Projektová dokumentace

Implementace překladače jazyka IFJ20

Formální jazyky a překladače

Tým 002, varianta 2

Jan Polišenský	xpolis04	40 %
Nina Štefeková	xstefe11	30 %
Martin Benovič	xbenov00	20 %
Sabína Švecková	xsveck00	10 %

8. prosince 2020

Obsah

1 Úvod

Cílem tohoto projektu bylo vytvořit překladač v jazyce C zpracovávající jazyk IFJ20, který vychází z programovajího jazyka GO. Tento překladač jsme implementovali jako soubor kooperujících nezávislých modulů lexikální analýzy, syntaktické analýzy, sémantické analýzy a generování kódu.

Ačkoli je tato míra nezávislost jednotlivých modulů netradiční, umožňuje pohodlnou změnu funkcnionality jedné části bez zásahu do jiné. Toho se v průběhu implementace ukázalo jako značná výhoda oproti klasické jednoprůchodové či dvouprůchodové implementaci.

2 Rozdělení práce v týmu

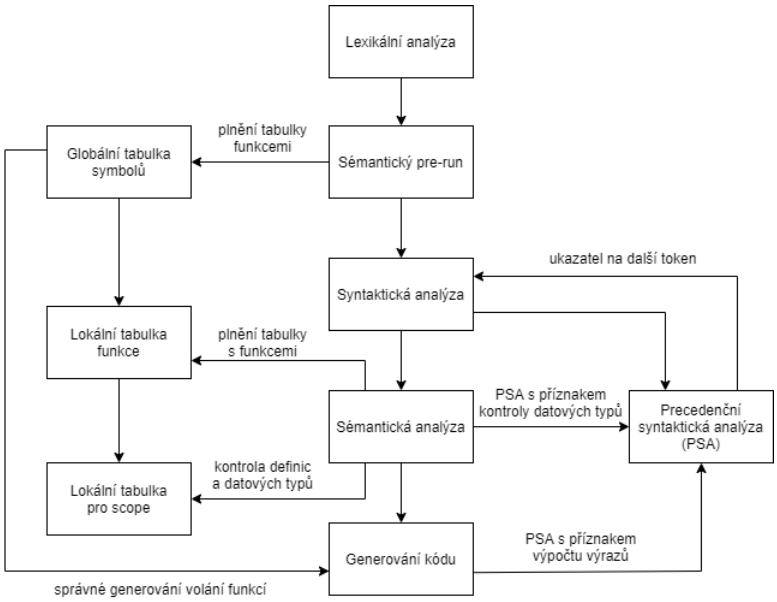
Jan Polišenský - návrh architektury, parser, generátor kódu, zpracování výrazů, tvorba gramatiky
Martin Benovič - dokumentace, testování, LL gramatika
Nina Štefeková - scanner, kód vestavěných funkcí
Sabína Švecková - scanner

2.1 Odůvodnění odchylek od rovnoměrnosti bodů

Složitá komunikace a absence prezenčních schůzek zapříčinily nerovnoměrné rozdělění v odpracovaných částech.

3 Implementace

Výsledná implementace vychází z níže uvedeného diagramu, vstupní kód je postupně sekvenčně zpracováván jednotlivými částmi našeho překladače. Hlavními bloky jsou pak lexikální analýza, sémantický prerun, syntaktická analýza a výsledné generování kódu.



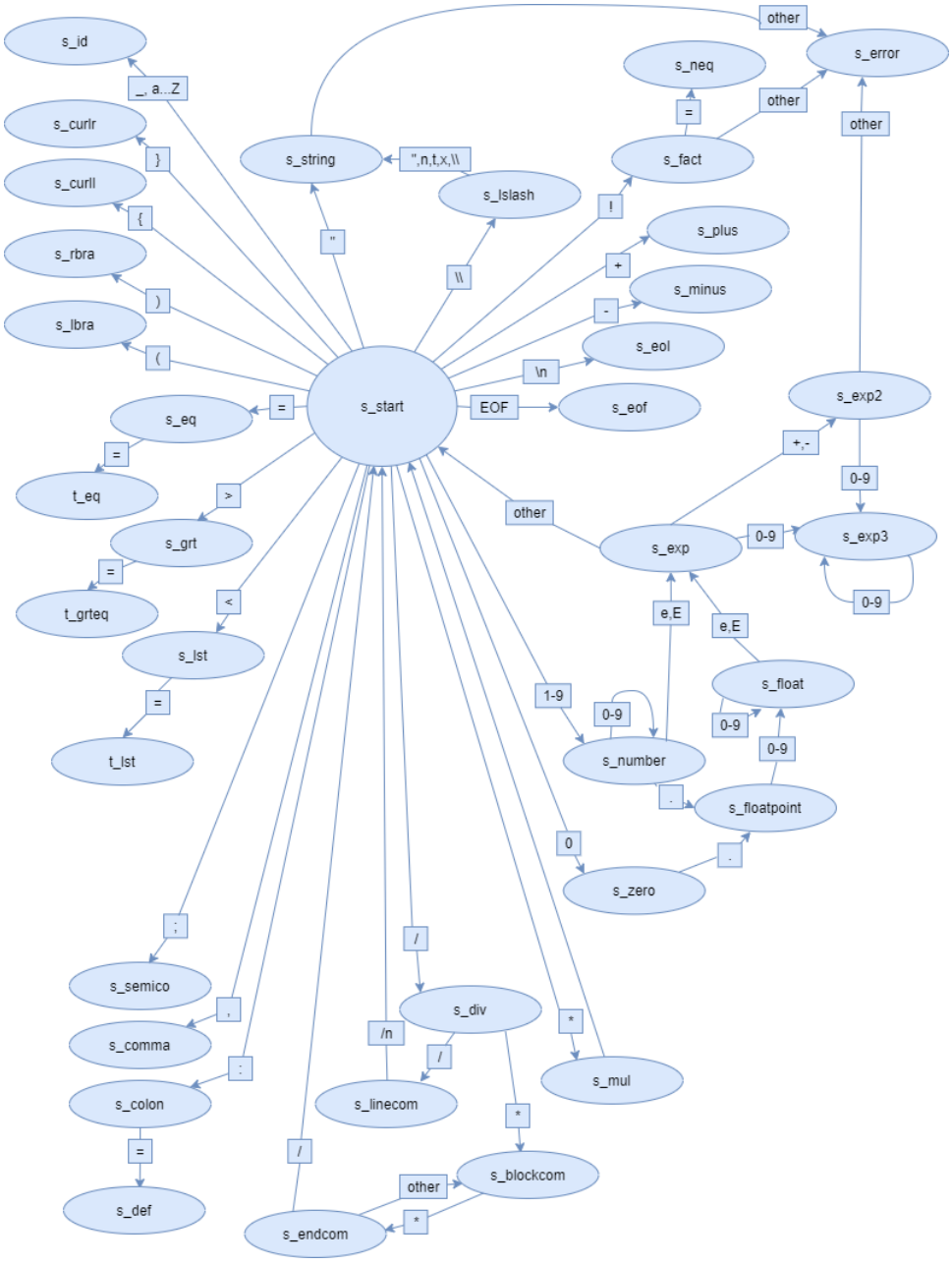
Obrázek 1: Schéma překladače

3.1 Lexikální analýza

Prvním krokem při tvorbě překladače byla implementace lexikální analýzy. Lexikální analyzátor jsme implementovali jako deteministický konečný automat. Před implementací jsme si automat navrhli, ale při samotné implementaci jsme ještě doplnili další stavy.

Automat jsme implementovali v jazyku C jako nekonečný `switch`, kde každý `case` odpovídá jednomu stavu automatu. Tento automat rozděluje zdrojový kód na jednotlivé tokeny. Tokeny jsou různé symboly, operátory, čísla, řetězce, identifikátory a klíčová slova. Pokud narazí na lexikální chybu, vrátí token `t_error`, který vyvolává chybu 1 při načítání tokenů v první části syntaktické analýzy.

Níže je pak uveden graf konečného automatu.



Obrázek 2: Schéma konečného automatu

3.2 Syntaktická a sémantická analýza

Před samotným spuštěním syntaktické analýzy je prováděn sémantický prerun. Jsou zde načítány tokeny ze scanneru a následně jsou zřetězovány do dvojsměrně vázaného lineárního seznamu, toto provázání tokenů nám umožuje se v průběhu překladu libovolně vracet či posouvat. Jsou zde také prováděny jednoduché sémantické akce a plnění globální tabulky symbolů. Tato funkce se nachází v souboru `preprocesor.c`.

Syntaktická a sémantická analýza je pak jádro celého překladače a nachází se v souboru `parser.c`. Syntaktická analýza je implementována metodou rekurzivního sestupu který se řídí LL-gramatikou. Sémantické akce jsou pak prováděny až po dokončení syntaktické analýza a jsou realizovány jako samostatný běh.

Během sémantické analýzy jsou pro každou funkci a blok kódu vytvářeny lokální tabulky symbolů pomocí kterých jsou kontrolovány korektní datové typy. Funkce, jejich parametry a návratové hodnoty jsou pak uloženy v globální tabulce symbolů

Níže je pak uvedena LL-tabulka a LL-gramatika podle níž je implementovaná syntaktická analýza

	EOL	KEYWORD	EOF	COMMA	ID	LBRA	CHECH_FOR_DEF_FUNCTION	EQ	ASSIGN	EXPR	ε
<n_prog>	1	2									
<n_func>	3	5	4								
<n_param_n>				7	8						6
<n_retvals>						10					9
<n_retval>				12							11
<n_fretval>	14			13							15
<n_body>	16, 19	19, 21, 22			19, 20		17, 19				18, 19
<n_body_id>				25				23	24		
<n_body_comma>									26		
<n_body_id_var>	29			28			27			28	
<n_fretvals>	30	31									32
<n_expr>				34						33	
<n_func_call>							35				
<n_call_param>				37, 38						38	36
<n_if>		39									
<n_for>		40									
<n_def>					42						41
<n_assign>					44						43

Tabulka 1: LL tabulka

1. <n_prog> -> EOL <n_prog>
2. <n_prog> -> KEYWORD ID <n_func>
3. <n_func> -> EOL <n_func>
4. <n_func> -> EOF
5. <n_func> -> KEYWORD ID LBRA <n_param_n> RBRA <n_retvals> CURLL <n_body> <n_retvals> CURLR <n_func>
6. <n_param_n> -> ε
7. <n_param_n> -> COMMA ID KEYWORD <n_param_n>
8. <n_param_n> -> ID KEYWORD <n_param_n>
9. <n_retvals> -> ε
10. <n_retvals> -> LBRA KEYWORD <n_retval> RBRA
11. <n_retval> -> ε
12. <n_retval> -> COMMA KEYWORD <n_retval>
13. <n_fretval> -> COMMA <n_expr> <n_fretval>
14. <n_fretval> -> EOL <n_fretval> EOL
15. <n_fretval> -> ε
16. <n_body> -> EOL <n_body> EOL
17. <n_body> -> <n_func_call> EOL <n_body>
18. <n_body> -> ε
19. <n_body> -> <n_body> EOL <n_expr>
20. <n_body> -> ID <n_body_id> EOL <n_body>
21. <n_body> -> <n_if> EOL <n_body>
22. <n_body> -> <n_for> EOL <n_body>
23. <n_body_id> -> EQ <n_body_id_var>
24. <n_body_id> -> ASSIGN <n_body_id_var>
25. <n_body_id> -> COMMA ID <n_body_id>
26. <n_body_comma> -> ASSIGN <n_body_id_var>
27. <n_body_id_var> -> <n_func_call>
28. <n_body_id_var> -> <n_expr>
29. <n_body_id_var> -> EOL
30. <n_fretvals> -> EOL
31. <n_fretvals> -> KEYWORD <n_expr> <n_fretval>
32. <n_fretvals> -> ε
33. <n_expr> -> expr
34. <n_expr> -> COMMA expr
35. <n_func_call> -> CHECK_FOR_DEF_FUNCTION LBRA <n_call_param> RBRA
36. <n_call_param> -> ε
37. <n_call_param> -> COMMA <n_expr> <n_call_param>
38. <n_call_param> -> <n_expr> <n_call_param>
39. <n_if> -> KEYWORD <n_expr> CURLL <n_body> CURLR KEYWORD CURLL <n_body> CURLR
40. <n_for> -> KEYWORD <n_def> SEMICO <n_expr> SEMICO <n_assign> CURLL <n_body> CURLR
41. <n_def> -> ε
42. <n_def> -> ID ASSIGN <n_expr>
43. <n_assign> -> ε
44. <n_assign> -> ID EQ <n_expr>

Tabulka 2: LL gramatika

3.3 Zpracování výrazů

Výrazy jsou zpracovávány pomocí precedenční syntaktické analýzy, jejíž implementace se nachází v souborech `psa_prerun.c`. a `PSA.c`. PSA je v naší implementaci klíčovým komponentem, neboť zpracování výrazu je třeba jak na syntaktické, tak i na sémantické úrovní, PSA je také nutné na korektní generování kódu výrazů. Z těchto požadavků pak plyne poněkud obsáhlejší implementace PSA, kterou bylo nutné ještě doplnit preprocesorem výrazů umístěným v souboru `psa_prerun.c`. PSA je tedy možné spouštět ve 3 režimech:

1. kontrola syntaxe výrazů
2. kontrola sémantiky výrazů
3. generování kódu z výrazů

PSA pro správné zpracování výrazů využívá vlastní zásobník a lokální tabulky symbolů. Níže je pak uvedena precedenční tabulka symbolů využitá v PSA.

	+	-	*	/	R	(	)	id	int	float	str	$
+	>	>	<	<	<	<	>	<	<	<	<	>
-	>	>	<	<	<	<	>	<	<	<	<	>
*	>	>	>	>	>	<	>	<	<	<	<	>
/	>	>	<	>	>	<	>	<	<	<	<	>
R	<	<	<	<		<	>	<	<	<	<	>
(	<	<	<	<	<	<		<	<	<	<	
)	>	>	>	>	>		>					>
id	>	>	>	>	>		>					>
int	>	>	>	>	>		>					>
float	>	>	>	>	>		>					>
str	>	>	>	>	>		>					>
$	<	<	<	<	<	<		<	<	<	<	

Tabulka 3: Precedenční tabulka

3.4 Tabulka symbolů

Tabulky symbolů jsou implementovány jako tabulky s rozptýlenými položkami s zřetězenými synonymy. Velikost mapovacího pole jsme dimenzovali aby nikdy nepřesáhla polovinu a je rovna prvočíslu 27457. Pro zřetězení synonym jsou využité linearní seznamy.
Funkce pro práci s tabulkou se nacházejí v souboru `symtable.c`.(kopírování tabulky, kontrola alokace tabulky, dealokace tabulky atd.).

3.5 Generování kódu

Hlavní funkce pro generován kódu se nachází v souboru `codegen.c`. Samotné generování pak probíhá v několika krocích. Nejprve je vygenerovaný kód vestavěných funkcí, tzv "base code", nachází se v souboru `code.c`.

Vstupní program je poté po jednotlivých řádcích procházen zdrojový kód a na základě prvního nebo i následujícího tokenu na každém řádu jsou volány jednotlivé funkce na generování konkrétních částí kódu (definice funkce, volání funkce). K vygenerování kódu z výrazů je pak použit precedenční syntaktický analyzátor s příznakem počítání výrazů. Ten k výpočtu výrazů využívá datový zásobník interpretu. Kvůli netradiční architektuře nastává v generátoru kódu problém s kontrolou platnosti jednotlivých proměnných v konkrétních blocích. Nejsou již totiž k dispozici lokální tabulky symbolů. Tento problém je řešen pomocí zásobníku na jehož vrhol se ukládá aktuálně platná proměnná daného jména.