BRNO UNIVERSITY OF TECHNOLOGY
FACULTY OF INFORMATION TECHNOLOGY

Formal languages and compilers

Imperative language IFJ17 compiler implentation

Team 006, variant II

December 7, 2017

Authors:

Leader	Name	Login	Points
	Daniš Tomáš	xdanis05	25%
•	Goldschmidt Patrik	xgolds00	25%
	Hamran Peter	xhamra00	25%
	Pšenák Kamil	xpsena02	25%

Extensions:

- BASE
- CYCLES
- FUNEXP
- GLOBAL
- IFTHEN
- SCOPE

Contents

1 Introduction

The task to implement IFJ17 compiler was given as a semestral project for courses IFJ and IAL. Group of old friends back from the high school has decided to accomplish it together and that is how the team 006 was created. After doing an initial problem decomposition, labor was divided as follows:

- **Tomáš Daniš** - Syntactic analysis, semantics checks, expressions evaluation, optimizations, tests definition
- **Patrik Goldschmidt** - Abstract data types and algorithms, documentation, lexical analyzer co-author, IFJ17 in-built functions, Makefile, script for regression tests
- **Peter Hamran** - Code generation, optimizations
- **Kamil Pšenák** - Lexical analysis, tests definition

This document further describes compiler development process and implementation details. Lexical analyzer FSA scheme as well as LL-grammar and LL-table are also included in the appendix section.

2 Components and interface

2.1 IFJ17 API file

For the special needs and mutually tied dependencies between various compiler components, we decided to create multi-purpose header file `ifj17_api.h`. This file contains common data structures like *token_t* or *symbol_t*, which are used among most of the source files. For the sake of the best clarity and code readability, API file is included by all major components which is usually the only needed dependency for data structures used in our implementation.

2.2 Lexical analyzer

Lexical analyzer (*scanner.c*) is implemented as a standard deterministic finite state automaton using *C* `switch` command. Lexical analysis is done on per-call basis, where syntactic analyzer repeats requests for a new token during the compilation process. To serve parser's call, scanner module reads a couple of characters until it reaches character that does not belong to the actual token. At this point, extra character read is saved to the internal scanner buffer. Buffer is then used during the next token request - its value serves as a beginning of new a token. Buffer is then set to the empty state and token forming continues by reading the input file.

Communication between scanner and parser modules is based on passing tokens represented by *token_t* structure. Token structure is composed of an identifier, token attribute and special expression attributes. Based on the read character sequence, scanner fills identification field appropriately and sets particular attribute if the token is a literal.

Lexical analyzer also provides error detection using states *error* and *str_error*. When an invalid character sequence is read, FSA switches to one of these error states. Reading states continues until white-space character or quotation mark in *str_error* state is read. Function then returns a lexical error. For advanced error recovery, scanner also supports line counting and error token value reporting. When the error occurs, user gets informed about the line at which the error occurred and also about a invalid token.

Scheme of lexical analyzer is shown in the figure 1 on page 8.

2.3 Syntactic analyzer

The core of the compiler is syntactic analyzer (*parser.c*). This module receives sequence of tokens from the scanner and validates if this sequence is a subset of the compiled language. If so, it also checks if the given construct is semantically correct and invokes generation of the output code. The module also analyses the semantics of the code to a certain extent to check for a possible optimizations.

The syntactic analysis was implemented with well-known recursive descent. This method was chosen because it was easier to implement, no extra stack was necessary and the resulting code structure is generally more readable. Expression analysis was implemented using PSA due to expression grammar. While it could be converted to an LL-grammar, we chose not to because of its simplicity and project requirements as well. LL-grammar and LL-table are included in the appendix section, pages 9 and 11 respectively.

However, our implementation deviates from the standards models in several ways:

- Placement of return statements is not checked using the LL-grammar but during the semantic analysis. This is done in order to simplify our LL-grammar.
- We needed to find the way how to successfully link the execution of LL-analysis to the PSA algorithm and vice versa. This was accomplished by each of the two modules checking if the received token does not belong to the other module. If so, the other module was invoked with this token already preloaded. In the assignment without extensions, this could have been potentially problematic. However, we decided to implement FUNEXP extension, which provides more general functionality and eliminates this discrepancy. More information about this issue is provided in the extension section on page 6.
- Way the *end* expression is found. Algorithm for PSA analysis presented in the IFJ course supposes that the end of the expression is the end of the whole input. However, in a real-life scenario, this is not the case. Programs in the IFJ17 language still continue after an expression has ended. Therefore, we needed a way to detect when a received token is no longer a part of the expression. This was accomplished by constructing the follow set the <*expr*> non-terminal. When one of the terminals from this set was received, we knew that it is no longer part of the expression and we artificially generated input of the PSA algorithm.
- After the end of the program, there can be any number of EOL characters which are not detected using the grammar but are discarded until an other character is reached. If this character is not EOF, a syntactic error has occurred.

Semantics analysis is executed concurrently with the syntactic analysis. It has been implemented using a semantic stack for data passing between functions, symbol table and global state variables. Global state variables provide information about the context of the program such as block depth, name of the current function (if any) or which block of the code is currently being analyzed. The semantic stack collects data from multiple related functions, which are then retrieved in a single place where they are validated and used for code generation. A set of rules that define input and output for each function can be found in the appendix section on page 13.

The symbol table is used for collecting information about functions and variables according to the standard implementation.

2.4 Code generator

Generator module creates output of the compilation process in the form of three address code IFJcode17. This code is then furtherly processed by interpreter *ic17int*, which was provided with the assignment.

Like scanner, code generation is invoked by calls from the parser during the compilation process. Communication between these 2 modules can be split into several stages. Firstly, parser sets up instruction operands and internal logic, which then affects the generation process. Data is passed via a *tstack* data structure, which was initially used for communication between the scanner and parser. Since generator needs almost all information contained in the token, we decided to re-use and slightly adjust this structure to facilitate parser and

generator communication. After the stack is set, parser calls generator routine with a specific ID that represents particular instruction to be generated. Every instruction follows specific stack management rules - number of stack elements to be pushed, popped, swapped or edited. Code is firstly generated into the instruction list `ilist_t`. This approach has several advantages, since the code can be optimized and is easily disposed if the compilation process fails. After optimization process is done, generator transforms instruction list contents into the instructions themselves. According to the assignment details, generated code gets printed to the standard output.

During the generation process, several issues need to be addressed. One of them is uniqueness of the generated variables and labels. Our solution solves this situation by using internal counters, whose value is preserved during the different invocations. Counter incrementation occurs after every expression evaluation, since temporary variable is created to store the result internally. Identifier of the generated variable is affected by parser, which specifies its name via value in the token structure. However, names for internal labels of loops, if statements and others are managed purely by the generator.

To achieve better generated program performance, compiler also provides few optimization techniques. Most of them are provided by parser, but generator handles some of them by itself. One of these optimizations is automatic in-built functions inlining. Using this method, in-built function bodies of *Asc()*, *Chr()*, *Length()* and *SubStr()* are straightly placed into the generated code. This effectively eliminates the cost of the jump instructions during the functions call.

2.5 Algorithms and data structures

2.5.1 Symbol table

According to the project requirements for variant II, symbol table is implemented using hash table. To achieve least hash collisions, acceptable performance and simplicity of implementation, Fowler-Noll-Vo (FNV) hash algorithm was chosen. Symbol table in this project uses FNV-1a variant with 32-bit flavor. This choice was inspired mainly by [1]. Specific implementation details and used magic numbers are based on one of the function author's blog [2].

Symbol table itself is implemented as a local array of pointers to *stab_item_t* structure. For the optimal performance/memory ratio, we decided to set a table size to 300 elements. Structure consist of the (*symbol_t*) data and pointer to the next item. Hash table items are thus mutually interconnected as a regular linked list. Key on which the hashing is based is the identifier contained in the symbol itself. Searching, inserting, deleting and other hash table operations are then based on the symbol ID contained in the hash table item.

Specialty of this hash table is that user is able to iterate though it using a *stab_iterator_t* structure and in-built functions *stab_iterate_set* and *stab_iterate_next*. This functionality comes handy during semantics checks and output code optimization.

2.5.2 PSA stack

Stack for precedence syntactic analysis (*tokenstack.c*) is composed of tokens with basic push, pop and top functionality. However, PSA algorithm requires specific stack functionality, so this stack was was needed to be adjusted accordingly. Returning topmost terminal is served via *tstack_get_topterm()* function. Implementation also provides functionality to push desired token after the topmost terminal. To facilitate specific PSA requirements, stack also contains a sentinel value, which represents $ in the PSA algorithm. This sentinel value is automatically added to the stack upon its initialization and can not be popped. Stack is then considered empty when it contains only a sentinel value.

Stack is also implemented fully dynamically. Dynamic size is achieved by internal implementation, which is basically a doubly-linked list. For reducing number of malloc calls, stack is pre-allocated upon initialization and when its maximum size reached, an expansion occurs. Using this approach, its size is theoretically limited only by the host's memory limitations.

2.5.3 Instruction list

Instruction list module *ilist.c* provides an interface for storing instructions prior their optimization and generation. List is internally represented as a doubly-linked list, which also stores number of instructions currently in the list. This value is used for output code optimization and can be determined using *ilist_size()* function. Instruction list module further provides other regular doubly-linked list functions discussed during IAL course.

2.5.4 Dynamic string

Dynamically-sized string module *str.c* provides C++ string pseudo-equivalent data structure. Module defines *string_t* data structure, which is internally represented as a current length of the string, allocated size and C string type *char* *. Dynamic string is pre-allocated to 128B and eventually reallocated to another +128B when its maximum size is reached. Manipulating the string is done via module functions, which support adding another char to the string, comparing between strings, conversion to the C-type strings and vice versa. This functionality is essential requirement for the scanner to build-up a token and pass it to the parser.

3 Development

3.1 Development cycle and project management

Considering a fact that project requirements were known before the start of the development process and their change was not expected, choosing a waterfall development method would be more than appropriate. However, based on the unnecessary administrative overhead and lack of experience of project's programmers, we decided to stick with agile development techniques.

Development itself was preceded by a study phase and problem decomposition. Labor was divided based on the experience, skills and personal preferences. Since the development and study stages were partially overlapping, agile development has proven to be very effective. Inner requirements between different modules were often changed, so we needed to adapt to those changes and even occasionally rework modules to support desired functionality.

To track our progress and discuss relevant issues, regular team meetings were held on weekly basis. For better cooperation, control version system *git* was used. Many agile development techniques like pair programming, continuous testing and regular code reviewing were also applied.

3.2 Testing and validation

As expected, many errors during the development process arose. Some errors were revealed using unit tests, but most of them appeared when we connected the modules together. These were linked mainly to incompatible interface between modules due to different interpretation of the issues by authors of modules. Our custom tests helped us to find most of these bugs and fix them.

Discussing this topic, we would also like to thank Martin Kobelka's team for providing numerous number of tests and other tools used for IFJ project development. These definitely helped many teams including ours.

4 Conclusion

We definitely have to admit that this project was rather challenging, though it still taught us a lot not only about compilers and their design, but also about bigger team projects and how important internal team management and communication is. Some time still could be used for debugging and optimization purposes, but we definitely think that our final product is in the good shape and it should be able to correctly process source code IFJ17, detect various errors and create output 3-address code ready for further interpretation.

References

[1] BOLD Ian. Hash algorithms efficiency comparison. [online], April 2012. Available at: https://softwareengineering.stackexchange.com/questions/49550/which-hashing-algorithm-is-best-for-uniqueness-and-speed. Seen 3.12.2017.

[2] NOLL Landon Curt. FNV Algorithm Overview. [online] June 2009, updated: March 2014. Available at: http://www.isthe.com/chongo/tech/comp/fnv/. Seen 3.12.2017.

5 Appendix

5.1 Extensions

5.1.1 BASE

For the BASE extension purpose, support for extra tokens `&b`, `&h` and `&o` needed to be added. After receiving this character sequence and at least one valid number belonging to the particular base, value gets converted to decimal using `strtol()` *C* function. If no valid number in specified base is received, value of 0 is implicitly returned. This process is done purely in the scanner module. Parser then receives decimal value in the token and is unaware that any conversion has happened.

5.1.2 CYCLES

CYCLE extension added support for `for` and `do...loop` cycle types. This also included support for loop flow control statements `exit` and `continue`. Implementation of this required us to track what cycles are we in at any point of the code analysis. This was accomplished using a semantic stack and a global variable counting cycle depth. `For` type cycles also required implementation of scope block variable validity tracking system. This eventually led us to the SCOPE extension implementation as well.

5.1.3 FUNEXP

FUNEXP extension required that functions are made as a part of expressions, which are normally analyzed with the PSA algorithm. However, argument list in the function call is not suitable to be analyzed by the PSA in a readable manner. Therefore, tighter cooperation between PSA and LL analysis was needed to be implemented. This is because argument list of functions is once again analyzed with LL analysis while the arguments themselves are expressions and are therefore analyzed with the PSA. In this manner, the two modules can call each other infinitely and we needed to ensure that correct data are passed between them.

The biggest problem was detecting whether during argument analysis, received right bracket is part of the expression or signifies the end of the function call. This was implemented by expanding the capabilities of our PSA table to be able to distinguish these two cases. In short, if an unmatched right bracket was found, the table reported the end of the function call. If this unmatched bracket was indeed an error, it was detected a few steps later during the LL-analysis.

5.1.4 GLOBAL

Extension GLOBAL required implementation of FREEBasic `global` and `static` variable types. To provide this functionality, variable shadowing was needed to be implemented. For this purpose, we used global symbol table shared for both variables and functions. Global symbol table combined with the global frame provided by IFJcode17 was sufficient for all the needs and this extension could be implemented.

5.1.5 IFTHEN

IFTHEN extension added support for standalone `If-then` and `elseif` statements. In this case, situation when the generator is unaware of number of labels and conditional jumps that have to be generated. This happens because we are unable to tell how many *elseif* statements will follow. Problem was addressed by redirecting the code generation to the auxiliary instruction list.

IFTHEN rozšírenie pridáva podporu osamelých If-then statementov a elseif statementov. V tomto prípade nastáva situácia kedy vopred nevieme ako je potrebné vygenerovať návestia a podmienené skoky pre IFJcode17 z dôvodu potencionálne viacnásobného výskytu elseif statementov. Tento problém sa odstránil pozastavením generácie kódu do inštrukčného listu a následným presmerovaním generácie tela podmieneného výrazu. Pri

narazení na ďalšiu časť podmieneného výrazu dokončíme generáciu predchádzajúce a vygenerované inštukcie sa pridajú do hlavného inštrukčného listu.

listy, prelinkovanie v generatore z dovodov, ze nevieme, kolko else-if sa tu bude nachadzat podmienena generacia lablov, rebranchovanie na tela

5.1.6 SCOPE

SCOPE extension required recursive block placement as well as correct visibility for shadowed variables in parent blocks. For this purpose, we implemented a variable depth stack (`vdstack.c`), which holds information about a single identifier and all its IFJcode17 possible names in various scope blocks. Each entry in this stack also contains information about scope block depth in which this name is used. At the end of the block, we iterated through the local symbol table and popped all the entries for the names that just went out of the scope. This proved to be advantageous because all visibility handling was done during code analysis and no extra checks were necessary in the generated code.

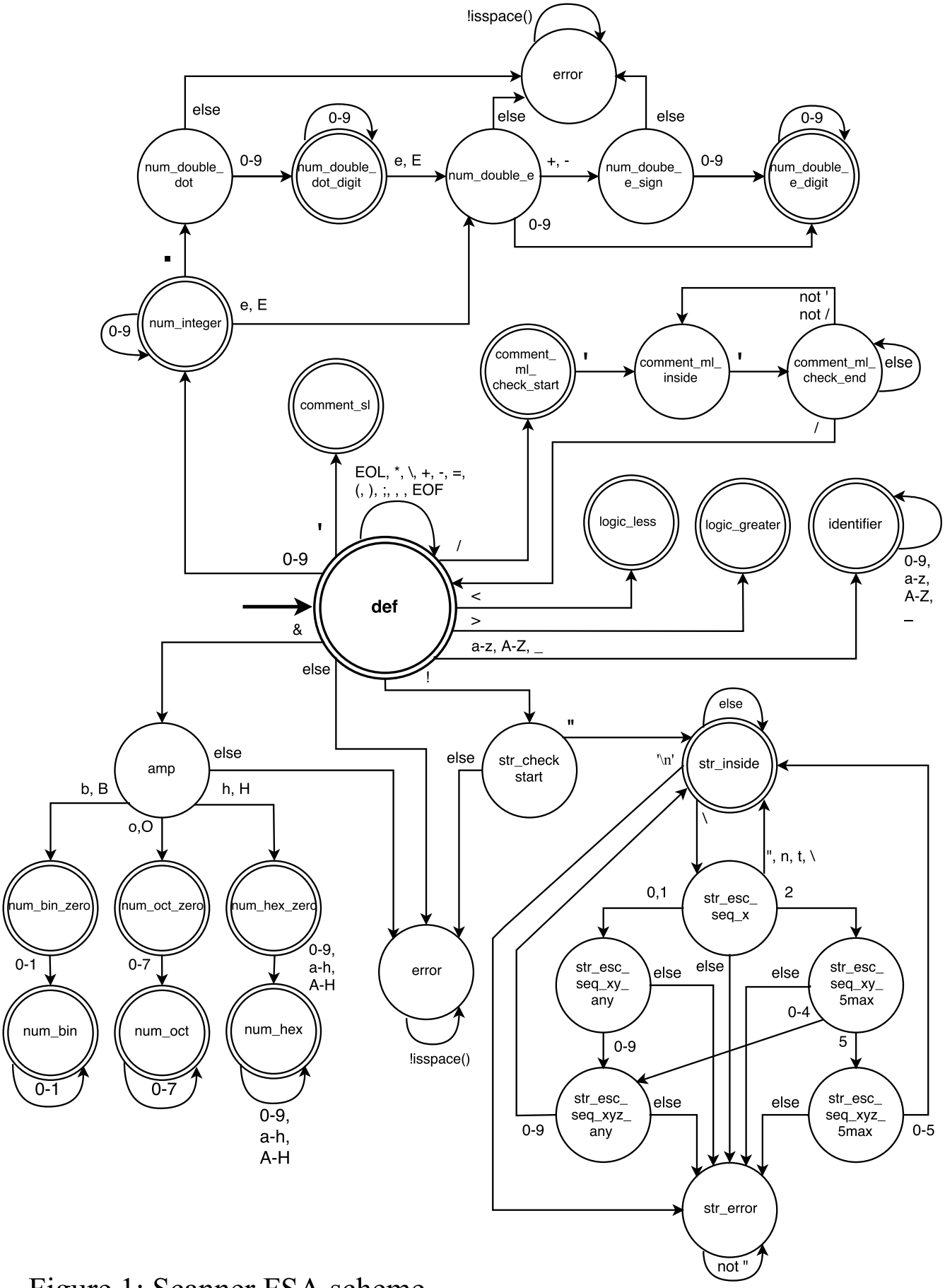

Figure 1: Scanner FSA scheme

Figure 2: LL-grammar

<*prog*>	→	<*decl_list*> Scope EOL <*stat_list*> End Scope EOF
<*decl_list*>	→	<*var_decl*> <*decl_list*>
<*decl_list*>	→	<*func_decl*> <*decl_list*>
<*decl_list*>	→	<*func_def*> <*decl_list*>
<*decl_list*>	→	<*empty_statement*>
<*decl_list*>	→	ε
<*func_decl*>	→	Declare Function id (<*param_list*>) As <*type*> EOL
<*func_def*>	→	Function id (<*param_list*>) As <*type*> EOL <*stat_list*> End Function EOL
<*var_decl*>	→	Dim <*opt_scope_modifier*> id As <*type*> <*opt_initialiser*> EOL
<*var_decl*>	→	Static id as <*type*> <*opt_initialiser*> EOL
<*param_list*>	→	<*param*> <*param_list_cont*>
<*param_list*>	→	ε
<*param*>	→	id As <*type*>
<*param_list_cont*>	→	, <*param*> <*param_list_cont*>
<*param_list_cont*>	→	ε
<*opt_scope_modifier*>	→	Shared
<*opt_scope_modifier*>	→	ε
<*opt_initialiser*>	→	= <*expr*>
<*opt_initialiser*>	→	ε
<*type*>	→	Integer
<*type*>	→	Double
<*type*>	→	String
<*stat_list*>	→	<*statement*> <*stat_list*>
<*stat_list*>	→	ε
<*statement*>	→	<*assignment*>
<*statement*>	→	<*read_statement*>
<*statement*>	→	<*print_statement*>
<*statement*>	→	<*scope_statement*>
<*statement*>	→	<*selection_statement*>
<*statement*>	→	<*iteration_statement*>
<*statement*>	→	<*return_statement*>
<*statement*>	→	<*empty_statement*>
<*statement*>	→	<*var_decl*>
<*statement*>	→	<*iteration_control_statement*>
<*assignment*>	→	id = <*expr*> EOL
<*read_statement*>	→	Input id EOL
<*print_statement*>	→	print <*expr*> ; <*expr_list*> EOL
<*expr_list*>	→	<*expr*> ; <*expr_list*>
<*expr_list*>	→	ε
<*scope_statement*>	→	Scope EOL <*stat_list*> End Scope EOL
<*selection_statement*>	→	If <*expr*> then EOL <*stat_list*> <*alternative_statement*> End If EOL
<*alternative_statement*>	→	elseif <*expr*> then EOL <*stat_list*> <*alternative_statement*>

<*alternative_statement*> → else EOL <*stat_list*>
<*alternative_statement*> → ε
<*iteration_statement*> → for id <*opt_type_decl*> = <*expr*> to <*expr*> <*opt_step*> EOL <*stat_list*> Next <*opt_id*> EOL
<*opt_type_decl*> → as <*type*>
<*opt_type_decl*> → ε
<*opt_step*> → step <*expr*>
<*opt_step*> → ε
<*opt_id*> → id
<*opt_id*> → ε
<*iteration_statement*> → do <*do_cycle*> EOL
<*do_cycle*> → EOL <*stat_list*> Loop <*opt_cond*>
<*do_cycle*> → <*cond*> EOL <*stat_list*> Loop
<*opt_cond*> → <*cond*>
<*opt_cond*> → ε
<*cond*> → <*do_mode*> <*expr*>
<*do_mode*> → until
<*do_mode*> → while
<*return_statement*> → return <*expr*> EOL
<*iteration_control_statement*> → <*control_statement*> <*cycle_type*> <*cycle_type_list*> EOL
<*control_statement*> → Exit
<*control_statement*> → Continue
<*cycle_type_list*> → , <*cycle_type*> <*cycle_type_list*>
<*cycle_type_list*> → ε
<*cycle_type*> → for
<*cycle_type*> → do
<*empty_statement*> → EOL

Figure 3: LL-table

	Declare	Function	Dim	Static	id	Shared	=	Integer	Double	String	Input	print	Scope	If	elseif	else	for	as	step	do	EOL	until
<prog>	1	1	1	1									1								1	
<decl_list>	3	4	2	2									6								5	
<func_decl>	7																					
<func_def>		8																				
<var_decl>			9	10																		
<param_list>					11																	
<param>					13																	
<param_list_cont>																						
<opt_scope_modifier>					17	16																
<opt_initialiser>							18														19	
<type>								20	21	22												
<stat_list>			23	23	23						23	23	23	23	24	24	23			23	23	
<statement>			33	33	25						26	27	28	29			30			30	32	
<assignment>					35																	
<read_statement>											36											
<print_statement>												37										
<expr_list>					38																39	
<scope_statement>													40									
<selection_statement>														41								
<alternative_statement>															42	43						
<iteration_statement>																	45			52		
<opt_type_decl>							47											46				
<opt_step>																			48		49	
<opt_id>					50																51	
<do_cycle>																					53	54
<opt_cond>																					56	55
<cond>																						57
<do_mode>																						58
<return_statement>																						
<iteration_control_statement>																						
<control_statement>																						
<cycle_type_list>																					65	
<cycle_type>																	66			67		
<empty_statement>																					68	
<func_call>																						
<arg_list>					70																	
<more_args_list>																						
<arg>					74																	
<built_in_func_call>																						

	while	return	Exit	Continue	;	,	(	then	Loop	End	)	to	EOF	Next	Length	Chr	Asc	Substr	Integer_literal	Double_literal	String_literal
<prog>																					
<decl_list>																					
<func_decl>																					
<func_def>																					
<var_decl>																					
<param_list>											12										
<param>																					
<param_list_cont>						14					15										
<opt_scope_modifier>																					
<opt_initialiser>																					
<type>																					
<stat_list>		23	23	23					24	24				24							
<statement>		31	34	34																	
<assignment>																					
<read_statement>																					
<print_statement>																					
<expr_list>							38								38	38	38	38	38	38	38
<scope_statement>																					
<selection_statement>																					
<alternative_statement>										44											
<iteration_statement>																					
<opt_type_decl>																					
<opt_step>																					
<opt_id>																					
<do_cycle>	54																				
<opt_cond>	55																				
<cond>	57																				
<do_mode>	59																				
<return_statement>		60																			
<iteration_control_statement>			61	61																	
<control_statement>			62	63																	
<cycle_type_list>						64															
<cycle_type>																					
<empty_statement>																					
<func_call>							69														
<arg_list>							70				71				70	70	70	70	70	70	70
<more_args_list>						72					73										
<arg>							74								74	74	74	74	74	74	74
<built_in_func_call>															75	78	77	76			

Figure 4: Precedence-table

Top/Input	+	-	*	/	\	=	< >	<	<=	>	>=	id	int_lit	double_lit	string_lit	(	)	end
+	>	>	<	<	<	>	>	>	>	>	>	<	<	<	<	<	>	>
-	>	>	<	<	<	>	>	>	>	>	>	<	<	<	<	<	>	>
*	>	>	>	>	>	>	>	>	>	>	>	<	<	<	<	<	>	>
/	>	>	>	>	>	>	>	>	>	>	>	<	<	<	<	<	>	>
\	>	>	<	<	>	>	>	>	>	>	>	<	<	<	<	<	>	>
=	<	<	<	<	<							<	<	<	<	<	>	>
< >	<	<	<	<	<							<	<	<	<	<	>	>
<	<	<	<	<	<							<	<	<	<	<	>	>
<=	<	<	<	<	<							<	<	<	<	<	>	>
>	<	<	<	<	<							<	<	<	<	<	>	>
>=	<	<	<	<	<							<	<	<	<	<	>	>
id	>	>	>	>	>	>	>	>	>	>	>						>	>
int_lit	>	>	>	>	>	>	>	>	>	>	>						>	>
double_lit	>	>	>	>	>	>	>	>	>	>	>						>	>
string_lit	>	>	>	>	>	>	>	>	>	>	>						>	>
(	<	<	<	<	<	<	<	<	<	<	<	<	<	<	<	<	=	
)	>	>	>	>	>	>	>	>	>	>	>						>	>
end	<	<	<	<	<	<	<	<	<	<	<	<	<	<	<	<	end	

Figure 4: Semantic stack rules

1. ***<prog>***
   Input: -
   Output: -

2. ***<decl_list>***
   Input: -
   Output: -

3. ***<var_decl>***
   Input: -
   Output: -

4. ***<opt_initialiser>***
   Input: -
   Output: Expression with type (NO_INIT in type if no initialiser)

5. ***<type>***
   Input: -
   Output: Read type (Integer, Double or String)

6. ***<opt_scope_modifier>***
   Input: -
   Output: Read scope modifier or NO_MODIFIER

7. ***<expr>***
   Input: -
   Output: Final expression resulting from the PSA algorithm

8. ***<func_decl>***
   Input: -
   Output: -

9. ***<func_def>***
   Input: -
   Output: -

10. ***<param_list>***
    Input: -
    Output: Number of parameters, followed by a reversed list of parameters

11. ***<param>***
    Input: -
    Output: Type and name of the read parameter

12. ***<param_list_cont>***
    Input: Number of already read parameters
    Output: Number of parameters, followed by a reversed list of parameters

13. ***<cycle_stat_list>***
    Input: List of cycle types with the caller on top (All only in non-recursive calls)
    Output: Input

14. ***<stat_list>***
    Input: List of cycle types with the caller on top (All only in non-recursive calls)
    Output: Input

15. ***<statement>***
    Input: List of cycle types
    Output: Input

16. ***<iteration_control_statement>***
    Input: List of cycle types
    Output: Input

17. ***<assigment>***
    Input: -
    Output: -

18. ***<read_statement>***
    Input: -
    Output: -

19. ***<print_statement>***
    Input: -
    Output: -

20. ***<expr_list>***
    Input: Number of read expressions so far
    Output: Number of read expressions, followed by a reversed list of these expressions

21. ***<selection_statement>***
    Input: List of cycle types
    Output: Input

22. ***<iteration_statement>***
    Input: List of cycle types
    Output: Input

23. ***<empty_statement>***
    Input: -
    Output: -

24. ***<return_statement>***
    Input: -
    Output: -

25. ***<alternative_statement>***
    Input: List of cycle types (Only in non-recursive calls)
    Output: Input

26. ***<opt_type_decl>***
    Input: -
    Output: Read type or NO_DECL

27. ***<opt_step>***
    Input: -
    Output: Step expression

28. ***<opt_id>***
    Input: -
    Output: Read identifier or NO_ID

29. ***<do_cycle>***
    Input: List of cycle types
    Output: Input

30. ***<opt_cond>***
    Input: -
    Output: Condition expression and do mode, or NO_COND

31. ***<cond>***
    Input: -
    Output: Condition expression, do mode

32. ***<do_mode>***
    Input: -
    Output: Do mode (Until or While)

33. ***<control_statement>***
    Input: -
    Output: Control statement (Continue or Exit)

34. ***<cycle_type_list>***
    Input: Initial cycle
    Output: Cycle type with number of occurences

35. ***<cycle_type>***
    Input: -
    Output: Cycle type (For or Do)

36. ***<built_in_func_call>***
    Input: -
    Output: Result expression containing the return value of the function call

37. ***<func_call>***
    Input: -
    Output: Number of parameters, followed by a reversed list of expressions representing them

38. ***<arg_list>***
    Input: -
    Output: Number of parameters, followed by a reversed list of expressions representing them

39. ***<more_args_list>***
    Input: Number of already read parameters followed by a reversed list of expressions representing them
    Output: Number of parameters, followed by a reversed list of expressions representing them

40. ***<arg>***
    Input: -
    Output: Read argument expression