vrátená hodnota `SUCCESS` (0).

2.2 Gramatické pravidla

(1) BODY → ε
(2) BODY → "function" func_id "(" PARAMETERS ")" ":" TYPE "{" BODY "}" BODY
(3) BODY → "if" "(" EXPRESSION ")" "{" BODY "}" "else" "{" BODY "}" BODY
(4) BODY → "while" "(" EXPRESSION ")" "{" BODY "}" BODY
(5) BODY → "return" RETURN ";" BODY
(6) BODY → FUNCTION_CALL ";" BODY
(7) BODY → ASSIGNMENT ";" BODY
(8) BODY → EXPRESSION ";" BODY
(9) ASSIGNMENT → var_id "=" ASSIGNMENT_PRIME
(10) ASSIGNMENT_PRIME → EXPRESSION
(11) ASSIGNMENT_PRIME → FUNCTION_CALL
(12) FUNCTION_CALL → func_id "(" ARGUMENTS ")"
(13) RETURN → ε
(14) RETURN → EXPRESSION
(15) RETURN → FUNCTION_CALL
(16) TYPE → PREFIX C_TYPE
(17) PREFIX → ε
(18) PREFIX → "?"
(19) C_TYPE → "int"
(20) C_TYPE → "float"
(21) C_TYPE → "string"
(22) C_TYPE → "void"
(23) ARGUMENTS → ε
(24) ARGUMENTS → EXPRESSION ARGUMENTS_PRIME
(25) ARGUMENTS_PRIME → ε
(26) ARGUMENTS_PRIME → "," EXPRESSION ARGUMENTS_PRIME
(27) PARAMETERS → ε
(28) PARAMETERS → C_TYPE var_id PARAMETERS_PRIME
(29) PARAMETERS_PRIME → ε
(30) PARAMETERS_PRIME → "," C_TYPE var_id PARAMETERS_PRIME

2.3 LL(1) tabulka

	"function"	func_id	"("	")"	":"	"{"	"}"	"else"	"while"	"if"	EXPR	"return"	";"	var_id	"="	"?"	"int"	"float"	"string"	"void"	","	$
BODY	2	12					1		4	3	8	5		9								1
ASSIGNMENT														9								
ASSIGNMENT_PRIME		11									10											
FUNCTION_CALL		12																				
RETURN		15									14		13									13
TYPE																16	16	16	16	16		
PREFIX																18						17
C_TYPE																	19	20	21	22		
ARGUMENTS				24							24											23
ARGUMENTS_PRIME				25																	26	
PARAMETERS				27													28	28	28	28		
PARAMETERS_PRIME				29																	30	

2.4 Spracovanie výrazov

Spracovanie výrazov sa deje oddelene od spracovania syntaxe, a to precedenčnou analýzou. Funkcie starajúce sa o toto spracovanie nájdeme v súbore ***expressions.c.*** Hlavnou funkciou je funkcia `parse_expression`, ktorá je volaná hlavným parserom v prípade, že parser narazí na výraz.

Dôležitou časťou prec. analýzy je zásobník reprezentovaný štruktúrou `expr_stack` pozostávajúcou z dĺžky zásobníka a zo štruktúry `expr_item`, ktorá reprezentuje jednotlivé prvky zásobníka. `expr_item` môže reprezentovať tri typy prvkov vyskytujúcich sa v precedenčnej analýze: znak dolár (znak neznámy v kontexte spracovania výrazu), terminál alebo neterminál.

Princíp fungovania precedenčnej analýzy je nasledovný: Na začiatku `parse_expression` je do zásobníku vložený znak dolár. Následne je cykle volané makro `next_tok` a na základe získaného tokenu je podľa precedenčnej tabuľky určené, ako sa má v analýze pokračovať. V tabuľke sa nachádzajú symboly:

- = - Vlož získaný token na zásobník a načítaj ďalší token
- < - Nastav najvrchnejšiemu termu na zásobníku príznak `breakpoint`, vlož získaný token na zásobník a načítaj ďalší token
- > - Vykonaj redukciu podľa známych pravidiel
- Prázdne políčko – chyba pri syntaktickej analýze

Analýza končí úspechom v prípade že načítaný token je interpretovaný ako dolár a na zásobník je v stave: dolár, neterminál (vrchol).

2.5 Precedenčná tabuľka výrazov

	$	(	)	+	-	*	/	.	===	!==	<	>	<=	>=	id
$		<		<	<	<	<	<	<	<	<	<	<	<	<
(		<	=	<	<	<	<	<	<	<	<	<	<	<	<
)	>		>	>	>	>	>	>	>	>	>	>	>	>	
+	>	<	>	>	>	<	<		>	>	>	>	>	>	<
-	>	<	>	>	>	<	<		>	>	>	>	>	>	<
*	>	<	>	>	>	>	>		>	>	>	>	>	>	<
/	>	<	>	>	>	>	>		>	>	>	>	>	>	<
.	>	<	>					>	>	>	>	>	>	>	<
===	>	<	>	<	<	<	<	<							<
!==	>	<	>	<	<	<	<	<							<
<	>	<	>	<	<	<	<	<							<
>	>	<	>	<	<	<	<	<							<
<=	>	<	>	<	<	<	<	<							<
>=	>	<	>	<	<	<	<	<							<
id	>		>	>	>	>	>	>	>	>	>	>	>	>	

2.6 Zjednodušená precedenčná tabuľka výrazov

	$	(	)	+-	*/	.	rop	id
$		<		<	<	<	<	<
(		<	=	<	<	<	<	<
)	>		>	>	>	>	>	
+-	>	<	>	>	<		>	<
*/	>	<	>	>	>		>	<
.	>	<	>			>	>	<
rop	>	<	>	<	<	<		<
id	>		>	>	>	>	>	

3. Tabuľka symbolov

V súboroch ***symtable.c*** a ***symtable.h*** sme implementovali tabuľku s rozptýlenými položkami (TRP) ako variantu zadania.

TRP sme sa rozhodli implementovať formou explicitného zreťazenia. Synonymá sme reťazili pomocou lineárne viazaných zoznamov.

Ako veľkosť mapovacieho poľa sme zvolili číslo prvočíslo ***49 157***, kvôli jeho vhodným vlastnostiam[1] a použili ***polynomial rolling hash*** mapovaciu funkciu. Generovaný hash sme zkomprimovali za pomoci ***MAD compression***[2], aby sa s ním dalo indexovať v tabuľke.

Tabuľka obsahuje položky typu `table_item_data`, ktoré reprezentujú funkciu alebo premennú. Taktiež obsahujú názov, ktorý sa aj vyžíva pre gerenovanie hash funckie

4. Generovanie kódu

Generovanie cieľového kódu IFJcode22, sme implementovali pomocou súboru ***generator.c***, ktorý je volaný v priebehu syntaktickej analýzy za chodu programu.

Ak sa token zhoduje s jednou z funkcií jazyka ***IFJ22***, zavolá sa príslušná funkcia definovaná ako makro, obsahujúca stringovú reprezentáciu danej funkcie v ***IFJcode22***. Okrem funkcií generujeme aj výstupný kód pre príkazy známe v jazyku ***IFJ22*** (napr. if, while, príkazy pre matematické operácie...).

Na záver, na základe vopred zadaného kódu, program vygeneruje a vypíše jeho trojadresný kód na štandardný výstup.

[1] https://planetmath.org/goodhashtableprimes
[2] http://www.cs.emory.edu/~cheung/Courses/253/Syllabus/Map/hash-func.html

5. Vlastné telo prekladača

V hlavnej funkcii ***main*** inicializujeme potrebné dátové štruktúry a ďalej voláme funkciu ***run_analysis***, ktorá ďalej riadi činnosť prekladača. Na konci je vrátený správny kód chyby a uvoľnené predtým alokované miesto.

Práca v tíme

Na zložení tímu sme sa dohodli ešte počas prvého týždňa semestra, nakoľko sme sa už všetci poznali.

Na projekte sme začali pracovať ku koncu septembra.
Z počiatku sme sa stretávali všetci spolu aby sme projektu a jednotlivým častiam pochopili všetci a mohli si tak rozdeliť našu prácu.

Popri súčasným prednáškam nám veľmi pomohli aj záznamy z minulého roku, vďaka ktorým sme mohli rýchlejšie postupovať.

Po spoločnom zhodnotení a pochopení častiam projektu sme si postupne začali prácu deliť a pracovať jednotlivo prípadne vo dvojiciach.

Prácu sme si rozdelili rovnomerne, každý člen teda dostal percentuálně hodnotenie 25%.

Po dukunčení jednotlivých častí, sme si otestovali naše riešenie pokusným odovzdaním, po ktorom sme pokračovali v opravách chýb. Nakoniec sme sa všetci spojili a vypracovali ku projektu túto dokumentáciu.

6. Rozdelenie práce mezi členmi tímu

6.1 Timotej Bučka (xbucka00)

- Dokumentácia, syntaktická analýza, sémantická analýza, analýza výrazov

6.2 Patrik Früštök (xfrust00)

- Dokumentácia, lexikálna analýza, tvorba testov, pridávanie komentárov

6.3 Daniel Mačura (xmacur09)

- Dokumentácia, syntaktická analýza, sémantická analýza, generácia kódu

6.4 Tobiáš Štec (xstect00)

- Dokumentácia, lexikálna analýza, syntaktická analýza, generácia kódu