Mini-Pascal compiler documentation

Suvi Sipilä, 014220326

Mariehamn May 20, 2020

UNIVERSITY OF HELSINKI
Computer Science Master's programme
CSM14205 - Code Generation

Contents

1 Instruction **1**

2 Architecture **1**

3 Context-free grammmar **3**

4 Regular expressions **4**

5 Language implementation-level decisions **5**

6 Semantic analysis **6**

7 Problems **6**

8 Error handling **7**

9 Program example **9**

10 Work hour log **10**

A Mini-Pascal Syntax **11**

1 Instruction

The program is written in C# (version .NET Core 3.1.1). Give the Mini-Pascal program code to the CodeExample.txt file. Execute the code at the root of the project by writing ***dotnet run*** in the terminal. Optionally, you can log how the CallStack behaves during execution by assigning a *true* value to the Interpreter. If there are no errors in the given program, CodeGenerator generates the target code in C-language in the GeneratorOutput.txt file.

2 Architecture

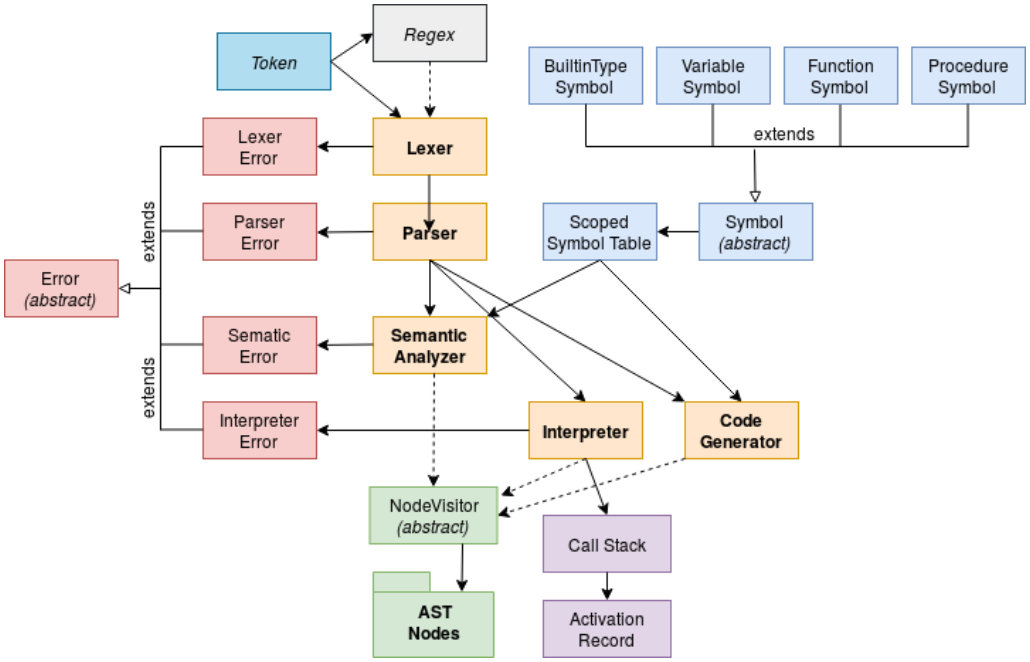

Figure 1: Architecture

The program reads the file containing the Mini-Pascal program code. The program delivers the text to Lexer, which reads and analyzes all the given characters and converts them into a token stream. Lexer uses rules defined by the Regex class. The Regex class sets rules that contain reserved keywords, character escapes, token types, letters, and digits. The Parser class gets Lexer as its parameter and analyzes the entire nested structure of the code and determines if the given token sequence is valid according to the grammar. The SemanticAnalyzer class analyzes the AST tree which Parser has parsed. The SemanticAnalyzer is responsible for visiting all the AST nodes (figure 2) with the help of NodeVisitor. It also uses ScopedSymbolTable to determine whether the types are correct, all variables are defined, and that the code itself makes sense. ScopedSymbolTable uses four classes, BuiltinTypeSymbol,

VariableSymbol, FunctionSymbol and ProcedureSymbol, which extends the abstract Symbol class. BuiltinTypeSymbol is responsible for built-in types such as *integer*, *string*, *Boolean*, etc. VariableSymbol class is responsible for variable symbols like *name = 'x'*, *type = 'int'*. FunctionSymbol and ProcedureSymbol classes are responsible for functions and procedures and their formal parameters. Interpreter and CodeGenerator classes are used only if there are no errors in the given program. The Interpreter visits all AST nodes with the help of NodeVisitor. Interpreter takes care of run-time values and stores them to the CallStack which uses ActivationRecord class. The values are stored in the activation records which can be accessed using the basic methods *Push*, *Pop* and *Peek*. CodeGenerator visits all AST nodes and generates the target code in C-language in the GeneratorOutput.txt file.

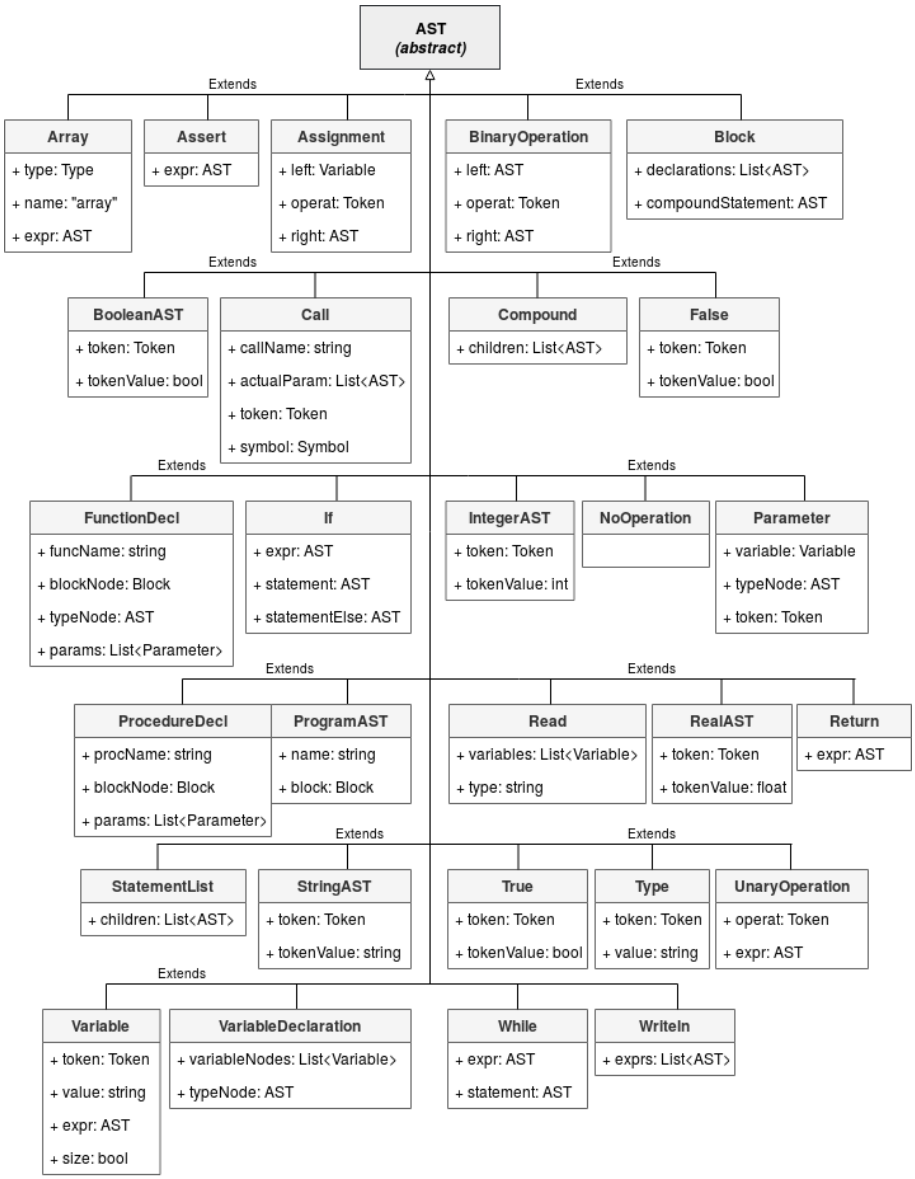

Figure 2: AST nodes

3 Context-free grammmar

Modified context-free grammar for Mini-Pascal.

```
<program>          ::= "program" <id> ";" { <block> } "."
<block>            ::= { <declarations> } { <compound> }
<declarations>     ::= <variableDecl> ";" | <procedureDecl>
                     | <functionDecl> |  ε
<compound>         ::= "begin" { <statementList> } "end"
<variableDecl>     ::= "var" <id> { "," <id> } ":" <type>
<procedureDecl>    ::= "procedure" <id> "(" <formalParamList> ")"
                       ";" <block> ";"
<functionDecl>     ::= "function" <id> "(" <formalParamList> ")" ":"
                       <type> ";" <block> ";"
<formalParamList>  ::= <formalParam> ";" { <formalParam> ";" }
                     | ε
<formalParam>      ::= <id> { "," <id> } ":" <type>
<type>             ::= "int" | "real" | "string" | "bool" | <array>
<array>            ::= "array" "(" { <expr> } ")" "of" <type>

<statement_list>   ::= <statement> ";" { <statement> }
<statement>        ::= <compound> | <variableDecl> | <call>
                     | <assigment> | <while> | <if> |  <read>
                     | <return> | <writeln> | <assert> | ε
<call>             ::= <id> "(" { <expr> { ", <expr> } } ")"
<assigment>        ::= <id> { "[" <expr> "]" } ":=" <call>
                     | <id> { "[" <expr> "]" } ":=" <expr>
<while>            ::= "while" <expr> "do" <statement>
<if>               ::= "if" <expr> "then" <statement> ";"
                       { "else" <statement> }
<read>             ::= "read" "(" <factor> { "," <factor> } ")"
<return>           ::= "return" { <expr> }
<writeln>          ::= "writeln" "(" <expr> { "," <expr> } ")"
<assert>           ::= "assert" "(" <expr> ")"

<expr>             ::= <term> { <operator> <term> }
<term>             ::= <factor> { <multiOperator> <factor> }
<factor>           ::= { "+" | "-" <factor> } | <literal>
                     | "true" | "false" | "(" <expr> ")"
                     | "not" <factor> | <id> "." "size"
                     | <call> | <id> { "[" <expr> "]" }

<multiOperator>    ::= "*" | "/" | "%" | "and"
<operator>         ::= "+" | "-" | "=" | "<" | "<>" | "<="
                     |  ">=" | ">" | "or"

<id>               ::= <letter> { <letter> | <digit> | "_" }
<literal>          ::= <integerLiteral> | <realLiteral>
                     |  <stringLiteral> | Boolean
<integerLiteral>   ::= <digits>
<digits>           ::= <digit> { <digit> }
<realLiteral>      ::= <digits> "." <digits> ["e" [ "+ " | "-"]
                       <digits>]
<stringLiteral>    ::= "\"" { <letter> | <escapeChar> } "\""
```

```
<letter>          ::= a | b | c | d | e | f | g | h | i | j | k
                    | l | m | n | o | p | q | r | s | t | u | v
                    | w | x | y | z | A | B | C | D | E | F | G
                    | H | I | J | K | L | M | N | O | P | Q | R
                    | S | T | U | V | W | Y | Z
<digit>           ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
<escapeChar>      ::= \\ | \a | \b | \f | \n | \r | \t | \v
```

4 Regular expressions

Token type	Regex	Explanation
Relational operators		
EQUAL	=	Equality comparison
INEQUALITY	<>	Not equal comparison
LESS	<	Less-than comparison
LESS_OR_EQUAL	<=	Less-or-equal-than comparison
GREATE_OR_EQUAL	>=	Greater-or-equal-than comparison
GREATER	>	Greater-than comparison
Negation		
NOT	**not**	Boolean operation, logical not
Adding operators		
PLUS	+	Integer and real addition, string concetenation
MINUS	-	Integer and real substraction
OR	**or**	Boolean operation, logical or
Multiplying operators		
MUL	*	Integer and real multiplication
DIV	/	Integer and real division
MODULO	%	Integer modulo
AND	**and**	Boolean operation, logical and
Special symbols		
LEFT_PARENTHESIS	(	Start of nested expression
RIGHT_PARENTHESIS	)	End of nested expression
LEFT_BRACKET	[	Start of array
RIGHT_BRACKET	]	End of array
ASSING	:=	Assigment
DOT	.	End of program, real number, and array size separator
COMMA	,	Variable, parameter, expression separator
SEMI	;	End of statement
COLON	:	Type definition
Keywords		
OR	**or**	Boolean operation, logical or
AND	**and**	Boolean operation, logical and
NOT	**not**	Boolean operation, logical not
IF	**if**	Start of if-statement
THEN	**then**	Part of if-statement
ELSE	**else**	Part of if-statement
OF	**of**	Array type definition
WHILE	**while**	Start of while-statement
DO	**do**	Part of while-statement
BEGIN	**begin**	Start of compound

END	**end**	End of compound
VAR	**var**	Variable definition
ARRAY	**array**	Array definition
PROCEDURE	**procedure**	Start of procedure
FUNCTION	**function**	Start of function
PROGRAM	**program**	Start of program
ASSERT	**assert**	Start of assert-statement
RETURN	**return**	Start of return-statement
Predefined id		
BOOLEAN	Boolean	Boolean type
FALSE	false	False type
INTEGER	integer	Integet type
READ	read	Start of read-statement, reads input stream
REAL	real	Real type
SIZE	size	Size of array
STRING	string	String type
TRUE	true	True type
WRITELN	writeln	Start of write-statement
Other		
INTEGER_CONST	[0-9]+	Integer value
REAL_CONST	[0-9]+ \. [0-9]+([eE][+-]?[0-9]+)	Real value
EOF		End of input
ID	[a-zA-Z0-9_]+	Identifier
ERROR	Error	Invalid ID-token

Figure 3: Mini-Pascal regular expressions

5 Language implementation-level decisions

Assignment and Call statement begins both with ID. This is solved in the Parser's *AssignmentStatement* method, where it checks to see if there is an ID token after the assign token and if lexer's current character is "(", in which case it goes to the *CallStatement* method. Otherwise, it is the assignment statement and it goes to the *Expr* method. If statement may or may not have else-part. The parser checks if the current token type is ELSE, and if not, it sets *continueList* parameter to *true*. This is a flag to the *StatementList* method, which typically adds a statement to the statement list when the current token type is SEMI (;). But if the else-part does not exist, the *StatementList* method does not know if new statements are coming after the if-statement. So, if the *continueList* parameter is *true*, the *StatementList* method adds a new statement even if the current token type is not SEMI. Factor is a left-recursive and there is a problem with size and Call statement, both beginning with ID. This is solved by checking whether the current character is dot or a left parenthesis. There is also a check on the Variable side to see if the next token is a left bracket, so it would be the array type variable.

6 Semantic analysis

1. Type checking
    - Variable, parameter, and function must have a type.
    - Assigned value to the variable, parameter, or function, must have a correct type.
2. Call
    - Call statement is accepted only in Procedures and Functions.
    - Call statement must have the correct number of arguments.
3. Variable and ID
    - Variable names must be defined in ScopedSymbolTable before they can be used.
    - Identifier must be unique.
4. Array
    - The array must have a pointer and the pointer must be a number in array assignment (for example A[**0**]).
    - The value of the pointer must be correct with the size of the array, and it cannot point outside of it.
5. Size
    - Operation *.size* can only be used in array types.
6. Boolean expression
    - Assert, If and While statements accept only Boolean expressions.

7 Problems

One of the biggest problems is with types - is the type an integer, string, real, Boolean, or Array. The program has a Type-AST, but it is not used on all AST nodes. For example, Read-AST has a type parameter as a string type. This is because I started this project by expanding the previous *Mini-PL Interpreter* project, and I didn't notice all the places that should have been fixed to fit this project. Because some AST nodes do not have Type-AST, checking types is complicated. Also, the fact that I had forgotten the Array type and implemented it as the last thing, made the type checking very difficult. Builtin type symbols do not have a Type parameter because the Builtin name defines what type of symbol it is (for example an integer, string, etc.). But in the array-case, it is not enough to know

that Builtin is an array type because we need to know what type of array it is. This problem is solved by SemanticAnalyzer's *VisitVariableDeclaration* method, where it adds an Array-AST as a symbol type. When checking the Builtin type symbols, the program can access the Array-AST node where is information about what is the type of array.

A function that has not yet been defined but is called by a recursive call was hard to implement. The problem is solved by SemanticAnalyzer's *VisitCall* method, where it adds the Call node in the *onHoldNodes* list. Then the *VisitFunctionDecl* method checks if the current function declaration has the same name as the Call node in the *onHoldNodes* list. If it is the same, then it visits the Call node and removes the node from the *onHoldNodes* list.

CodeGeneration class was easy to implement with the help of NodeVisitor. It is done in almost the same way and logic as SemanticAnalyzer. It adds a scope level to each statement, making it easier to keep track of which scope level the statement belongs to. The output is added to the string parameter, and that makes the class so long. This could have been done with some other solution to make the code cleaner. In fact, the whole project should be refactored. One difficult task was adding the semicolon to the end of the statement. This is solved by the *AddSemicolonIfNeeded* method, which checks if the last character is a semicolon or not.

8 Error handling

Errors are handled in Lexer, Parser, SemanticAnalyzer, and Interpreter. If Lexer, Parser or SemanticAnalyzer detects an error, they add the error to the error list and continue analyzing the rest of the source program. For example, if the Parser detects an error, it reports the error and skips to the next statement. If the ID has an incorrect character that is not in the Regex lists, the Lexer reports the error and generates an Error-token, so that the Parser knows that the ID is invalid. Once the entire program is analyzed and if errors are found, the program prints the errors and stops running. If there are no errors, then the program goes to the Interpreter, which detects run-time errors, for example, if the user input is incorrect. If the Interpreter does not detect errors, CodeGeneration class generates a GeneratorOutput.txt file. Below are examples of errors and how to handle them.

```
program Main;
  var hello : string;
  var ä : string;
  var x;
  begin
    x := 5;
    hello := "hello;
    ä := 2;
    writeln(ä);
  end.
```

```
>>LEXER ERROR
>>Invalid character: ä, position=3:7
>>
>>PARSER ERROR
>>Unexpected token -> Token(ERROR, ä, position=3:8)
>>
>>PARSER ERROR
>>Unexpected token -> Token(SEMI, ;, position=4:8)
>>
>>LEXER ERROR
>>End of input while scanning a string literal
>>
>>LEXER ERROR
>>Invalid character: ä, position=8:5
>>
>>LEXER ERROR
>>Invalid character: ä, position=9:13
>>
>>SEMANTIC ERROR
>>Variable must have a type -> Token(ERROR, ä, position=3:8)
>>
>>SEMANTIC ERROR
>>Variable must have a type -> Token(ERROR, x, position=4:7)
>>
>>SEMANTIC ERROR
>>Variable name not found -> Token(ID, x, position=6:5)
```

```
program Main;
  var a : integer;
  begin
    a := "a";
    writeln(a);
  end.

>>SEMANTIC ERROR
>>Wrong type! Should be integer, not a string ->
>>Token(STRING, a, position=4:13)
```

```
program Main;
  function F (n: integer) : integer;
  begin
    writeln(n);
    if n < 1 then return n;
    else
      return F(n-1);
  end;

  begin
    var i, c : integer;
    read (i, c);
    writeln (F(i, c));
  end.

>>SEMANTIC ERROR
>>Wrong number of arguments -> Token(ID, F, position=13:14)
```

9 Program example

```
program Recursion;

function F(n: integer) : integer;
begin
  writeln(n);
  if n = 0 then
    return 1;
  else
    return M(n-1);
end;

function M(n: integer) : integer;
begin
  writeln(n);
  if n = 0 then return 0;
  else return F(n-1);
end;

begin
  var i : integer;
  i := 5;
  while i > 1 do F(i);
end.

>>5
>>4
>>3
>>2
>>1
>>0
```

```
// CodeGeneratorOutput.txt
#include <stdio.h>
#include <assert.h>
#include <stdbool.h>

int Recursion0() {

int F1(int n2) {
printf(n2);
if2 n2 = 0
goto ifStatement2;
else
goto ifStatementElse2;
ifStatement2:
return 1;

ifStatementElse2:
return M(n2 - 1);
}
```

```
int M1(int n2) {
printf(n2);
if2 n2 = 0
goto ifStatement2;
else
goto ifStatementElse2;
ifStatement2:
return 0;

ifStatementElse2:
return F(n2 - 1);
}

int i1;
i1 = 5;
while1 i1 > 1
goto whileStatement1;
whileStatement1:
F(i1);

return 0;
}
```

10 Work hour log

Date	Time	Work done
06.04.2020	4h	Building a project (expanding the previous Mini-PL)
08.04.2020	6h	Scopes and SymbolTable
22.04.2020	6h	Starting CodeGeneration class, learning C
27.04.2020	6h	AR and CallStack classes, log method
28.04.2020	6h	Procedure, Call and Function
29.04.2020	6,5h	Return-AST, AR fix so it returns rv value, starting type checking
04.05.2020	6h	Boolean, True, False -AST, fix Read statement
05.05.2020	6h	Fixed Error classes so they don't throw Error right away, While-statement
06.05.2020	6h	CodeGeneration class
11.05.2020	6h	CodeGeneration class, recursive calls
12.05.2020	10h	CodeGeneration, modulo, fixed return so it does not have to have return value, Array type
13.05.2020	9h	Array continues, CodeGeneration and array
18.05.2020	9h	Array size and its errors, Regex class, documenting
19.05.2020	8h	Documentation, diagrams, testing, Error token
20.05.2020	9h	Documentation, fixed bug in recursive call
TOTAL HOURS: 103,5h		

Figure 4: Work log

A Mini-Pascal Syntax

Syntax of Mini-Pascal (Spring 2020)

Mini-Pascal is a simplified (and slightly modified) subset of Pascal. Generally, the meaning of the features of Mini-Pascal programs are similar to their semantics in other common imperative languages, such as C.

1. A Mini-Pascal program consist of series of functions and procedures, and a main block. The subroutines may call each other and may be (mutually) recursive. Within the same scope (procedure, function, or block), identifiers must be unique but it is OK to redefine a name in an *inner* scope.
2. A **var** parameter is passed *by reference*, i.e. its address is passed, and inside the subroutine the parameter name acts as a synonym for the variable given as an argument. A called procedure or function can freely read and write a variable that the caller passed in the argument list.
3. Mini-Pascal includes a C-style **assert** statement. If an assertion fails the system prints out a diagnostic message and halts execution.
4. The Mini-Pascal operation *a.size* only applies to values of type **array of** *T* (where *T* is a simple type). There are only one-dimensional arrays. Array types are compatible only if they have the same element type. Arrays' indices begin with zero. The compatibility of array indices and array sizes is checked at run time (usually).
5. By default, variables in Pascal are not initialized (with zero or otherwise); so they may initially contain rubbish (random) values.
6. A Mini-Pascal program can print numbers and strings via the predefined special routines *read* and *writeln*. The stream-style input operation *read* makes conversion of values from their text representation to appropriate internal numerical (binary) representation.
7. Pascal is a case non-sensitive language, which means you can write the names of variables, functions and procedures in either case.
8. The Mini-Pascal *multiline comments* are enclosed within curly brackets and asterisks as follows: "{* . . . *}".
9. Note that the names *Boolean*, *false*, *integer*, *read*, *real*, *size*, *string*, *true*, *writeln* are treated in Mini-Pascal as "predefined identifiers", i.e., it is allowed to use them as regular identifiers in Mini-Pascal programs.

The arithmetic operator symbols '+', '-', '*', and '/' represent the following functions, where T is either "*integer*" or "*real*".

```
"+" : (T, T) -> T            // addition
"-" : (T, T) -> T            // subtraction
"*" : (T, T) -> T            // multiplication
"/" : (T, T) -> T            // division
```

The operator '%' represents integer modulo operation. The operator '+' *also* represents string concatenation:

```
"%" : (integer, integer) -> integer          // integer modulo
"+" : (string, string) -> string             // string concatenation
```

The operators "**and**", "**or**", and "**not**" represent Boolean operations:

```
"or"  : (Boolean, Boolean) -> Boolean        // logical or
"and" : (Boolean, Boolean) -> Boolean        // logical and
"not" : (Boolean) -> Boolean                 // logical not
```

The relational operators "=", "<>", "<", "<=", ">=", ">" are overloaded to represent the comparisons between two values of the same type, with the obvious meanings. They can be applied to values of the types *int*, *real*, *string*, *Boolean*.

Context-free syntax notation for Mini-Pascal

The syntax definition is given in so-called *Extended Backus-Naur* form (EBNF). In the following Mini-Pascal grammar, the use of curly brackets "{ ... }" means 0, 1, or more repetitions of the enclosed items. Parentheses may be used to group together a sequence of related symbols. Brackets ("[" "]") may be used to enclose optional parts (i.e., zero or one occurrence). Reserved keywords are marked bold (as "**bold**"). Operators, separators, and other single or multiple character tokens are enclosed within quotes (as ":="). Note that the syntax given below also specifies the precedence of operators (via productions defined at different hierarchical levels).

Context-free grammar

<program> ::= "**program**" *<id>* "**;**" { *<procedure>* | *<function>* } *<main-block>* "**.**"

<procedure> ::= "**procedure**" *<id>* "(" parameters ")" "**;**" *<block>* "**;**"

<function> ::= "**function**" *<id>* "(" parameters ")" "**:**" *<type>* "**;**" *<block>* "**;**"

<var-declaration> ::= "**var**" *<id>* { "**,**" *<id>* } "**:**" *<type>*

<parameters> ::= ["**var**"] *<id>* "**:**" *<type>* { "**,**" ["**var**"] *<id>* "**:**" *<type>* } | <empty>

<type> ::= *<simple type>* | *<array type>*

<array type> ::= "**array**" "**[**" [*<integer expr>*] "**]**" "**of**" *<simple type>*

<simple type> ::= *<type id>*

<block> ::= "**begin**" *<statement>* { "**;**" *<statement>* } ["**;**"] "**end**"

<statement> ::= *<simple statement>* | *<structured statement>* | *<var-declaration>*

<empty> ::=

<simple statement> ::= *<assignment statement>* | *<call>* | *<return statement>* |
< read statement> | *<write statement>* | *<assert statement>*

<assignment statement> ::= *<variable>* ":=" *<expr>*

<call> ::= *<id>* "**(**" *<arguments>* "**)**"

<arguments> ::= expr { "**,**" expr } | *<empty>*

<return statement> ::= "**return**" [expr]

<read statement> ::= "*read*" "**(**" *<variable>* { "**,**" *<variable>* } "**)**"

<write statement> ::= "*writeln*" "**(**" *<arguments>* "**)**"

<assert statement> ::= "**assert**" "**(**" *<Boolean expr>* "**)**"

<structured statement> ::= *<block>* | *<if statement>* | *<while statement>*

<if statement> ::= "**if**" *<Boolean expr>* "**then**" *<statement>* |
"**if**" *<Boolean expr>* "**then**" *<statement>* "**else**" *<statement>*

<while statement> ::= "**while**" *<Boolean expr>* "**do**" *<statement>*

<expr> ::= *<simple expr>* |
<simple expr> *<relational operator>* *<simple expr>*

<simple expr> ::= [*<sign>*] *<term>* { *<adding operator>* *<term>* }

<term> ::= *<factor>* { *<multiplying operator>* *<factor>* }

<factor> ::= *<call>* | *<variable>* | *<literal>* | "**(**" *<expr>* "**)**" | "**not**" *<factor>* | *< factor>* "**.**" "*size*"

<variable> ::= *<variable id>* ["**[**" *<integer expr>* "**]**"]

<relational operator> ::= "**=**" | "**<>**" | "**<**" | "**<=**" | "**>=**" | "**>**"

<sign> ::= "+" | "-"

<negation> ::= "**not**"

<adding operator> ::= "+" | "-" | "**or**"

<multiplying operator> ::= "*" | "/" | "**%**" | "**and**"

Lexical grammar

<id> ::= *<letter>* { *<letter>* | *<digit>* | "**_**" }

<literal> ::= *<integer literal>* | *<real literal>* | *<string literal>*

<integer literal> ::= *<digits>*

<digits> ::= *<digit>* { *<digit>* }

<real literal> ::= *<digits>* "." *<digits>* ["**e**" [*<sign>*] *<digits>*]

<string literal> ::= "\"" { < *a char or escape char* > } "\""

<letter> ::= a | b | c | d | e | f | g | h | i | j | k | l | m | n | o |
p | q | r | s | t | u | v | w | x | y | z | A | B | C |
D | E | F | G | H | I | J | K | L | M | N | O | P
| Q | R | S | T | U | V | W | X | Y | Z

<digit> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

<special symbol or keyword> ::= "+" | "-" | "*" | "%" | "=" | "<>" | "<" | ">" | "<=" | ">=" |
"(" | ")" | "[" | "]" | ":=" | "." | "," | ";" | ":" | "**or**" |
"**and**" | "**not**" | "**if**" | "**then**" | "**else**" | "**of**" | "**while**" | "**do**" |
"**begin**" | "**end**" | "**var**" | "**array**" | "**procedure**" |
"**function**" | "**program**" | "**assert**" | "**return**"

<predefined id> ::= "*Boolean*" | "*false*" | "*integer*" | "*read*" | "*real*" | "*size*" |
"*string*" | "*true*" | "*writeln*"
