



Dokumentácia k projektu
z IFJ/IAL
**IMPLEMENTÁCIA PREKLADAČA
JAZYKA IFJ20**
Tím 123, varianta I

Členovia tímu:

Martin Novotný Mlinárčsik (xnovot1r) **25%**

Šimon Feňko (xfenko01) **25%**

Andrej Hýroš (xhyros00) **25%**

Peter Čellár (xcella01) **25%**

9. decembra 2020

Obsah

1	Úvod	2
2	Etapy projektu	2
3	Práca v tíme	2
3.1	Komunikácia v tíme	2
3.2	Správa kódu	2
3.3	Rozdelenie práce	2
4	Implementácia	3
4.1	Lexikálna analýza	3
4.2	Syntaktická analýza - rekurzívny zostup	3
4.3	Precedenčná syntaktická analýza	3
4.4	Sémantická analýza	3
4.5	Generovanie kódu	3
4.6	Testovanie	3
4.7	Tvorba Dokumentácie	4
5	Dátové štruktúry	4
5.1	Obojsmerne viazaný zoznam tokenov	4
5.2	Binárne vyhľadávacie stromy	4
5.3	Zasobník ukazateľov na binárne stromy	4
5.4	Zasobník precedenčnej analýzy	4
6	Zhrnutie	4
7	Prílohy	5

1 Úvod

Cieľom projektu[1] bolo vytvorenie prekladača v jazyku C, ktorý prečíta zdrojový kód zapísaný v zdrojovom jazyku IFJ20, ktorý je zjednodušenou podmnožinou jazyka Go a preloží ho do cieľového jazyka IFJcode19 (medzikód). Vybrali sme si variantu I. a to implementovanie tabuľky symbolov pomocou binárneho vyhľadávacieho stromu.

Osobitná vďaka patrí Ing. Zbynkovi Křivkovi Ph.D, ktorý nám pohotovo odpovedal na všetky naše dotazy cez Discord.

2 Etapy projektu

1.Návrh štruktúry programu:

- a) konečný automat⁷
- b) precedenčná tabuľka⁷
- c) LL tabuľka⁷
- d) LL gramatika⁷

2.Lexikálna analýza^{4.1}

3.Syntaktická analýza - rekurzívny zostup^{4.2}

4.Precedenčná syntaktická analýza^{4.3}

5.Sémantická analýza^{4.4}

6.Generovanie kódu^{4.5}

7.Testovanie^{4.6}

8.Tvorba Dokumentácie^{4.7}

3 Práca v tíme

3.1 Komunikácia v tíme

Ako hlavný komunikačný kanál sme zvolili discord (hovory), poprípade messenger. Kvôli zhoršenej epidemiologickej situácii, sme úplne obmedzili osobné stretnutia, čo sa odrazilo na práci, ktorá bola trochu náročnejšia.

3.2 Správa kódu

Ako verzovací systém sme použili Git, hostovaný na službe Github. Každý z členov už pracoval s Gitom v minulosti na inom projekte, takže nám to prišlo najjednoduchšie. Tento spôsob nám vyhovoval najviac, pretože každý člen tímu mohol pridávať a upravovať zdrojový kód.

3.3 Rozdelenie práce

Martin Novotný Mlinárčsik: Syntaktická analýza, precedenčná analýza, testovanie

Šimon Feňko: Lexikálna analýza, Konečný automat, Dokumentácia, testovanie

Andrej Hýroš: Sémantická analýza, práca s dátovými štruktúrami, testovanie

Peter Čellár: Sémantická analýza, práca s dátovými štruktúrami, testovanie

4 Implementácia

4.1 Lexikálna analýza

Lexikálny analyzátor (LA), bola časť projektu, ktorú sme implementovali ako prvú. Je navrhnutý ako konečný automat, podľa ktorého bol lexikálny analyzátor implementovaný. Ten postupne načíta jednotlivé znaky. Zaviedli sme obojsmernú komunikáciu medzi syntaktickým analyzátorom (SA) a (LA). SA bol potom schopný čítať tokeny a tiež vracať naspäť do LA. Tokeny boli čítané pri volaní LA zo štandardného vstupu. Pokiaľ sa analyzátor nedostane do požadovaného stavu, vracia lexikálnu chybu (hodnota 1).

4.2 Syntaktická analýza - rekurzívny zostup

Na implementáciu syntaktického analyzátoru bola použitá metóda rekurzívneho zostupu, ktorá vychádza z vytvorenej LL gramatiky.

Na začiatku bola zostavená LL gramatika, z ktorej bola následne odvodená LL tabuľka. Pre každé pravidlo bola vytvorená funkcia, ktorá simuluje dané pravidlo. Okrem výrazov parser ukladá všetky tokeny aj do ďalšieho zoznamu, ktorý pošle sémantickej analýze.

4.3 Precedenčná syntaktická analýza

Výrazy sú v programe vyhodnocované za pomoci precedenčnej analýzy. Syntaktický analyzátor ukladá tokeny výrazu do obojsmerne viazaného listu, ktorý je odoslaný funkciou precedenčnej analýzy, kde vystupuje ako vstup. Algoritmus je založený na algoritme z prednášok, kde sa tokeny postupne ukladajú zo vstupu na zásobník a na základe precedenčnej tabuľky je určované, v akom poradí sa bude výraz vyhodnocovať.

4.4 Sémantická analýza

Sémantická analýza využíva dva prechody cez zoznam tokenov. Prvý prechod uloží do globálneho stromu informácie o funkciách. Druhý prechod vstúpi do tela funkcií a vykoná analýzu v nich.

Na pracovanie s funkciami používame globálny strom, ktorý obsahuje názov funkcie a dátovú štruktúru `functionData` v ktorej sú uložené dáta o názvoch, počtoch a typov parametrov a návratových hodnôt. Služi na kontrolu či už id funkcie existuje, kontrolu typu argumentov vo volaní funkcie a kontrolu typu vracanej premennej/literálu.

Pre prácu s premennými a literálmi sme vytvorili lokálny strom, ktorý obsahuje názov, typ a hodnotu premennej. Služi na kontrolu či už id premennej existuje a či sa typy premenných a literálov zhodujú.

Pre kontrolu platnosti rámcov sme použili dátovú štruktúru `MainStack`. V nej sú uložené ukazatele na lokálne stromy, ktoré sa vytvoria pri nájdení novej funkcie, `foru`, `ifu`. Pri ukončení funkcie, `foru`, `ifu` sa lokálne stromy uvoľnia zo zásobníku a následne sa zrušia.

4.5 Generovanie kódu

Implementácia sémantickej analýzy nám zabrala viac času ako sme očakávali a preto nám nezostal dostatok času na implementáciu generátoru. Namiesto snahy spraviť generátor aspoň čiastočne čo by mohlo spôsobiť viac škody ako osohu sme sa rozhodli úplne a správne implementovať sémantickú analýzu.

4.6 Testovanie

Testovali sme pravidelne a osobitne každú časť za pomoci kratších a základných testov. Na základe toho sme sa zakaždým uisťovali o správnosti kódu a chyby mohli byť okamžite odstránené.

4.7 Tvorba Dokumentácie

Tvorba dokumentácie patrila medzi finálne časti projektu, ktoré sme robili. Na tvorbu sme použili LaTeX.

5 Dátové štruktúry

5.1 Obojsmerne viazaný zoznam tokenov

Implementovali sme obojsmerne viazaný zoznam tokenov, do ktorého ukladá parser všetky tokeny. Takýto zoznam potom využíva precedenčná analýza(ukladajú sa sem výrazy) a semantická analýza(ukladajú sa sem všetky tokeny zo zdrojového súboru).

Každý prvok zoznamu obsahuje ukazatele na predchádzajúci a nasledujúci prvok zoznamu, a dátovú štruktúru obsahujúcu informácie o samotnom tokene.

5.2 Binárne vyhľadávacie stromy

Implementácia tabuľky symbolov využitím binárnych vyhľadávacích stromov bol požiadavok vyplývajúci zo zadania. Implementovali sme dva druhy stromov, jeden pre deklarované funkcie ukadajúci identifikátor funkcie, počet, mená a typy parametrov, rovnako ako počet a typ návratových hodnôt. Druhý strom ukladá informácie o samotných premenných, menovite identifikátor a typ. Každý uzol obsahuje ukazatele na ľavý a pravý podstrom.

5.3 Zasobník ukazateľov na binárne stromy

Zasobník ukazateľov na binárne stromy bol implementovaný pre potreby kontrolovania deklarácie premenných v rámci programu. Pri implemetácii boli využité znalosti aj časti kódu pochádzajúce z prvej úlohy z predmetu IAL.

5.4 Zasobník precedenčnej analýzy

Zásobník precedenčnej analýzy je súčasťou algoritmu demonštrovaného na prednáškach. Na stack sa ukladajú symboly, ktoré boli spracované pomocou operácií precedenčnej analýzy.

Pozn. Pri implemetácii dátových štruktúr boli využité znalosti aj časti kódu pochádzajúce z úloh z predmetu IAL.

6 Zhrnutie

Približne sme vedeli do čoho ideme keďže dvaja členovia tímu už robili projekt z ifj. Tým sme mali zostavený rýchlo, dohodli sme sa na komunikácii a na verzovacom systéme. Vzhľadom na epidemiologickú situáciu, sme boli nútení úplne obmedziť osobné stretnutia. V dôsledku čoho bola komunikácia zťažnená. Lexikálnu a syntaktickú analýzu sme spravili pomerne rýchlo vzhľadom k preberaniu učiva na prednáškach. Celkovo sme projekt robili priebežne s prednáškami z IFJ a IAL. Sémantická analýza nás zaskočila svojou komplexnosťou a zobrala nám príliš veľa času. Z toho dôvodu sme nestihli spraviť generátor kódu. Náš postup v práci sme priebežne testovali a podľa testov sme ho ladili. Na konci sme otestovali projekt ako celok. Projekt nám dal veľa znalostí o fungovaní prekladačov a prakticky sme si prešli preberanú látku z IFJ a IAL čo nám dalo hlbšie pochopenie učiva. Taktiež nám tento projekt dal cenné skúsenosti svojou komplexnosťou.

7 Prílohy



Obr. 1: Konečný automat

	+	-	*	/	()	==	!=	>=	<=	<	>	i	\$
+	2	2	1	1	1	2	2	2	2	2	2	2	1	2
-	2	2	1	1	1	2	2	2	2	2	2	2	1	2
*	2	2	2	2	1	2	2	2	2	2	2	2	1	2
/	2	2	2	2	1	2	2	2	2	2	2	2	1	2
(1	1	1	1	1	3	1	1	1	1	1	1	1	4
)	2	2	2	2	4	2	2	2	2	2	2	2	4	2
==	1	1	1	1	1	2	2	2	2	2	2	2	1	2
!=	1	1	1	1	1	2	2	2	2	2	2	2	1	2
>=	1	1	1	1	1	2	2	2	2	2	2	2	1	2
<=	1	1	1	1	1	2	2	2	2	2	2	2	1	2
<	1	1	1	1	1	2	2	2	2	2	2	2	1	2
>	1	1	1	1	1	2	2	2	2	2	2	2	1	2
i	2	2	2	2	4	2	2	2	2	2	2	2	4	2
\$	1	1	1	1	1	4	1	1	1	1	1	1	1	4

Obr. 2: Precedenčná tabuľka

	package	main	eol	eof	func	id	()	,	{	}	return	exp	if	else	for	;	:=	=	int	float64	string
PROG	1																					
DEF_FUNC				3	2																	
PARAMS						4		5														
FUNC_RETLIST_BODY							8			9												
TYPE						30														27	28	29
PARAMS_N								7	6													
FUNC_BODY						11		10		11										11	11	11
STAT			20			17						20	20		18		19					
OPTIONAL_RET												14	13									
RETURN_TYPE						31				32										31	31	31
REQUIRED_RED												12										
EXP_N			16						15	16												
ID_N/CALL_FUNC							46		46									45	45			
FOR_DEF						21											22					
FOR_ASSIGN						43				44												
INIT_DEF																						
ID_N									25									23	24			
RETURN_TYPE_N									33	34								26	26			
CALL_FUNC/ASSIGN							36						35									
FUNC_PARAMS								40					39									
OPT_ID						37	38						38									
FUNC_PARAMS_N								42	41													

Obr. 3: LL Tabuľka

1. <PROG> -> package main eol <DEF_FUNC> eof
2. <DEF_FUNC> -> func id (<PARAMS>) <FUNC_RETLIST_BODY> eol
 <DEF_FUNC>
3. <DEF_FUNC> -> ε
4. <PARAMS> -> id <TYPE> <PARAMS_N>
5. <PARAMS> -> ε
6. <PARAMS_N> -> , id <TYPE> <PARAMS_N>
7. <PARAMS_N> -> ε
8. <FUNC_RETLIST_BODY> -> (FUNC_BODY
9. <FUNC_RETLIST_BODY> -> { eol STAT OPTIONAL_RET }
10. <FUNC_BODY> ->) { eol STAT OPTIONAL_RET }
11. <FUNC_BODY> -> RETURN_TYPE { eol STAT REQUIRED_RET eol }
12. <REQUIRED_RET> -> return exp EXP_N
13. <OPTIONAL_RET> -> return eol
14. <OPTIONAL_RET> -> ε
15. <EXP_N> -> , exp EXP_N
16. <EXP_N> -> ε
17. <STAT> -> id ID_N/CALL_FUNC
18. <STAT> -> if exp { eol STAT eol } else { eol STAT } eol STAT
19. <STAT> -> for FOR_DEF ; exp ; FOR_ASSIGN { eol STAT } eol STAT
20. <STAT> -> ε
21. <FOR_DEF> -> id INIT_DEF exp
22. <FOR_DEF> -> ε
23. <INIT_DEF> -> :=
24. <INIT_DEF> -> =
25. <ID_N> -> , id ID_N
26. <ID_N> -> ε
27. <TYPE> -> int
28. <TYPE> -> float64
29. <TYPE> -> string
30. <TYPE> -> id
31. <RETURN_TYPE> -> TYPE RETURN_TYPE_N
32. <RETURN_TYPE> -> ε
33. <RETURN_TYPE_N> -> , TYPE RETURN_TYPE_N
34. <RETURN_TYPE_N> -> ε
35. <CALL_FUNC/ASSIGN> -> exp EXP_N eol STAT
36. <CALL_FUNC/ASSIGN> -> (FUNC_PARAMS) eol STAT
37. <OPT_ID> -> id
38. <OPT_ID> -> ε
39. <FUNC_PARAMS> -> exp FUNC_PARAMS_N
40. <FUNC_PARAMS> -> ε
41. <FUNC_PARAMS_N> -> , exp FUNC_PARAMS_N
42. <FUNC_PARAMS_N> -> ε
43. <FOR_ASSIGN> -> id ID_N = exp EXP_N
44. <FOR_ASSIGN> -> ε
45. <ID_N/CALL_FUNC> -> ID_N INIT_DEF OPT_ID CALL_FUNC/ASSIGN
46. <ID_N/CALL_FUNC> -> (FUNC_PARAMS) eol STAT

Obr. 4: LL Gramatika

Citácie

- [1] Zbyněk Křivka, Lukáš Zobal, Dominika Regéciová. *Formální jazyky a překladače*. [ONLINE]. 2020. URL: <https://wis.fit.vutbr.cz/FIT/st/cfs.php.cs?file=/course/IFJ-IT/projects/ifj2020.pdf>.