Tým xhricma00 varianta vv-BVS

Marek Hric	Mikuláš Lešiga	Roman Andraščík	Adam Veselý
xhricma00	xlesigm00	xandrar00	xvesela00

December 5, 2024

Rozdelenie bodov

xhricma00: 25%
xlesigm00: 25%
xandrar00: 25%
xvesela00: 25%

Rozšírenia

ORELSE
UNREACHABLE
BOOLTHEN
FOR
WHILE
FUNEXP

Rozdelenie prace :

Marek Hric :

- Lexikálna analýza
- Generovanie kodu

Mikuláš Lešiga :

- Syntaktická analýza
- Semantická analýza
- LL Gramatika
- Tabulky

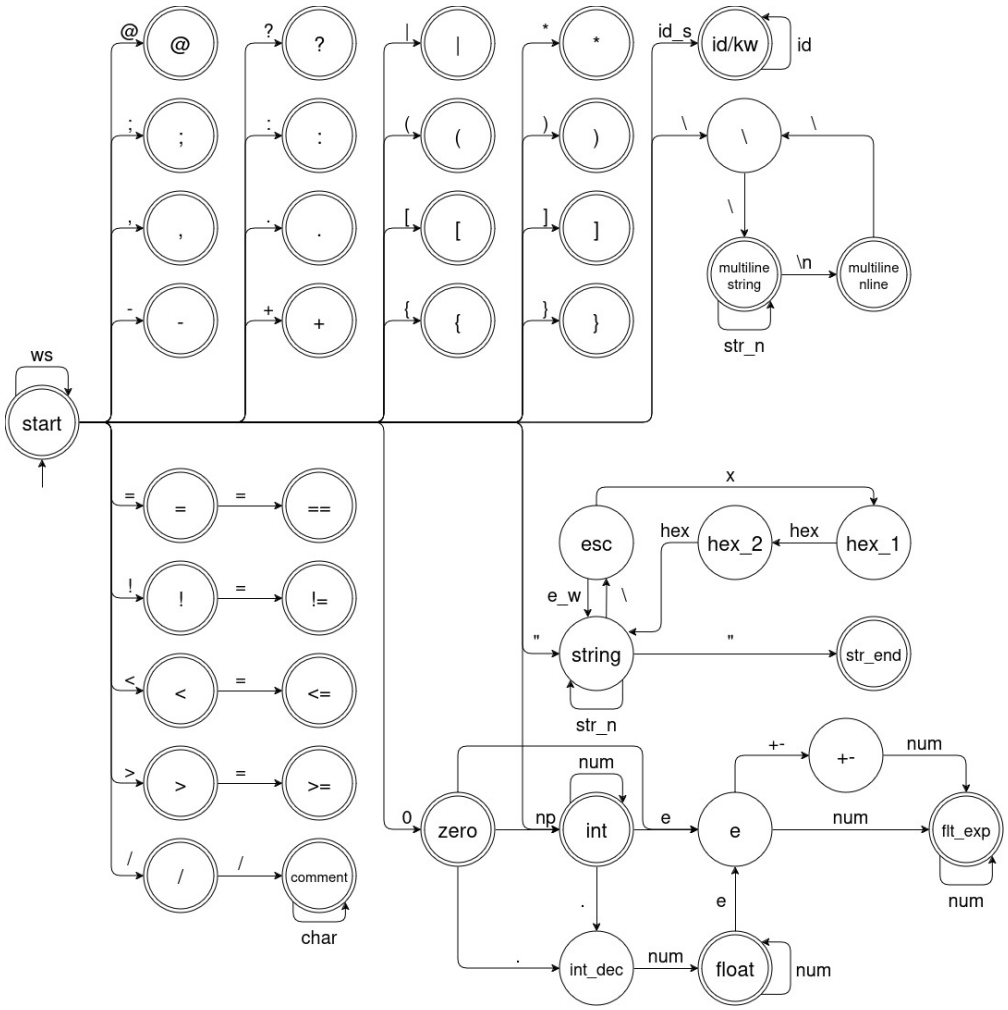
Roman Andraščík :

- Testy
- Pomocné funkcie
- Symtable
- Dokumentácia

Adam Veselý :

- Syntaktická analýza
- Semantická analýza
- Dátové štruktúry

Diagram konečného automatu :



Legenda:

- **ws** - white space
- **id_s** - znaky zaciatku identifikatora (_a-zA-Z)
- **id** - znaky identifikatora (_a-zA-Z0-9)
- **num** - cisla (0-9)
- **np** - cisla bez nuly (1-9)
- **hex** - hexa cisla (0-9a-fA-F)
- **e** - exponent (eE)
- **char** - lubovolny znak
- **e_w** - znaky escape sekvencii bez x (n,t,r)
- **str_n** - lubovolny znak bez $\backslash n$

LL-gramatika :

1. $< prog > \rightarrow < prolog >< function\_def >< end >$
2. $< prolog > \rightarrow$ const ID = @$import(< expression >)$;
3. $< end > \rightarrow EOF$
4. $< function\_def > \rightarrow$ pub fn ID$(< param_list >) < function_type >< block >< function_def >$
5. $< function\_def > \rightarrow \varepsilon$
6. $< param\_list > \rightarrow ID :< type\_complete >< comma\_par\_found >$
7. $< param\_list > \rightarrow \varepsilon$
8. $< comma\_par\_found > \rightarrow , < param\_list >$
9. $< comma\_par\_found > \rightarrow \varepsilon$
10. $< block > \rightarrow \{< statement >\}$
11. $< statement > \rightarrow < var\_declaration >< statement >$
12. $< statement > \rightarrow ID < ID\_found >< statement >$
13. $< statement > \rightarrow < if\_statement >< statement >$
14. $< statement > \rightarrow < for\_loop >< statement >$
15. $< statement > \rightarrow < while\_loop >< statement >$
16. $< statement > \rightarrow < return\_statement >< statement >$
17. $< statement > \rightarrow < break >< statement >$
18. $< statement > \rightarrow < continue >< statement >$
19. $< statement > \rightarrow \varepsilon$
20. $< ID\_found > \rightarrow =< asgn\_found >;$
21. $< ID\_found > \rightarrow (< expression\_list >);$
22. $< ID\_found > \rightarrow :< while\_loop >$
23. $< var\_declaration > \rightarrow$ const ID $:< type\_complete >=< asgn\_found >;$
24. $< var\_declaration > \rightarrow$ var ID $:< type\_complete >=< asgn\_found >;$
25. $< if\_statement > \rightarrow if(< expression >) < if\_found >$

26. $< if\_found > \ \rightarrow \ < optional\_value >< block >< else\_statement >$
27. $< else\_statement > \ \rightarrow \ else < block >$
28. $< else\_statement > \ \rightarrow \ \varepsilon$
29. $< optional\_value > \ \rightarrow \ |ID|$
30. $< optional\_value > \ \rightarrow \ \varepsilon$
31. $< while\_loop > \ \rightarrow \ while(< expression >) < optional\_value >< optional\_statements >< block >< else\_statement >$
32. $< return\_statement > \ \rightarrow \ return < expression >;$
33. $< expression\_list > \ \rightarrow \ < expression >< comma\_expr\_found >$
34. $< expression\_list > \ \rightarrow \ \varepsilon$
35. $< comma\_expr\_found > \ \rightarrow \ , < expression\_list >$
36. $< comma\_expr\_found > \ \rightarrow \ \varepsilon$
37. $< type > \ \rightarrow \ i32$
38. $< type > \ \rightarrow \ f64$
39. $< type > \ \rightarrow \ []u8$
40. $< type > \ \rightarrow \ bool$
41. $< for\_loop > \ \rightarrow \ for(< expression >) < optional\_value >< block >$
42. $< optional\_statements > \ \rightarrow \ :< block >$
43. $< optional\_statements > \ \rightarrow \ \varepsilon$
44. $< type\_complete > \ \rightarrow \ < question\_mark >< type >$
45. $< question\_mark > \ \rightarrow \ ?$
46. $< question\_mark > \ \rightarrow \ \varepsilon$
47. $< single\_statement > \ \rightarrow \ < var\_declaration >$
48. $< single\_statement > \ \rightarrow \ ID < ID\_found >$
49. $< single\_statement > \ \rightarrow \ < if\_statement >$
50. $< single\_statement > \ \rightarrow \ < for\_loop >$
51. $< single\_statement > \ \rightarrow \ < while\_loop >$

52. $< single\_statement > \rightarrow < return\_statement >$
53. $< single\_statement > \rightarrow continue;$
54. $< single\_statement > \rightarrow break;$
55. $< function\_type > \rightarrow < type >$
56. $< function\_type > \rightarrow void$
57. $< asgn\_found > \rightarrow < expression >$
58. $< continue > \rightarrow continue;$
59. $< break > \rightarrow break < cycle\_current\_control >$
60. $< break > \rightarrow continue < cycle\_current\_control >$
61. $< cycle\_current\_control > \rightarrow ID$
62. $< cycle\_current\_control > \rightarrow \varepsilon$

LL-tabulka :

	const	ID	=	@	import	<expression>	(	)	;	EOF	pub	fn	ε	:	,	{	}	.	var	if	else	|	while	return	i32	f64	[	]	u8	void	bool	for	?	break	continue
<prog>	1																																		
<prolog>	2																																		
<end>										3																									
<function_def>											4		5																						
<param_list>		6											7																						
<comma_par_found>													9		8																				
<block>																10																			
<statement>	11	12											19						11	13			15	16								14		17	18
<ID_found>			20				21							22				23																	
<var_declaration>	24																		25																
<if_statement>																				26															
<if_found>													27									27													
<else_statement>													29								28														
<optional_value>													31									30													
<while_loop>																							32												
<return_statement>																								33											
<expression_list>						34							35																						
<comma_expr_found>													37		36																				
<type>																									38	39	40				41				
<for_loop>																																42			
<optional_statements>													44	43																					
<type_complete>													45																				45		
<question_mark>													47																				46		
<single_statement>	48	49																	48	50			52	53								51		54	55
<function_type>																									56	56	56			57	56				
<break>																																		58	
<continue>																																			59
<asgn_found>						60																													
<cycle_current_control>		61											62																						

Precendecna-tabulka :

	.?	!	* /	+ -	orelse	r	and	or	(	)	i	$
.?	<								<	>	<	>
!		<	>	>	>	>	>	>	<	>	>	>
* /		<	>	>	>	>	>	>	<	>	<	>
+ -		<	<	>	>	>	>	>	<	>	<	>
orelse		<	<	<	>	>	>	>	<	>	<	>
r		<	<	<	<		>	>	<	>	<	>
and		<	<	<	<	<	>	>	<	>	<	>
or		<	<	<	<	<	<	>	<	>	<	>
(	<	<	<	<	<	<	<	<	<	=	<	
)	>	>	>	>	>	>	>	>		>		>
i	>		>	>	>	>	>	>		>		>
$	<	<	<	<	<	<	<	<	<		<	:)

Lexikálna analýza

Riešenie lexikálnej analýzy sme začali vytvorením diagramu deterministického konečného automatu. Následne sme na jeho základe začali vypracovávať implementáciu. Implementácia sa nachádza v súbore *scanner.c,* ktorý pracuje s tokenmi deklarovanymi v súbore *token.h*. Hlavnou funkciou *scanner.c* je funkcia *get_token*. Pre uľahčenie práce a prehľadnosti kódu sme si deklarovali niekoľko makier, ktoré sú extensivne používané v hlavnej funkcii. Funkcia *get_token* berie postupne znaky zo štandardného vstupu a vytvára token. Tokenu je priradeny jeho typ a hodnota, ktorá mu odpovedá. Funkcia začína určovaním jednoznakovych tokenov, ktoré vie určite hneď na začiatku. Pokračuje identifikáciou komentárov, ktoré následne ignoruje. Po identifikácii komentárov zisťuje či sa jedna o ID alebo Keyword, pri kľúčových slovách sa následne určuje aj ich typ. Ak sa nejedna ani o jedno pokračuje kontrolou dátových typov, pri ktorých ukladá aj ich hodnoty.

Syntaktická analýza

Riešenie syntaktickej analýzy sme započali vytvorením LL gramatiky, LL tabuľky a precendecnej tabuľky. Následne na ich základe sme vypracovali súbor *parser.c a exp_parser.c.*Tieto súbory pracujú s uzlami deklarovanymi v súbore *ast.h*. Spustenie syntaktickej analýzy započne zavolanim funkcie *Parse()*. Tato funkcia postupne prechádza cez tokeny a priradzuje ich do uzlov, pomocou ktorých postupne tvorí abstraktný syntakticky strom na základe LL gramatiky. Súbor *parser.c* ďalej riadi aj precedencnu analýzu volaním funkcii zo súboru *exp_parser.c*. Tento súbor vytvorí strom výrazov, ktorý je následne pripojení do syntaktického stromu.

Semantická analýza

Sémantická analýza je implementovana v súboroch *sem_anal.c , symtable.c, sem_anal.h a symtable.h*. Spustenie sémantickej analýzy započne zavolanim funkcie *analyse()*. Sémantická analýza je založená na rekurzivnom prechode AST stromu, ktorý je prevzatí od funkcie *parse()*, ktorá ho vygenerovala. Funkcia ďalej využíva globálne deklarovany AST strom, v ktorom sa nachádzajú built in funkcie. Pri generácií nového AST stromu sa do neho vkladajú built in funkcie práve z tohto globálneho stromu. Funkcia *analyse()* po spustení hľadá *main* a následne postupne rekurzivne prechádza cez AST strom, kde kontroluje validitu dátových typov uložených v ASTNode štruktúrach. Po tejto kontrole započne aj kontrola navratovych hodnôt. Po úspešnej validacii dát predáva nový AST strom funkcii *codegen()*, ktorá začína generaciu kódu. Ak validacia neprebehne

úspešne, vyhlási sematicku chybu.

Symtable, ktorý tato funkcia využíva je implementovany ako AVL strom.

Generovanie kodu

Generátor je implementovany v súboroch *codegen_priv.h. codegen.h a codegen.c* .Spustenie generácie kódu započne zavolanim funkcie *codegen()*. Kód je generovaný na základe rekurzivneho prechadzania abstraktivneho syntaktického stromu podľa dátového typu uloženého v štruktúre *ASTNode*. Kód ďalej využíva aj pomocný Linked List na ukladanie deklarovanych premenných v danej funkcii.

Dátové štruktúry

Character Buffer

Implementovane v súboroch *c_buff.c c_buff.h*.

Implementácia Character Buffer je využitá hlavne v časti Scanner, kde slúži na bezpreblemove získavanie dať a ich následnú validaciu. Na prácu so scannerom ho neskôr využívajú aj časti Parser a Expression Parser. Štruktúra obsahuje klasické funkcie *c_buff_init, c_buff_free, c_buff_enqueue, c_buff_dequeue, c_buff_is_empty*.

Dynamic String

Implementovane v súboroch *dyn_str.c, dyn_str.h*.

Implementácia dynamického reťazca je využitá hlavne v Scanner časti programu, kde sprostredkuvava validaciu a uschovavanie dať, neskôr je použitá aj v časti Codegen, kde slúži na uľahčenie validacie dať. Štruktúra dynamického reťazca obsahuje klasické funkcie *dyn_str_init, dyn_str_grow, dyn_str_append, dyn_str_append_str a dyn_str_free*.

Stack

Implementovane v súboroch *stack.c, stack.h*.

Implementáciu nášho zásobníku využívame v Expression Parser časti programu. Štruktúra zásobníku je implementovana s klasickými funkciami *stackInit, stackPush, stackPop, stackIsEmpty, stackClear a stackGetTop*. Zásobník sme zvolili pre jeho optimálny prístup k dátam a zachovanie jednoduchosti kódu.

Poznámky

BOOLTHEN

Nepodporuje :

- Viac relačných operátorov za sebou