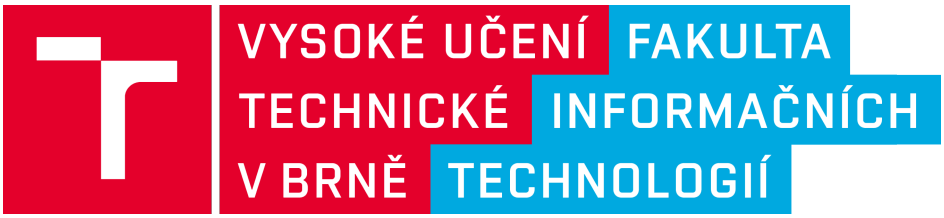

Dokumentácia k projektu

Prekladač jazyka IFJ22

Tým xdobia15, varianta BVS

Rozšírenia:

FUNEXP

Riešitelia:

Matúš Dobiáš (xdobia15) 25%

Oliver Nemček (xnemce08) 25%

Richard Blažo (xblazo00) 25%

Martin Packa (xpacka00) 25%

December 8, 2022

1 Návrh

1.1 Lexikálna analýza [*scanner.c*]

Prvá časť projektu ktorá bola implementovaná je scanner, ktorý realizuje lexikálnu analýzu. Podľa konečného automatu ktorý sme na začiatku navrhli sme v implementovali funkciu `get_token` ktorá načíta znaky zo štandartného vstupu a prevedie ich na token (dátový typ `token_t`). Token sa skladá z 2 zložiek: `token_type`, čo je enumerovaný dátový typ obsahujúci všetky typy tokenov, a `value`, čo je reťazec obsahujúci presnú postupnosť znakov, ktoré daný token tvoria.

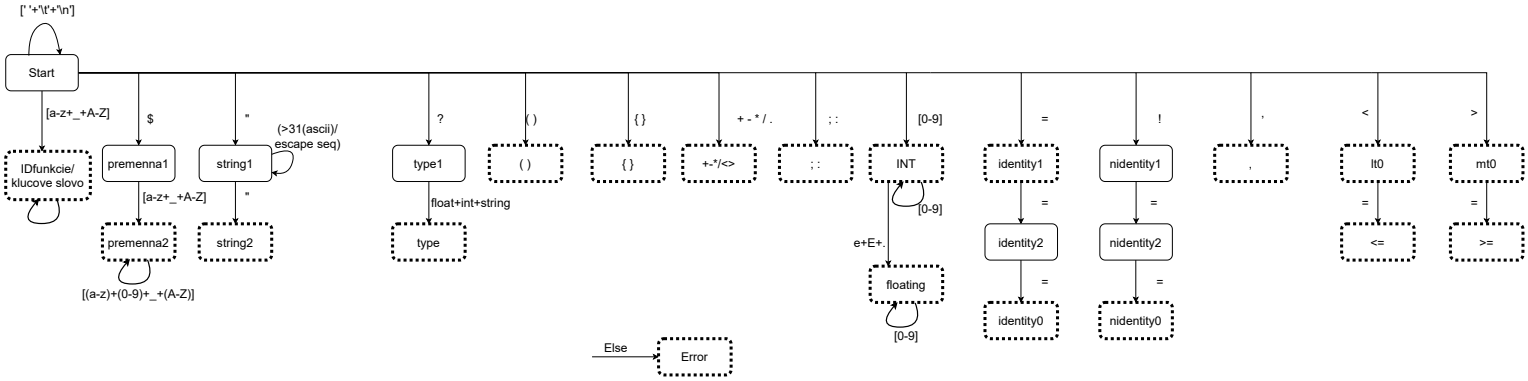

Figure 1: Návrh konečného automatu pre lexikálny analyzátor

1.2 Syntaktická analýza [*parser.c*]

Syntaktická analýza je realizovaná pomocou rekurzívneho vzostupu. Tento rekurzívny postup je založený na LL-gramatike a tabuľke. Zostup je realizovaný pomocou rekurzívneho volania funkcií, kde skoro každé pravidlo má vlastnú funkciu. Funkcia `p_body` je používaná pri spracovaní hlavného tela, tela funkcie a tiel if, else a while. Naša implementácia porušuje pravidlo LL(1) gramatiky v prípade, keď sa na vstup dostáva token s identifikátorom premennej, vtedy musíme načítať ďalší token a zistiť, či sa jedná o výraz alebo o priradenie. V prípade výrazu predá syntaktická analýza riadenie precedenčnej syntaktickej analýze.

LL tabulka

LL gramatika

1.3 Precedenčná syntaktická analýza [*bottom_ up.c*]

Pre spracovanie výrazov je použitá metóda syntaktickej analýzy zdola hore. Jej jadro tvorí funkcia `expr`, ktorá overí, či sa jedná o valídny výraz podľa precedenčnej gramatiky, a vytvorí abstraktný syntaktický strom pre tento výraz. Každý prvok (t.j. terminál alebo neterminál) je reprezentovaný dátovým typom `unit_t`, ktorý obsahuje nasledujúce:

1. int ttyp : Typ prvku. Značí, o aký terminál alebo neterminál ide. Každý token a neterminál má priradené číslo (Tieto čísla sú uvedené vo figúre 2). Môže sa stať, že viac typov tokenov

majú priradené jedno číslo, ak v kontexte výrazu majú ten istý význam (napr. int a float). Práve podľa typu sa vyberajú akcie v precedenčnej tabulke a pravé strany pravidiel pri redukcii.

2. token_t value : Z akého tokenu neterminál vzdnikol. Využíva sa pri vytváraní listov abstraktného syntaktického stromu.

3. ASSnode_t* uzol : Uzol abstraktného syntaktického stromu priradený k prvku. Informácia o uzle sa využíva keď sa daný prvok redukuje, aby sa mohla nadradená operácia naviazať na operandy.

Každý načítaný token sa najprv konvertuje na typ `unit_t`, a následne sa z precedenčnej tabulky určí akcia. Buď sa prvok vloží na zásobník, alebo sa na zásobníku redukuje podľa pravej strany pravidla. To sa deje vo funkcii `cmp_to_rule`, ktorá nielen vráti neterminál korespondujúci k ľavej strane použitého pravidla, ale aj pri redukcii vytvorí uzol abstraktného syntaktického stromu.

Precedenčná tabulka

Precedenčná gramatika

1.4 Sémantická analýza [*semantic.c*]

Sémantická analýza prebieha v dvoch fázach: zároveň so syntaktickou analýzou a počas behu programu. V prvej fáze kontroluje definície funkcii, premenných, prípad prázdnej návratovej hodnoty (`return;`) a prípad návratovej hodnoty volania funkcie (`return funkcia();`). Tieto kontroly sú umožnené tabuľkou symbolov. V druhej fáze sa počas behu programu použitím funkcie `type` kontrolujú typové kompatibility v prípadoch, kde sa nedajú overiť za syntaktickej analýzy.

1.5 Generovanie kódu [*ASS.c*]

Generácia kódu prebieha Postorder priechodom stromom a printovaním na štandardný výstup. Výnimkou sú `If`, `While`, `Fcall` a `Fdec` ktoré majú zložitú stromovú štruktúru. Generácia kódu začína vygenerovaním vstavaných funkcií a definovaním premenných deklarovaných priamo v zdrojovom kóde ifj22.

2 Popis rozhrania

2.1 get_token

Funkcia `get_token` je volaná zo syntaktického analyzátoru kedykoľvek potrebuje nový token zo vstupu. Parameter int skip je nastavený na 1, keď sa očakáva token prológu, inak je 0.

2.2 Spracovanie výrazov v syntaktickej analýze

Funkcia `expr`, ktorá tvorí základ precedenčnej analýzy je volaná z modulu *Parser.c*, a to vždy keď parser očakáva výraz. Funkcia `expr` má parametre first a second, do ktorých má parser možnosť vložiť tokeny, ktoré sa vložia na vstup precedenčnej analýzy. To sa využíva v prípadoch, kedy si parser musí vopred načítať tokeny aby zistil, či sa jedná o výraz. Ďalej má funkcia `expr` parametre `end` a `end2`, ktoré slúžia ako ukončovač výrazu. Sú dva, pretože niekedy si parser nieje istý, čím bude výraz končiť, v tomto prípade prvý z týchto dvoch ktorý sa vyskytne vo výraze sa vezme ako

ukončovač výrazu (t.j. v kontexte precedenčnej analýzy sa zmení na $). Špeciálnym prípadom je znak ')', ktorý sa zmení na $ iba v prípade, ak sme vo výraze na nulovej úrovni zanorenia zátvoriek.

2.3 Generácia kódu

Funkcia `expr` vracia parseru abstraktný syntaktický strom pre výraz, ktorý bol spracovaný. Parser tento strom napojí na strom ktorý vytvára pre celý program, nad ktorým potom zavolá funkciu `print_code`, ktorá z daného stromu vygeneruje výsledný kód.

3 Použité dátové štruktúry

3.1 Obojsmerne viazaný lineárny zoznam [*c206.c*]

Obojsmerne viazaný lineárny zoznam bol využitý pri precedenčnej syntaktickej analýze ako náhrada za zásobník. Dôvodom je, že sa pri nej často pracuje s prvkami ktoré nie sú na vrchole, a možnosť mať určitý prvok "aktívny" je tu veľmi užitočná (aktívny je vždy najvyšší terminál).

3.2 Tabulka symbolov [*symtable.c , symdll.c*]

Tabulka symbolov implementovaná binárnym vyhľadávacím stromom obsahuje uzly ktoré okrem kľúča a vetiev obsahujú aj typ premennej a informácie o funkcií, ktorými sú počet a typy jej parametrov, typ návratovej hodnoty a informáciu či bola definovaná.
Tabulka symbolov je vytvorená pre každú deklaráciu funkcie a pre hlavné telo programu, ktoré sú potom uložené do obojsmerne viazaného zoznamu

4 Práca v týme

4.1 Spôsob práce v týme

Prekladač sme začali vyvýjať približne v polovici semestra. Pri práci sme k verzovaniu používali systém Git a testy sme robili ku každej časti prekladača manuálne. Komunikovali sme primárne cez platformu discord, zriedkavo osobne. Prácu sme sa snažili rozdeliť tak, aby každý pracoval na oddelenom celku, pričom sme sa snažili minimalizovať rozhranie.

4.2 Rozdelenie práce v týme

Matúš Dobiáš 25%
Generovanie kódu
Sémantická analýza
Návrh konečného automatu

Oliver Nemček 25%
Lexikálna analýza
Precedenčná syntaktická analýza
Dokumentácia

Richard Blažo 25%
Syntaktická analýza
Sémantická analýza
Návrh LL Gramatiky

Martin Packa 25%
Lexikálna analýza
Testovanie
Dokumentácia

5 Zdroje

1. IAL domáce úlohy

2. Prednášky IFJ

Figure 2: Precedenčná tabulka

		0	1	2	3	4	5	6	7	8	9	10	11	12
		null	.	string	i	+	-	*	/	relacny	identity	(	)	$
0	null		>			>	>	>	>	>	>		>	>
1	.	<	>	<		>	>	<	<	>	>	<	>	>
2	string		>							>	>		>	>
3	i					>	>	>	>	>	>		>	>
4	+	<	>		<	>	>	<	<	>	>	<	>	>
5	-	<	>		<	>	>	<	<	>	>	<	>	>
6	*	<	>		<	>	>	>	>	>	>	<	>	>
7	/	<	>		<	>	>	>	>	>	>	<	>	>
8	relacny	<	<	<	<	<	<	<	<	>	>	<	>	>
9	identity	<	<	<	<	<	<	<	<	<	>	<	>	>
10	(	<	<	<	<	<	<	<	<	<	<	<	=	
11	)		>			>	>	>	>	>	>		>	>
12	$	<	<	<	<	<	<	<	<	<	<	<		

Figure 3: Precedenčná gramatika

```
-3<exp> -> i
-2<null> -> null
-4<strexp> -> string
  <null> -> (<null>)
  <exp> -> (<exp>)
  <strexp> -> (<strexp>)
  <exp> -> <exp>||<nnull> + <exp>||<nnull>
  <exp> -> <exp>||<nnull> - <exp>||<nnull>
  <exp> -> <exp>||<nnull> * <exp>||<nnull>
  <exp> -> <exp>||<nnull> / <exp>||<nnull>
  <strexp> -> <strexp>||<nnull> dot <strexp>|<nnull>
-5 S -> (S)
  S -> <exp>|<nnull> relid1 <exp>|<nnull>
  S -> <exp>|<nnull> relid2 <exp>|<nnull>
  S -> <strexp>|<nnull> relid <strexp>|<nnull>
  S -> <strexp>|<nnull> relid2 <strexp>|<nnull>
  S -> <strexp>|<exp> relid2 <strexp>|<exp>

  integer, ffloat = i
  lte, mte, lt, mt, = relid1
  identity, nidentity = relid2
  null, ID_variable = null
```

Figure 4: LL tabulka

	$	<?php	declare	function	fid	)	}	if	while	id	;	expr	return	?>	,	void	?int	?string	?float	int	string	float	str_lit	int_lit	float_lit
<start>		1	1																						
<prolog>		2	2																						
<prolog1>		3																							
<prolog2>			4																						
<body>	14			5	11			6	7	8		10	12	13									9	9	9
<fparams>						16											15	15	15	15	15	15			
<nparam>						18									17										
<rettype>																20	19	19	19	19	19	19			
<type>																	21	22	23	24	25	26			
<nodefbody					32		34	27	28	29		31	33										30	30	30
<ifstat>					36							35													
<whilestat>					38							37													
<assigned>					40							39											41	41	41
<const>																							42	43	44
<fcall>					45																				
<callargs>					46	47				46													46	46	46
<ncallargs>						49									48										
<retval>					50					50	52	51											50	50	50
<vals>					55					53													54	54	54

Figure 5: LL gramatika

```
1. <start> -> <prolog> <body>

2. <prolog> -> <prolog1> <prolog2>

3. <prolog1> -> <?php
4. <prolog2> -> declare(strict_types=1);

5. <body> -> function fid ( <fparams> ):<rettype> { <nodefbody> } <body>
6. <body> -> if (<ifstat> <body>
7. <body> -> while (<whilestat> <body>
8. <body> -> id = <assigned>; <body>
9. <body> -> <const>; <body>
10. <body> -> <expr>; <body>
11. <body> -> <fcall>; <body>
12. <body> -> return <retval>; <body>
13. <body> -> ?>
14. <body> -> eps

15. <fparams> -> <type> id <nparam>
16. <fparams> -> eps

17. <nparam> -> , <type> id <nparam>
18. <nparam> -> eps

19. <rettype> -> <type>
20. <rettype> -> void

21. <type> -> ?int
22. <type> -> ?string
23. <type> -> ?float
24. <type> -> int
25. <type> -> string
26. <type> -> float

27. <nodefbody> -> if (<ifstat> <nodefbody>
28. <nodefbody> -> while (<whilestat> <nodefbody>
29. <nodefbody> -> id = <assigned>; <nodefbody>
30. <nodefbody> -> <const>; <nodefbody>
31. <nodefbody> -> <expr>; <nodefbody>
32. <nodefbody> -> <fcall>; <nodefbody>
33. <nodefbody> -> return <retval>; <nodefbody>
34. <nodefbody> -> eps

35. <ifstat> -> <expr>) {<nodefbody>} else {<nodefbody>}
36. <ifstat> -> <fcall>) {<nodefbody>} else {<nodefbody>}

37. <whilestat> -> <expr>) {<nodefbody>}
38. <whilestat> -> <fcall>) {<nodefbody>}

39. <assigned> -> <expr>
40. <assigned> -> <fcall>
41. <assigned> -> <const>

42. <const> -> str_lit
43. <const> -> int_lit
44. <const> -> float_lit

45. <fcall> -> fid(<callargs>)

46. <callargs> -> <vals> <ncallargs>
47. <callargs> -> eps

48. <ncallargs> -> , <vals> <ncallargs>
49. <ncallargs> -> eps

50. <retval> -> <vals>
51. <retval> -> <expr>
52. <retval> -> eps

53. <vals> -> id
54. <vals> -> <const>
55. <vals> -> <fcall>
```