DESCRIPTION OF COURSEWORK SOURCE LANGUAGE

GRAMMAR NOTATION

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

GRAMMAR FOR THE SCANNER

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

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

<character-literal>    ::=  ' <graphic> '

<graphic>              ::=  <letter>
                           |<digit>
                           |<operator>
                           |.
                           |!
                           |?
                           |_
                           | the space character

<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 | z | 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>               ::=  <command>

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

<single-command>        ::=  <identifier> ( := <expression> | ( <parameters> ) )
                            |if <expression> then <single-command>
                             else <single-command>
                            |while <expression> do <single-command>
                            |let <declaration> in <single-command>
                            |begin <command> end
                            |skip

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

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

<parameters>            ::=  <parameter> ( , <parameter> )*

<parameter>             ::=  <expression>
                            |var <identifier>

<type-denoter>          ::=  <identifier>

<expression>            ::=  <secondary-expression> [ ? <expression> : <expression> ]

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

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

ABSTRACT GRAMMAR, SEMANTICS AND TYPE RULES

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

- Executing a program executes the command and exits.

```
<command>      ::=  <command> ; <command>                             SequentialCommand
                   |<identifier> := <expression>                      AssignCommand
                   |<identifier> ( <parameters> )                     CallCommand
                   |if <expression> then <command> else <command>     IfCommand
                   |while <expression> do <command>                   WhileCommand
                   |let <declaration> in <command>                    LetCommand
                   |begin <command> end                               BeginCommand
                   |skip                                              SkipCommand
```

- A sequential command C1; C2 is executed by executin C1 then executing C2.
- 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 parameters P then calling the function or procedure F with these parameters. The type of the parameters must match the parameter types given in the declaration of F.
- 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 do C 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.
- let D in C is executed as follows. The declaration D are elaborated then then C is executed.
- begin C end is executed by simply executing C.
- The skip command does nothing when executed.

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

- Declarations D1; D2 are elaborated by elaborating D1 then elaborating D2.
- The declaration const I ~ E is elaborated 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 elaborated by binding I to a new variable of type T. The variable's value is undefined.
- The declaration var I : T := E is elaborated as follows. First E is evaluated. I is then bound to a new variable of type T. The variable's value is set to the result obtained from the evaluation of E. The type of E must be T.

```
<parameters>   ::=  <parameters> , <parameters>                       ParameterSequence
                   |<expression>                                      ValueParameter
                   |var <identifier>                                  VarParameter
```

- A parameter lists is evaluated by evaluating each parameter in turn, left to right.
- An expression parameter is evaluated by evaluating that expression.
- Expression parameters are passed by value, variable parameters are passed by reference.
- Parameters types must match the types used in the function or procedure declaration.

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

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

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

- Evaluation of an integer literal is simply the value of the integer literal. The type of the expression is Integer.
- Evaluation of a character literal is simply the value of the charcter literal. The type of the expression is Char.
- Evaluation of a variable or constant name gives the current value associated with that variable or constant. 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.
- E1 ? E2 : E3 is evaluated as follows. First E1 is evaluated. If it is true then E2 is evaluated and this becomes the value of the expression, otherwise E3 is evaluated and becomes the values of the expression. E1 must be of type Boolean and E2 and E3 must be of the same type. The type of the expression is the type of E2 and E3.
- 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 = + or –, and E must be of type Integer. Evaluation of +E is equivalent to evaluation E and -E is evaluted by evaluating E then negating its value.
- A bracketed expression (E) is evaluated by simply evaluating E. The type of the expression is the same as E's type.

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

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

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

- $d_1 d_2 d_3 \ldots d_n$ has a value of $d_1 * 10^n + d_2 * 10^{n-1} + d_3 * 10^{n-2} + \ldots + d_n$
- 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).
  - Note in particular that declarations in D may not refer to other declarations in D
    e.g. The following is **not** allowed, as a is not is scope when b is declared

```
let
     const a ~ 1;
     const b ~ a    ! ERROR: a is not in scope yet
in
     ! a and b are in scope now
     skip
```

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