Introduction to the ***Source of Light Programming Language (SOL)***

During the Survey of programming language course Dr. Arthur Hanna taught us how to develop a programming language using the metalanguage BNFi to specify SOL syntax . He advised and guided thoughout the syntax, BNF, documentation and the course concepts. This documentation illustrates Source of Light Programming Language (SOL). *SOL is* a simple, but Turing completeii, programming language. The Single Source file named with the extension .sol is contained the SOL Program. SOL have been developed to support.

- single-line and multi-line (block) comments.
- literals for each of the scalar data types integer, float, character, boolean, and string**
- explicitly defined scalar variables and named constants with scalar data types integer, float, character, and boolean (but not string because *SOL* does not support string as a full-fledged data type)**
- explicitly defined integer, float, character, and boolean n-dimensional **array** variables (and/or 1-dimensional **associate arrays**[iii])**
- **statically-allocated global** scope and program module **local** scope variables and constants**
- an eclectic list of unary and binary operators**
- a multiple-assignment-statement**
- several "standard" **structured flow-of-control** statements (an `IF`-statement, a `FOR`-statement, a `DOWHILE`-statement, but no `GOTO`-statement with the corresponding "icky" statement labels)**
- **assertions****
- `ENTER`-statement and `DISPLAY`-statement for console formatted text ENTER/output**
- **multi-module** programs (**program** module** **procedure**, **function**, and **handler subprogram** modules with **ASK-by-value**, **ASK-by-result**, **ASK-by-value/result**, and **ASK-by-reference parameter passing**)**
- automatically-allocated function, procedure, and handler subprogram module formal parameters and local scope variables and constants in **dynamically-allocated activation records****
- non-recursive and recursive calls of functiand procedure subprogram modules (accomplished with an expression-resident prefix function referencing syntax, a `ASK`-statement, and a `SENDBACK`-statement)**
- a simplistic form of **exception handling** (accomplished with handler subprogram modules and a `RAISE`-statement, an `EXIT`-statement, and a `RESUME`-statement).

Using the **metalanguage BNF**[iv] to specify *SOL* syntax

```
<SOLProgram>            ::= { <dataDefinitions> }*
                            { (( <PROCEDUREDefinition> | <FUNCTIONDefinition> | <HANDLERDefinition> )) }*
                            <PROGRAMDefinition> EOPC

<dataDefinitions>       ::= <variableDefinitions> | <constantDefinitions>

<variableDefinitions>   ::= VAR <identifier> : <datatype> [ [ <LBUBRange> { , <LBUBRange> }* ] ]
                            { , <identifier> : <datatype> [ [ <LBUBRange> { , <LBUBRange> }* ] ] }* .

<LBUBRange>             ::= [ (( + | - )) ] <integer> : [ (( + | - )) ] <integer>

<constantDefinitions>   ::= CON <identifier> : <datatype> := <literal>
                            { , <identifier> : <datatype> := <literal> }* .

<datatype>              ::= (( INT | FLT | BOOL | CHR ))

<PROGRAMDefinition>     ::= PROGRAM
                               { <dataDefinitions> }*
                               { <statement> }*
                            STOP
```

```
<PROCEDUREDefinition> ::= PROCEDURE <identifier> [ ( <formalParameter> { , < formalParameter > }* ) ]

                             { <dataDefinitions> }*

                             { <statement> }*

                          STOP


<FUNCTIONDefinition>  ::= FUNCTION <identifier> : <datatype> ( [ <formalParameter> { , < formalParameter >
}* ] )

                             { <dataDefinitions> }*

                             { <statement> }*

                          STOP


<HANDLERDefinition>   ::= HANDLER <identifier> ( <formalParameter> )

                             { <dataDefinitions> }*

                             { <statement> }*

                          STOP


<formalParameter>     ::= [ (( ENTER | OUTPUT | ENTEROUTPUT | ASSIGN )) ] <identifier> : <datatype> [ [ { ,
}* ] ]


<statement>           ::= { <assertion> }*

                          ((
                               <DISPLAYStatement>

                             | <ENTERStatement>

                             | <assignmentStatement>

                             | <CHECKStatement>
                             | <WHENStatement>
                             | <DOWHILEStatement>
                             | <ASKStatement>
                             | <SENDBACKStatement>
                             | <RAISEStatement>
                             | <EXITStatement>
                             | <RESUMEStatement>
                            ))
                            { <assertion> }*


<assertion>            :: { <expression> }


<DISPLAYStatement>      ::= DISPLAY (( <string> | <expression> | ENDOFLINE ))

                              { , (( <string> | <expression> | ENDOFLINE )) }* .


<ENTERStatement>      ::= ENTER [ <string> ] <variable> .


<assignmentStatement> ::= <variable> { , <variable> }* := <expression> .


<CHECKStatement>         ::= CHECK ( <expression> ) THEN
                               { <statement> }*

                          { ELSECHECK ( <expression> ) THEN
                               { <statement> }* }*
                          [ ELSE
```

```
                                  { <statement> }* ]
                               STOP

<DOWHILEStatement>    ::= DO

                                  { <statement> }*

                               WHILE ( <expression> )

                                  { <statement> }*

                               STOP


<WHENStatement>        ::= WHEN <variable> := <expression> TO <expression> [ BY <expression> ]
                                  { <statement> }*
                               STOP

<ASKStatement>       ::= ASK <identifier> [ ( (( <expression> | <variable> ))

                                              { , (( <expression> | <variable> )) }* ) ] .


<SENDBACKStatement>     ::= SENDBACK [ ( <expression> ) ] .


<RAISEStatement>      ::= RAISE <identifier> ( <expression> ) .


<EXITStatement>       ::= EXIT .


<RESUMEStatement>     ::= RESUME .


<expression>          ::= <conjunction> { (( OR | NOR | XOR )) <conjunction> }*
```

```
<conjunction>          ::= <negation> { (( AND | NAND )) <negation> }*

<negation>             ::= [ NOT ] <comparison>

<comparison>           ::= <comparator> [ (( < | <= | = | > | >= | (( != | <> )) )) <comparator> ]

<comparator>           ::= <term> { (( + | - )) <term> }*

<term>                 ::= <factor> { (( * | / | % )) <factor> }*

<factor>               ::= [ (( ORD  | CHR  | INT   | FLT  |
                                ABS  | UP   | LOW   | PRED |
                                SUCC | ISUP | ISLOW | +    | - )) ]^v <secondary>

<secondary>            ::= <primary> [ (( ^ | ** )) <primary> ]

<primary>              ::= <variable> | ( <expression> ) | <literal>
                         | <identifier> ( [ <expression> { , <expression> }* ] )
                         | <identifier> LB ( <expression> )
                         | <identifier> UB ( <expression> )

<variable>             ::= <identifier> [ [ <expression> { , <expression> }* ] ]

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

```
<literal>              ::= <integer> | <float> | <character> | <boolean> | <string>

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

<float>                ::= <digit> { <digit> }* . <digit> { <digit> }* [ E [ - ] <digit> { <digit> }* ]

<character>            ::= ' <ASCIICharacter> '                        || *Note* escape both \ and ' with \

<boolean>              ::= true | false

<string>               ::= " { <ASCIICharacter> }* "                   || *Note* escape both \ and " with \

<letter>               ::= A | B | ... | Z | a | b | ... | z

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

<ASCIICharacter>       ::= || Every DISPLAYable ASCII character in range [ ' ','~' ]

<comment>              ::= ** { <ASCIICharacter> }* EOLC               ** single-line comment
                         | %^ { (( <ASCIICharacter> | EOLC )) }* ^%    %^ multi-line (block) comment ^%
```

Non- Terminal Symbols:

- <SOL Program>
- <PROGRAMDefinition>
- <statement>
- <CheckStatement>
- <WhenStatement>
- <identifier>
- <string>
- <ASCIICharacter>
- <comment>
- <DOWHILEStatement>
- <DISPLAYStatement>
- <ENTERStatement>
- <AssignmentStatement>
- <ASKStatement>
- <SENDBACKStatement>
- <RAISEStatement>
- <EXITStatement>
- <RESUMEStatement>

Pseudo Terminal symbols:

- <IDENTIFIER>
- <INTEGER>
- <STRING>
- <EOPTOKEN>
- <UNKTOKEN>

Reserved words (Terminal Symbols):

- PROGRAM
- STOP **Stands for** (END In C/ C++)
- DISPLAY **Stands for** (DISPLAY in C/ C++)
- ENDOFLINE
- OR
- NOR
- XOR
- AND
- NAND
- NOT
- TRUE (T)
- FALSE (F)
- VAR
- INTNUM
- BOOL
- CON
- ENTER **Stands for** (ENTER in C/C++)
- CHECK **Stands for** (IF in C/C++))
- THEN
- CHECKIF **Stands for** (Else IF in C/C++))
- ELSE
- DO
- WHILE
- WHEN **Stands for** (For in C/C++))
- TO

- BY
- PROCEDURE
- ENTER
- OUTPUT
- ENTEROUTPUT
- ASSIGN **Stands for** (Assign in C/C++))
- ASK **Stands for** (ASK in C/C++))
- SENDBACK (SENDBACK in C/C++))

Punctuation:

- COMMA
- PERIOD
- OPARENTHESIS
- CPARENTHESIS
- COLON
- COLONEQ
- OBRACE
- CBRACE
- FUNCTION

Operators:

- LT
- LTEQ
- EQ
- GT
- GTEQ

- `NOTEQ, // <> and !=`
- `PLUS`
- `MINUS`
- `MULTIPLY`
- `DIVIDE`
- `MODULUS`
- `ABS`
- `POWER // ^ and **`
- `INC`
- `DEC`

Terminal Symbol:

- `Program`
- `}`
- `//, /<, >/`
- `||, !|, ^|`
- `&, !&`
- `out`
- `NO, !`
- `+`
- `<, <=, ==, >, >=, <>, !=`
- `+, -`
- `*, /, %`
- `abs, +, -`
- `^, **`
- `True, False`

3. Static semantics: Once the *SOL* compiler runs, its checks that the programmer follow both the BNF syntax rules and the static semantic rules. The *SOL* Program cannot be translated into machine code if it has program has any syntax errors.

1) In order to run the program appropriately an identifier which names a variable or constant should be used correctly.
2) Values must be used with operators in the correct ways. For instance, you cannot add a float value and an integer value.

- **Dynamic semantics:** You can use any amount of white space characters (blanks, tabs, and end of lines) among the source code tokens because *SOL* is free-format languages.
- *SOL* is a case-neutral programming language, so it is not a case sensitive programming language.

Note: SOL `<DisplayStatement>`, are terminated by the reserved word `STOP`.

Note: SOL `<DisplayDefinition>`, begin with reserved words `PROGRAM`, and are always terminated by the reserved word `STOP`.

- **`<comment>`**
  - The user can add a single line comments in any source line.
  - A single-line comment extends from the triple solidus ** prefix (used to mark the beginning of the single-line comment) to the end of the line containing the triple **.
  - Single-line comments are represented as white space at the end of the source line in which the comment appears.

```
<comment>              ::= ** { <ASCIICharacter> }* ENDOFLINE              **single-line comment
```

For instance,

```
**--------------------

**Compute i

**--------------------
```

```
ENTER "i? " i.

i := ((a+b)*12 % 1)-5.

DISPLAY "i is ",i, ENDOFLINE.      **Display results
```

- A **multi-line (block) `<comment>`**
  - May appear any place white-space may appear.
  - The block comment extends from the `%` prefix (used to mark the beginning of the block comment) to the `^` suffix (used to mark the ending of the block comment). Block comments may be nested as illustrated in the example shown below.

```
<comment>             ::= %^ { (( <ASCIICharacter> | ENDOFLINE )) }* ^%   ** multi-line (block) comment
```

For instance,

```
%^---------------------
**Compute n

---------------------^%

ENTER "n? " n.

n := ((a+b)*10 % 8)-5.

DISPLAY "n is ",n,ENDOFLINE.      %^ Display

                               Results ^%
```

- A **<SOLProgram>**
  - is an optional collection of 1 or more global data definitions** followed by an optional collection of 0 or more

subprogram module definitions** followed by a required program module definition.
  - Each *SOL* program should consist of at least 1 module, the program module.
  - Once a *SOL* program start execution, the flow-of-control begins with the first statement in the program module's list of statements and flow-of-control continues until it terminates.

```
<SOLProgram>          ::= <PROGRAMDefinition> ENDOFLINE
```

- A **<PROGRAMDefinition>**
  - is used as definition for unnamed program module.
  - Provides an optional data definitions section for local variables and constants.
  - Provides a set of 0 or more executable statements which make up the program module body.

```
<PROGRAMDefinition>   ::= START

                          { <statement> }*

                          STOP
```

`<literal>` constants may be **`<integer>`**, **`<float>`**, **`<character>`**, **`<boolean>`**, or **`<string>`**. An integer literal must be unsigned and must be an element of the range `[ 0,` 2^{15}`-1 = 32767 ]`** a float literal must be unsigned and include at least 1 digit before the decimal point and at least 1 digit after the decimal point** a character literal is delimited with single quotes `'` and may contain any 1 DISPLAYable character** the reserved words `true` and `false` are the only 2 boolean literals** and a string literal is delimited by double quotes `"` and may contain 0 or more DISPLAYable characters (Note Character literals may contain a single quote by coding it using the escape sequence `\'`** string literals may contain a double quote by coding it using the escape sequence `\"`** and character and string literals may contain a backslash character `\` by coding it using the escape sequence `\\`).

```
<literal>              ::= <integer> | <float> | <character> | <boolean> | <string>

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

<float>                ::= <digit> { <digit> }* . <digit> { <digit> }* [ E [ - ] <digit> { <digit> }* ]

<character>            ::= ' <ASCIICharacter> '                       || *Note* escape both \ and ' with \

<boolean>              ::= true | false

<string>               ::= " { <ASCIICharacter> }* "                  || *Note* escape both \ and " with \
```

For example, the `DISPLAY`-statement shown below displays the line `"Howdy", he exclaimed!`

```
DISPLAY "\"Howdy\", he exclaimed!",ENDOFLINE.
```

and the assignment-statement shown below assigns a single quote to the character variable `singleQuote`

```
VAR singleQuote: CHR.
```

```
singleQuote := '\''.
```

- **<identifier>**
  - The Programmer can use <identifier> to name global variables and constants** program module variables and constants** procedure, function, and handler subprogram modules** subprogram module formal parameters** and subprogram module variables and constants.
  - When the identifier it is specified in the header of the module's definition it is called a Subprogram module identifier.
  - A variable can be defined when it is specified in a data definitions section.
  - A formal parameter can be defined when it is specified in a subprogram module formal parameter list.

```
 <identifier>          ::= <letter> { ([ <letter> | <digit> | _ ]) }*
```

A **`<SOLProgram>`** is an optional collection of 1 or more global data definitions** followed by an optional collection of 0 or more subprogram module definitions** followed by a required program module definition: Each *SOL* program must consist of at least 1 module, the **program module**. When a *SOL* program begins execution, **flow-of-control** begins with the first statement in the program module's list of statements and flow-of-control continues until it terminates when the statement which ends the list of program module statements completes execution.

```
<SOLProgram>          ::= { <dataDefinitions> }*
                            { (( <PROCEDUREDefinition> | <FUNCTIONDefinition> | <HANDLERDefinition> )) }*
                            <PROGRAMDefinition> EOPC
```

A **`<dataDefinitions>`** is a possibly empty list of variable and/or constant definitions. A variable is named-with-`<identifier>`, may be a scalar or a multi-dimensional array, has an explicit data type, and may not be given an initial value. A constant is named-with-`<identifier>`, must be scalar, has an explicit data type, and must be given an initial literal value. Every *SOL* variable or constant must be defined before it can be legally referenced.

Variables and constants defined in the `<dataDefinitions>` block which precedes all module definitions have **global** scope. The scope of a module variable or constant identifier (including a subprogram module formal parameter) is **local** scope, a scope which extends from the point of definition to the last statement in the body of the module in which it is defined. It is a static semantic error to reference an undefined identifier or a defined but not-in-scope identifier. It is a static semantic error to multiply-define an identifier in the same scope (global or local)** however, when the scope of an identifier is both global scope and local scope, a reference to the overloaded identifier is a reference to the locally-scoped identifier. (Note There is no *SOL* syntax for referencing the globally-scoped identifier when it collides with a locally-scoped identifier.) A formal parameter identifier is defined when it is specified in the formal parameter list of its subprogram module definition. It is a static semantic error (1) when an array variable index range lower-bound is greater than its corresponding upper-bound** and (2) when the data type of a constant is not the same as the data type of the literal used to define the constant.

```
<dataDefinitions>     ::= <variableDefinitions> | <constantDefinitions>

<variableDefinitions> ::= VAR <identifier> : <datatype> [ [ <LBUBRange> { , <LBUBRange> }* ] ]
                          { , <identifier> : <datatype> [ [ <LBUBRange> { , <LBUBRange> }* ] ] }* .

<LBUBRange>           ::= [ (( + | - )) ] <integer> : [ (( + | - )) ] <integer>

<constantDefinitions> ::= CON <identifier> : <datatype> := <literal>
                          { , <identifier> : <datatype> := <literal> }* .

<datatype>            ::= (( INT | FLT | BOOL | CHR ))
```

A **`<PROGRAMDefinition>`** defines the unnamed program module by providing (1) an optional data definitions section for local variables and constants** and (2) a set of 0 or more executable statements which make up the program module body.

```
<PROGRAMDefinition>   ::= PROGRAM
                             { <dataDefinitions> }*
                             { <statement> }*
                          STOP
```

A **program module body** consists of 0 or more statements which begin execution when the program is run. Flow-of-control begins with the first statement in the program module's body** flow-of-control terminates after the last statement is executed. There is no explicit instruction used to terminate program module execution** instead, flow-of-control terminates when the flow-of-control "runs into" the program module `STOP` reserved word.

A **`<PROCEDUREDefinition>`** defines a globally-scoped, named procedure subprogram module by providing (1) a globally-unique identifier to name the module** (2) an optional list of formal parameters** (3) an optional data definitions section for local variables and constants** and (4) a set of 0 or more executable statements which make up the procedure subprogram module body. A procedure module may be defined with no formal parameters, but then the parentheses used to enclose the formal parameter list must not be specified.

```
<PROCEDUREDefinition> ::= PROCEDURE <identifier> [ ( <formalParameter> { , < formalParameter > }* ) ]
                             { <dataDefinitions> }*
                             { <statement> }*
```

```
                                  STOP
```

A **procedure subprogram module body** consists of 0 or more statements which are executed when the procedure is referenced with a `ASK`-statement (see `<ASKStatement>` description). Flow-of-control begins with the first statement in the procedure body** flow-of-control terminates after the procedure executes a `SENDBACK`-statement. When the procedure terminates, flow-of-control resumes with the statement immediately following the `ASK`-statement which referenced the procedure. When a procedure body does not contain a `SENDBACK`-statement, the flow-of-control terminates when the procedure "runs into" the procedure `STOP` reserved word. See `<formalParameter>` description for parameter-passing details.

A **`<FUNCTIONDefinition>`** defines a globally-scoped, named **pure** function subprogram module by providing (1) a globally-unique identifier to name the module** (2) the function scalar SENDBACK type** (3) a required list of 0 or more formal parameters** (4) an optional data definitions section for local variables and constants** and (5) a set of 0 or more executable statements which make up the function subprogram module body. A function module may be defined with no formal parameters, but the parentheses used to enclose formal parameter list must still be specified. It is a static semantic error when any of the formal parameters is not defined as an `IN` parameter ("pure" means "no side-effects allowed" via `OUT`, `IO`, or `ASSIGN` parameters or using a global variable as an l-value).

```
<FUNCTIONDefinition>  ::= FUNCTION <identifier> : <datatype> ( [ <formalParameter> { , < formalParameter > }* ] )
                              { <dataDefinitions> }*
                              { <statement> }*
                          STOP
```

A **function subprogram module body** consists of 0 or more statements which are executed when the function is referenced in an expression (see "function reference" discussion below). Flow-of-control begins with the first statement in the function's body** flow-of-control terminates when the function executes a `SENDBACK`-statement. When the function terminates, flow-of-control resumes the evaluation of the expression which referenced the function by replacing the reference to the function with the function SENDBACK value. It is a static semantic error when the data type of the `SENDBACK`-statement expression does not match the data type of the function SENDBACK type. It is a run-time exception when a function body does not contain a `SENDBACK`-statement in <u>every</u> possible path of flow-of-control. (<u>Note</u> It is a run-time exception because the flow-of-control analysis to determine that a `SENDBACK`-statement will be executed regardless of which path the function follows to termination is very difficult to do at translation time, but exceedingly easy to do at run-time!)

A **`<HANDLERDefinition>`** defines a globally-scoped, named, single-`IN` parameter, special-purpose subprogram module designed to handle a specific **programmer-defined exception**. The definition provides (1) a globally-unique identifier to name the module** (2) 1 required `IN` formal parameter** (3) an optional data definitions section for local variables and constants** and (4) a set of 0 or more executable statements which make up the handler subprogram module body.

```
<HANDLERDefinition>   ::= HANDLER <identifier> ( <formalParameter> )
                              { <dataDefinitions> }*
                              { <statement> }*
                          STOP
```

A **handler subprogram module body** consists of 0 or more statements which are executed when the handler is referenced with a `RAISE`-statement (see `<RAISEStatement>` description). Flow-of-control begins with the first statement in the handler’s body** flow-of-control terminates when the handler executes a `RESUME`-statement or an `EXIT`-statement. When the handler terminates with a `RESUME`-statement, flow-of-control resumes with the statement which follows the `RAISE`-statement which invoked the handler (see `<RESUMEStatement>` description). When the handler terminates with an `EXIT`-statement, a predefined exception is raised (see `<EXITStatement>` description). It is a run-time exception when a handler module terminates without executing either a `RESUME`-statement or an `EXIT`-statement.

The **function subprogram module reference** syntax (found in `<primary>`) consists of a function subprogram module identifier suffixed by 0 or more comma-separated actual parameter expressions enclosed in parentheses `(...)`. The actual parameter expressions are evaluated and the expression results are used to initialize the corresponding `IN` formal parameters. When parameter passing is completed, flow-of-control is transferred to the first statement of the body of the function being referenced. Execution of a `SENDBACK`-statement causes flow-of-control to SENDBACK to the evaluation of the expression containing the function reference. It is a static semantic error when the actual parameters contained in the function reference do not exactly match in number and data type with the formal parameters specified in the function subprogram module definition.

```
<primary>             ::= <variable> | ( <expression> ) | <literal>
                          | <identifier> ( [ <expression> { , <expression> }* ] )
                          | <identifier> LB ( <expression> )
                          | <identifier> UB ( <expression> )
```

The **procedure subprogram module `<ASKStatement>`** actual parameter expressions corresponding to `IN` formal parameters are evaluated. `OUT` and `IO` formal parameters must be l-values (must be scalar variables or array variable elements) and `ASSIGN` actual parameters must be l-values (must be scalar variables or array variable elements or arrays), so no evaluation is necessary. Each `IN` formal parameter is initialized with the value of the corresponding actual parameter expression specified in the `ASK`-statement** each `IO` formal parameter is initialized with the value of the corresponding actual parameter variable specified in the `ASK`-statement** and each `ASSIGN` formal parameter is initialized with a reference-to (address-of) the corresponding actual parameter variable specified in the `ASK`-statement. When parameter passing is completed, flow-of-control is transferred to the first statement of the procedure being referenced. Execution of a `SENDBACK`-statement causes flow-of-control to SENDBACK to the statement immediately after the `ASK`-statement which made the procedure reference. It is a static semantic error when the `ASK`-statement actual parameters do not exactly match in number and data type with the formal parameters specified in the procedure subprogram module definition.

```
<ASKStatement>       ::= ASK <identifier> [ ( (( <expression> | <variable> ))
                                              { , (( <expression> | <variable> )) }* ) ] .
```

The **procedure module** **`<SENDBACKStatement>`** assigns the value of each `OUTPUT` and `INPUTOUTPUT` formal parameter variable to the corresponding actual parameter variable specified in the `SENDBACK`-statement** flow-of-control is transferred to the statement which immediately follows the `SENDBACK`-statement which referenced the procedure. `IN` formal parameters are not part of the semantics of the procedure `SENDBACK`-statement. It is a static semantic error when a procedure `SENDBACK`-statement attempts to SENDBACK a value.

The expression which is required in a **function module** `<RETURNStatement>` is evaluated** flow-of-control is transferred to the expression which contains the reference to the function. The value of the `SENDBACK`-statement expression is used as the value returned by the reference to the function module. It is a static semantic error when the `SENDBACK`-statement does not contain an expression or when the data type of the expression does not match the data type of the function module SENDBACK type.

```
<RETURNStatement>     ::= SENDBACK [ ( <expression> ) ] .
```

It is a static semantic error when a `SENDBACK`-statement occurs in the program module or a handler subprogram module.

A **`<formalParameter>`** may be `IN` (which specifies **ASK-by-value**), `OUT` (**ASK-by-result**), `IO` (**ASK-by-value/result**), or `ASSIGN` (**ASK-by-reference**). A formal parameter defined without an `IN`, `OUT`, `IO`, or `ASSIGN` specifier is `IN` (ASK-by-value) by default. An array parameter is specified by the syntax `[...]` containing 0 or more commas **`','`** such that 0 commas means a 1-dimensional array, 1 comma means a 2-dimensional array, et cetera. It is a static semantic error when an array formal parameter is not specified as a `ASSIGN` parameter. It is a static semantic error when the number of dimensions of an array variable actual parameter does not match the corresponding formal parameter and when the data type of the array elements do not match.

```
<formalParameter>     ::= [ (( IN | OUT | IO | ASSIGN )) ] <identifier> : <datatype> [ [ { , }* ] ]
```

All function subprogram module formal parameters must be `IN` parameters because all *SOL* functions must be **pure functions** (Note By definition, pure functions SENDBACK a value but do not have any **side-effects**). A function module may legally read-reference global variables and constants (this is, use them as r-values), but it is a static semantic error for a function module to write-reference global variables (that is, use the global variables as an l-value).

Procedure module formal parameters may be `INPUT`, `OUTPUT`, `INPUTOUTPUT`, and/or `ASSIGN` parameters.

A handler module must have exactly 1 `INPUT` parameter.

An **actual parameter** corresponding to an `IN` formal parameter is an r-value which results from evaluation of an expression** an actual parameter corresponding to an `OUT` or an `IO` formal parameter must be scalar variable or array variable element (an l-value)** and a `ASSIGN` formal parameter must be scalar variable or array variable element (an l-value) or array identifier. The data type of each actual parameter must exactly match the data type of its corresponding formal parameter** that is, *SOL* does not support **coercions** (implicit cast operations).

An **`<expression>`** computes a scalar integer, float, character, or boolean value. (Note **Eager evaluation** is intended for all expressions** however, *SOL* has no side-effecting operators, so **short-circuit** (**lazy) evaluation** is also allowed.)

```
<expression>          ::= <conjunction> { (( OR | NOR | XOR )) <conjunction> }*
<conjunction>         ::= <negation> { (( AND | NAND )) <negation> }*
<negation>            ::= [ NOT ] <comparison>
<comparison>          ::= <comparator> [ (( < | <= | = | > | >= | <> )) <comparator> ]
<comparator>          ::= <term> { (( + | - )) <term> }*
<term>                ::= <factor> { (( * | / | % )) <factor> }*

<factor>              ::= [ (( ORD  | CHR  | INT   | FLT  |

                               ABS  | UP   | LOW   | PRED |

                               SUCC | ISUP | ISLOW | +    | - )) ] <secondary>
```

```
<secondary>           ::= <primary> [ (( ^ | ** )) <primary> ]

<primary>             ::= <variable> | ( <expression> ) | <literal>

                        | <identifier> ( [ <expression> { , <expression> }* ] )

                        | <identifier> LB( <expression> )

                        | <identifier> UB( <expression> )
```

SOL **binary arithmetic operators** are addition (lexeme +)** subtraction (-)** multiplication (*)** division (/)** integer remainder (%)** and exponentiation (^ or **, both lexemes are allowed). The **unary** arithmetic operators are **identity** (+)** **negation** (-)** and **absolute value** (ABS, a unary operator, not a function). The evaluation of each binary arithmetic operator always requires 2 integer or 2 float operands and always results in a scalar integer or a scalar float value. The **signatures** of the arithmetic operators shown below are the only combinations of operands and operator allowed. It is a static semantic error when operator operands do not match of the allowed signatures** **mixed-mode** arithmetic expressions are not allowed. To "un-mix" the data type, use appropriate cast operators (see discussion below).

- `  +: integer × integer → integer`
- `  -: integer × integer → integer`
- `  *: integer × integer → integer`
- `  /: integer × integer → integer`
- `  +: float × float → float`
- `  -: float × float → float`
- `  *: float × float → float`
- `  /: float × float → float`
- `  %: integer × integer → integer`
- `  ^: float × integer → float`
- ` **: float × integer → float`
- `  +: integer → integer`
- `  -: integer → integer`
- `ABS: integer → integer`
- `  +: float → float`

- `  -: float → float`
- `ABS: float → float`

SOL **character unary operators** are cast-integer-to-character (`CHR`)** cast-character-to-integer (`ORD`)** cast-float-to-integer (`INT`)** cast-integer-to-float (`FLT`)** convert-character-to-uppercase (`UP`)** convert-character-to-lowercase (`LOW`)** **predicate** is-character-uppercase? (`ISUP`)** predicate is-character-lowercase? (`ISLOW`)** predecessor-character (`PRED`)** and successor-character (`SUCC`). The signatures of the character unary operators are shown below. It is a static semantic error when the `CHR` operand is not an integer** when the `ORD`, `UP`, `LOW`, `ISUP`, `ISLOW`, `PRED`, `SUCC` operand is not a character** when the `INT` operand is not a float** or when the `FLT` operand is not an integer.

- `  CHR: integer → character`
- `  ORD: character → integer`
- `  INT: float → integer`
- `  FLT: integer → float`
- `   UP: character → character`
- `  LOW: character → character`
- ` ISUP: character → boolean`
- `ISLOW: character → boolean`
- ` PRED: character → character`
- ` SUCC: character → character`

Boolean values may be combined using the **boolean binary conjunctive operators** (`AND`, `NAND`)** the boolean binary **disjunctive** operators (`OR`, `NOR`, `XOR`)** and the boolean unary **negation** operator (`NOT`). The evaluation of the `AND`, `NAND`, `OR`, `NOR`, and `XOR` binary boolean operators <u>always requires</u> 2 boolean operands and <u>always</u> results in a scalar boolean value. The evaluation of the `NOT` unary boolean operator <u>always requires</u> 1 boolean operand and <u>always</u> results in a scalar boolean value. The signatures of the boolean operators are shown below. It is a static semantic error when the boolean operator operands are not boolean.

- ` AND: boolean × boolean → boolean`
- `NAND: boolean × boolean → boolean`
- `  OR: boolean × boolean → boolean`
- ` NOR: boolean × boolean → boolean`
- ` XOR: boolean × boolean → boolean`

- `NOT: boolean → boolean`

Integer, float, and character values can be compared to each other using the less than (<), the less than or equal (<=), the equal (=), the greater than (>), the greater than or equal (>=), and the not equal (<> or `!=`) binary **relational** or **comparison** operators. The relational operators always compute in a scalar boolean value. The signatures of the binary comparison operators are shown below. It is a static semantic error when the binary comparison operator does not have 2 same-type operands Notes (1) boolean values cannot be compared** and (2) boolean and integer and character values may not be intermixed in the way which the C/C++ programming languages allow intermixing or the interchangeability of boolean and integer and character values. Note *SOL* is designed to be more **type sensitive** (more **strongly typed**) than C/C++** therefore, the data type of the expressions required in the `CHECK`-statements and `DOWHILE`-statements must be boolean.

- `     <: integer × integer → boolean`
- `    <=: integer × integer → boolean`
- `     >: integer × integer → boolean`
- `    >=: integer × integer → boolean`
- `     =: integer × integer → boolean`
- ` != <>: integer × integer → boolean`
- `     <: float × float → boolean`
- `    <=: float × float → boolean`
- `     >: float × float → boolean`
- `    >=: float × float → boolean`
- `     =: float × float → boolean`
- ` != <>: float × float → boolean`
- `     <: character × character → boolean`
- `    <=: character × character → boolean`
- `     >: character × character → boolean`
- `    >=: character × character → boolean`
- `     =: character × character → boolean`
- ` != <>: character × character → boolean`

SOL is like most other programming languages in that it allows the use of paired parentheses in an expression to change the **order of operator evaluation** from that based solely on the operators' predefined **precedences** and/or **associativities**. For example, the 2 assignment-statements shown below contain expressions with 2 different operator evaluation orders, `- *` and `* -`, respectively.

```
x := (y-z)*54.

x := y-z*54.
```

It is a run-time error when an integer or float **overflow**, a float underflow, or a character range violation occurs during the run-time evaluation of an expression. The *SOL* program terminates with an appropriate error message displayed on the user's terminal.

***SOL* Operators**	**Precedence**	**Associativity**
`LB UB`	`1 = highest`	`Non-associative`
`^ or **`	`2`	`Non-associative`
`ABS +(unary) -(unary) ORD CHR INT FLT UP LOW ISUP ISLOW PRED SUCC`	`3`	`Non-associative (unary operators)`
`* / %`	`4`	`Left-to-right`
`+(binary) -(binary)`	`5`	`Left-to-right`
`< <= = > >= <> or !=`	`6`	`Left-to-right`
`NOT`	`7`	`Left-to-right`
`AND NAND`	`8`	`Left-to-right`
`OR NOR XOR`	`9 = lowest`	`Left-to-right`

SOL uses the following `<primary>` syntax to "compute" array dimension attributes for the **lower-bound `LB(<expression>)`** of the index range for dimension `<expression>` for the array variable named with `<identifier>` and the **upper-bound `UB(<expression>)`** of the index range for dimension `<expression>` for the array variable named `<identifier>`.

```
<primary>             ::= <identifier> LB( <expression> )

                        | <identifier> UB( <expression> )
```

For example, for the array variable definitions shown below, `x1s LB(1)` evaluates to 1, `x1s UB(1)` evaluates to 3, `x2s LB(2)` evaluates to 5, and `flags UB(3)` evaluates to 7.

```
VAR x1s[1:3]: INT.

VAR x2s[1:3,5:11]: INT.

VAR flags[1:3,0:2,3:7]: BOOL.
```

The **`<DISPLAYStatement>`** adds computed value of integer, float, character, boolean expressions, and string literals to the console output buffer using only the minimum number of characters required to format the value being output. Notes (1) The boolean value `true` is output as `"T"` and `false` as `"F"`** (2) output of `ENDOFLINE` causes the output buffer to be displayed as a 1 line on the user's terminal and then empties the output buffer (`ENDOFLINE` is the only way to add the end-of-line character to the output buffer)** and (3) *SOL* does not allow for detailed formatting of value added to the output buffer.

```
<DISPLAYStatement>      ::= DISPLAY (( <string> | <expression> | ENDOFLINE )) { , (( <string> |
<expression> | ENDOFLINE )) }* .
```

Here are several examples of `DISPLAY`-statements which use the integer variable definition of `x`

```
VAR x: INT.
```

`DISPLAY`-Statement	**Output**

`x := +11711.`
`DISPLAY "x is",x,".",ENDOFLINE.`
	`x is11711.`

`DISPLAY`-Statement	**Output**
`x := -5678.` `DISPLAY "x is ",x,".",ENDOFLINE.`	`x is -5678.`
`DISPLAY '\'',(2 <= 2),"' and ",(2 <> 2),'.',ENDOFLINE.`	`'T' and F.`

The **`<ENTERStatement>`** prompts with the string when specified (otherwise prompts with the default prompt `"? "`), ENTERs a 1 carriage SENDBACK-terminated record from the console keyboard, then stores the integer (float, character, or boolean)-equivalent of the characters contained in the line into the integer (float, character, or boolean) variable specified. The user's ENTER must be coded according to the syntax rules for *SOL* integer, float, character, and boolean literals. It is a run-time error when the ENTER characters cannot be converted to the appropriate data value.

```
<ENTERStatement>      ::= ENTER [ <string> ] <variable> .
```

For example, the execution of the `ENTER`-statement shown below

```
ENTER "x? " x.
```

looks like this on the console screen

```
x? □
```

where ☐ represents the **text cursor** whereas the execution of the ENTER-statement shown below

```
ENTER x.
```

looks like this on the console screen

```
? ☐
```

The **`<assignmentStatement>`** has the semantics of a classic assignment-statement** namely, the expression is evaluated yielding a scalar integer, float, character, or boolean value, then the value is assigned to the 1-or-more variable(s) on the left of the `:=` operator. No coercion of right-hand side data type to left-hand side data type is permitted** however, several casting operators are provided (for example, see the unary operators `CHR`, `ORD`, `INT`, and `FLT`). It is a static semantic error when the data type of 1-or-more of the variables does not match the data type of the expression.

```
<assignmentStatement> ::= <variable> { , <variable> }* := <expression> .
```

Here are some simple examples,

```
VAR x: INT,y: INT.

VAR a: FLT,b: FLT.

VAR c: CHR.

VAR flag1: BOOL,flag2: BOOL,flag3: BOOL.
```

```
x := 1.

a,b := 3.142.

flag1,flag2,flag3 := false.

c := TOLOW(CHR(ORD('A'))).
```

The **`<CHECKStatement>`** has the semantics of the classical CHECK-statement** specifically,

1. The `CHECK` expression is evaluated. When the expression is true, the set of statements between the `CHECK` and the first `ELSECHECK` or the `ELSE` or the `STOP` (CHECK no `ELCHECK` or `ELSE` clause are specified) is executed. All the other sets of statements in the `CHECK`-statement are <u>not</u> executed.

2. When the `CHECK` expression is false, the `ELSECHECK` expressions are evaluated from top-to-bottom until a true `ELCHECK` expression is found and its set of statements is executed. The set of statements for bypassed false `ELSECHECK` expressions are <u>not</u> executed and all the remaining sets of statements in the `CHECK`-statement are <u>not</u> executed.

3. When the `CHECK` expression is false and <u>every</u> `ELSECHECK` expression is also false, the set of statements between the `ELSE` and the `STOP` (CHECK the `ELSE` is specified) is executed and <u>all</u> other sets of statements are <u>not</u> executed.

It is a static semantic error when the `CHECK` expression is not boolean data type.

```
<CHECKStatement>          ::= CHECK ( <expression> ) THEN
                                  { <statement> }*

                             { ELSECHECK ( <expression> ) THEN
                                  { <statement> }* }*
                             [ ELSE
                                  { <statement> }* ]
                               STOP
```

The **<DOWHILEStatement>** is a combo form of both of the classic **unbounded loops**** this is, it is can be used as a **pre-test loop** and a **post-test** loop, but in it most general form it is an unbounded **mid-test loop**! Here are its semantics.

```
<DOWHILEStatement>    ::= DO

                                  { <statement> }*

                               WHILE ( <expression> )

                                  { <statement> }*

                               STOP
```

1. the statements between `DO` and `WHILE` are executed unconditionally

2. the expression is evaluated

3. when the expression is false, go to step 6.

4. execute the statements in the body of the loop between the `)` and the `STOP`

5. go to step 1.

6. next-statement

The **`<FORStatement>`** has the semantics of a classical bounded pre-test loop (also called a **counted loop**). (Note The variable specified between the reserved word `FOR` and the `:=` is called the **loop-control variable** and must be explicitly defined.)

```
<FORStatement>        ::= FOR <variable> := <expression> TO <expression> [ BY <expression> ]
                             { <statement> }*
                          STOP
```

1. The three expressions are evaluated only 1 time before the first execution of the loop's body** the expression values are stored in **temporaries** *E1*, *E2* and *E3*, respectively. The first expression value, *E1*, is a lower-bound when the `BY` expression value, *E3*, is positive but is an upper-bound when *E3* is negative. The second expression value, *E2*, is an upper-bound when *E3* is positive but is a lower-bound when *E3* is negative. Note When the `BY` expression syntax is omitted, *E3* always defaults to +1 implying *E1* is treated as a lower-bound and *E2* as an upper-bound. It is a static semantic syntax error when (1) the loop-control variable is not integer data type** or (2) 1 or more of the expressions are not integer expressions. It is a run-time error when *E3* = 0.

2. the loop control variable is initialized with *E1*

3. case (*E3* > 0): when the loop control variable > *E2*, go to step 7

case (*E3* < 0): when the loop control variable < *E2*, go to step 7

4. execute the statements in the body of the loop

5. add *E3* to the loop control variable

6. go to step 3

7. next-statement

Each **`<assertion>`**[vi] contains an `<expression>` which is evaluated when encountered during normal flow-of-control. When the assertion's expression evaluates to true, program execution proceeds normally** however, when the expression evaluates to false, a run-time error occurs.

```
<assertion>            :: { <expression> }
```

The handling of an assertion-failure exception displays `"Run-time error #1 near line #..."` where the line number is that of the source code line which contains the `{` which marks the beginning of the failed assertion in the message. Execution of the program is aborted. It is a static semantic error when an assertion expression does not evaluate to boolean data type. Note Assertions are normally used as **preconditions** and **postconditions** of statements.

The **`<RAISEStatement>`** **raises** a user-defined **exception** by specifying the `<identifier>` of the exception handler subprogram module and specifies an scalar integer, float, character, or boolean exception object which is passed to the handler module as the module's only parameter.

```
<RAISEStatement>       ::= RAISE <identifier> ( <expression> ) .
```

The `RAISE`-statement has semantics like the procedure module `ASK`-statement** that is, the actual parameter expression is evaluated, the handler's `IN` formal parameter is initialized with the value of the actual parameter expression, then flow-of-control is transferred to the first statement of the body of the handler being referenced. It is a static semantic error when the data type of the actual parameter expression does not match the data type of the formal parameter specified in the handler subprogram module definition.

A handler module <u>must</u> terminate execution with a **`<RESUMEStatement>`** or an **`<EXITStatement>`**. When a handler module terminates with a `RESUME`-statement, program flow-of-control resumes with the statement following the `RAISE`-statement which invoked the handler. When a handler module terminates with an `EXIT`-statement, the program aborts execution by raising the predefined exception `"Run-time error #5 near line #X..X"`. It is a static semantic error when either a `RESUME`-statement or an `EXIT`-statement occurs in the body any module except a handler module.

```
<EXITStatement>       ::= EXIT .
```

```
<RESUMEStatement>     ::= RESUME .
```

Some alternative *SOL* structured flow-of-control statements

The very simple **`<CHECKStatement>`** described here has the semantics of a classical CHECK-statement** specifically, the expression is evaluated and when the expression is true, the set of statements between the `CHECK` and the `ELSE` (or `STOP` if no `ELSE` clause is specified) is executed and the set of statements between the `ELSE` and the `STOP` (if it is specified) is not executed** but when the expression is false, the set of statements between the `CHECK` and the `ELSE` (or `STOP` if no `ELSE` clause is specified) is not executed and the set of statements between the `ELSE` and the `STOP` (if it is specified) is executed. It is a static semantic error when the expression is not boolean data type.

```
<CHECKStatement>         ::= CHECK ( <expression> ) THEN
                                { <statement> }*
                             [ ELSE
                                { <statement> }* ]
                             STOP
```

Here is an ugly example—ugliness manifest in the required duplication of the `STOP` reserved word—which fully justifies the syntax of the *SOL* `CHECK`-statement.

```
CHECK ( x < y ) THEN
   x := x+1.
ELSE CHECK ( x > y ) THEN
   DISPLAY x.
ELSE
   y := x+y.
STOP
STOP
```

The classic **`<WHILEStatement>`** has the semantics of a classical unbounded pre-test loop

```
<WHILEStatement>      ::= WHILE ( <expression> ) DO
                             { <statement> }*
                          STOP
```

1. the expression is evaluated (Note It is a static semantic error when the expression is not boolean data type)

2. when the expression is false, go to step 5

3. execute the statements in the body of the loop

4. go to step 1

5. next-statement

Here is a simple example,

```
WHILE ( x < y )
```

```
   DISPLAY x.

   x := x+1.

STOP
```

The classic **`<DOWHILEStatement>`** has the semantics of a classical unbounded pre-test loop

```
<DOWHILEStatement>    ::= DO
                             { <statement> }*
                          WHILE ( <expression> ) .
```

1. execute the statements in the body of the loop

2. the expression is evaluated (Note It is a static semantic error when the expression is not boolean data type)

3. when the expression is true, go to step 1

4. next-statement

Here is a simple example,

```
DO

   DISPLAY x.

   x := x+1.

WHILE ( x < y ).
```

Note The classic pre-test and post-test unbounded loops just illustrated should not be considered wrong, only limited in their capacity to express unbounded looping. The *SOL* `DOWHILE`-statement is admirable because it provides the capacity to express pre-test, mid-test, and post-test unbounded iteration using the syntax and semantics of just 1 statement. See https://en.wikipedia.org/wiki/Control_flow on August 15, 2017 for more detail.

ENDNOTES:

---

[i] Manar Abouhenidi uses the **metalanguage BNF** to describe the **grammar** or **context-free syntax** of the ***Source of Light Programming Language* (*SOL*)**. Here are some hints for understanding and learning to use BNF to describe the context-free syntax of the programming language you will develop this semester.

The *SOL* grammar is the list of **production rules** which, when followed, are used to write syntactically-correct *SOL* **programs**. Each rule has 1 **non-terminal symbol** left-hand side which is separated from the rule's right-hand side by `::=`. The right-hand side of a rule is a sequence of non-terminal and **terminal symbols** and BNF **metasymbols** which "spell out" the expression of the left-hand side.

There is always exactly 1 non-terminal which appears on the left hand side of only 1 rule—often called the **goal symbol** of the language—for example, the *SOL* goal symbol is `<SOLProgram>`.

Every symbol in BNF is a **non-terminal** symbol, a **terminal** symbol, or a **metasymbol**. Every syntactically-correct *SOL* **source program** is a sequence of terminal symbols which "spell out" or express an instance of the *SOL* goal symbol. Note Several of *SOL*'s grammar rules have right-hand sides which contain unbounded iteration (see note 4) in the next paragraph), so there are an infinite number of syntactically-correct *SOL* source programs!

The BNF metasymbology is interpreted in these ways

1) `<x>` means "`x` is a non-terminal symbol"**

2) `::=` means "is defined as"**
3) `|` means "or"**
4) `{x}*` means "`x` repeated 0 or more times"**
5) `{x}n` means "`x` repeated exactly `n` times"**
6) `[ x ]` means "`x` is optional"**
7) an underlined metasymbol like `[` and `]` means "`[` and `]` should be treated as terminal symbols"**
8) `(( x1 | x2 | ... | xn ))` means "chose exactly 1 from the list of alternates `x1`,`x2`,...,`xn`"** and
9) `||` introduces a BNF comment which extends to end-of-line.

[ii] In computability theory, a system of data-manipulation rules (such as a computer's instruction set or a programming language) is said to be **Turing complete** or **computationally universal** if it can be used to simulate any **single-taped Turing machine**. The concept is named after English mathematician Alan Turing. A classic example is the lambda calculus.

A closely related concept is that of Turing equivalence—2 computers *P* and *Q* are called equivalent if *P* can simulate *Q* and *Q* can simulate *P*. Thus, a Turing-complete system is one which can simulate a Turing machine. According to the Church–Turing thesis, which conjectures that the Turing machines are the most powerful computing machines, for every real-world computer there exists a Turing machine which can simulate its computational aspects. Universal Turing machines can simulate any Turing machine and by extension the computational aspects of any possible real-world computer.

To show that something is Turing complete, it is enough to show that it can be used to simulate some Turing complete system. For example, an **imperative language** is Turing complete if it has conditional branching (for example, "if" and "goto" statements, or a "branch if zero" instruction) and the ability to change an arbitrary amount of memory locations (for example, the ability to maintain an arbitrary number of variables). Since this is almost always the case, most (if not all) imperative languages are Turing complete if the limitations of finite memory are ignored. Taken from http://en.wikipedia.org/wiki/Turing_completeness on May 18, 2015.

[iii] An **associative array** (also called map, symbol table, or dictionary) is an abstract data type composed of a collection of (key,value) pairs, such that each key appears just once in the collection. Operations associated with this data type allow: (1) add a pairs to the collection** (2) removal a

---

pair from the collection** (3) modify of the value of an existing pair** and (4) lookup of the value associated with a particular key. A standard solution to the dictionary problem is a hash table. Many programming languages include associative arrays as primitive data types** those languages which do not often provide them in software libraries form. Taken from http://en.wikipedia.org/wiki/Associative_array on May 19, 2015.

[iv] Manar Abouhenidi uses the **metalanguage BNF** to describe the **grammar** or **context-free syntax** of the ***Source of Light Programming Language* (*SOL*)**. Here are some hints for understanding and learning to use BNF to describe the context-free syntax of the programming language you will develop this semester.

The *SOL* grammar is the list of **production rules** which, when followed, are used to write syntactically-correct *SOL* **programs**. Each rule has 1 **non-terminal symbol** left-hand side which is separated from the rule's right-hand side by **::=**. The right-hand side of a rule is a sequence of non-terminal and **terminal symbols** and BNF **metasymbols** which "spell out" the expression of the left-hand side.

There is always exactly 1 non-terminal which appears on the left hand side of only 1 rule—often called the **goal symbol** of the language—for example, the *SOL* goal symbol is `<SOLProgram>`.

Every symbol in BNF is a **non-terminal** symbol, a **terminal** symbol, or a **metasymbol**. Every syntactically-correct *SOL* **source program** is a sequence of terminal symbols which "spell out" or express an instance of the *SOL* goal symbol. Note Several of *SOL*'s grammar rules have right-hand sides which contain unbounded iteration (see note 4) in the next paragraph), so there are an infinite number of syntactically-correct *SOL* source programs!

The BNF metasymbology is interpreted in these ways

10) `<x>` means "`x` is a non-terminal symbol"**
11) **::=** means "is defined as"**
12) `|` means "or"**
13) `{x}*` means "`x` repeated 0 or more times"**

14) `{x}n` means "`x` repeated exactly `n` times"**
15) `[ x ]` means "`x` is optional"**
16) an underlined metasymbol like [and] means "`[` and `]` should be treated as terminal symbols"**
17) `(( x1 | x2 | ... | xn ))` means "chose exactly 1 from the list of alternates `x1`,`x2`,...,`xn`"** and
18) `||` introduces a BNF comment which extends to end-of-line.

[v] Possible additions to `<factor>` are `(( EXP | LN | LOG | SIN | COS | TAN | SEC | CSC | COT ))`

[vi] In computer programming, an assertion is a predicate (a true-or-false expression) placed in a program to indicate that the developer thinks that the predicate is always true at that place. When an assertion evaluates to false at run-time, an assertion failure results (which typically causes execution to abort). Programmers can use assertions to help specify programs and to reason about program correctness. For example, a **precondition**—an assertion placed at the beginning of a section of code—determines the set of states under which the programmer expects the code to execute. A **postcondition**—placed at the end of a section of code—describes the expected state at the end of execution. Generally, then, an assertion is used to verify that an assumption made by the programmer during the implementation of the program remains valid when the program is executed.

Note that assertions are distinct from routine error-handling. Assertions document logically impossible situations and discover programming errors: when the impossible occurs, something fundamental is clearly wrong with the program. This is distinct from error handling: most error conditions are possible, although some may be extremely unlikely to occur. Using assertions as a general-purpose error handling mechanism is unwise: assertions do not allow for recovery from errors** an assertion failure will normally halt the program's execution abruptly. Assertions also do not display a user-friendly error message. Taken from http://en.wikipedia.org/wiki/Assertion_(software_development) on May 18, 2015.