VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
FAKULTA INFORMAČNÍCH TECHNOLOGIÍ

Documentation

Implementation of translator for the imperative language IFJ22

Team xmarsha15, variant TRP

Teamleader: Martin Maršalek - xmarsa15 - 25%

Tomáš Frátrik - xfratr01 - 25%

Andrej Smatana - xsmata03 - 25%

János László Vasík - xvasik05 - 25%

No implemented extensions

Brno, December 7, 2022

Contents

1 Divison of implementation

- Martin Maršalek:
  - Basic parser and symbol table implementation
  - Symbol table stack implementation
  - Parser structure implementation
  - Draft of LL grammar
  - Implementation of semantic checks
  - Correction and debug of symbol table and syntax rules
- Tomáš Frátrik:
  - Draft and implementation of precedence table
  - Implementation of expression processing
  - Implementation of expression stack
  - Code generation while processing expressions
  - Debug of code generator
  - Correction and debug of expr.c
- Andrej Smatana:
  - Draft of token types and FSM states
  - Debug and refactor of scanner module
  - Implementation of string_t data type
  - Error code interface
  - Draft and debug of code generator
  - Debug and refactor of expression processing
- János László Vasík:
  - Draft of FSM
  - Correction of LL grammar
  - Draft, debug and refactor of scanner module
  - Refactor of syntax rules
  - Draft of interfaces between modules
  - Documentation

2 Lexical analysis

2.1 Scanner - tokenizer

The lexical analysis of incoming tokens was implemented in the file *scanner.c (.h)*. The main function of this unit is the function *scanToken()*, which on request from the interface of the syntactic analyzer unit processed the next token from the input source, which is the *stdin*, and returned either an error code or a token which was represented as a data structure *token_t*. The processing of tokens was implemented the following way. Upon request, the scanner started a *while cycle*, which went on until the managing final state machine either got to an end state with no more valid characters to read from input and returned a token, or encountered a lexically incorrect character sequence, in which case the scanner returned *ERR_LEX_ANALYSIS*. If the analysis of the current token did not find any errors, the scanner module would set the token's type based on the state the FSM ended in and the characters that the current token contained. Also in the case of identifier and constant tokens it attached the corresponding sequence of characters into the token it later returned. The diagram of states of the final state machine is illustrated in subsection 2.3.

2.2 Parser - interface between lexical and syntactic analysis

The interface between lexical and syntactic analysis was implemented in the file *parser.c (.h)*. The main function of this unit is the function *getNextToken()*. The structure *parser_t* holds two tokens at all time, and upon calling *getNextToken()* from the syntactic analyzer the parser unit replaces the first token in the *parser_t* structure with the token it stored as second, and calls the function *scanToken()* of *scanner.c*, which replaces the second token in the structure.

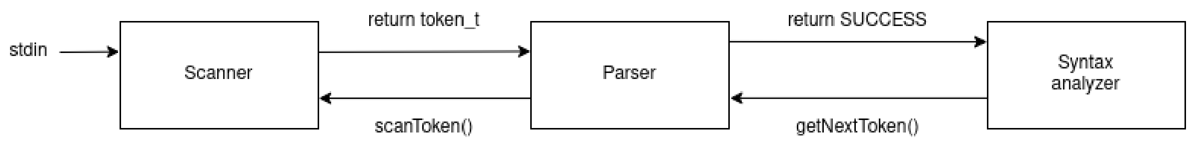
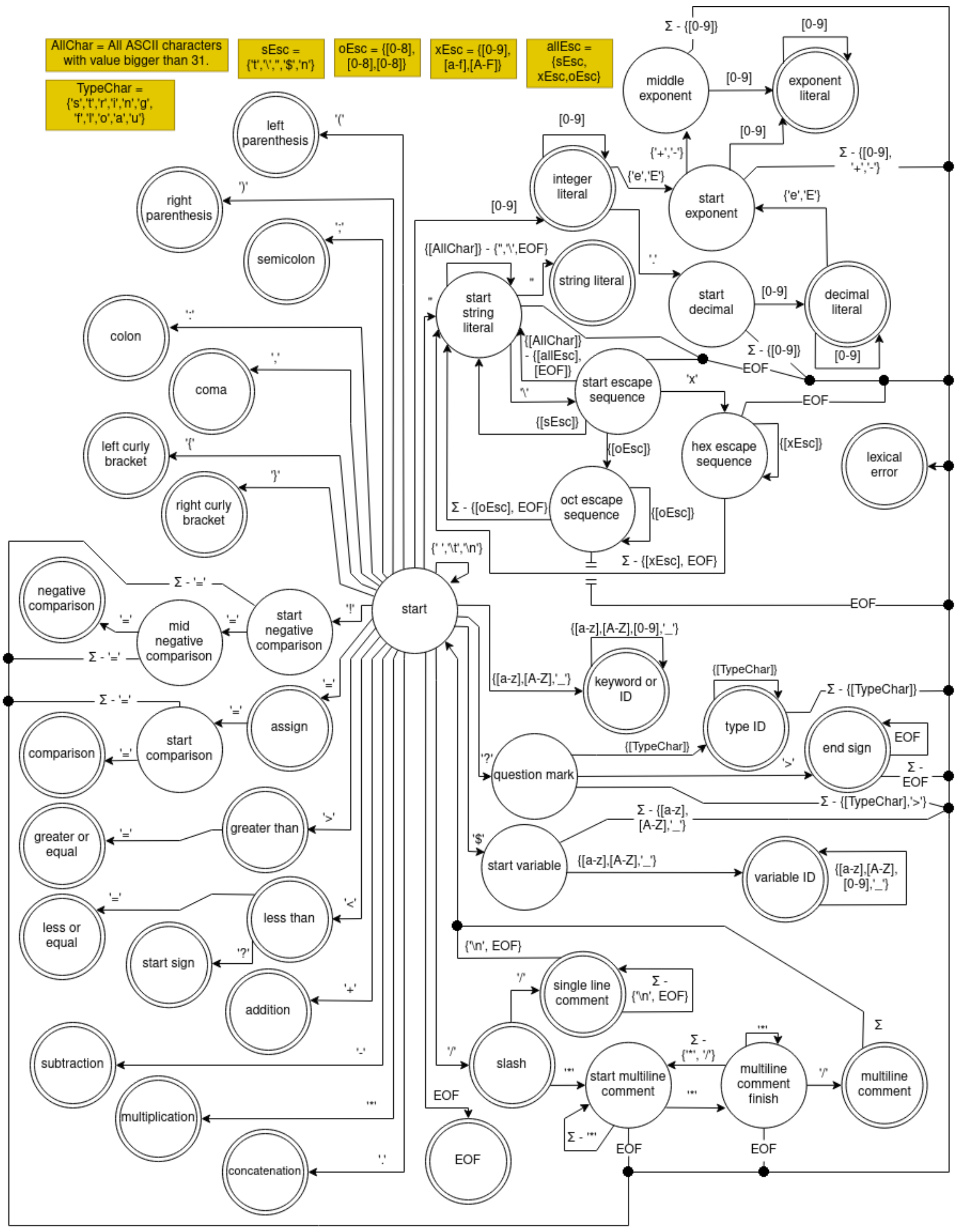


Figure 1: Interface between the lexical and syntactic analyzer units

2.3 Final state machine diagram



3 Syntactic analysis

3.1 Syntax rules

In the file *syntax_rules.c(.h)* is the implementation of the directive functions which cover the top to bottom part of the syntactic analysis. The rules of the implemented LL grammar are to be found in subsection 3.2. The implementation of rules was made with a recursive descent algorithm that begins descending from the first rule function, which is thus the main function of this file, *rProgram()*. When reaching given terminals and non-terminals, code is generated and the semantic and syntactic checks are being made. All the needed data for syntactical analysis is being held by the structure *parser_t*, into which upon calling *getNextToken()* the next token is copied by the parser unit. This part of the syntactical analysis processes everything apart from the expressions that are found in the input stream. The symbol table needed for the semantic analysis was realized using a hash table (structure *htab_t*), and for the same checks inside function bodies a one way list of such hash tables (structure *sym_stack_t*) was used.

3.2 LL grammar

1. <**program**> → <prolog> <units> <program_end>

2. <**prolog**> → "<?php" "declare(strict_types=1)" ";"

3. <**units**> → <unit> <units>
4. <**units**> → ε

5. <**unit**> → <function_definition>
6. <**unit**> → <statements>

7. <**function_definition**> → "function" ID "(" <params> ")" ":" <type>
   "{" <statements> "}"

8. <**params**> → <param> <params_n>
9. <**params**> → ε

10. <**param**> → <type> $ID

11. <**param_n**> → "," <param>

12. <**params_n**> → <param_n> <<params_n>>
13. <**params_n**> → ε

14. <**type**> → INT
15. <**type**> → STRING
16. <**type**> → FLOAT

17. <**statements**> → <variable_statement> <statements>
18. <**statements**> → <conditional_statement> <statements>
19. <**statements**> → <while_loop_statement> <statements>
20. <**statements**> → <function_call_statement> <statements>
21. <**statements**> → <return_statement> <statements>
22. <**statements**> → <expression> <statements>
23. <**statements**> → ε

24. <**variable_statement**> → <assignment_statement>
25. <**variable_statement**> → <expression>
26. <**variable_statement**> → $ID ";"

27. <**assignment_statement**> → $ID "=" <expression>
28. <**assignment_statement**> → $ID "=" <function_call_statement>

29. <**conditional_statement**> → "if" "(" <expression> ")" "{" <statements> "}"
"else" "{" <statements> "}"

30. <**while_loop_statement**> → "while" "(" <expression> ")"
"{" <statements> "}"

31. <**function_call_statement**> $\rightarrow$ ID "(" <arguments> ")" ";"

32. <**return_statement**> $\rightarrow$ "return" <return_value>

33. <**arguments**> $\rightarrow$ <term> <arguments_n>
34. <**arguments**> $\rightarrow$ ε

35. <**argument_n**> $\rightarrow$ "," <term>

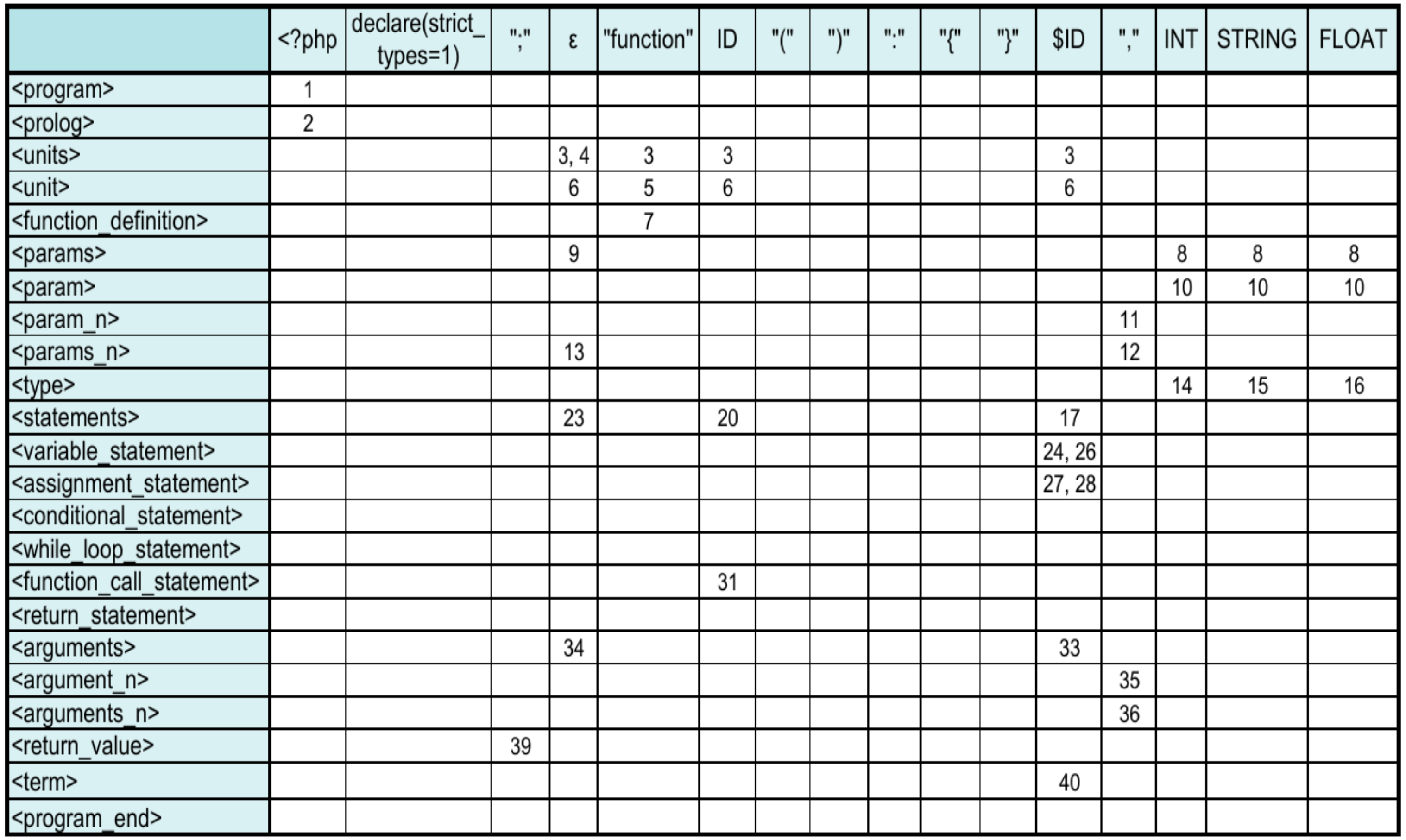
36. <**arguments_n**> $\rightarrow$ <argument_n> <arguments_n>
37. <**arguments_n**> $\rightarrow$ ε

38. <**return_value**> $\rightarrow$ <expression>
39. <**return_value**> $\rightarrow$ ";"

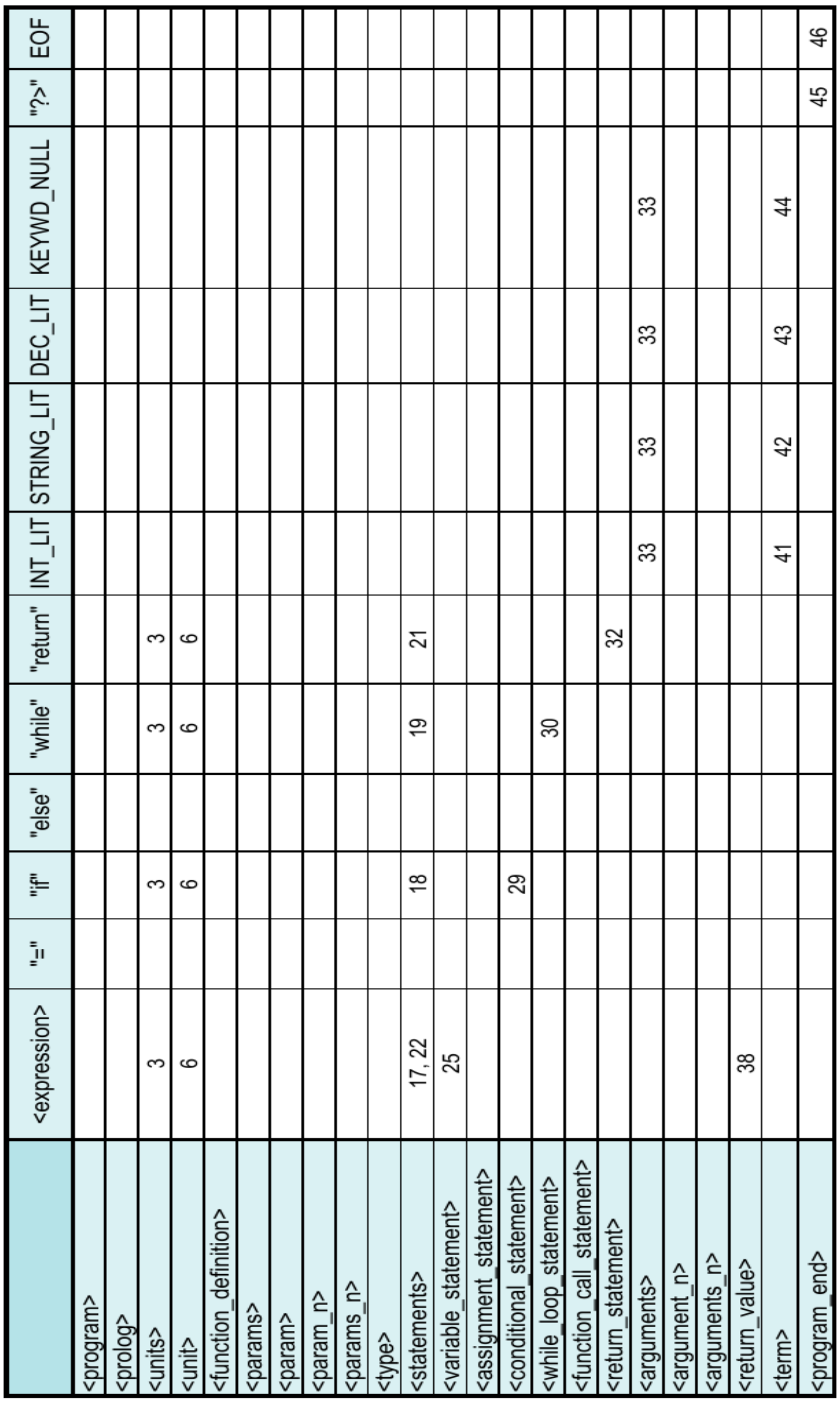
40. <**term**> $\rightarrow$ $ID
41. <**term**> $\rightarrow$ INT_LITERAL
42. <**term**> $\rightarrow$ STR_LITERAL
43. <**term**> $\rightarrow$ DEC_LITERAL
44. <**term**> $\rightarrow$ KEYWD_NULL

45. <**program_end**> $\rightarrow$ "?>" EOF
46. <**program_end**> $\rightarrow$ EOF

3.3 LL table



	<?php	declare(strict_ types=1)	";"	ε	"function"	ID	"("	")"	":"	"{"	"}"	$ID	","	INT	STRING	FLOAT
<program>	1															
<prolog>	2															
<units>				3, 4	3	3						3				
<unit>				6	5	6						6				
<function_definition>					7											
<params>				9										8	8	8
<param>														10	10	10
<param_n>													11			
<params_n>				13									12			
<type>														14	15	16
<statements>				23		20						17				
<variable_statement>												24, 26				
<assignment_statement>												27, 28				
<conditional_statement>																
<while_loop_statement>																
<function_call_statement>						31										
<return_statement>																
<arguments>				34								33				
<argument_n>													35			
<arguments_n>													36			
<return_value>			39													
<term>												40				
<program_end>																

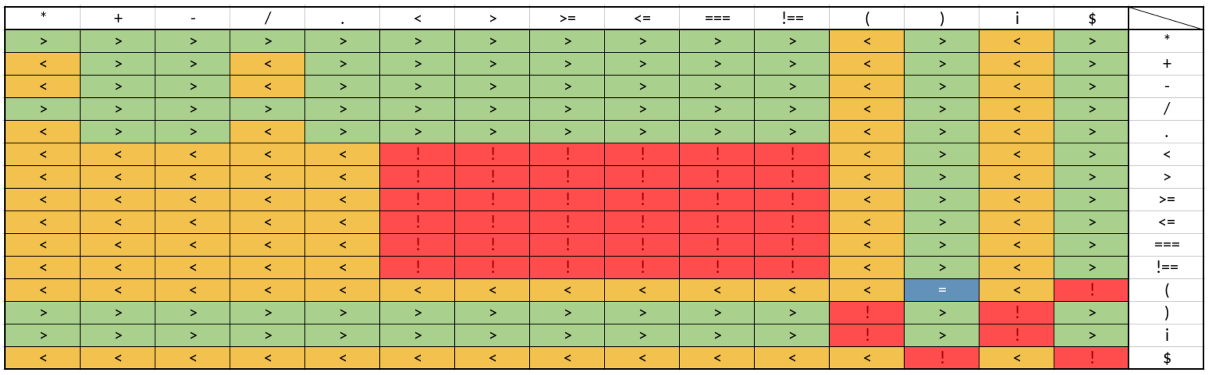


	<expression>	"="	"if"	"else"	"while"	"return"	INT_LIT	STRING_LIT	DEC_LIT	KEYWD_NULL	"?>"	EOF
<program>												
<prolog>												
<units>	3		3		3	3						
<unit>	6		6		6	6						
<function_definition>												
<params>												
<param>												
<param_n>												
<params_n>												
<type>												
<statements>	17, 22		18		19	21						
<variable_statement>	25											
<assignment_statement>												
<conditional_statement>			29									
<while_loop_statement>					30							
<function_call_statement>												
<return_statement>						32						
<arguments>							33	33	33	33		
<argument_n>												
<arguments_n>												
<return_value>	38											
<term>							41	42	43	44		
<program_end>											45	46

3.4 Expression processing

When the top down analysis derives an expression, it calls the main function, *exprParse()*, of the expression processor unit that is implemented in the file *expr.c(.h)*. At the beginning of the processing a stack of type *eStack_t* is initialized (see: subsection 7.5), then the type of the incoming token and the type of the closest terminal on the stack is changed to match the expression parser's own token type system. These types are defined in the file *expr.h*. Upon making these conversions, based on the received types, an operation from the precedence table (see: subsection 3.5) is chosen, and executed. Possible operations include shifting with or without an indent, reduction based on rules (which is explained in subsection 3.6), and a reserved error operation which terminates processing. While the processing of said terminals is being done, code regarding this part of the analysis is being generated and semantic checks are made. The incoming tokens are read from the *parser_t* struct and thus upon receiving a token that signals the end of given expression, either by causing an error or being correct, the analysis simply returns to top down parsing.

3.5 Precedence table



*	+	-	/	.	<	>	>=	<=	===	!==	(	)	i	$	
>	>	>	>	>	>	>	>	>	>	>	<	>	<	>	*
<	>	>	<	>	>	>	>	>	>	>	<	>	<	>	+
<	>	>	<	>	>	>	>	>	>	>	<	>	<	>	-
>	>	>	>	>	>	>	>	>	>	>	<	>	<	>	/
<	>	>	<	>	>	>	>	>	>	>	<	>	<	>	.
<	<	<	<	<	!	!	!	!	!	!	<	>	<	>	<
<	<	<	<	<	!	!	!	!	!	!	<	>	<	>	>
<	<	<	<	<	!	!	!	!	!	!	<	>	<	>	>=
<	<	<	<	<	!	!	!	!	!	!	<	>	<	>	<=
<	<	<	<	<	!	!	!	!	!	!	<	>	<	>	===
<	<	<	<	<	!	!	!	!	!	!	<	>	<	>	!==
<	<	<	<	<	<	<	<	<	<	<	<	=	<	!	(
>	>	>	>	>	>	>	>	>	>	>	!	>	!	>	)
>	>	>	>	>	>	>	>	>	>	>	!	>	!	>	i
<	<	<	<	<	<	<	<	<	<	<	<	!	<	!	$

3.6 Grammar rules for expressions

1. **E** → i
2. **E** → (E)
3. **E** → 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

4 Symbol table

The chosen implementation method of the symbol table is the variant TRP (hash table). This structure is implemented in the file *symtable.c (.h)*.

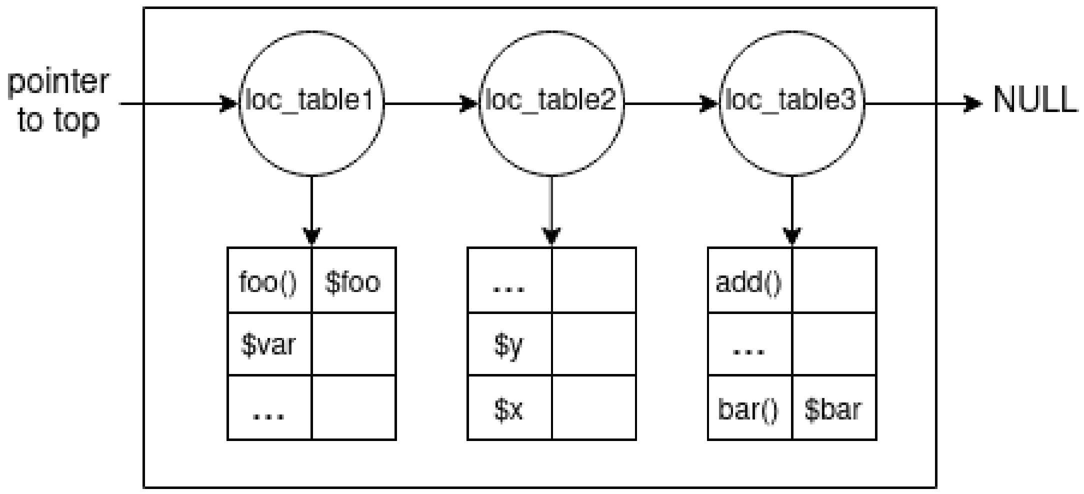
4.1 Implementation of symbol table

All elements of the symbol table are being held by the structure *htab_t*, which is detailed in subsection 7.3. The hashing function of the table was chosen to be implemented via an SDBM algorithm because of its good distribution of hash values. For the purpose of storing and processing functions and variables in the main body of the input code there is a main symbol table located in the *parser_t* structure. The structure of the data in an element of the hash table is as follows:

- type – used to identify if it's a function or variable ID
- param_IDs – if a function is held, this char pointer points to the array of parameters
- param_count – holds number of parameters

4.2 Stack of symbol tables

For checking variables, argument and return values in the local scope of a function, a stack implemented as a one way list of said symbol tables is also part of the *parser_t* data structure. This one way list is located in the file *sym_table_stack.c (.h)*, and is illustrated in the following figure.



5 Semantic analysis

5.1 Semantic checks in top down

Semantic checks are performed in two stages: compile-time and run-time. During compilation, we check for re-definitions of functions and variables, also when referencing a variable, we check whether that variable has been defined, this happens in *syntax_rules.c*, using the symbol table. At run-time, there are checks for the types and the amount of arguments being passed when calling a function, based on the function's parameters. These checks are included in the generated code via helper variables. Similarly, there are checks performed for function return types. Some of this is performed directly in *syntax_rules.c*, and the rest via functions defined in *generator.c*.

5.2 Semantic checks in bottom up

Semantic checks in bottom up are being done in the function *gen_expr_checkType()* at run-time. Firstly, we jump to the section that is related to the given operation. After that we will check what is the type of first variable, and what is the type of second variable. Following this we can determine if those variables can occur in combination with the current operation, or if not, if we need to convert the variables to another data type. Last semantic check is done at the end of expression, when we check if we return the correct data type.

6 Code generation

6.1 Code generation in top down

Much of the code is generated directly in the various rule functions, defined in the file *syntax_rules.c*. Specifically in statement rules, rules for function definitions and their parameters, and function calls and their arguments. For some of the code generation, functions defined in *generator.c* are used, such as: *genFunctionLabel()*, *genFunctionEnd()*, *genFunctionParamType()*, *genFunctionRetType()*, etc.

6.2 Code generation in bottom up

Code generating in bottom up part is dependent on the last read tokens in top down. If we are at an expression without any assignment, we don't generate any code, in other parts we do. Generation is done as follows, firstly we define our variables or constants with function *generateCode_defvar()*, then we specify operation with function *generateCode_operation*, after that semantic checks are performed in function *gen_expr_checkType()* and final result is computed in function *gen_expr_compute()*. The final stage contains checking if the last token is *')'* or *';'*, because we need to return boolean value in the case when the last token is *')'* and some other non-boolean value when last token is *';'*.

7 Data structures and types

7.1 String type - *string_t*

This data type and structure is used to dynamically allocate memory for read identifiers, string variables and constants. It is defined in the file *str.c (.h)*. This structure holds the allocated string, the number of characters it contains and the size of the total allocated memory.

7.2 Token type - *token_t*

The token data structure is used to hold tokens read by the scanner unit, hence its implementation is located in *scanner.c (.h)*. A token has two properties, a type, which is set by the scanner, and an attribute, which can either be an integer, a double, a keyword or the previously mentioned *string_t* data structure.

7.3 Hash table - symbol table

The main element of this structure is defined as *htab_t*, it functions similarly to the string structure as it holds the amount of buckets allocated, the current number of initialized items in the table, and a pointer to the list of said items. This similarity comes from the fact that the number of total buckets is dynamically allocated and deallocated when needed, just like in the structure *string_t*.

7.4 Double linked list - code generator

Double linked list is defined in the file *dll.c (.h)*. It is primarily used for storing arguments of called functions being pushed into stack. As the CDECL calling convetion is used, arguments of called functions are being inserted into DLL by function *DLLInsertLast()* from left to right. At the moment of code generation for interpreter, the data of elements are being printed to stdout in reversed order using *DLLPrintAllReversed()* as the last given argument needs to be pushed as first.

7.5 Expression stack - syntactic analysis

A stack to aid in expression processing has been implemented in the source file *expr_stack.c (.h)*. Its structure is also based on a linked list. The main structure, *eStack_t* holds the pointer to the first element in the list, and the current number of elements. Each given element contains three properties and a pointer to the next element. The said properties are a type (which different from the types in the *token_t* structure), a token of type *token_t* and an ID, which serves as an index for code generation.