MINISQUARE-22

LANGUAGE SPECIFICATION

GRAMMAR NOTATION

```
<non-terminal>
terminal
x | y - either x or y
x*    - zero or more xs
x+    - one or more xs
[x]   - zero or one x
```

GRAMMAR FOR THE SCANNER

```
<identifier>          ::=  <digit>* <letter> ( <letter> | <digit> )*

<integer-literal>     ::=  <digit> <digit>*

<character-literal>   ::=  { <graphic> }

<graphic>             ::=  <letter> | <digit> | space | ?

<operator>            ::=  + | - | * | / | < | > | = | !

<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
```

Reserved words and punctuation symbols from the parser grammar must also be recognised as tokens.

Comments and white space can be placed anywhere and have no meaning, except for separating other items.

Comments begin with a &, followed by any characters, and last until the end of the line.

Spaces, tabs and end of line characters are all white space.

GRAMMAR FOR THE PARSER

```
<program>                ::=  <single-command>

<command>                ::=  <single-command> ( ; <single-command> )*

<single-command>         ::=  blank (i.e. nothing)
                             |<identifier> ( ~ <expression> | ( <parameter> ) )
                             |? <expression> => <single-command>
                             |if ( <expression> ) then <single-command>
                              else <single-command>
                             |while ( <expression> ) <single-command> wend
                             |loop <single-command> while ( <expression> )
                              <single-command> repeat
                             |let <declaration> in <single-command>
                             |begin <command> end

<declaration>            ::=  <single-declaration> ( ; <single-declaration> )*

<single-declaration>     ::=  const <identifier> ~ <expression>
                             |var <identifier> ~ <type-denoter>

<parameter>              ::=  blank (i.e. nothing)
                             |<expression>
                             |var <identifier>

<type-denoter>           ::=  <identifier>

<expression>             ::=  <primary-expression> ( <operator> <primary-expression> )*

<primary-expression>     ::=  <integer-literal>
                             |<character-literal>
                             |<identifier> [ ( <parameter> ) ]
                             |<operator> <primary-expression>
                             |( <expression> )
```

ABSTRACT GRAMMAR, SEMANTICS AND TYPE RULES

```
<program>      ::=  <command>                                          Program
```

- Executing a program executes the command and exits.

```
<command>      ::=  blank (i.e. nothing)                               BlankCommand
                   |<identifier> ~ <expression>                        AssignCommand
                   |<identifier> ( <parameter> )                       CallCommand
                   |? <expression> => <command>                        QuickIfCommand
                   |if ( <expression> ) then <command> else <command>  IfCommand
                   |while ( <expression> ) <command> wend              WhileCommand
                   |loop <command> while ( <expression> ) <command>    LoopCommand
                    repeat
                   |let <declaration> in <command>                     LetCommand
                   |begin <command> end                                BeginCommand
                   |<command> ; <command>                              SequentialCommand
```

- A blank command does nothing when executed.
- An assignment command **V ~ E** is executed by first evaluating **E** then assigning that value to variable **V**. **V** and **E** must have the same type.
- A call command **F(P)** is executed by first evaluating the parameter **P** then calling the function or procedure **F** with these parameters. If **F** is a function that returns a value, this should be discarded. The number and type of the parameters must match the parameter types given in the declaration of **F**.
- A quick if command **? E => C** is executed as follows. **E** is evaluated. If it is true then **C** is executed. **E** must be have a type of Boolean.
- An if command **if (E) then C1 else C2** is executed as follows. **E** is evaluated. If it is true then **C1** is executed, otherwise **C2** is executed. **E** must be have a type of Boolean.
- A while command **while (E) C wend** is executed as follows. **E** is evaluated and if it is true then **C** is executed. This is repeated until **E** evaluates to false. When this happens, the command finishes. **E** must have a type of Boolean.
- A loop command **loop C1 while (E) C2 repeat** is executed as follows. **C1** is executed. **E** is evaluated and if it is true then **C2** is executed. This process is repeated (execute **C1**, evaluate **E**, execute **C2** if **E** was true) until **E** is false. When this happens, the command finishes. **E** must have a type of Boolean.
- **let D in C** is executed as follows. The declarations in **D** are performed then **C** is executed. More information is given in the Identifier Rules section of this document.
- **begin C end** is executed by simply executing **C**.
- A sequential command **C1; C2** is executed by executing **C1** then executing **C2**.

```
<declaration>  ::=  <declaration> ; <declaration>                      Declarations
                   |const <identifier> ~ <expression>                  ConstDeclaration
                   |var <identifier> ~ <type-denoter>                  VarDeclaration
```

- Declarations **D1; D2** are performed by performing **D1** then performing **D2**.
- The declaration **const I ~ E** is performed by evaluating the expression **E** then binding this value to **I**. The type of **I** is the type of **E**.
- The declaration **var I ~ T** is performed by binding **I** to a new variable of type **T**. The variable's value is initially undefined.

More information about identifiers appears in the Identifier Rules section of this document.

```
<parameter>    ::=  blank (i.e. nothing)                               BlankParameter
                   |<expression>                                       ValueParameter
                   |var <identifier>                                   VarParameter
```

- An empty parameter does nothing when evaluated.
- A value parameter is evaluated by evaluating the expression.

- Expression parameters are passed by value, variable parameters are passed by reference.

```
<type-denoter>::=  <identifier>                                          TypeName
```

- Identifier must be a known type; Boolean, Integer or Char.

```
<expression>   ::=  <integer-literal>                                    IntegerExpression
                   |<character-literal>                                  CharExpression
                   |<identifier>                                         IdExpression
                   |<identifier> ( <parameter> )                         CallExpression
                   |<expression> <operator> <expression>                 BinaryExpression
                   |<operator> <expression>                              UnaryExpression
                   |( <expression> )                                     BracketExpression
```

- Evaluation of an integer expression is simply the value of the integer literal. The type of the expression is Integer.
- Evaluation of a char expression is simply the value of the charcter literal. The type of the expression is Char.
- Evaluation of an ID expression gives the current value associated with that variable or constant in the current scope. The type of the expression is the same as the type of the variable or constant.
- Evaluation of a function call is performed as follows. First the parameters are evaluated. The function is then called. The value of the expression is the value returned by the function. The types of the parameters must match the function declaration and the function must have a return type. The type of the expression is the return type of the function. The call must be to a function, **not** to a procedure, i.e. a value must be returned.
- Binary expressions **E1 op E2** are evaluated as would be expected, with the usual rules of mathematical precedence. If the operator is anything other than **=** then both **E1** and **E2** must be Integers. If the operator is **=** then **E1** and **E2** must be of the same type. If the operator is **+**, **-**, ***** or **/** then the expression type is Integer. Otherwise, the expression type is Boolean.
- Unary expressions **OE** must have **O = !** and **E** must be of type Boolean. **!E** is evaluted by evaluating **E** then taking its logical negation.
- A bracketed expression **(E)** is evaluated by simply evaluating **E**. The type of the expression is the same as **E**'s type.

```
<identifier>   ::=  <digit>* <letter> ( <letter> | <digit> )*          Identifier

<operator>     ::=  + | - | * | / | < | > | = | !                       Operator

<integer-literal> ::=  <digit> <digit>*                                 IntegerLiteral
```

- $\mathbf{d_1\ d_2\ d_3 \ldots d_n}$ has a value of $\mathbf{d_1} * 10^{n} + \mathbf{d_2} * 10^{n-1} + \mathbf{d_3} * 10^{n-2} + \ldots + \mathbf{d_n}$
- The type of an integer literal is Integer.

```
<character-literal> ::=  { <graphic> }                                  CharacterLiteral
```

- The value of a character literal is the single graphical character it represents
- The type of a character literal is Char

IDENTIFIER RULES

- Identifiers are case sensitive.
- Reserved words may not be used as identifiers.
- All variables and constants must be declared before being used.
- Only variables can be assigned to.
- Scopes are nested and created through a let command.
- Two identifiers with the same name cannot be declared in the same scope, though an identifier may be declared with the same name as an identifer in a containing scope.
- The let command **let D in C** functions as follows
    - A new scope is created inside the current scope.
    - Each declaration in **D** is processed and new constants and variables created.
    - **C** executes with the variables and constants that were in the containing scope plus those in **D**.
    - All declarations in **D** are local to **C** (and any nested scopes inside **C**).

STANDARD ENVIRONMENT

The following items are defined in the standard environment

Types

- `Integer`
- `Char`
- `Boolean`

Constants

• `true`	– of type `Boolean`
• `false`	– of type `Boolean`

Functions

• `chr(i: Integer) : Char`	– get the character with code i
• `ord(c: Char) : Integer`	– get the code for c
• `eof() : Boolean`	– true if the end of file has been reached, or false otherwise
• `eol() : Boolean`	– true if the end of line has been reached, or false otherwise

Procedures

• `get(var c: Char)`	– get a character from input
• `getint(var i: Integer)`	– get an integer from input
• `put(c: Char)`	– print a character to output
• `putint(I: Integer)`	– print an integer to output
• `puteol()`	– print an end of line character to output