BRNO UNIVERSITY OF TECHNOLOGY
FACULTY OF INFORMATION TECHNOLOGY

Implementation of compiler for imperative language IFJ20

Extensions: BOOTHEN, BASE, FUNEXP, MULTIVAL, UNARY

Team 016, option II

Gulčíková Sabína	**(xgulci00)**	**25 %**
Burda Luděk	(xburda13)	25 %
Fridrich David	(xfridr08)	25 %
Zaťovič Martin	(xzatov00)	25 %

Contents

1 Introduction

The aim of this project was to create a program in C language, which would process source code in IFJ20 language. This code would further be translated into IFJcode20, with proper return codes indicating the correctness of both source code and valid run of the compiler.
Several modules focusing on different responsibilities of compiler were implemented in order to achieve proper translation from source program to functionally equivalent target program. Detailed description of individual modules and their cooperation can be found in the following sections of this documentation.

2 Implementation

Because of the complexity of this project, creation of several coordinated components was necessary. According to the general structure of compiler, appropriate modules were implemented in separate, intercommunicating modules. Detailed description of each unit is provided in the following sections.

2.1 Lexical analysis - Scanner

The very first implemented module in this project was the lexical analyzer. The implementation of lexical analyzer was based on the previously designed finite state automaton 1. Each accepting state of automaton represents valid token. List of valid tokens can be found in legend of states of finite state automaton 2, as well as in the enumerator `tokenType` located in the `scanner.h` file.
Analysis starts once the main function `gettToken` is called for the first time, and operates until token `EOF` is processed. Whole process is operated by `switch`, in which each `case` represents corresponding state in the finite automaton. In this way, proper simulation of the automaton is achieved.
Recognized tokens are stored in structure `Token` which can be found in `scanner.h` file. These contain both type and value of the token stored as a string.
Scanner operates on a ”greedy” basis, meaning that it keeps reading character from a stream of file until valid token is discovered. In case it encounters invalid sequence of characters, `LEXICAL_ERROR` is returned with value 1, proper error message is displayed, and the analysis is aborted.

2.2 Syntax analysis - Parser

Our approach of syntax analysis composes of recursive descent through nonterminals of the LL grammar. Each nonterminal has its own function in code, except the nonterminal describing sequence of identifiers and the sequence of expressions. In the beginning, our concept was completely made of recursive descent, but after implementing the semantic analysis we decided to change our approach of the sequences. The reasons are described in the semantic analysis section.

2.2.1 Precedence analysis

The precedence analysis was implemented on the basis of pushdown automaton. Its inputs are obtained by calling of the `getToken` function and processed using the bottom-up method. Hence the name, the automaton uses the stack and a precedent table to determine the order of used rules and to validate correctness of syntax. If a rule is found, code generation functions are called. Set of all the rules along with the precedent table can be found in the appendix.

2.3 Semantic analysis

`T`he semantic analysis is composed mostly of the interaction with the table of symbols. Every time an identifier is encountered in an expression, we check its existence and the data type its supposed to be. These data are inserted into symtable in the process of variable definitions or parsing the arguments of defined functions. We decided to do a workaround of the recursive descent approach at checking the identifier and expression sequences. The reason is because our implementation of semantic checking is based on storing the identifiers and return values of corresponding expressions in two separate arrays of values. These arrays are then used for either various semantic checks or inserting data about variables into the table of symbols.

2.4 Code generation

Code generation is the final process of the entire compilation. Its implementation can be found within file `generator.c`. This unit is responsible for generating code to IFJcode20. It takes in 3AC (in addition to some special self-made instructions) which is then generated respectfully. Custom instructions are sequences of basic instructions serving purpose like definition of a function or definition of if-statements. Final code is generated to `stdout`. Comments are part of the output for better understanding of inbuild functions.

3 Implemented data structures

In the following sections, we will focus on special data structures that were implemented in this project. Their realisation and our capability of their usage was based on the knowledge gained from Algorithms lectures.

3.1 Symtable

Data structure used for holding values of variables and functions. Specific implementation was having global variable to hold pointer to the whole data structure, which can be further divided into smaller segments. Global variable `symGlobal`is defined as pointer to the whole symtable. There are 2 main data structures of type struct which are `symtableGlobalItem` and `symtableItem`. On the highest level, there is a symtable of `symtableGlobalItem` items (functions) and each one has an array of `symtableItem` type → one item of this array represents scope in each given function. Each scope then has its own symtable which is to be filled with variables. There is number of supportive functions to manipulate with the data such as: `symtableItemGetGlobal`, `pushArg` or `addScope` to add/delete scopes, add items and functions to data structures or check if item is in symtable.

3.2 Stack

The implementation of abstract data type - stack was essential for implementation of extended pushdown automaton. The stack is used to store all the symbols, until operation '>' is determined by the precedent table. In this case the symbols on the top of the stack are scanned until a separator is found and then they are compared to the left sides of the rules. If they match a rule is found. Operation '<' pushes the currently analyzed symbol along with a separator onto the stack. Operation '=' only occurs when the symbol ')' is found and a symbol '(' was previously pushed onto the stack. It results in a simple push of ')' onto the stack.

3.3 Custom string

The module Custom string allows us to work with a dynamic string whose length is easily adjusted whenever needed. Possibility of doing so is necessary for lexical analysis, which reads tokens of initially unknown length. Thanks to function `appendChar` found in `dynamic_string.c` newly read character can be appended to already existing sequence of characters. These can be further converted into a string literal of fixed length.
It is also possible to erase the content by the `eraseDynamicString` function. When discarding completely, it is necessary to free up the allocated memory by `freeDynamicString` as well.
This module was created on the basis of `str.c` and `str.h` files, which can be found in the public IFJ course pages with private materials.

4 Teamwork and cooperation

This project was created in cooperation of four students. Each person has had a dedicated module that he or she was working on.
The responsibility for individual tasks was evenly divided between all team members. Because of the complexity of this project it was necessary to establish boundaries and deadlines which would be followed during the realization process. By achieving the objectives that were set at the beginning of our collaboration, seamless cooperation was ensured.
Particular division of tasks was as follows.

4.1 Division of tasks

- Burda Luděk - Parser, Semantic Analysis, Documentation
- Fridrich David - Symtable, Code Generation, Documentation
- Gulčíková Sabína - Finite State Automaton, Lexical Analysis, Tests, Documentation
- Zaťovič Martin - Precedence Analysis, Documentation

4.2 Communication and version control

The current situation with global pandemic made any meetings of personal kind impossible. Therefore it was necessary to communicate over digital platforms, in order to keep steady workflow. Specifically, all of our communication was carried out in personalized Discord server, where we shared all resources and notes, planned sessions and discussed all problems that have appeared during the process of implementation.
Updated versions of files were kept in private repository on GitHub, which allowed for effective communication and version control.

5 Conclusion

To conclude, working on this project was an enriching experience. Getting the grasp of the theory behind compilers gave us perfect ability to comprehend how modern compilers work. It will be easier for us to understand why the code that we have written could not be compiled in the future, and what is the actual difference in semantic, syntactic and lexical errors. The complexity of this project taught us how to ensure proper time management, how to cooperate in small team, and how to overcome any possible obstacles that have appeared in the process of implementation. In addition to the aforementioned, we were given the opportunity to apply our theoretical knowledge of abstract data types in use, playing an important role in elaborate large-scale project.

A Finite state automaton for the base of lexical analysis

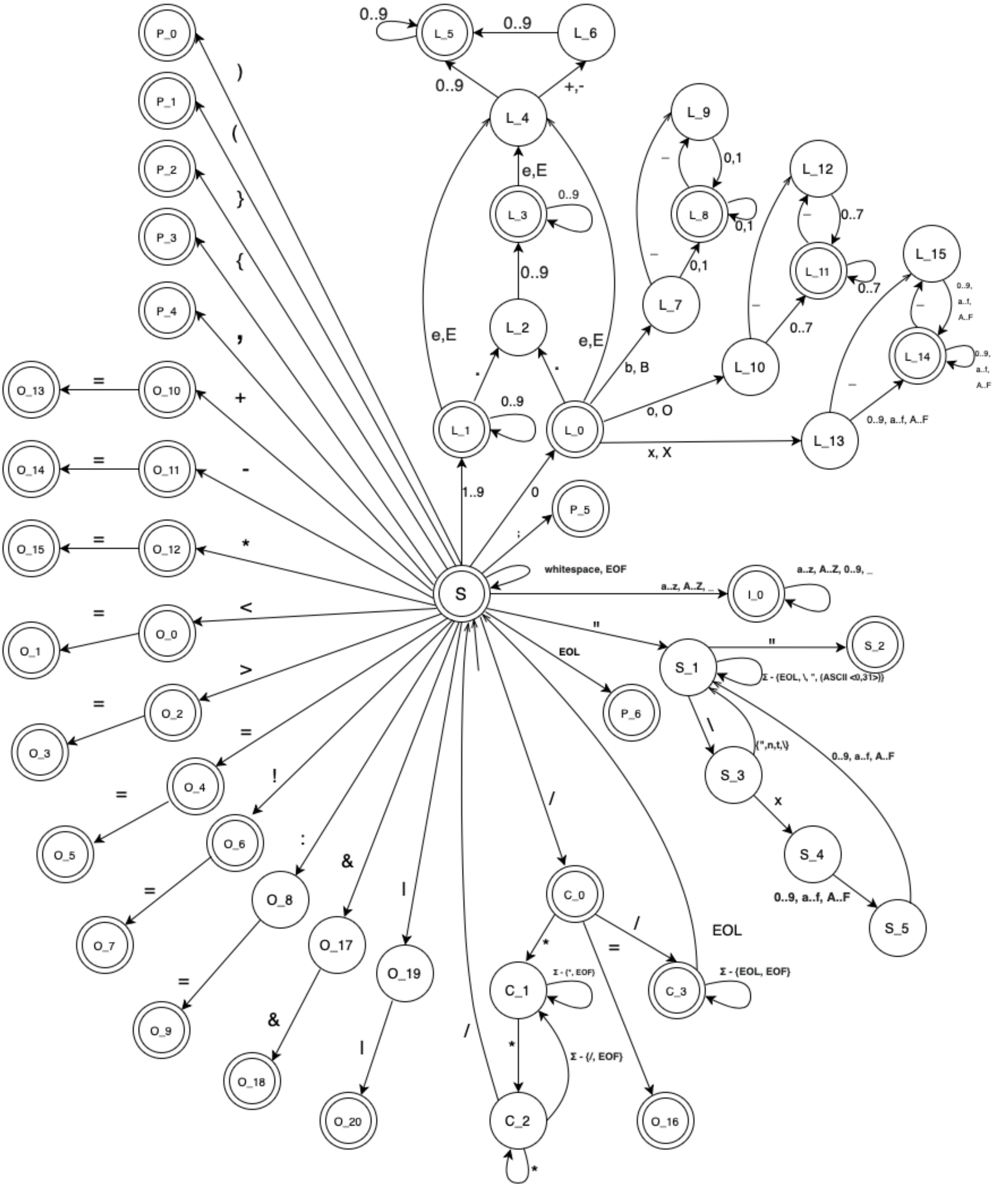
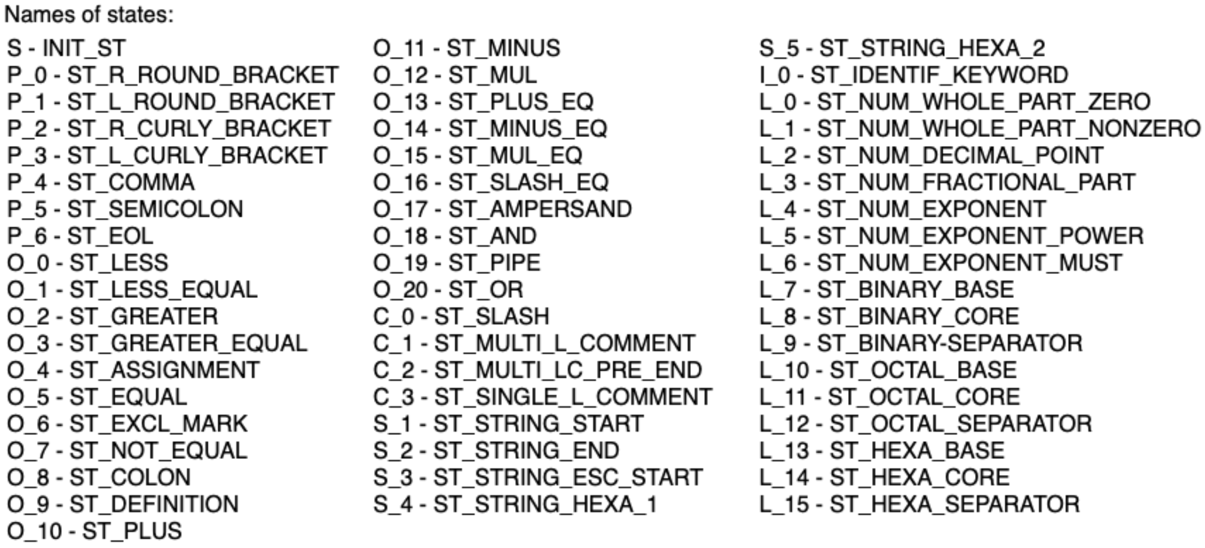


Figure 1: Finite state automaton for lexical analyzer

B Legend for finite state automaton

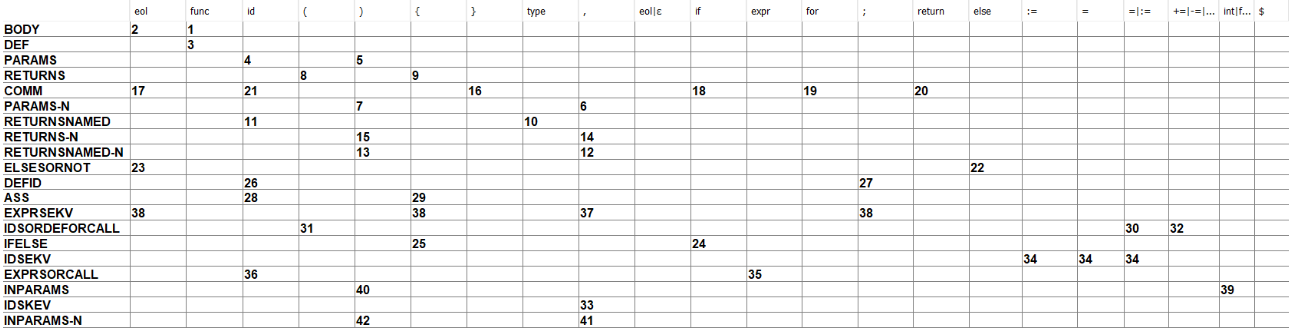


Names of states:

S - INIT_ST
P_0 - ST_R_ROUND_BRACKET
P_1 - ST_L_ROUND_BRACKET
P_2 - ST_R_CURLY_BRACKET
P_3 - ST_L_CURLY_BRACKET
P_4 - ST_COMMA
P_5 - ST_SEMICOLON
P_6 - ST_EOL
O_0 - ST_LESS
O_1 - ST_LESS_EQUAL
O_2 - ST_GREATER
O_3 - ST_GREATER_EQUAL
O_4 - ST_ASSIGNMENT
O_5 - ST_EQUAL
O_6 - ST_EXCL_MARK
O_7 - ST_NOT_EQUAL
O_8 - ST_COLON
O_9 - ST_DEFINITION
O_10 - ST_PLUS
O_11 - ST_MINUS
O_12 - ST_MUL
O_13 - ST_PLUS_EQ
O_14 - ST_MINUS_EQ
O_15 - ST_MUL_EQ
O_16 - ST_SLASH_EQ
O_17 - ST_AMPERSAND
O_18 - ST_AND
O_19 - ST_PIPE
O_20 - ST_OR
C_0 - ST_SLASH
C_1 - ST_MULTI_L_COMMENT
C_2 - ST_MULTI_LC_PRE_END
C_3 - ST_SINGLE_L_COMMENT
S_1 - ST_STRING_START
S_2 - ST_STRING_END
S_3 - ST_STRING_ESC_START
S_4 - ST_STRING_HEXA_1
S_5 - ST_STRING_HEXA_2
I_0 - ST_IDENTIF_KEYWORD
L_0 - ST_NUM_WHOLE_PART_ZERO
L_1 - ST_NUM_WHOLE_PART_NONZERO
L_2 - ST_NUM_DECIMAL_POINT
L_3 - ST_NUM_FRACTIONAL_PART
L_4 - ST_NUM_EXPONENT
L_5 - ST_NUM_EXPONENT_POWER
L_6 - ST_NUM_EXPONENT_MUST
L_7 - ST_BINARY_BASE
L_8 - ST_BINARY_CORE
L_9 - ST_BINARY-SEPARATOR
L_10 - ST_OCTAL_BASE
L_11 - ST_OCTAL_CORE
L_12 - ST_OCTAL_SEPARATOR
L_13 - ST_HEXA_BASE
L_14 - ST_HEXA_CORE
L_15 - ST_HEXA_SEPARATOR

Figure 2: Legend of states of finite state automaton

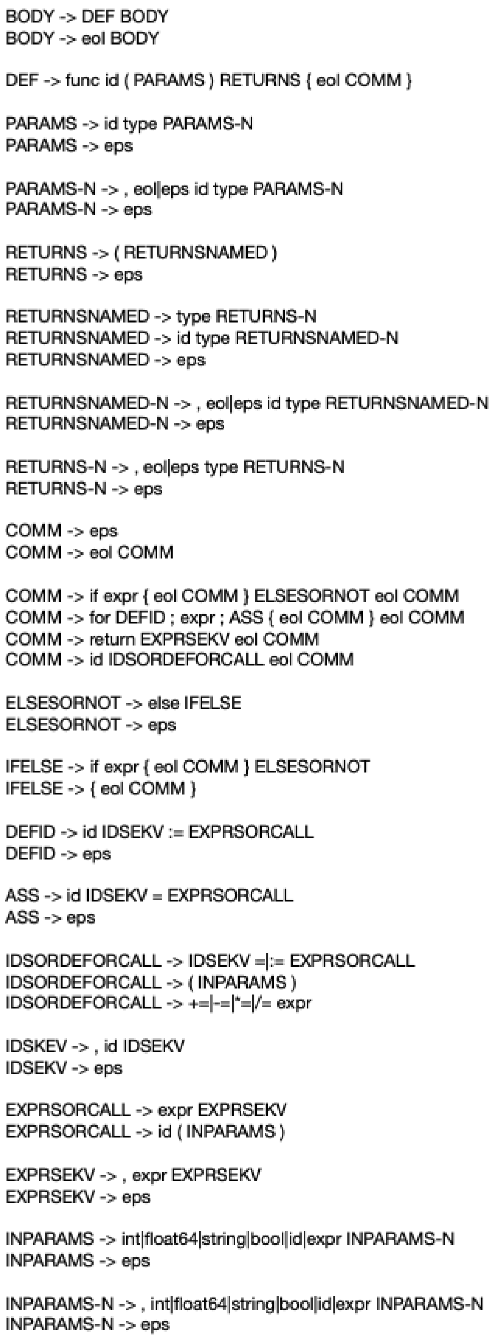
C LL-grammar table



	eol	func	id	(	)	{	}	type	,	eol|ε	if	expr	for	;	return	else	:=	=	=|:=	+=|-=|...	int|f...	$
BODY	2	1																				
DEF		3																				
PARAMS			4		5																	
RETURNS				8		9																
COMM	17		21				16				18		19		20							
PARAMS-N					7				6													
RETURNSNAMED			11					10														
RETURNS-N					15				14													
RETURNSNAMED-N					13				12													
ELSESORNOT	23															22						
DEFID			26											27								
ASS			28			29																
EXPRSEKV	38					38			37					38								
IDSORDEFORCALL				31															30	32		
IFELSE						25					24											
IDSEKV																	34	34	34			
EXPRSORCALL			36									35										
INPARAMS					40																39	
IDSKEV									33													
INPARAMS-N					42				41													

Figure 3: LL-table used for syntax analysis

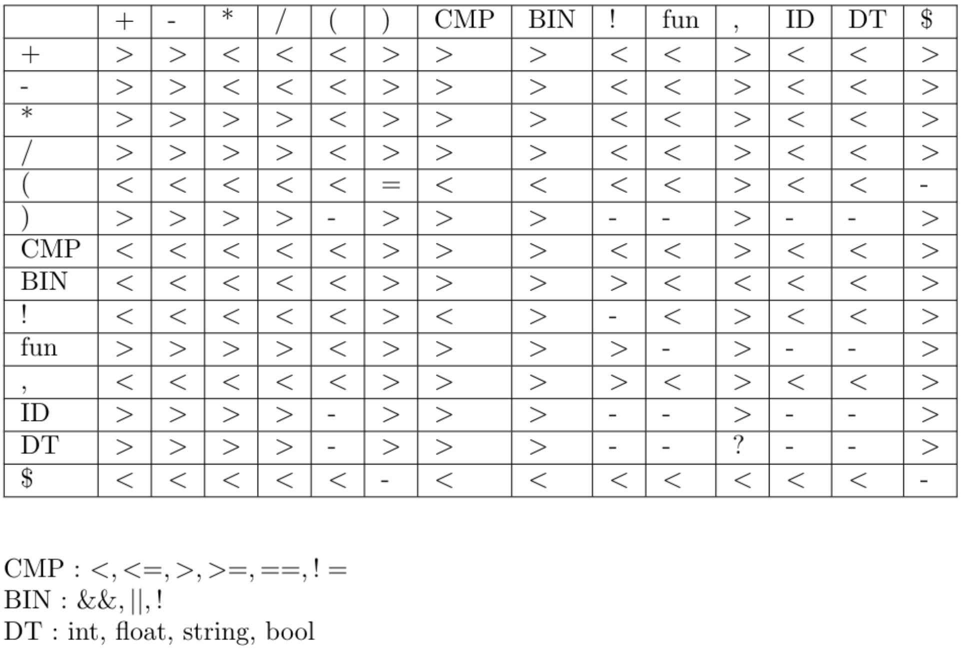
D LL-grammar



```
BODY -> DEF BODY
BODY -> eol BODY

DEF -> func id ( PARAMS ) RETURNS { eol COMM }

PARAMS -> id type PARAMS-N
PARAMS -> eps

PARAMS-N -> , eol|eps id type PARAMS-N
PARAMS-N -> eps

RETURNS -> ( RETURNSNAMED )
RETURNS -> eps

RETURNSNAMED -> type RETURNS-N
RETURNSNAMED -> id type RETURNSNAMED-N
RETURNSNAMED -> eps

RETURNSNAMED-N -> , eol|eps id type RETURNSNAMED-N
RETURNSNAMED-N -> eps

RETURNS-N -> , eol|eps type RETURNS-N
RETURNS-N -> eps

COMM -> eps
COMM -> eol COMM

COMM -> if expr { eol COMM } ELSESORNOT eol COMM
COMM -> for DEFID ; expr ; ASS { eol COMM } eol COMM
COMM -> return EXPRSEKV eol COMM
COMM -> id IDSORDEFORCALL eol COMM

ELSESORNOT -> else IFELSE
ELSESORNOT -> eps

IFELSE -> if expr { eol COMM } ELSESORNOT
IFELSE -> { eol COMM }

DEFID -> id IDSEKV := EXPRSORCALL
DEFID -> eps

ASS -> id IDSEKV = EXPRSORCALL
ASS -> eps

IDSORDEFORCALL -> IDSEKV =|:= EXPRSORCALL
IDSORDEFORCALL -> ( INPARAMS )
IDSORDEFORCALL -> +=|-=|*=|/= expr

IDSKEV -> , id IDSEKV
IDSEKV -> eps

EXPRSORCALL -> expr EXPRSEKV
EXPRSORCALL -> id ( INPARAMS )

EXPRSEKV -> , expr EXPRSEKV
EXPRSEKV -> eps

INPARAMS -> int|float64|string|bool|id|expr INPARAMS-N
INPARAMS -> eps

INPARAMS-N -> , int|float64|string|bool|id|expr INPARAMS-N
INPARAMS-N -> eps
```

Figure 4: LL-grammar used for syntax analysis

E Precedence table

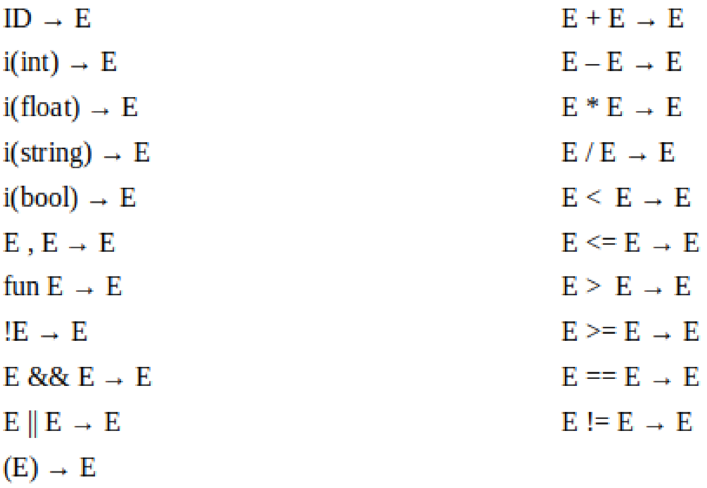


	+	-	*	/	(	)	CMP	BIN	!	fun	,	ID	DT	$
+	>	>	<	<	<	>	>	>	<	<	>	<	<	>
-	>	>	<	<	<	>	>	>	<	<	>	<	<	>
*	>	>	>	>	<	>	>	>	<	<	>	<	<	>
/	>	>	>	>	<	>	>	>	<	<	>	<	<	>
(	<	<	<	<	<	=	<	<	<	<	>	<	<	-
)	>	>	>	>	-	>	>	>	-	-	>	-	-	>
CMP	<	<	<	<	<	>	>	>	<	<	>	<	<	>
BIN	<	<	<	<	<	>	>	>	>	<	<	<	<	>
!	<	<	<	<	<	>	<	>	-	<	>	<	<	>
fun	>	>	>	>	<	>	>	>	>	-	>	-	-	>
,	<	<	<	<	<	>	>	>	>	<	>	<	<	>
ID	>	>	>	>	-	>	>	>	-	-	>	-	-	>
DT	>	>	>	>	-	>	>	>	-	-	?	-	-	>
$	<	<	<	<	<	-	<	<	<	<	<	<	<	-

CMP : <, <=, >, >=, ==, ! =
BIN : &&, ||, !
DT : int, float, string, bool

Figure 5: Precedence table

F Precedence rules



ID → E
i(int) → E
i(float) → E
i(string) → E
i(bool) → E
E , E → E
fun E → E
!E → E
E && E → E
E || E → E
(E) → E

E + E → E
E – E → E
E * E → E
E / E → E
E < E → E
E <= E → E
E > E → E
E >= E → E
E == E → E
E != E → E

Figure 6: Set of all rules used by precedence analysis