Syntax of Mini-Pascal (Spring 2016)

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. Mini-Pascal allows nested functions and procedures, i.e. subroutines which are defined within another. A nested function is invisible outside of its immediately enclosing function, but can access preceding visible local objects (data, functions, etc.) of its immediately enclosing function as well as of any function(s) which, in turn, encloses that function.
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 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 usually checked at run time.
5. By default, variables in Pascal are not initialized (with zero or otherwise); so they may initially contain rubbish values.
6. A Mini-Pascal program can print numbers and strings via the predefined special routines *read* and *writeln*. The stream-style IO makes conversion of values from their text representation to their 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*, *integer*, *false*, *read*, *real*, *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>* "**;**" *<block>* "**.**"

<declaration> **::=** "**var**" *<id>* { **,** *<id>* } "**:**" *<type>* "**;**" |
"**procedure**" *<id>* "(" parameters ")" "**;**" *<block>* "**;**" |
"**function**" *<id>* "(" parameters ")" "**:**" *<type>* "**;**" *<block>* "**;**"

<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>* | *<declaration>*

<empty> **::=**

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

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

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

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

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

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

<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**"

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