Semantics for Basic SPLAT 1.2

<program> ::= **`program`** <decls> **`begin`** <stmts> **`end`** `;`

<decls> will contain zero or more variable or function declarations. All of the declared items must have unique labels. SPLAT is case-sensitive, and does not allow for function overloading. All of the labels referred to in the body statements <stmts> must correspond to one of the declared items (function or variable) in <decls>.

The state of a SPLAT program is defined by the values stored in the variables declared in <decls>. All variables have an initial value which is determined by their type. For example, all **`Integer`** variables have initial value 0, all **`Booleans`** have initial value **`false`**, and all **`String`** variables have the empty string as their initial values.

When a SPLAT program is executed, the individual statements of <stmts> are executed in sequence, which may change the program state, and may send output to the console window.

<var-decl> ::= <label> **`:`** <type> `;`

A variable may be declared in the context of the program, or in the context of an individual function. A variable declared in the context of the program is accessible to the statements in the body of the program, but not in the functions. Likewise, a variable declared in the context of a function is only accessible within the body of that function, and is not accessible to other functions or the program body.

<func-decl> ::= <label> **`(`** <params> **`)`** **`:`** <ret-type> **`is`** <loc-var-decls> **`begin`** <stmts> **`end`** `;`

Within the context of a function, the labels of the locally declared variables and parameters must all be different from one another, and also cannot be the same as any function label of the program. However, local function variables and parameters do not have to be different from other variables or parameters declared elsewhere.

When a function is called, a new local state is created for the function which stores the values for the function's <params> parameters and <loc-var-decls> variables. At the beginning of the call, the values associated with the calling arguments are passed by value (i.e., copied) into the parameters, and the variables take on the initial values for their respective types. The statements of the body may access or change the values stored in these variables and parameters, call any functions declared in the program (including the same function), and output to the console window. However, when the function has completed execution, the variables and parameters go out of scope, and the local state is discarded.

If the return type <ret-type> is **`void`**, then the function cannot have a statement of the form **`return`** <expr> `;`. Such a function completes execution when a statement of the form **`return ;`** is encountered, or the final statement of the body is executed.

If the return type of the function it non-void, then the function may not contain a simple **`return ;`** statement, but its body statements must be *expr-return-terminated*. This means that the last statement of <stmts> must be of the form:

1. **`return`** <expr> **`;`**
2. **`if`** <expr> **`then`** <stmts> **`else`** <stmts> **`end if;`** *where both <stmts> are expr-return-terminated*

For these functions, the <expr> on all return statements must be of the same type as <ret-type>. When such functions end execution, the evaluated value of <expr> is returned and the original call site takes on this value.

<stmt> ::= <label> **`:=`** <expr> **`;`**

Assignment statements are executed by first evaluating the value of <expr> relative to the current state of the program or function, and then the value of the variable or parameter associated with <label> is updated within the state. The type of <expr> must match that of the declared type of <label>.

<stmt> ::= **`while`** <expr> **`do`** <stmts> **`end while`** **`;`**

| **`if`** <expr> **`then`** <stmts> **`else`** <stmts> **`end if`** **`;`**

| **`if`** <expr> **`then`** <stmts> **`end if`** **`;`**

While loops and if conditional statements in SPLAT behave in the same manner as in other C-based imperative languages. The type of <expr> in all of these cases must be **`Boolean`**.

<stmt> ::= <label> **`(`** <args> **`)`** **`;`**

Function call statements begin with the evaluation of the <args> relative to the current state, and then the execution of the function <label>, where the argument values are passed-by-value (i.e., copied) to the corresponding <params> of the function. For this reason, the number and types of <args> must match those of the declared <params> of the function.

The functions of function call statements also must have a declared <ret-type> of **`void`**. Note that calls to such functions cannot change the current state, since SPLAT is pass-by-value, and does not allow for side-effects.

```
<stmt>      ::=    print <expr>  ;
            |      print_line  ;
```

Expression print statements output a string representation of the current value of <expr> to the console window, with no additional surrounding spaces. Note that a string literal (or other literals) can be used in place of <expr>, to provide for different messages, and extra "padding" spaces in output, though SPLAT string literals cannot contain "escape characters". To skip to the next line of output, the **`print_line;`** statement should be used.

```
<stmt>      ::=    return <expr>  ;
            |      return  ;
```

Return statements can only be used in the body statements of a function definition, and not in the main program body. Rules for their use in functions are given above.

```
<expr>      ::=    ( <expr>  <bin-op>  <expr> )
            |      ( <unary-op>  <expr> )
<bin-op>    ::=    and | or | > | < | == | >= | <= | + | - | * | / | %
<unary-op>  ::=    not | -
```

Binary and unary operation expressions must always be surrounded by parenthesis in basic SPLAT to prevent any ambiguity (and to simplify parser creation!) Type rules for their use in Basic SPLAT are as follows:

Operators	**Legal Argument Type(s)**	**Resulting Type**
`and, or, not`	Boolean	Boolean
`>, <, >=, <=`	Integer	Boolean
`==`	Integer, Boolean, or String (argument types must match)	Boolean
`-, *, /, %`	Integer	Integer
`+`	Integer or String (argument types must match)	Integer or String (same as arguments)

Operator expressions are evaluated by first evaluating their individual expression arguments relative to the current state, and then preforming the operation on the resulting values. The operation result values are computed in the same manner as in other C-based imperative languages, with the following exceptions:

- For **==**, a value-equality check is performed on the two arguments, resulting in a Boolean value
- For **+**, Integer addition is performed on Integer arguments, whereas for String arguments, String concatenation is performed.

```
<expr>        ::=     <label> ( <args> )
```

Function call expressions begin with the evaluation of the <args> relative to the current state, and then the execution of the function <label>, where the argument values are passed-by-value (i.e., copied) to the corresponding <params> of the function. For this reason, the number and types of <args> must match those of the declared <params> of the function.

Functions of function call expressions must have a declared <ret-type> of something other than **`void`**, since they must have a resulting value upon evaluation. Furthermore, the type of the expression is the same as <ret-type>.

Note that calls to functions cannot change the current state, since SPLAT is pass-by-value, and does not allow for side-effects. However, the returned value can be assigned to a variable or parameter as part of a larger assignment statement.