Dokumentácia projektu z predmetov IFJ a IAL

Implementace překladače imperativního jazyka IFJ20

Tím 021, varianta I

Výčet identifikátorů implementovaných rozšíření: FUNEXP, MULTIVAL, UNARY

Timotej Ponek (xponek00) - 32%
Marek Precner (xprecn00) - 20%
Kristián Královič (xkralo05) - 24%
Marek Valko (xvalko11) - 24%

Obsah

1 Návrh a implementácia

1.1 Lexikálna analýza

Vstupný súbor reprezentovaný znakmi je nutné rozdeliť na „tokeny“. O túto činnosť sa stará tzv. „scanner“, ktorý je reprezentovaný deterministickým konečným automatom, ktorý je vytvorený pomocou konštrukcie `switch` v jazyku C (viz prílohu **Obr. 1**).
Hlavnou funkciou lexikálnej analýzy je funkcia `getToken`, ktorá načíta znaky zo vstupného súboru a následne ho uloží do predom vytvorenej štruktúry. Štruktúra tokenu obsahuje ako aj typ tokenu, tak aj hodnotu tokenu, ktorá je reprezentovaná ako typ `union` a taktiež booleovskú hodnotu `isID` ktorá označuje či sa jedná o identikifátor.
Ďalej sa v skeneri chadáza funkcia `checkKey`. ktorá slúži na identifikáciu kľúčových slov. V prípade, že skener načíta neplatný znak (znak, ktorý nie je podporovaný v jazyku IFJ20), tak skener prechádza do stavu `LEX_ERR`, ktorý signalizuje výskyt lexikálnej chyby.

1.2 Syntaktická analýza

Naša syntaktická analýza je založená na metóde rekurzívneho zostupu, kde názvy jednotlivých funkcií reprezentujú jednotlivé neterminály v gramatike.
Syntaktická analýza začína funkciou `parse()`, ktorou sa začína v podstate celý preklad. V `parse()` sa nastavia počiatočné hodnoty dátových štruktúr a pomocných premenných, zavolá sa funkcia `GetToken()`, a ak bol prvý načítaný token valídny sa ďalej volá funkcia `start()`. `start()` a všetky ostatné funkcie, reprezentujúce jednotlivé neterminály v gramatike, sú založené na „switchi“ a na základe typu tokenu.
V „parseri“ sa vykonávajú aj niektoré sémantické akcie, ako napríklad kontrola, že sú parametre a „return“ typy novo definovanej funkcie korektné, že dátový typ a počet výrazov na ľavej strane priradenia odpovedá počtu na pravej strane priradenia.
Parser riadi taktiež generovanie kódu, a to najmä definovanie premenných, priradenie, volanie funkcií, a tvorbu `if-else` a `for` konštrukcií jazyka IFJ20.
Syntaktická analýza končí, ak typ tokenu nevyhovuje pravidlu gramatiky, v ktorom sa parser práve nachádza a vracia zodpovedajúci „error“ kód. Ďalej syntaktická analýza končí, ak nastala chyba alokácie pamäte alebo bol načítaný token typu `lex_err`.

1.3 Sémantická analýza

Sémantický analyzátor začína svoju prácu, keď je zavolaný z parseru pravidlom gramatiky `<expr>`. Je založený na precedenčnej analýze. Analyzátor má vlastnú premennú typu `token` (semanticToken) nedostupnú v parseri. Pri zavolaní sa skopíruje `currentToken` do `semanticToken`, a po dokončení semantickej analýzy sa nezávisle na „error“ kóde znovu skopíruje obsah `semanticToken` do `currentToken`. Toto nastane v pomocnej funkcii, ktorá volá ďalšiu funkciu, kde už prebieha jadro sémantickej analýzy.
Analyzátor je po príchode prvého tokenu založený na „switchi“ ktorý podľa typu tokenu zvolí, do ktorého z troch ďalších „switchov“ (pre `int`, `float64` a `string`) sa má skočiť. V týchto „switchoch“ ostáva sémantická analýza, až pokiaľ nie je načítaný token, ktorý zmení „error“ kód z `CODE_OK` na nejaký chybový kód alebo sa nevyprázdni `Tokenstack` a zavolá `return`. Ak je prvý token identifikátor, vyhľadá sa v tabuľke symbolov. Podľa jeho typu sa skočí na príslušný „switch“ (`int`, `float64`, `string`). Ak nie je v tabuľke symbolov, očakáva sa, že ide o nedefinovanú funkciu. Načíta sa ďalší token, ak je typu „(“, tak sa načíta ďalší token a rekurzívne sa zavolá semantická analýza. Po ukončení analýzy prvého parametra funkcie, sa volá vo `while` rekurzívne semántická analýza, ak je typ tokenu „,“. Po skončení `while`, ak je typ tokenu „)“, tak sa vloží názov funkcie (ktorý je uložený v pomocnej premennej) do tabuľky symbolov. Z hodnôt uložených na `idAssignStack` sa odvodia typy parametrov funkcie a vložia k funkcii do tabuľky symbolov.

Ďalej sa volá `DeriveFromIds`, ktorá zo „stacku“ identifikátorov na ľavej strane priradenia vloží na základe ich typov „return“ typy k funkcii do tabuľky symbolov. Rekurzívne sa volá analyzátor aj pre už definovanú funkciu.
Ak sa v priebehu analýzy samotného výrazu narazí na token reprezenrujúci `ID` funkcie, skontroluje sa, či funkcia vracia iba jeden vyhovujúci typ.
Pre analýzu samotného výrazu každého z troch datových typov, ktoré jazyk IFJ20 podporuje, ak je načítaný token vyhovujúci, je potlačený na `Tokenstack`. Jeho typ je uložený do premennej `lastype` a načíta sa další token. Podľa premennej `lastype` sa ďalej redukuje obsah „stacku“, čiže napríklad, ak príde token typu `int` a `lastype` je `add`, načítame další token a podľa jeho typu redukujeme. V tomto prípade ak by bol další token `add`, `sub` alebo jeden z porovnávacích operátorov, vyberieme z `Tokenstacku` dva tokeny, o ktorých vopred vieme že sú tam. Vygenerujeme inštrukciu sčítania a pushneme na „stack“ token typu `int`. Takto pokračujeme ďalej, až kým narazíme na terminál končiaci syntaktickú analýzu. Potom vyberáme z `Tokenstacku` tokeny, až dokým sa nevyprázdni.
Keď je `Tokenstack` prázdny, výsledny typ výrazu (`int/float64/string/bool`) uložíme na `idAssignStack`, ktorý použije parser pri vykonávaní sémantických kontrôl porovnania typu výrazu, s typom identifikátoru, ktorý je ku nemu priraďovaný, kontrole počtu terminálov na ľavej a pravej strane priradenia, prípadne očakávame podmienku `if/for`, tak či je typ výrazu `bool`. Týmito akciami sa vyprázdni `idAssignStack` pre ďalšie použitie v sémantickom analýzátore.

1.4 Generovanie kódu

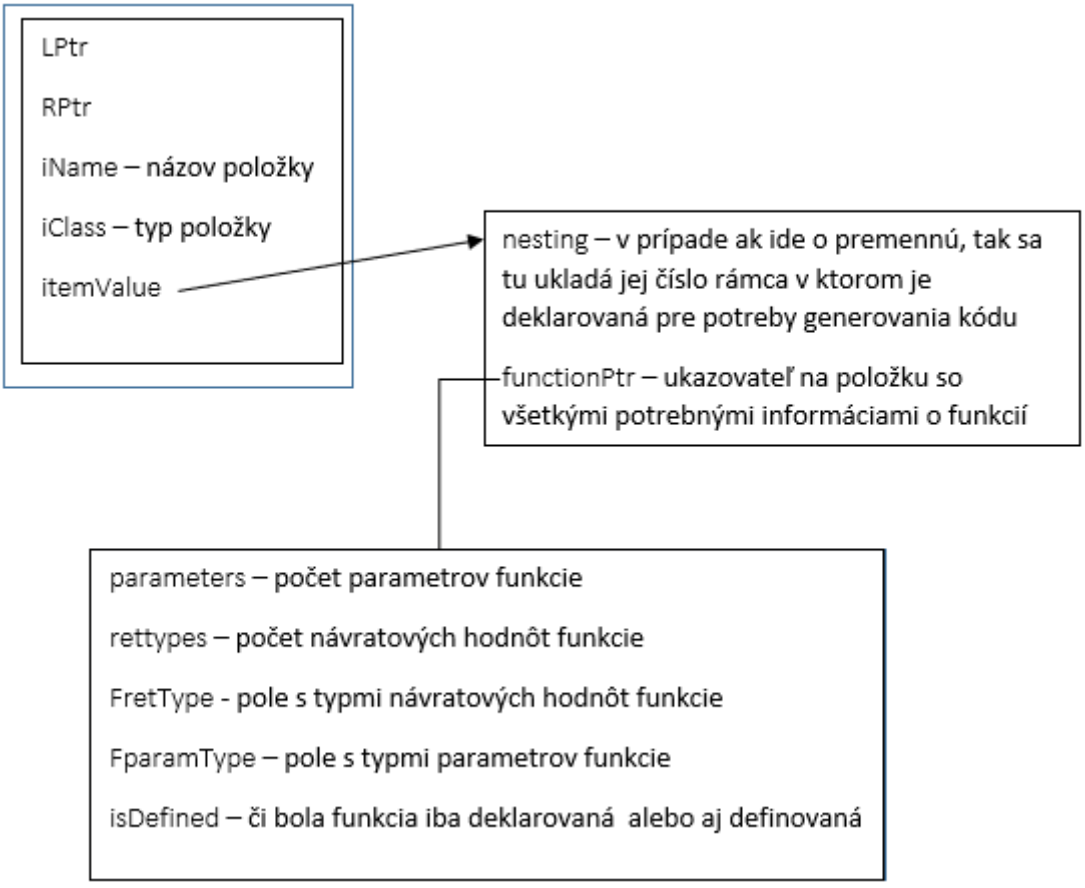
Generovanie kódu IFJCODE20 prebieha v rámci sémantickej a syntaktickej analýzy. Samotné generovanie inštrukcií je realizované v module `codegen.c`, v ktorom sa nachádza množstvo pomocných funkcií pre výpis inštrukcií jazyka IFJCODE20.
Jednotlivé funkcie sú volané z modulov `parser.c` a `expr.c`. Každá uživateľská funkcia je tvorená návestím v tvare `$HELP1identifikator`.
Pre každú definovanú premennú generujeme unikátny názov na základe zanorenia bloku, v ktorom sa parser práve nachádza a „scopu“ v tabuľke symbolov. Návestia pre konštrukcie `if/else` a `for` generujeme na základe zanorenia v bloku, kde premennú reprezentujúcu výskyt znakov „ v parseri vždy inkrementujeme. Ešte predtým si uložíme jej hodnotu a pri vyskočení z bloku načítaním tokenu „“, ak nasleduje `else` tak máme stále k dispozícií korektnú hodnotu pre označenie else návestia. Naviac všetky lokálne premenné (majme na mysli premenné, ktoré nemajú žiadne zanorenie vo funkcii), sú označené číslami „0_1“.
Pre každý `for loop` ukladáme všetky deklarácie do „linked listu stringov“, a deklarácie generujeme až na konci najvonkajšieho `for loop`. Pri interpretácií kódu sa skáče na návestie s definíciami a po skončení `for loop` sa toto návestie preskakuje.

1.4.1 Generovanie funkcií

Pred každým volaním funkcie sa pomocou príkazu `CREATEFRAME` vytvorí dočasný rámec, na ktorý sa uložia jednotlivé parametre uživateľskej funkcie. Následne použitím príkazu `CALL`, sa skočí na návestie uživateľskej funkcie. Pri vstupe do funkcie sa pomocou príkazu `PUSHFRAME`, uložia hodnoty do lokálneho rámca a následne sú použíté na vykonávanie funkcie. Pri návrate sú použité príkazy `POPFRAME` a `RETURN`.

2 Algoritmy a dátové štruktúry

Tabuľku symbolov máme implementovanú podľa nášho zadania formou binárneho stromu. Pre každý blok sa vytvára tabuľka symbolov zvlášť, ukladá sa na „stack“ tabuliek symbolov a pri ukončení bloku sa tabuľka maže a „popuje“ zo stacku. Máme implementované aj prislušné operácie k tabuľke symbolov a hľadaniu v tabuľkách symbolov na „stacku“.
`TokenStack` je štruktúra, používaná pri sémantickej analýze na analýzu výrazov. `ItemStack` má dvojaké využitie, používame ho na ukladanie „id“ na ľavej strane priradenia/definície, ale aj na ukladanie výsledných typov výrazov na pravej strane.
`Linkedlist` používame na ukladanie kódu, ktorý sa musí vytlačiť na špecifickom mieste, a až po vykonaní potrebných pravidiel gramatiky, v našom prípade keď sa chceme vyhnúť redefinícií premenných vo `for loop`.
Štruktúry `TokenStack` a `ItemStack`, a operácie nad nimi su uložené v súbore `expr.h` a `expr.c`. `Linkedlist` v súbore `codegen.h` a `codegen.c`, ostatné v `symtable.h` a `symtable.c`.



Obr. 1: Tabuľka ku štruktúre tabuľky symbolov

3 Práca v tíme

3.1 Rozdelenie práce v tíme

Timotej Ponek (xponek00)

- Vedúci tímu
- Generovanie kódu
- Sématická analýza
- Syntaktická analýza

Marek Precner (xprecn00)

- Implementácia kódu
- Dokumentácia

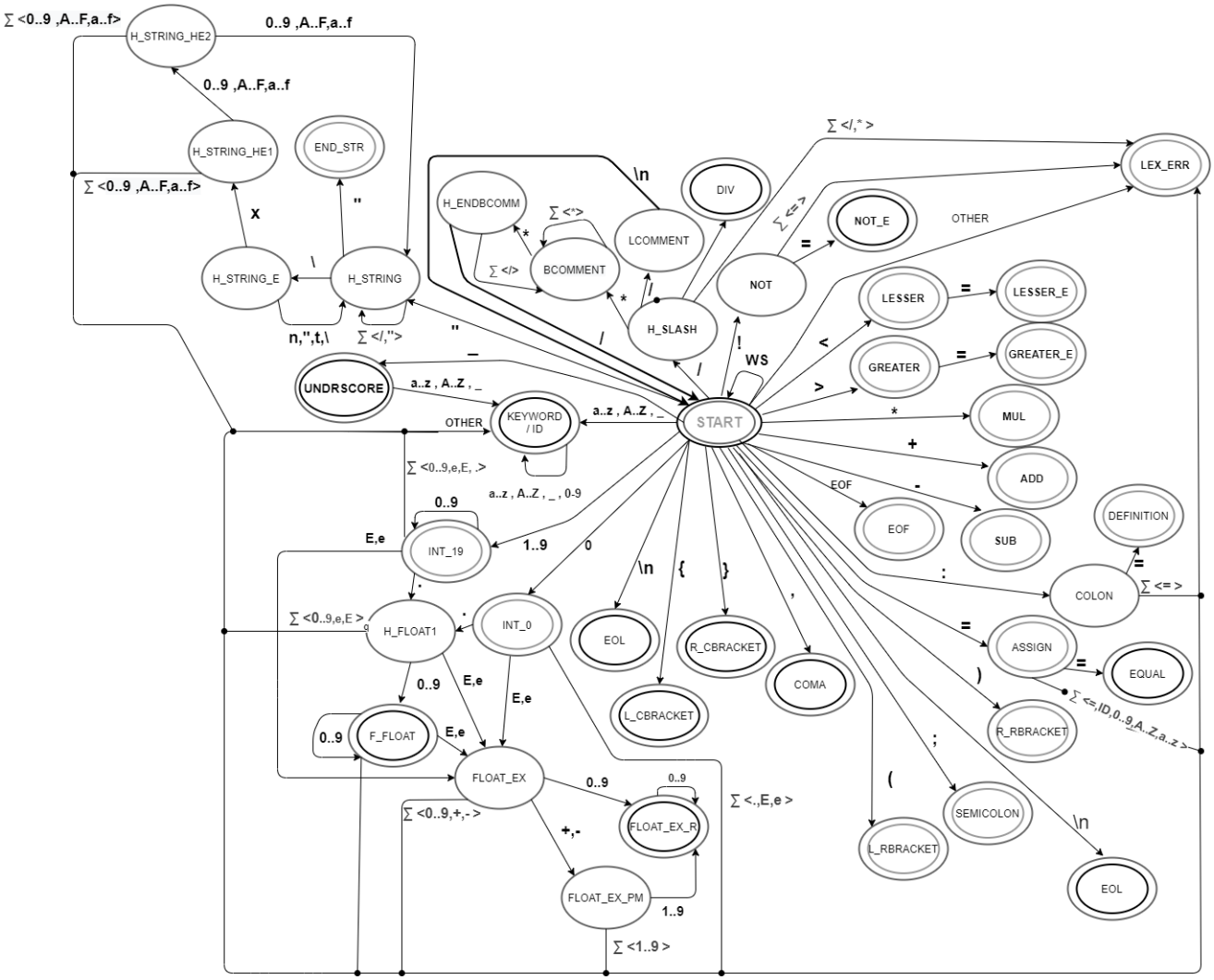
Kristián Královič (xkralo05)

- Plánovanie a organizácia
- Generovanie kódu
- Lexikálna analýza

Marek Valko (xvalko11)

- Sématická analýza
- Generovanie kódu

4 Diagram konečného automatu



Legenda :

WS - všetky whitespace znaky okrem \n
a..z - všetky znaky malej abecedy
A..Z - všetky znaky veľkej abecedy
0..9 - čísla od 0 po 9
∑ <0..9,+,- > - Všetky znaky okrem znakov v hranatej zátvorke

Obr. 2: Diagram konečného automatu špecifikujúci lexikálny analyzátor

5 LL - gramatika

1. <prog> -> PACKAGE MAIN <fnc list> <main fnc> <fnc list> EOF
2. <fnc list> -> <fnc> <fnc list>
3. <fnc list> -> ϵ
4. <fnc> -> FUNC F_ID (<parameters> <return types>) { <stat list>
5. <parameters> -> ID <type> <param_n>
6. <parameters> ->)
7. <param_n> -> , <parameters>
8. <param_n> ->)
9. <return types> -> (<retList>
10. <return types> -> ϵ
11. <retList> ->)
12. <retList> -> <retType>
13. <retType> -> <type> <retType_n>
14. <retType_n> -> , <retType>
15. <retType_n> ->)
16. <stat> -> <return>
17. <return> -> RETURN <expr> <expressions>
18. <main fnc> -> FUNC MAIN() { <stat list>
19. <stat list> -> <stat> <stat list>
20. <stat> -> { <statlist>
21. <stat> -> ID <FigAD>
22. <stat> -> _ <FigAD>
23. <FigAD> -> , <next_id>
24. <FigAD> -> := <expr> <expressions>
25. <FigAD> -> = <expr> <expressions>
26. <FigAD> -> (<expr>, <expressions
27. <expressions> -> , <expr> <expressions>
28. <expressions> -> ϵ EOL
29. <next_id> -> ID <FigAD>
30. <next_id> -> _ <FigAD>
31. <type> -> FLOAT64
32. <type> -> INT

33. <type> -> STRING
34. <stat> -> <fnc>
35. <stat> -> <return>
36. <stat> -> <if>
37. <if> -> IF <expr> { EOL <stat list> <else>
38. <else> -> ELSE { <statlist>
39. <else> -> EOL
40. <stat> -> <for>
41. <for> -> FOR <for_decl> { EOL <stat list>
42. <for_decl> -> <decl> ;<expr>; <assign i>
43. <for_decl> -> ;<expr>; <assign i>
44. <for_decl> -> ;<expr>;
45. <decl> -> <FigAD>
46. <assign_i> -> <FigAD>

Tabuľka 1: LL - gramatika riadiaca syntaktickú analýzu

6 LL - tabuľka

	PACKAGE MAIN	EOF	FUNC F_ID	{	ID	)	,	(	RETURN	FUNC MAIN	_	:=	=	EOL	FLOAT64	INT	STRING	IF	ELSE	FOR	$
<prog>	1	1																			
<fnc list>																					3
<fnc>			4	4																	
<parameters>					5	6															
<param_n>						8	7														
<return types>								9													10
<retList>						11															
<retType>																					
<retType_n>						15	14														
<stat>				20	21						22										
<return>									17												
<main fnc>										18											
<stat list>																					
<FigAD>							23	26				24	25								
<expressions>							27							28							28
<next_id>					29						30										
<type>															31	32	33				
<if>				37										37				37			
<else>				38										39					38		
<for>				41										41						41	
<for_decl>																					
<assign_i>																					

Obr. 3: LL - tabuľka použítá pri syntaktickej analýze

7 Precedenčná tabuľka

V precedenčnej analýze nám príde token typu `id`, z tabuľky symbolov zistíme jeho typ. Ak je vyhovujúci pre práve spracovaný výraz, pokračuje sa v analýze a na `Tokenstack` sa ukladá token zisteného datového typu. Ak z tabuľky symbolov zistíme, že ide o `id` funkcie, skontrolujeme, či má len jednu vyhovujúcu návratovú hodnotu a rekurzívne sa zavolá analýza výrazu pre parametre funkcie.
Po dokončení analýzy pre parametre funkcie (pri ktorej sa vygeneroval potrebný kód a návratová hodnota funkcie sa „pushla“ na datový zásobník) sa na `Tokenstack` ukladá len token so zodpovedajúcim datovým typom pre analyzovaný výraz. Tým pádom sa na `Tokenstacku` nikdy nemôže objaviť `id` funkcie a preto ani v precedenčnej tabuľke nie je. Keďže máme rozšírenie `FUNEXP`, podporujeme aj odriadkovanie vo výrazoch po aritmetických, reťazcových a relačných operátoroch čo je naznačené aj v precedenčnej tabuľke.

	+	-	*	/	(	)	==	<=	>=	!=	string	int, float	EOL	$
+	>	>	<	<	<	>	>	>	>	>	<	<	=	>
-	>	>	<	<	<	>	>	>	>	>		<	=	>
*	>	>	>	>	<	>	>	>	>	>		<	=	>
/	>	>	>	>	<	>	>	>	>	>		<	=	>
(	<	<	<	<	<	=	<	<	<	<	<	<	=	>
)	>	>	>	>		>	<	<	<	<	<	<		
==	>	>	>	>	<	>					<	<	=	>
<=	>	>	>	>	<	>					<	<	=	>
>=	>	>	>	>	<	>					<	<	=	>
!=	>	>	>	>	<	>					<	<	=	>
string	>					>	>	>	>	>				
int, float	>	>	>	>		>	>	>	>	>				
EOL											>	>	=	>
$	<	<	<	<	<	<	<	<	<	<	<	<	<	

Obr. 4: Precedenčná tabuľka použitá pri syntaktickej analýze výrazov

8 Záver

Projekt nás spočiatku zaskočil svojou zložitosťou a svojím rozsahom. Až po získaní potrebných informácií na prednáškach IFJ, sme začali na projekte pracovať.
Náš tím sme si vytvorili hneď na začiatku semestra, keďže sme už spolu v minulosti pracovali. Dohodli sme sa na používaní komunikačných prostriedkov, kde sme si rozdelili prácu.
V priebehu vývoja sme sa stretli s menšími problémami, ktoré sme ale vyriešili buď použitím fóra vo WISe, alebo použitím IFJ kánalu na Discorde. Správnosť riešenia projektu sme si overili automatickými testami a pokusným odovzdaním, pomocou ktorého sme boli schopní projekt ešte viac doladiť.
Celkovo nám tento projekt priniesol veľa skúseností, a objasnil nám preberanú látku v predmetoch IFJ a IAL.