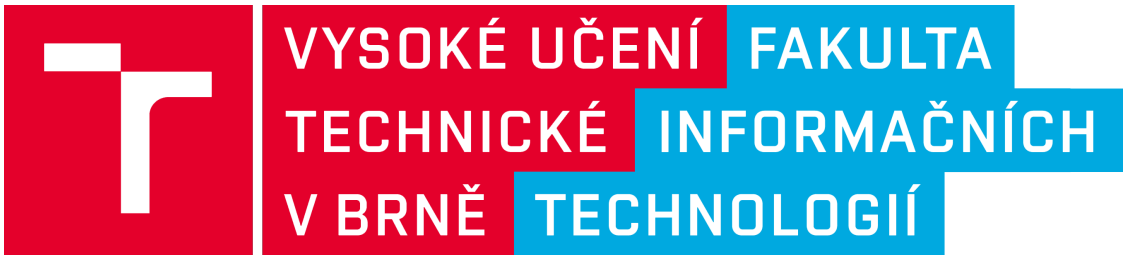


Dokumentace

Implementace překladače imperativního jazyka IFJ21

Tým 035, varianta I

vedoucí Štěpán Bakaj (xbakaj00) 25%
David Chocholatý (xchoch09) 25%
Adam Kaňkovský (xkanko00) 25%
Radek Šerejch (xserej00) 25%

Bez implementace rozšíření

Brno, 8. prosince 2021

Obsah

1 Práce v týmu

1.1 Rozdělení práce mezi jednotlivé členy týmu

- Štěpán Bakaj
  - návrh automatu pro lexikální analýzu
  - datový typ string_t a zásobník symbolů
  - obousměrně vázaný seznam (dll)
  - návrh precedenční tabulky
  - implementace syntaktického analyzátoru (psa)
  - implementace sémantického analyzátoru (psa)
  - testování, překlad a spuštění překladače

- David Chocholatý
  - návrh automatu pro lexikální analýzu
  - implementace lexikálního analyzátoru
  - zásobník parametrů a jednosměrně vázaný seznam (ids_list_t)
  - LL gramatika a LL tabulka
  - implementace syntaktického analyzátoru (parser)
  - implementace sémantického analyzátoru (parser)
  - testování a dokumentace

- Adam Kaňkovský
  - návrh automatu pro lexikální analýzu
  - datové typy pro lexikální analyzátor
  - návrh precedenční tabulky a tabulky symbolů
  - binární vyhledávací strom
  - jednosměrně vázaný seznam (sym_linked_list)
  - generování kódu
  - testování (sémantický analyzátor, generování kódu)

- Radek Šerejch
  - návrh automatu pro lexikální analýzu
  - datový typ token_t
  - implementace lexikálního analyzátoru
  - návrh precedenční tabulky
  - implementace syntaktického analyzátoru (psa)
  - implementace sémantického analyzátoru (psa)
  - testování (lexikální analyzátor, syntaktický analyzátor)
  - generování kódu (psa)

2 Implementace překladače

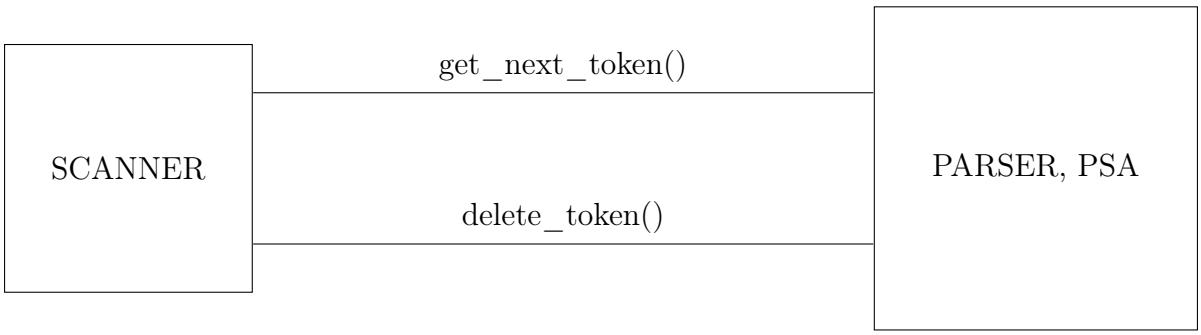
2.1 Lexikální analýza

2.1.1 Základní struktura lexikálního analyzátoru

Lexikální analyzátor je implementován v souboru *scanner.c (.h)*. Jeho hlavní funkcí implementující vnitřní logiku je funkce ***get_next_token()***. Načítání znaků ze standardního vstupu *stdin* probíhá pomocí cyklu *while*. Znaky jsou načítány, dokud není na vstupu konec souboru *EOF*. Pokaždé, kdy konečný automat přejde do koncového stavu je provádění funkce ukončeno a je navrácen načtený token. Při opětovném zavolání funkce je v cyklu *while* čten opět následující znak ze vstupu. Po ukončení provádění výše zmíněného cyklu bez stavu, který by navracel token, a tím ukončil provádění funkce, je kontrolováno, zda se konečný automat nachází v koncovém stavu. Tato situace je validní, pokud se konečný automat nachází v počátečním a zároveň koncovém stavu *init*. Funkce *get_next_token()* navrací hodnotu datového typu *token_t**, který obsahuje typ tokenu a jeho atribut. Datový typ atributu je odvozen na základě typu tokenu. Definice datového typu *token_t* je implementována v souboru *scanner.h*.

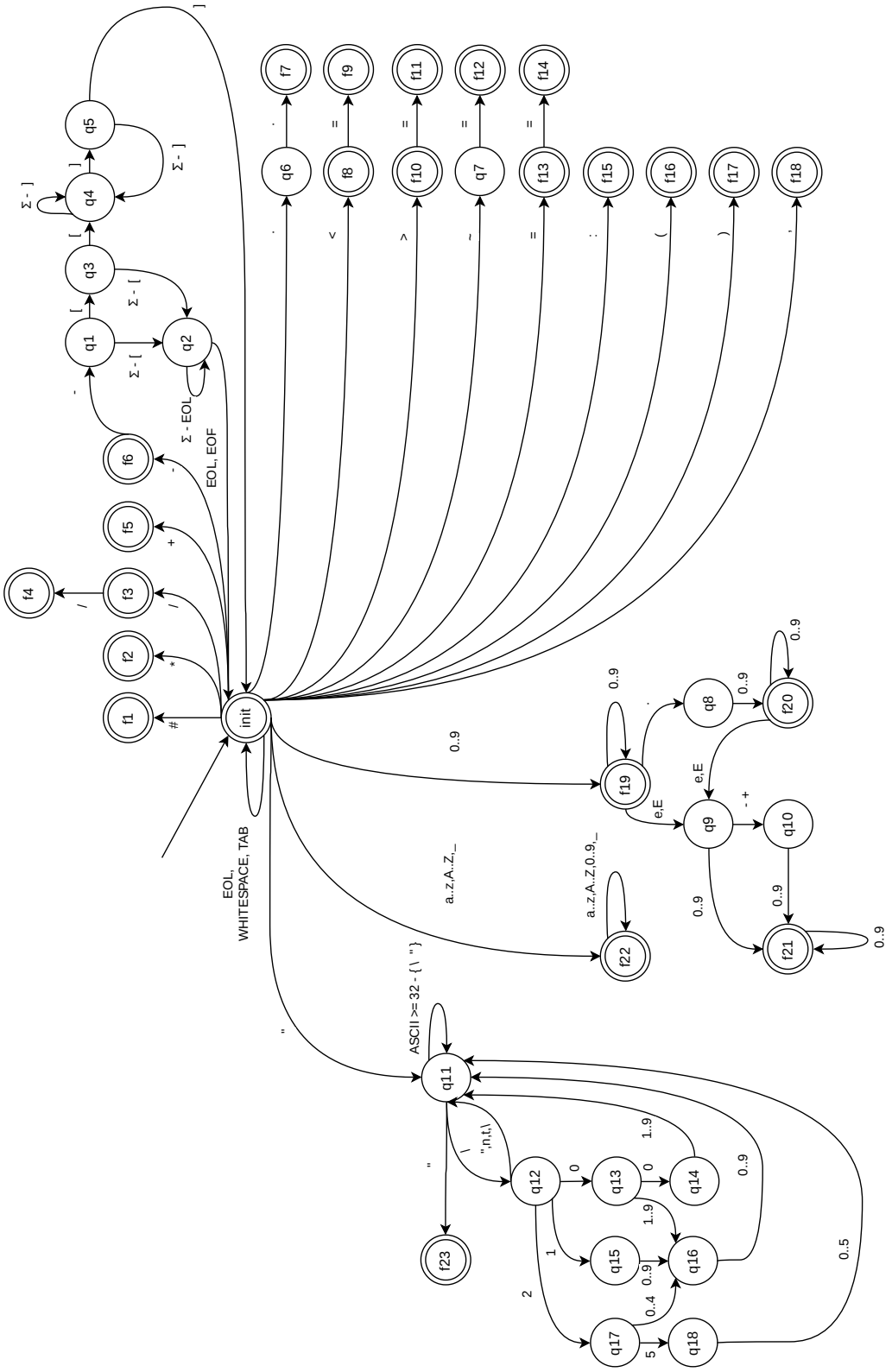
2.1.2 Rozhraní pro syntaktický analyzátor

Rozhraní mezi lexikálním (*scanner*) a syntaktickým (*parser, psa*) analyzátorem tvoří dvě funkce. První z nich je funkce ***get_next_token()***, načítající vstupní tokeny, a druhá je funkce ***delete_token()***, která slouží pro uvolnění paměti zaalokované pro již načtený token.



Obrázek 1: Rozhraní mezi lexikálním a syntaktickým analyzátorem

2.1.3 Diagram konečného automatu



Obrázek 2: Diagram konečného automatu

2.1.4 Legenda diagramu konečného automatu

- **q1** S_ONE_LINE_COMMENT
- **q2** S_ONE_LINE_COMMENT_CONTENT
- **q3** S_LEFT_SQUARE_BRACKET
- **q4** S_BLOCK_COMMENT_CONTENT
- **q5** S_RIGHT_SQUARE_BRACKET
- **q6** S_DOT
- **q7** S_TILDE
- **q8** S_DECIMAL_POINT
- **q9** S_EXP
- **q10** S_EXP_PLUS_MINUS
- **q11** S_STRING_CONTENT
- **q12** S_ESC_SEQ_BACKSLASH
- **q13** S_ESC_SEQ_ZERO
- **q14** S_ESC_SEQ_DOUBLE_ZERO
- **q15** S_ESC_SEQ_ONE
- **q16** S_ESC_SEQ_X_X
- **q17** S_ESC_SEQ_TWO
- **q18** S_ESC_SEQ_TWO_FIVE

- **f1** S_CHAR_CNT
- **f2** S_MUL
- **f3** S_DIV
- **f4** S_INT_DIV
- **f5** S_PLUS
- **f6** S_MINUS
- **f7** S_CONCAT
- **f8** S_LESS_THAN
- **f9** S_LESS_EQ
- **f10** S_GTR_THAN
- **f11** S_GTR_EQ
- **f12** S_NOT_EQ
- **f13** S_ASSIGN
- **f14** S_EQ
- **f15** S_COLON
- **f16** S_LEFT_BRACKET
- **f17** S_RIGHT_BRACKET
- **f18** S_COMMA
- **f19** S_INT
- **f20** S_DECIMAL
- **f21** S_DECIMAL_W_EXP
- **f22** S_IDENTIFIER_KEYWORD
- **f23** S_STRING

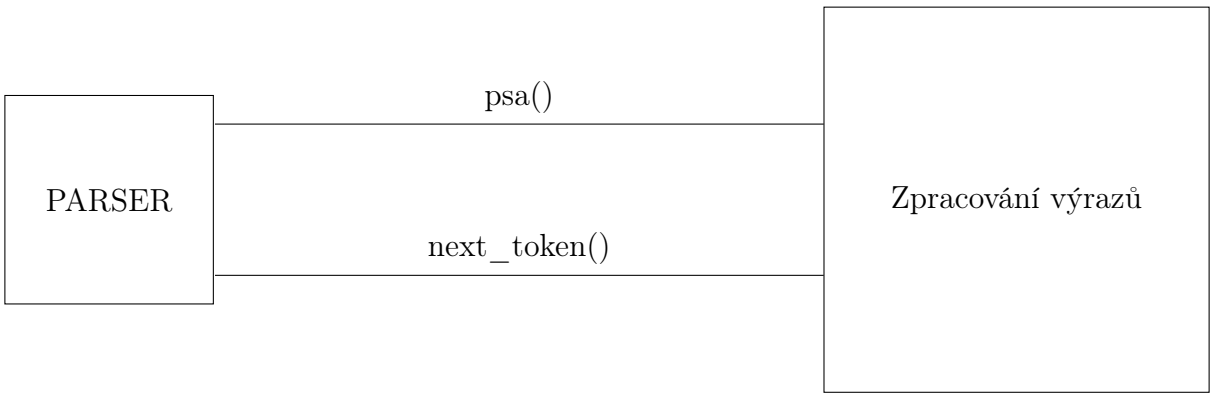
2.2 Syntaktická analýza

2.2.1 Parser

Implementace parseru se nachází v souboru *parser.c (.h)*. Syntaktická část parseru implementuje syntaktickou analýzu založenou na LL gramatice metodou rekurzivního sestupu [2]. Výše zmíněný soubor obsahuje také z pohledu logiky hlavní funkci celého překladače: ***parser()***. Tato funkce provádí všechny nezbytné úkony před započetím analýzy i po jejím skončení a volá funkci *prog()*, která zpracovává počáteční pravidlo (prolog) každého vstupního programu. Všechny potřebné parametry pro zpracování syntaktické a sémantické analýzy vstupního programu jsou uloženy ve struktuře, která je předávána jako parametr pomocí ukazatele. Tento parametr je datového typu *p_data_ptr_t*. Načtení tokenu lexikálním analyzátorem se provádí pomocí funkce ***next_token()***. Ta zajišťuje zavolání obou funkcí sloužících jako rozhraní mezi lexikálním a syntaktickým analyzátorem, a to funkcí *delete_token()* pro uvolnění paměti již zpracovaného tokenu a *get_next_token()* pro načtení následujícího tokenu. Parser zpracovává všechny části LL gramatiky až na výrazy. Ty jsou předávány pro zpracování metodou precedenční syntaktické analýzy.

2.2.2 Rozhraní mezi parserem a zpracováním výrazů (psa)

Rozhraní mezi parserem a zpracováním výrazů (psa) tvoří funkce ***psa()***. Ta je v parseru zapouzdřena do funkce *expression()*. Jako parametr funkce *psa()* je opět předáván ukazatel na strukturu obsahující všechna potřebná data datového typu p_data_ptr_t. Část syntaktického analyzátoru pro zpracování výrazů využívá funkci *next_token()* pro zpracování vstupních tokenů.



Obrázek 3: Rozhraní mezi parserem a zpracováním výrazů

2.2.3 LL gramatika

1. <prog> → require “ifj21” <main_b>

2. <main_b> → function id (<params>) <ret_func_types> <stats> end <main_b>
3. <main_b> → global id : function (<arg_def_types>) <ret_def_types> <main_b>
4. <main_b> → id (<args>) <main_b>
5. <main_b> → ε

6. <stats> → local id : <type> <assign> <stats>
7. <stats> → if exp then <stats> else <stats> end <stats>
8. <stats> → while exp do <stats> end <stats>
9. <stats> → return <ret_vals> <stats>
10. <stats> → id <id_func> <stats>
11. <stats> → ε

12. <id_func> → <n_ids> = <as_vals>
13. <id_func> → (<args>)

14. <params> → id : <type> <n_params>
15. <params> → ε
16. <n_params> → , id : <type> <n_params>
17. <n_params> → ε

18. <n_ids> → , id <n_ids>
19. <n_ids> → ε

20. <vals> → exp <n_vals>
21. <n_vals> → , exp <n_vals>

22. <n_vals> → ε

23. <as_vals> → <vals>
24. <as_vals> → id (<args>)

25. <ret_vals> → <vals>
26. <ret_vals> → ε

27. <assign> → = <assign_val>
28. <assign> → ε

29. <assign_val> → exp
30. <assign_val> → id (<args>)

31. <term> → id
32. <term> → <const>

33. <args> → <term> <n_args>

34. <args> → ε
35. <n_args> → , <term> <n_args>
36. <n_args> → ε

37. <arg_def_types> → <func_def_types>
38. <arg_def_types> → ε

39. <ret_func_types> → : <func_types>
40. <ret_func_types> → ε

41. <ret_def_types> → : <func_def_types>
42. <ret_def_types> → ε

43. <func_types> → <type> <n_func_types>
44. <n_func_types> → , <type> <n_func_types>
45. <n_func_types> → ε

46. <func_def_types> → <type> <n_func_def_types>
47. <n_func_def_types> → , <type> <n_func_def_types>
48. <n_func_def_types> → ε

49. <type> → integer
50. <type> → number
51. <type> → string
52. <type> → nil

53. <const> → int_value
54. <const> → double_value
55. <const> → string_value
56. <const> → nil

Poznámka: "exp"- označení pro výraz

2.2.4 LL tabulka

	require	function	global	id	local	if	while	return	end	else	,	=
<prog>	1											
<main_b>		2	3	4								
<stats>				10	6	7	8	9	11	11		
<id_func>											12	12
<params>				14								
<n_params>											16	
<n_ids>											18	19
<vals>												
<n_vals>				22	22	22	22	22	22	22	21	
<as_vals>				24								
<ret_vals>				26	26	26	26	26	26	26		
<assign>				28	28	28	28	28	28	28		27
<assign_val>				30								
<term>				31								
<args>				33								
<n_args>											35	
<arg_def_types>												
<ret_func_types>				40	40	40	40	40	40			
<ret_def_types>		42	42	42								
<func_types>												
<n_func_types>				45	45	45	45	45	45		44	
<func_def_types>												
<n_func_def_types>		48	48	48							47	
<type>												
<const>												

Obrázek 4: LL tabulka, část 1

	(	)	int_value	double_value	string_value	nil	integer	number	string	:	exp	$
<prog>												
<main_b>												5
<stats>												
<id_func>	13											
<params>		15										
<n_params>		17										
<n_ids>												
<vals>											20	
<n_vals>												
<as_vals>											23	
<ret_vals>											25	
<assign>												
<assign_val>											29	
<term>			32	32	32	32						
<args>		34	33	33	33	33						
<n_args>		36										
<arg_def_types>		38					37	37	37			
<ret_func_types>										39		
<ret_def_types>										41		42
<func_types>							43	43	43			
<n_func_types>												
<func_def_types>							46	46	46			
<n_func_def_types>		48										48
<type>						52	49	50	51			
<const>			53	54	55	56						

Obrázek 5: LL tabulka, část 2

2.2.5 Zpracování výrazů

Zpracování výrazů je implementováno v souboru *psa.c (.h)*. To je prováděno metodou precedenční syntaktické analýzy [2]. Nejprve je nainicializován zásobník (kapitola 3.1.1), kam se postupně budou ukládat symboly z precedenční tabulky, a na vrchol je vložen znak reprezentující konec čtení. Symboly jsou datového typu *psa_table_symbol_enum*. Poté je načten symbol z vrcholu zásobníku a symbol ze vstupu. Pro tyto symboly jsou určeny indexy pro čtení z precedenční tabulky (funkce ***get_index_enum()*** a ***get_index_token()***). Podle vyčtené hodnoty je buď přidán symbol na zásobník bez zarážky, se zarážkou, anebo je spuštěna redukce části výrazu na zásobníku. Redukce výrazu se provádí tím způsobem, že je nejprve určeno, kolik symbolů se nachází na vrcholu zásobníku do zarážky. Maximálně se může jednat o tři symboly. Poté pomocí funkce ***test_rule()*** se zjistí, zda existuje pro dané symboly pravidlo na redukci. Pravidla jsou definována v *psa_rules_enum*. Pomocí daného pravidla se zredukuje část výrazu. Redukce výrazu proběhne úspěšně pouze tehdy, pokud je na vstupu token daného výrazu a na vrcholu zásobníku je symbol pro konec čtení (počítáno bez výsledného neterminálu).

2.2.6 Precedenční tabulka

	#	+	-	*	/	//	..	<	>	<=	>=	~=	"=="	(	)	I	s	$
#		>	>	>		>		>	>	>	>	>	>	<	>	<	<	>
+	<	>	>	<	<	<		>	>	>	>	>	>	<	>	<		>
-	<	>	>	<	<	<		>	>	>	>	>	>	<	>	<		>
*	<	>	>	>	>	>		>	>	>	>	>	>	<	>	<		>
/	<	>	>	>	>	>		>	>	>	>	>	>	<	>	<		>
//	<	>	>	>	>	>		>	>	>	>	>	>	<	>	<		>
..							>						>	<	>	<	<	>
<	<	<	<	<	<	<		>	>	>	>	>	>	<	>	<	<	>
>	<	<	<	<	<	<		>	>	>	>	>	>	<	>	<	<	>
<=	<	<	<	<	<	<		>	>	>	>	>	>	<	>	<	<	>
>=	<	<	<	<	<	<		>	>	>	>	>	>	<	>	<	<	>
~=	<	<	<	<	<	<		>	>	>	>	>	>	<	>	<	<	>
"=="	<	<	<	<	<	<	<	>	>	>	>	>	>	<	>	<	<	>
(	<	<	<	<	<	<	<	<	<	<	<	<	<	<	"="	<	<	
)		>	>	>	>	>	>	>	>	>	>	>	>		>			>
I		>	>	>	>	>	>	>	>	>	>	>	>		>			>
s		>	>	>		>	>	>	>	>	>	>	>		>			>
$	<	<	<	<	<	<	<	<	<	<	<	<	<	<		<	<	

Obrázek 6: Precedenční tabulka

2.2.7 Gramatika pro výrazy

1. E → i
2. E → #E
3. E → (E)
4. E → E .. E
5. E → E + E
6. E → E − E
7. E → E ∗ E
8. E → E / E
9. E → E // E
10. E → E = E
11. E → E ∼= E
12. E → E <= E
13. E → E >= E
14. E → E < E
15. E → E > E

2.3 Tabulka symbolů

Dle výběru varianty zadání I je tabulka symbolů implementována binárním vyhledávacím stromem. Ten je implementován v souboru *symtable.c (.h)*.

2.3.1 Návrh tabulky symbolů

Prvek v tabulce symbolů obsahuje následující elementy:

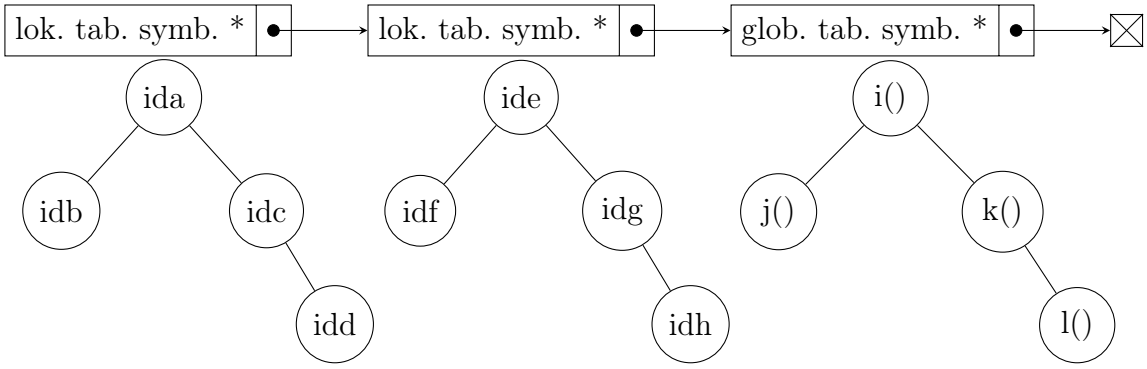
- declared - proměnná/funkce byla deklarována
- defined - proměnná/funkce byla definována
- data_type - datový typ proměnné
- params_count - počet parametrů definice funkce
- params_type_count - počet parametrů deklarace funkce
- returns_def_count - počet návratových typů definice funkce
- returns_count - počet návratových typů deklarace funkce
- first_param - ukazatel na první parametr definice funkce
- first_type_param - ukazatel na první parametr deklarace funkce
- first_def_ret - ukazatel na první návratový typ definice funkce
- first_ret - ukazatel na první návratový typ deklarace funkce

Všechny elementy prvku v tabulce jsou uloženy ve struktuře datového typu *symData_t*. Při inicializaci dat prvku jsou všechny elementy nastaveny na výchozí hodnotu a prvek pracuje pouze s ukazatelem na danou strukturu.

Jako element prvku tabulky symbolů není ukládána informace, zda daný identifikátor patří proměnné nebo funkci. Tuto informaci lze vydedukovat z definice jazyka. Pokud-li se nachází identifikátor v tabulce jiné než globální (tedy v lokální tabulce), jedná se o identifikátor proměnné (v jazyku IFJ21 nelze definovat vnořené funkce a v hlavním těle programu nelze deklarovat proměnné). Naopak jestliže se identifikátor nachází v globální tabulce, jedná se o identifikátor funkce.

2.3.2 Uložení tabulek symbolů do jednosměrně vázaného seznamu

Jednotlivé tabulky symbolů jsou vloženy do jednosměrně vázaného seznamu. Jeho implementace se nachází v souboru *sym_linked_list.c (.h)*. Tento způsob uložení tabulky reprezentující blok slouží pro vyhledávání identifikátoru proměnné v tabulkách symbolů. Lineární procházení seznamu se využívá při ověřování deklarace či definice proměnné. Identifikátory funkcí se nachází pouze v úplně posledním prvku seznamu a to v tabulce reprezentující globální tabulku celého programu.



* - ukazatel na kořen binárního stromu reprezentujícího tabulku symbolů

Obrázek 7: Diagram jednosměrně vázaného seznamu

2.4 Sémantická analýza

2.4.1 Parser

Sémantický analyzátor provádějící sémantickou analýzu vstupního programu mimo zpracování výrazů je implementován v souboru *parser.c*. Zmíněný analyzátor využívá tabulku symbolů a datovou strukturu obsahující potřebná data pro syntaktickou a sémantickou analýzu datového typu p_data_ptr_t.

2.4.2 Zpracování výrazů

Psa provádí sémantickou analýzu vždy při redukci podle pravidla. Než je redukováno, přidá se vždy při vkládání na zásobník k symbolu ještě jeho datový typ za účelem jeho kontroly při redukci. Pro získání datového typu je používána funkce ***get_ type()***. Při redukci se kontroluje kompatibilita datových typů a výsledný typ je uložen spolu s redukovaným neterminálem.

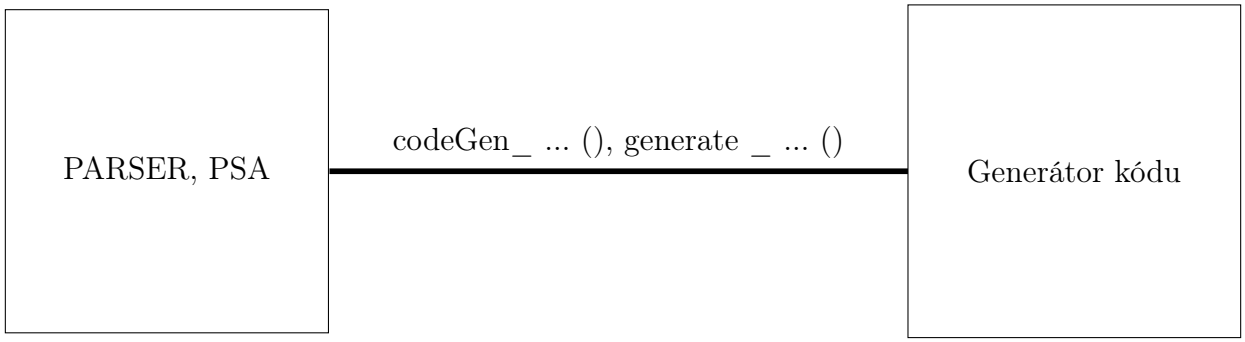
2.5 Generování kódu

2.5.1 Vlastní generování kódu

Ve většině případů je generovaný kód vkládán pomocí pomocných funkcích přímo na standardní výstup stdout. Pouze v případu cyklu while jsou všechny nedeklarační instrukce proměnných vkládány nejdříve do obousměrně vázaného seznamu a na konci cyklu jsou tyto výstupy vytisknuty na standardní výstup. Stínění je poté řešeno pomocí proměnné *scale*, která je přidávána za jméno proměnné a poté jsou tyto výsledné názvy vkládány na zásobník. Zásobník je používán při každé práci s proměnnou. Proměnné jsou mazány při vynoření z danného bloku (scale). Všechny podmínky *if* a cykly *while* jsou od sebe odlišené pomocí speciálního identifikátoru, který je také vkládán na zásobník a do něhož jsou identifikátory těchto příkazů přidány při zavolání výpisu daných instrukcí a nakonci jejich funkčí částí odebrány.

2.5.2 Rozhraní generátoru kódu

Jako rozhraní mezi parserem, psa a generátorem kódu slouží funkce ve tvaru *codeGen_ ... ()* a ve tvaru *generate_ ... ()*. Každá tato funkce vygeneruje určitou přidělenou část kódu jazyka IFJcode21. Zmíněné funkce jsou implementovány v souboru *code_ generator.c (.h)*.



Obrázek 8: Rozhraní mezi parserem, psa a generátorem kódu

3 Datové struktury a datové typy

3.1 Datové struktury

3.1.1 Zásobník symbolů (zpracování výrazů)

Zásobník symbolů je implementovám v souboru *symstack.c (.h)*. Zmíněný zásobník je používán při precedenční syntaktické analýze. Kromě základních funkcí push, pop, a mnohých dalších, byly implemetovány další potřebné funkce. Jsou to funkce ***symbol_stack_top_terminal()*** a ***symbol_stack_insert_after_top_terminal()***. Funkce *symbol_stack_top_terminal()* vrací ukazatel na nejvrchnější terminál na zásobníku a *symbol_stack_insert_after_top_terminal()* slouží pro vložení symbolu za vrchní terminál. Struktura obsahuje symboly, které jsou získávány pomocí funkce ***get_symbol_from_token()***. Spolu se symbolem je ukládán datový typ, který se používá při sémantické kontrole, a ukazatel na další prvek seznamu.

3.1.2 Zásobník parametrů (parser)

Zásobník parametrů je implementován v souboru *paramstack.c (.h)*. Slouží pro uložení parametrů volání funkce a je využíván v parseru. Tyto parametry jsou vkládány nejprve na zásobník za účelem docílení jejich opačného pořádí před jejich předáním generátoru kódu.

3.1.3 Binární vyhledávací strom (tabulka symbolů)

Binární vyhledávací strom reprezentující tabulku symbolů je implementován v souboru *symtable.c (.h)*. Jeho implementace je založena na standardním přístupu [1]. Při vkládání nového elementu je prováděna hluboká kopie z důvodu předávání parametru jako ukazatele na nový element.

3.1.4 Jednosměrně vázaný seznam (tabulka symbolů)

Jednosměrně vázaný seznam je implementován v souboru *sym_linked_list.c (.h)*. Jako jeho elementy jsou vkládány tabulky symbolů, přesněji řečeno ukazatele na kořenový uzel dané tabuky symbolů. Zmíněný seznam je procházen lineárně, což je využíváno například při získávání informace, zda proměnná či funkce byla deklarována nebo definována.

3.1.5 Jednosměrně vázaný seznam (identifikátory)

Jednosměrně vázaný seznam sloužící pro ukládání identifikátorů proměnných je implementován v souboru *ids_list.c (.h)*. Jeho využití je především při zpracování příkazu přiřazení s vícenásobným přiřazením do proměnných. Konkrétně se jedná například o kontrolu deklarace daných proměnných nebo jejich datových typů. Při předání identifikátorů proměnných generátoru kódu jsou předávány odzadu.

3.1.6 Obousměrně vázaný seznam (generování kódu)

Obousměrně vázaný seznam je implementovám v souboru *dll.c (.h)*. Slouží pro uchování vygenerovaného kódu. Pro konstrukci cyklu while jsou všechny deklarace posunuty před cyklus. Toho je docíleno pomocí funkce ***DLL_InsertLast()***, pomocí které je vkládán generovaný kód do seznamu, a poté pomocí funkce ***DLL_PrintAll()*** jsou všechny položky vytisknuty na standardní výstup.

3.2 Datové typy

3.2.1 string_t

Datový typ *string_t* je implementován v souboru *string.h*. Funkce pro práci s daným datovým typem jsou implementovány v souboru *string.c*. Zmíněný datový typ reprezentuje dynamický řetězec a je využíván v lexikálním analyzátoru při načítání znaků řetězce, identifikátoru, klíčového slova, celého nebo desetinného čísla.

3.2.2 token_t

Datový typ *token_t* je implementován v souboru *scanner.h*. Tento datový typ reprezentuje token načtený lexikálním analyzátorem. Obsahuje typ tokenu a jeho atribut. Datový typ atributu je odvozen na základě typu tokenu.

Reference

[1] Martin Mareš a Tomáš Valla. *Průvodce labyrintem algoritmů*. CZ.NIC, Praha, 2017.

[2] Alexander Meduna. *Elements of Compiler Design*. Taylor and Francis. Taylor & Francis Informa plc, 2008.