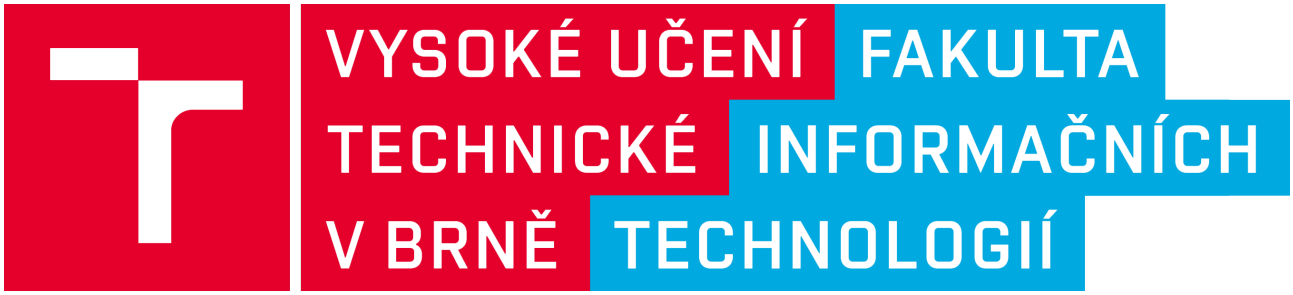


Projektová dokumentácia

Implementácia prekladača imperatívneho jazyka IFJ20

Tým 117, varianta 1

9. decembra 2020

Matej Horník	**(xhorni20)**	40 %
Filip Brna	(xbrnaf00)	25 %
Matúš Tvarožný	(xtvaro00)	25 %
Procházka Ivo	(xproch0h)	10 %

Obsah

1 Úvod

Zadaním projektu bolo implementovať prekladač imperatívneho jazyka IFJ20 do predmetov IFJ a IAL v jazyku C.Prekladač načíta program napísaný v jazyku IFJ20 zo štandardného vstupu a preloží ho do cieľového jazyka IFJcode20 na štardantný výstup.

1.1 Popis jazyka IFJ20

Jazyk IFJ20 je zjednodušenou podmnožinou jazyka GO, čo je staticky typovaný imperativní jazyk. V programovacom jazyku IFJ20 záleží na veľkosti písmen pri identifikátoroch a kľúčových slov (tzv. case-sensitive).

2 Štruktúra

Projekt sa skladá zo štyroch základných častí, ktoré budú bližšie opísané v nasledujúcich podkapitolách. Základne časti projektu sú :

- Lexikálny analyzátor
- Syntaktický analyzátor
- Sémantický analyzátor
- Generator kódu IFJcode20

2.1 Lexikálny analyzátor

Na začiatku sme implementovali lexikálny analyzátor v súbore `scanner.c`, ktorého základom je deterministický konečný automat (ďalej iba DKA). Hlavnou funkciou v tomto súbore je funkcia `get_token`, ktorá číta jednotlivé znaky a pomocou príkazu `switch` prechádza do nasledujúcich stavov podľa DKA až kým nevyhodnotí lexikálne správny token, hodnoty tokenu sú ďalej ukladané do štruktúry `token` pomocou čísel z prislúchaceho enumu, inak vracia `LEX_ERR`. Lexikálny analyzátor musí mať nadstavený vstupný súbor, ktorý obsahuje program napísaný v jazyku IFJ20.

Pri úspešnom vyhodnotení tokenu sa správne uvoľní všetká alokovaná pamäť o túto činnosť sa stará nami definovana funkcia `cleaner`. Matematické a relačné operátory sú vyhodnotené vcelku rýchlo a jednoducho, identifikátory, reťazce a čísla vyžadujú viacej prechodov a používajú dynamický reťazec, pomocné funkcie na prácu s ním sú definované v súbore `str.c`. Pri identifikátore sa kontrolujú povolené znaky na základe pozície v reťazci a na záver sa identifikátor porovná so všetkými kľúčovými slovami; ak sa nájde zhoda, je to kľúčové slovo, inak je to identifikátor.O túto prácu sa stará funkcia `process_identifier`. Reťazce sú ohraničené dvojitými úvodzovkami (") a môžu obsahovať escape sekvenciu. Pre tento prípad existuje špeciálny stav `STATE_STRING_BACKSLASH`, do ktorého sa prechádza pri prečítaní znaku spätné lomítko, (\) za ktorým môže nasledovať escape sekvencia podľa zadania.

2.2 Syntaktický analyzátor

Syntaktická analýza je hlavnou časťou programu implementová v súbore `parser.c` . Je založená na LL-gramatike a podľa pravidiel LL-tabuľky prechádza zdrojovým súborom rekurzívnym zostupom. Funkcie v syntaktickom analyzátore predstavujú netereminály skladajúce sa z nami definovaných pravidiel LL-gramatiky. Toto platí pre celú syntaktickú analýzu s výnimkou spracovania výrazov, ich spracovávanie je implementované v súbore `expression.c` pomocou precedenčnej tabuľky. V rámci syntaktickej analýzy je volaná funkcia lexikálneho analyzátoru `get_token`, ktorá v prípade validného tokenu uloží atribúty daného tokenu do štruktúry `token`, ktorá je súčasťou globálnej štruktúry `parser_data`. Na základe tohto tokenu sa syntaktický analyzátor rozhoduje, ktoré pravidlo z LL-tabuľky uplatniť a ktorú z funkcií následne zavolať. V prípade, že nastane rozpor s pravidlami LL gramatiky vracia `parser.c` chybovú hlášku `SYN_ERR`.

2.2.1 Precedenčná syntaktická analýza

Precedenčná syntaktická analýza(PSA) bola použitá pri spracovávaní výrazov. Je implementovaná vo vlastnom module `expression.c`. Keď syntaktický analyzátor narazí na výraz volá funkciu `expression()`, ktorá je definovaná v `expression.h`. Pri spracovávaní výrazu si PSA ukladá aktuálny symbol na zásobník, ktorý je implementovaný v `stack.c`, a spracuje daný symbol podľa precedencnej tabuľky. Keď spracuje aktuálny symbol zavolá si další token zo `scanneru` alebo zredukuje zásboník podľa jedného z pravidiel. Ak PSA narazí na symbol "<"z precedenčnej tabuľky tak si na zásobník uložíme zarážku aby sme vedeli pokiaľ máme redukovat dané pravidlo. PSA spracováva symboly až dovtedy dokým nenarazí na symbol $ a taktiež na zásobníku bude symbol "$"v daný moment je PSA úspešná a vracia `SYN_OK`. Pri spracovávaní výrazov sa taktiež kontroluje sémantika aritmetických, relačných operacií a kontroluje či použité identifikátory sú definované.

2.3 Sémantický analyzátor

Sémantická analýza je súčasťou syntaktickej analýzy a je taktiež implementová v súbore `parser.c`. V globálnej štruktúre `parser_data` sa nachádza binárny strom `BT_global`, ktorý pozostáva z identifikátorov funkcií a `BT_stack`, ktorý je stack zložený z binarných stromov, ktoré obsahujú identifikátory lokálnych premenných. Binárny strom, ktorý sa nachádza na vrchole zásobníka reprezentuje súhrn aktuálnych (najzanorenejších) premenných. Po ukončení daného zanorenia je aktuálny binárny strom uvolnený zo zásobníka a s ním je uvolnená aj všetká potrebná alokovaná pamäť. Binárne stromy v sémantickom analyzátore slúžia aj na kontrolu či dané identifikátory existujú a či súhlasí ich dátový typ, návratová hodnota. V prípade, že syntaktický analyzátor volá pomocnú funkciu `expression()`, ktorá zistí vo výraze sémantickú chybu tak vracia návratový kod príslušný danej chybe.

2.4 Generator kódu IFJcode20

Generátor kódu je súboru funkcií, ktoré sa volajú pri syntaktickej analýze a pri práci s aritmetickými výrazmi. Stará sa o výsledné vypísanie výrazov v jazyku ifjcode20. Kód se vypíše na štandartný výstup po ukončení všetkých analýz. Na začiatku generovania sa vypíšu pomocné premenné, ktoré sa využívajú napríklad pri konkatenanci. Návestie sa od seba líši indexom a typom, kde typ značí či sa jedná o for alebo if.

3 Použité algoritmy a dátové štruktúry

3.1 Tabuľka symbolov

Tabulka je implementovaná pomocou binárneho stromu. Ako kľúč pre vyhľadávanie v strome používame hash z identifikátoru (názov funkcie alebo premmenej). Použitá hash funkcia je sdbm http://www.cse.yorku.ca/ oz/hash. Keby sa náhodou stalo, že sa hash z identifikátoru zhoduje tak pripojíme nový identifikátor k už existujucej vetve stromu ako viazaný zoznam. Implementácia algoritmov pre binárny strom je pomocou rekurzií. Základ algoritmov bol prevzatý z prednášiek IAL a upravený pre naše potreby. V binárnom strome si uchovávame informácie o funkciách a identifikátoroch. Pri funkciách si taktiež ukládame typy vstupných parametrov a typy výstupných hodnôt. V hlavičkovom súbore symtable.h sú definované funkcie `BT_search`, `BT_insert`, `BT_delete` a `BT_dispose` ktoré sa používajú pri práci s binárnym stromom.

3.2 Dynamický reťazec

Vytvorili sme štruktúru `str_struct` pre prácu s reťazcami dynamickej dĺžky, ktorú používame pre ukladanie identifikátoru alebo reťazca u atribútu tokenu v lexikálnej analýze. Štruktúra a operácie nad ňou sú popísané v hlavičkovom súbore `str.h`. Veľký základ implementačných detailov tvoria algoritmy ktoré nám boli poskytnuté v jednodochom interprete (http://www.fit.vutbr.cz/study/courses/IFJ/public/project/). Implementáciu sme si máličko upravili a pridali vlastné funkcie. Štruktúra v sobe ukladá ukazovateľ na reťazec, dĺžku reťazca a velikosť alokovanej pamäte.

3.3 Zásobník symbolov pre precedenčnú syntaktickú analýzu

Implementovali sme zásobník symbolov, ktorý používame pri spracovaní výrazov. Zásobník má implementované základné operácie nad zásobníkom ako sú `stack_push`, `stack_pop`, `stack_top` a špeciálnu funkciu `stack_push_after_top_term` , ktorá pridá symbol za najvrchnejší terminál. Štruktura položky zásobníka obashuje symbol a typ daného symbolu, ktorý sa inicializuje iba pri identifikátoroch, číselných literáloch alebo reťazcoch. Štruktúra taktiež obsahuje ukazateľ na dalšiu položku

3.4 Zásobník tabuľky symbolov

Implementovali sme zásobník binárnych stromov , ktorý používame pri viacnásobnej definície premenných v programe pri zanorovaní v ifoch alebo foroch. Zásobník má implementované základné operácie nad zásobníkom ako sú `BT_stack_push`, `BT_stack_pop`, `BT_stack_top` a špeciálnu funkciu `BT_stack_search`, ktorá sa používa pri zistení či použitá premenná je definovaná. Štruktúra položky zásobníka obsahuje binárny strom. Štruktúra taktiež obsahuje ukazovateľ na ďalšiu položku.

3.5 Fronta dynamických reťazcov

Implementovali sme frontu dynamických reťazcov, ktorú používame pri ukladaní typov vstupných parametrov a výstupných hodnôt danej funkcie v binárnom strome. Taktiež frontu využívame pri sémantických kontrolách pri priraďovaní k premenným a volaní funkcií. Fronta má implementované základné operácie ako sú `id_queue_push`, `id_queue_pop`, `id_queue_top`. Štruktúra položky fronty obsahuje dynamický reťazec. Štruktúra taktiež obsahuje ukazovateľ na ďalšiu položku.

4 Práca v tíme

4.1 Spôsob práce v tíme

Na projektu sme začali pracovat v oktobri (říjnu). Prácu sme si delili postupne, nemali sme od začiatku stanovený kompletný plán rozdelenia práce. Najprv sme sa sústredili na teoretickú časť projektu (DKA,LL gramatika, LL tabulka). Neskôr sme začali postupne pracovať na lexikalnej, syntaktickej, sémantickej analýze a na generovaní kódu.

4.2 Verzovací systém

Na úvod sme skonštatovali, že najpraktickejšie bude použit verzovací system Git, konkrétne sme sa rozhodli použit stránku GitHub na ktorej sme využívali možnosť zdieľaného repozitáru. Git nám umožnil pracovať na viacerých úlohách na projekte súčasne.

4.3 Komunikácia

Komunikácia medzi členmi tímu prebiehala výlučne prostredníctvom aplikácie Discord, kde sme si medzisebou, bud písali, volali alebo pracovali spoločne na jednej zdieľanej obrazovke. Pri projekte chýbala možnosť osobnej komunikácie čo berieme ako výrazné negatívum.

4.4 Rozdelenie práce

Prácu sme si rozdelili na základe osobitých požiadaviek členov a ich skúseností, každý tak dostal percentuálne hodnotenie, ktoré odpovedá jeho percentuálnemu podielu práce vykonanej na projekte.

Prácu sme si rozdelili následovne:

Meno	Povinnosti
Matej Horník	vedenie tímu, konzultácie,lexikálna analýza, syntaktická analýza, sémantická analýza, generovanie kódu, testovanie, diagramy
Filip Brna	lexikálna analýza, syntaktická analýza, dokumentácia, testovanie, diagramy
Matúš Tvarožný	lexikálna analýza, sématická analýza, dokumentácia, testovanie, diagramy
Ivo Procházka	generovanie kódu

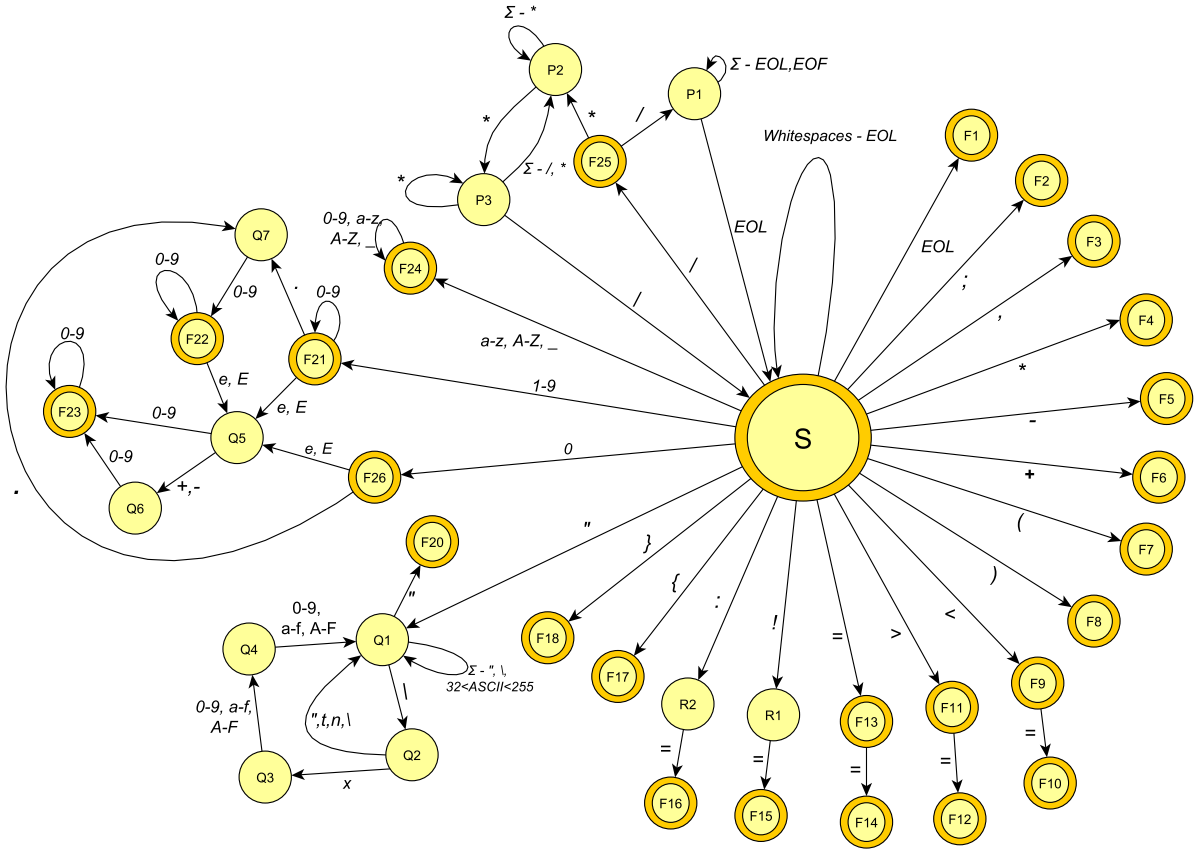
Tabuľka 1: Rozdelenie práce

5 Záver

Celkovo hodnotíme tento projekt kladne, hoci na začiatku vyzeral pomerne zložito. No po čase, keď sme postupne nabrali vedomosti o tvorbe prekladačov sme začali s vývojovým procesom.Tím na začiatku pozostával z troch členov a pomocou fakultného discord serveru bol tím doplnený o chýbajúceho člena, ktorý počas celej doby neprejevoval taký záujem a zodpovednosť o projekte a práci na ňom ako to vyznelo z našej spoločnej počiatočnej komunikácie. Projekt nás naučil a objasnil nám prácu s Gitom, ktorá je mimoriadným prínosom do nášho profesíjneho života. Na projekte sme pracovali do posledných dní. V priebehu vývoja sme narazili na pár problemov ale všetky sa nám podarilo vyriesiť či už konzultáciami s vyučujúcim alebo svojpomocne. Určite tento projekt prispel k zlepšeniu našich programátorských schopností a pochopeniu zložitosti prekladača.

5.1 Zdroje

Zdroje sme čerpali z dostupných výukových materiálov predmetov IFJ a IAL. Informácie sme čerpali najmä z prednášok, democvičení a prezentácií dostupných v IS FIT.



Obr. 1: Diagram konečného automatu špecifikujuci lexikální analyzátor

1. <start> -> package main EOL <eol> <prog> EOF
2. <eol>-> EOL <eol>
3. <eol>-> ε
4. <eof>-> EOF
5. <eof>-> ε
6. <prog>-> func ID (<params>) <return_value> { EOL <eol> <body> }
   <eof> EOL <eol> <prog>
7. <prog>-> ε
8. <body>-> if <expression> { EOL <eol> <body> } else { EOL <eol> <body> }
   EOL <eol> <body>
9. <body> -> for <definition> ; <expression> ; <assignment>
   { EOL <eol> <body> } EOL <eol> <body>
10. <body> -> ID := <expression> EOL <eol> <body>
11. <body> -> <ids> = ID(<argument>) EOL <eol> <body>
12. <body> -> <ids> = <values> EOL <eol> <body>
13. <body> -> ε
14. <body> -> return <list_of_return_values> EOL <eol> <body>
15. <body> -> ID (<argument>) EOL <eol> <body>
16. <definition> -> ID := <expression>
17. <definition> -> ε
18. <assignment> -> ID = <expression>
19. <assignment> -> ε
20. <ids>   -> ID <id_n>
21. <id_n> -> , ID <id_n>
22. <id_n> -> ε
23. <list_of_return_values> -> ε
24. <list_of_return_values> -> <values>
25. <values> -> <expression> <values_n>
26. <values_n> -> ,<expression> <values_n>
27. <values_n> -> ε
28. <params> -> ID <type> <params_n>
29. <params> -> ε
30. <params_n> -> , ID <type> <params_n>
31. <params_n> ->ε
32. <return_value> -> (<type> <return_value_n>)
33. <return_value> -> ε
34. <return_value_n> -> , <type> <return_value_n>
35. <return_value_n> -> ε
36. <type> -> int
37. <type> -> float64
38. <type> -> string
39. <argument> -> <value> <argument_n>
40. <argument> -> ε
41. <argument_n> -> ,<value> <argument_n>
42. <argument_n> -> ε
43. <value> -> INT_VALUE
44. <value> -> FLOAT_VALUE
45. <value> -> STRING_VALUE
46. <value> -> ID

Tabuľka 2: LL – gramatika riadiaca syntaktickú analýzu

Tabuľka 3: LL – množiny First, Follow

	EMPTY	FIRST	FOLLOW
<start>	∅	{package}	{ EOF }
<eol>	eps	{EOL}	{ func, if, for, ID, return, } , EOF }
<eof>	eps	{EOF}	{EOL}
<prog>	eps	{func}	{ EOF }
<body>	eps	{if, for, ID, return}	{ } }
<definition>	eps	{ID}	{ ; }
<assignment>	eps	{ID}	{ { }
<ids>	∅	{ID}	{ = }
<id_n>	eps	{,}	{ = }
<list_of_return_values>	eps	{}	{ EOL }
<values>	∅	{expression}	{ EOL }
<values_n>	eps	{,}	{ EOL }
<params>	eps	{ID}	{) }
<params_n>	eps	{,}	{) }
<return_value>	eps	{(}	{ { }
<return_value_n>	eps	{,}	{) }
<type>	∅	{int, float, string}	{ , ,) }
<argument>	eps	{INT_VALUE, FLOAT_VALUE, STRING_VALUE,, ID}	{) }
<argument_n>	eps	{,}	{) }
<value>	∅	{INT_VALUE, FLOAT_VALUE, STRING_VALUE, ID}	{ , ,) }

	package	func	EOL	EOF	if	for	ID	return	;	{	}	(	)	"="	,	int	float	string	INT_VALUE	FLOAT_VALUE	STRING_VALUE	EXPRESSION
<start>	1.																					
<eol>		3.	2.	3.	3.	3.	3.	3.			3.											
<prog>		6.		7.																		
<body>					8.	9.	10.,11.,12.,15.	14.			13.											
<definition>							16.		17.													
<assignment>							18.			19.												
<ids>							20.															
<id_n>														22.	21.							
<list_of_return_values>			23.																			24.
<values>																						25.
<values_n>			27.												26.							
<params>							28.						29.									
<params_n>													31.		30.							
<return_value>										33.		32.										
<return_value_n>													35.		34.							
<type>																36.	37.	38.				
<argument>							39.						40.						39.	39.	39.	
<argument_n>													42.		41.							
<value>							46.									43.	44.	45.				
<eof>			5.	4.																		

Tabuľka 4: LL – tabulka použitá pri syntaktickej analýze

Tabuľka 5: Precedenčná tabulka použitá pri precedenčnej syntaktickej analýze výrazov

	+	-	*	/	RO	(	)	i	$
+	>	>	<	<	>	<	>	<	>
-	>	>	<	<	>	<	>	<	>
*****	>	>	>	>	>	<	>	<	>
/	>	>	>	>	>	<	>	<	>
RO	<	<	<	<		<	>	<	>
(	<	<	<	<	<	<	=	<	
)	>	>	>	>	>		>		>
i	>	>	>	>	>		>		>
$	<	<	<	<	<	<		<	

RO = Relacne operatory