VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
FAKULTA INFORMAČNÍCH TECHNOLOGIÍ

Dokumentácia

Implementace překladače imperativního jazyka IFJ20

Tým 139, Varianta I

Riešitelia:

Sabitov Nurzhan, xsabit00 (vedúci) **0%**

Matějka Martin, xmatej55 **33%**

Závodský Ľubomír, xzavod14 **34%**

Kocman Matej, xkocma07 **33%**

6. 12. 2020

1. Rozdelenie práce:

Hneď po zostavení tímu sme si určili komunikačný kanál, čo v našom prípade predstavovala aplikácia Discord. Na prvom online stretnutí sme si prácu rozdelili nasledovne.

- Sabitov Nurzhan mal na starosti vedenie tímu, a generáciu kódu.
- Matějka Martin mal na starosti správu úložiska pre verzovací systém (v našom prípade GitHub) a sémantickú analýzu.
- Závodský Ľubomír ako najskúsenejší tímový člen začal lexikálnou analýzou a nakoniec implementáciou built-in funkcií v IFJcode20.
- Kocman Matej dostal za úlohu syntaktickú analýzu.

Avšak po asi treťom týždni prestal Sabitov Nurzhan komunikovať s tímom a jeho úlohy „zdedil“ Martin Matějka.

2. Úvod

Našou úlohou bolo implementovať prekladač jazyka IFJ20 v jazyku C. Prekladač by mal byť schopný načítať zdrojový kód v jazyku IFJ20 zo štandardného vstupu a následne ho preložiť do cieľového jazyka IFJcode20, pričom by mal byť schopný detegovať nesprávne napísaný kód z pohľadu lexikálneho, syntaktického a sémantického pričom ak rozpozná chybu, vráti jeden z chybových kódov určených v zadaní. Na začiatku sme sa rozhodli, že náš prekladač bude fungovať metódou syntaxou riadeného prekladu, ako aj bolo neskôr doporučené na prednáškach.

3. Implementácia

3.1. Lexikálna analýza

Lexikálna analýza je prvá časť projektu, ktorú sme potrebovali pre funkčný prekladač. Začali sme tým, že sme si navrhli deterministický konečný automat. Tento automat sme upravili potom tak, že podľa lexikálnych pravidiel rozdelí vstupný jazyk IFJ20 na lexémy, ktoré sa následne prevedú na tokeny. Lexémy, ktoré neodpovedajú IFJ20 sú rozpoznané a v takom prípade lexikálna analýza prevedie chybný lexém na token typu chyba a ten je poslaný do syntaktického analyzátoru. V takom prípade vracia prekladač chybový kód 1. Tokeny sa posielajú do syntaktického analyzátora pomocou štruktúry, v ktorej sa tokeny natypujú. Lexikálna analýza taktiež filtruje zo vstupu komentáre a dokumentačné reťazce. Túto metódu sme zvolili, kvôli zjednodušeniu riešenia projektu v ďalších častiach.

3.2. Syntaktická analýza

Pre syntaktickú analýzu sme zvolili metódu rekurzívneho zostupu, kedy každé pravidlo gramatiky má svoju vlastnú funkciu ktoré môže byť volaná rekurzívne. Vytvorili sme si LL tabuľku gramatických pravidiel podľa ktorej sme analýzu zhotovili. Počas zhotovovania syntaktickej analýzy sme nenarazili na žiadne výrazné problémy až na chyby, ktoré prichádzali z lexikálnej analýzy – tá sa vlastne ladila celú dobu trvania projektu. Našu bezproblémovú implementáciu potvrdilo aj vyhodnotenie z nultého odovzdania, kde syntaktická analýza utŕžila skóre 97%. Avšak už pri implementovanej analýze sme narazili na dva problémy v gramatike, ktoré sme vyriešili tzv. „hotfixmi“ a to vlastne vložením menších pomocných pravidiel do existujúcich. Výhodou bolo, že sme nemuseli prerábať celú gramatiku. Jednalo sa napríklad o prípad definície void funkcie bez argumentov štýlom *main()()*. Použité sú pre ňu 3 makrá z toho jedno atypické *PEEK_TOKEN()* ktoré slúžilo v prípade detegovania nasledujúceho tokenu, kde token bol uložený do pomocnej globálnej premennej a pri volaní *GET_TOKEN()* bol znova načítaný.

3.2.1. Precedenčná analýza výrazov

Na parsovanie výrazov sme použili precedenčnú analýzu, pre ktorú bolo treba zhotoviť precedenčnú tabuľku. V precedenčnej analýze sme prevádzali tokeny z lexikálnej analýzy na tokeny pre precedenčnú analýzu, ktoré značne zjednodušili jej implementáciu. Precedenčná tabuľka bola vložená ako dvojrozmerné pole znakov kde figurovali znaky z teórie ako < , >, = a prázdny znak pre nedefinované pravidlo. Vďaka poradiu enumerácie *expr_lexem* sme boli schopní získať pozíciu pravidla v tabuľke značne jednoducho. Precedenčná analýza využíva zásobník implementovaný ako jednosmerne lineárne viazaný zoznam uložený v súbore *stack.c*, samotná analýza je uložená v súbore *expr.c*.

3.3. Sémantická analýza

Sémantickú analýzu v našom prípade predstavujú sémantické kontroly vsadené do syntaktickej analýzy. Využíva veľa globálnych premenných, ktoré indikujú, v akej časti kódu sa analýza nachádza alebo uchovávajú ukazovateľ na globálny strom funkcií, ukazovateľ na zásobník lokálnych stromov apod. Na porovnávanie typu a počtu návratových hodnôt sme využívali štruktúru nafukovacieho poľa implementovaného v *dynstring.h*, ktorý sme vlastne využili ako zásobník s indexáciou. Pri vkladaní novej funkcie či premennej do binárneho stromu sme zároveň kontrolovali, či náhodou funkcia už nie je definovaná. Globálnu premennú *uniq_scope*, sme v každom novom scope inkrementovali a jej hodnotu sme vždy uložili pri definícií novej premennej. Týmto sme zaručili unikátnosť názvu každej premennej kvôli generovaniu kódu. Naopak pri hľadaní premenných sme využívali zásobník stromov implementovaný taktiež v súbore *symtable.c*, kde funkcia prehľadala všetky predošlé stromy. Netypickou implementáciou bol aj spôsob kontroly funkcií, ktoré v dobe sémantickej kontroly ešte neboli definované. V tomto prípade sa volanie takejto funkcie ukladalo na tzv. error stack (*sem_estack.c*). Na konci parsovania sa pomocou funkcie *EStack_solveproblems* vyhodnotilo, či všetky volania boli validné. Tým pádom nám stačil jeden priechod vstupným kódom.

3.4. Generácia kódu

Generácia kódu sa vykonávala postupne popri sémantických kontrolách. Pri každej kontrole bola zavolaná funkcia z generácie kódu s príslušnými argumentami pre generáciu. Tento kód sa ukladal do nafukovacieho poľa, ktoré bolo definované ako globálna premenná. Pre vyriešenie problematiky viacnásobnej definície premennej v cykle sme zvoli spôsob vytknutia všetkých definícií premenných na začiatok funkcie. Preto existovali štyri globálne premenné s vygenerovaným kódom, jedna pre výsledný kód a tri pomocné, kde do jednej sa ukladali inštrukcie v hlavičke funkcie a preddefinované pomocné premenné, do ďalšej sa ukladali len definície nových premenných a do poslednej ostatné inštrukcie. Tieto tri premenné sa na konci vyhodnocovania funkcie spojili dokopy a vložili sa do premennej pre výsledný kód. Na odlíšenie názvov identifikátorov sme si určili, že vo výslednom kóde sa generáciou vytvorené pomocné premenné budú začínať znakom *&*, názvy návestí začínali znakom *!* a užívateľské premenné znakom *%*. Aritmetické operácie boli generované pomocou zásobníkových inštrukcií v precedenčnej analýze. Priradenie hodnoty do konkrétnej premennej sa teda používala inštrukcia *POPS*. Pri priraďovaní hodnôt do viacerých premenných sme netypicky využívali funkciu *nstring_get_and_delete(Nstring *source,Nstring *new)*. Fungovala tak, že z nafukovacieho poľa, kde sme mali dopredu uložené unikátne identifikátory oddelené znakom | (*source*) nahradila najvrchnejší identifikátor znakmi % a vrátila ho v nafukovacom poli *new* ktoré bolo použité ako argument pre generovanie kódu.

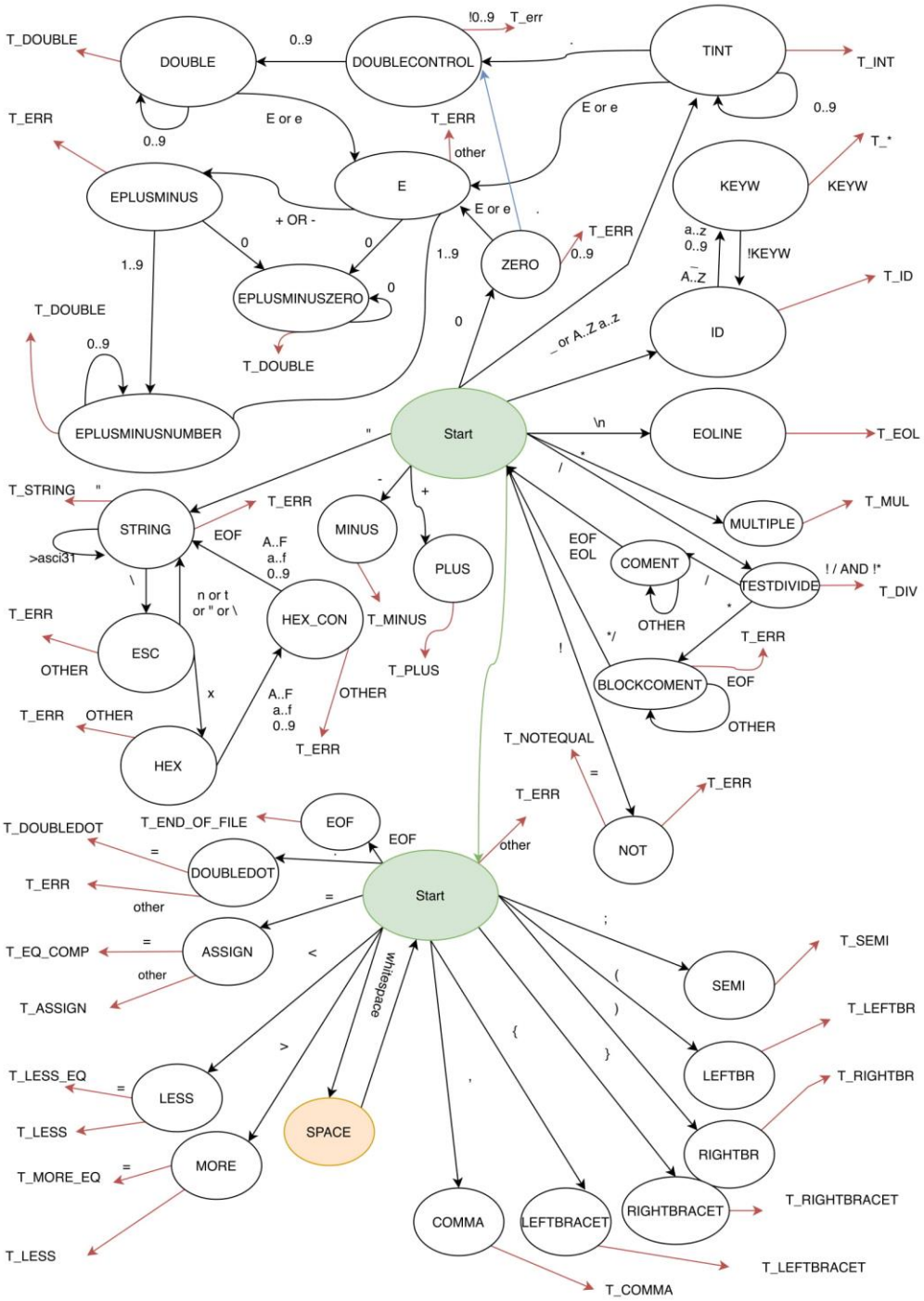
4. Záver

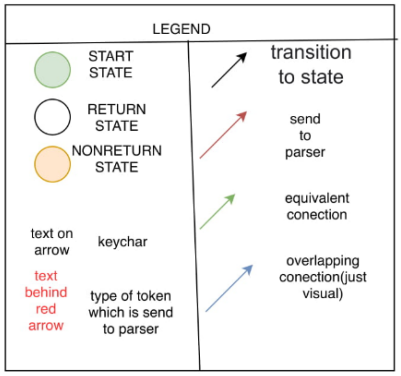
Projekt nám dal zabrať a aj napriek absencií jedného člena sme boli schopní projekt odovzdať k nultému a prvému pokusnému odovzdaniu. Najväčším problémom avšak zostáva rýchlosť nášho internetového

pripojenia, pretože bolo málokedy možné sa počuť bez problémov. S výsledkom našej práce sme spokojní i keď pripúšťame, že výsledný kód nie je tak čitateľný a uhladený ako by sme si predstavovali. Veľkou pomocou bolo aj to, že na naše otázky ohľadom projektu včasne reagoval, p. *Ing. Zbyněk Křivka, Ph.D.*

5. Prílohy

5.1. Graf konečného automatu





*RETURN TOKEN TYPE:
T_WELSE,T_
WFLOAT64,T_WFOR,T_WFUNC,
T_WIF,T_WINT,T_WPACKAGE
,T_WRETURN,T_WSTRING

5.2. LL tabuľka

	package	id	EOL	EOF	func	(	)	{	}	,	;	int	float64	string	int_lit	float_lit	str_lit	:=	=	expr	ε	if	for	else	return
<prog>	1		1																						
<func-list>				3	2																				
<func>					4																				
<param-list>		5					6																		
<param-next>							8			7															
<data-type-list>						9		10																	
<data-type-next>							12			11															
<data-type>												13	14	15											
<stat-list>		16							17													16	16		16
<stat>		39																				18	19		38
<def-stat>		20									21														
<assign-stat>		22						23																	
<id-list>		24																							
<id-next>										25									26						
<expr-list>																				27					
<expr-next >			29					29		28															
<term-list>		30					31								30	30	30								
<term-next>							33			32															
<term>		34													35	37	36								
<id-stat>						42				41								40							
<expr-or-id>		44																		43					
<opt-eol>	46	46	45	46	46																	46	46		46

5.3. LL gramatika

1.	<prog> → <opt-eol> package id EOL <opt-eol> <func-list>
2.	<func-list> → <func> <opt-eol> <func-list>
3.	<func-list> → EOF
4.	<func> → func id (<param-list> <data-type-list> <stat-list> EOL
5.	<param-list> → id <data-type> <param-next>
6.	<param-list> →)
7.	<param-next> → , id <data-type> <param-next>
8.	<param-next> →)
9.	<data-type-list> → (<data-type> <data-type-next>
10.	<data-type-list> → { EOL <opt-eol>
11.	<data-type-next> → , <data-type> <data-type-next>
12.	<data-type-next> →) { EOL <opt-eol>
13.	<data-type> → int
14.	<data-type> → float64
15.	<data-type> → string
16.	<stat-list> → <stat> EOL <opt-eol> <stat-list>
17.	<stat-list> → }
18.	<stat> → if expr { EOL <opt-eol> <stat-list> else { EOL <opt-eol> <stat-list>
19.	<stat> → for <def-stat> ; expr ; <assign-stat> { EOL <stat-list>
20.	<def-stat> → id := expr
21.	<def-stat> → ε
22.	<assign-stat> → <id-list> <expr-list>
23.	<assign-stat> → ε
24.	<id-list> → id <id-next>
25.	<id-next> → , id <id-next>
26.	<id-next> → =
27.	<expr-list> → expr <expr-next>
28.	<expr-next > → , expr <expr-next>
29.	<expr-next > → ε
30.	<term-list> → <term> <term-next>
31.	<term-list> →)
32.	<term-next> → , <term> <term-next>
33.	<term-next> →)
34.	<term> → id
35.	<term> → int_literal
36.	<term> → str_literal
37.	<term> → float_literal
38.	<stat> → return <expr-list>
39.	<stat> → id <id-stat>
40.	<id-stat> → := expr
41.	<id-stat> → <id-next> <expr-or-id>
42.	<id-stat> → (<term-list>

43.	<expr-or-id> → <expr-list>
44.	<expr-or-id> → id (<term-list>
45.	<opt-eol> → EOL <opt-eol>
46.	<opt-eol> → ε

5.4. Tabuľka precedenčnej analýzy

	+	-	*	/	(	)	i	<	>	<=	>=	==	!=	,	$
+	>	>	<	<	<	>	<	>	>	>	>	>	>		>
-	>	>	<	<	<	>	<	>	>	>	>	>	>		>
*	>	>	>	>	<	>	<	>	>	>	>	>	>		>
/	>	>	>	>	<	>	<	>	>	>	>	>	>		>
(	<	<	<	<	<	=	<	<	<	<	<	<	<		
)	>	>	>	>		>		>	>	>	>	>	>		>
i	>	>	>	>		>		>	>	>	>	>	>		>
<	<	<	<	<	<	>	<	Podľa zadania budú relačné operátory len v if a for podmienkach a nedajú sa na seba viazať							>
>	<	<	<	<	<	>	<								>
<=	<	<	<	<	<	>	<								>
>=	<	<	<	<	<	>	<								>
==	<	<	<	<	<	>	<								>
!=	<	<	<	<	<	>	<								>
,															>
$	<	<	<	<	<		<	<	<	<	<	<	<		