SPL Manual
($Revision: 1.27 $)

Gary T. Leavens
Leavens@ucf.edu

October 17, 2024

1 The language SPL

The Simple Programming Language (SPL) is a block-structured, procedural, imperative programming language used for COP 3402 (Systems Software) at UCF.

This document describes the syntax and semantics of SPL.

2 Syntax

The syntax of SPL is defined in this section. First the lexical syntax of SPL is defined. This is followed by the context-free syntax, which builds on the lexical syntax.

2.1 Lexical Syntax

The lexical syntax of SPL is defined by the lexical grammar shown in Figure 1. Curly brackets ({ and }) are not terminal symbols in this grammar; curly brackets are only used to denote an arbitrary number of repetitions of some nonterminal; for example {⟨letter-or-digit⟩} means 0 or more repetitions of the nonterminal ⟨letter-or-digit⟩. In the grammar, some character classes are described in English, these are described in a Roman font between double quotation marks (" and ").

All of the terminal symbols that are possible productions of ⟨punctuation⟩, ⟨reserved-word⟩, and ⟨rel-ops⟩ represent tokens in the grammar, but those nonterminals themselves are not used in the contex-free grammar in Figure 2.

All characters matched by the nonterminal ⟨ignored⟩ are unused by the context-free grammar, and so should be ignored by the lexer, except for purposes of counting line numbers.

2.2 Context-Free Syntax

The context-free syntax of SPL is defined by the context-free grammar shown in Figure 2. The start symbol of this grammar is ⟨program⟩.

⟨ident⟩ ::= ⟨letter⟩ {⟨letter-or-digit⟩}
⟨letter⟩ ::= `a` | `b` | `...` | `y` | `z` | `A` | `B` | `...` | `Y` | `Z`
⟨number⟩ ::= ⟨digit⟩ {⟨digit⟩}
⟨digit⟩ ::= `0` | `1` | `2` | `3` | `4` | `5` | `6` | `7` | `8` | `9`
⟨letter-or-digit⟩ ::= ⟨letter⟩ | ⟨digit⟩
⟨plus⟩ ::= `+`
⟨minus⟩ ::= `-`
⟨mult⟩ ::= `*`
⟨div⟩ ::= `/`

⟨punctuation⟩ ::= **`.`** | **`;`** | `=` | **`,`** | **`:=`** | `(` | `)`
⟨reserved-word⟩ ::= **`const`** | **`var`** | **`proc`** | **`call`** | **`begin`** | **`end`**
| **`if`** | **`then`** | **`else`** | **`while`** | **`do`** | **`read`** | **`print`** | **`divisible`** | **`by`**
⟨rel-ops⟩ ::= `==` | `!=` | `<` | `<=` | `>` | `>=`

⟨ignored⟩ ::= ⟨blank⟩ | ⟨tab⟩ | ⟨vt⟩ | ⟨formfeed⟩ | ⟨eol⟩ | ⟨comment⟩
⟨blank⟩ ::= "A space character (ASCII 32)"
⟨tab⟩ ::= "A horizontal tab character (ASCII 9)"
⟨vt⟩ ::= "A vertical tab character (ASCII 11)"
⟨formfeed⟩ ::= "A formfeed character (ASCII 12)"
⟨newline⟩ ::= "A newline character (ASCII 10)"
⟨cr⟩ ::= "A carriage return character (ASCII 13)"
⟨eol⟩ ::= ⟨newline⟩ | ⟨cr⟩ ⟨newline⟩
⟨comment⟩ ::= ⟨percent-sign⟩ {⟨non-nl⟩} ⟨newline⟩
⟨percent-sign⟩ ::= `%`
⟨non-nl⟩ ::= "Any character except a newline"

Figure 1: Lexical grammar of SPL. The grammar uses a `terminal font` for terminal symbols and a **`bold terminal font`** for reserved words. Note that all ASCII letters (a-z and A-Z) are included in the production for ⟨letter⟩. Again, curly brackets $\{x\}$ means an arbitrary number of (i.e., 0 or more) repetitions of x.

⟨program⟩ ::= ⟨block⟩ `.`

⟨block⟩ ::= **`begin`** ⟨const-decls⟩ ⟨var-decls⟩ ⟨proc-decls⟩ ⟨stmts⟩ **`end`**

⟨const-decls⟩ ::= {⟨const-decl⟩}
⟨const-decl⟩ ::= **`const`** ⟨const-def-list⟩ `;`
⟨const-def-list⟩ ::= ⟨const-def⟩ | ⟨const-defs⟩ `,` ⟨const-def⟩
⟨const-def⟩ ::= ⟨ident⟩ `=` ⟨number⟩

⟨var-decls⟩ ::= {⟨var-decl⟩}
⟨var-decl⟩ ::= **`var`** ⟨ident-list⟩ `;`
⟨ident-list⟩ ::= ⟨ident⟩ | ⟨idents⟩ `,` ⟨ident⟩

⟨proc-decls⟩ ::= {⟨proc-decl⟩}
⟨proc-decl⟩ ::= **`proc`** ⟨ident⟩ ⟨block⟩ `;`

⟨stmts⟩ ::= ⟨empty⟩ | ⟨stmt-list⟩
⟨empty⟩ ::=
⟨stmt-list⟩ ::= ⟨stmt⟩ | ⟨stmt-list⟩ `;` ⟨stmt⟩
⟨stmt⟩ ::= ⟨assign-stmt⟩ | ⟨call-stmt⟩ | ⟨if-stmt⟩
  | ⟨while-stmt⟩ | ⟨read-stmt⟩ | ⟨print-stmt⟩ | ⟨block-stmt⟩
⟨assign-stmt⟩ ::= ⟨ident⟩ `:=` ⟨expr⟩
⟨call-stmt⟩ ::= **`call`** ⟨ident⟩
⟨if-stmt⟩ ::= **`if`** ⟨condition⟩ **`then`** ⟨stmts⟩ **`else`** ⟨stmts⟩ **`end`**
  | **`if`** ⟨condition⟩ **`then`** ⟨stmts⟩ **`end`**
⟨while-stmt⟩ ::= **`while`** ⟨condition⟩ **`do`** ⟨stmts⟩ **`end`**
⟨read-stmt⟩ ::= **`read`** ⟨ident⟩
⟨print-stmt⟩ ::= **`print`** ⟨expr⟩
⟨block-stmt⟩ ::= ⟨block⟩

⟨condition⟩ ::= ⟨db-condition⟩ | ⟨rel-op-condition⟩
⟨db-condition⟩ ::= **`divisible`** ⟨expr⟩ **`by`** ⟨expr⟩
⟨rel-op-condition⟩ ::= ⟨expr⟩ ⟨rel-op⟩ ⟨expr⟩
⟨rel-op⟩ ::= `==` | `!=` | `<` | `<=` | `>` | `>=`

⟨expr⟩ ::= ⟨term⟩ | ⟨expr⟩ ⟨plus⟩ ⟨term⟩ | ⟨expr⟩ ⟨minus⟩ ⟨term⟩
⟨term⟩ ::= ⟨factor⟩ | ⟨term⟩ ⟨mult⟩ ⟨factor⟩ | ⟨term⟩ ⟨div⟩ ⟨factor⟩
⟨factor⟩ ::= ⟨ident⟩ | ⟨number⟩ | ⟨sign⟩ ⟨factor⟩ | `(` ⟨expr⟩ `)`
⟨sign⟩ ::= ⟨minus⟩ | ⟨plus⟩

Figure 2: Context-free grammar for the concrete syntax of SPL. The grammar uses a `terminal font` for terminal symbols, and a **`bold terminal font`** for reserved words. As in EBNF, curly brackets $\{x\}$ means an arbitrary number of (i.e., 0 or more) repetitions of x. Note that curly braces are not terminal symbols in this grammar.

3 Semantics

This section describes the semantics of SPL.

Nonterminals discussed in this section refer to the nonterminals in the context-free grammar of SPL's concrete syntax, as defined in Figure 2.

In SPL, all constants and variables denote (32 bit) integers.

3.1 Blocks

The execution of a ⟨block⟩ declares the named constants, variables, and procedures found within it, initializes the constants and variables, and then it executes the statements in textual order (i.e., from left to right, top to bottom). For example the block `begin print 1; print 2 end` first prints 1 and then it prints 2. If there are no statements, then the block does nothing; thus, the block `begin end` has no effect when executed.

Any run-time errors encountered cause during execution of a statement cause the entire block's execution to terminate with that error.

3.2 Potential Scopes and Declaration Scopes

A SPL program can contain nested scopes, as a ⟨block⟩ contains blocks for each of the procedures declared within it and statements may themselves be blocks.

A (potential) *scope* is a ⟨block⟩; that is, it is the area of the program's text between the block's **begin** and **end** reserved words.

A *declaration scope* in SPL is an area of program text that extends from just after the first mention of the name being declared in a declaration form (e.g., a ⟨proc-decl⟩) to the end of the surrounding potential scope (i.e., to the end of the surrounding ⟨block⟩). For example, the SPL program in Figure 3 has a nested scope, the ⟨block⟩ that defines the procedure `nested`, and which contains a declaration of `x` that shadows the declaration in the surrounding (top-level) block.

```
begin
  const x = 10;
  proc nested
  begin
    const x = 3;
    print x      % prints 3
  end;
  call nested
end.
```

Figure 3: A SPL program with nested procedure scope and a declaration (of the constant `x` that shadows the declaration at the top level). When run, this program will print the numeral 3 to standard output.

Due to nesting of potential scopes, a declaration of an ⟨ident⟩ in a nested block, which is also declared in a surrounding potential scope, causes a hole in the surrounding identifier's declaration scope; the resulting hole is as big as the declaration scope of the inner declaration. (That is, an ⟨ident⟩ may be declared in a nested potential scope even if it is declared in a surrounding potential scope.) SPL uses static scoping; thus, uses of a name declared again in a nested block refer to the closest textually-surrounding declaration of that name instead of other, shadowed, declarations in surrounding potential scopes. However, it is an error if an ⟨ident⟩ is declared more than once in a potential scope, as either a constant, a variable, or a procedure.

Since a procedure declaration mentions the name of the procedure being declared before the ⟨block⟩ that defines its meaning, recursive calls of procedures are allowed. For example the SPL program shown in Figure 4 has a recursive procedure `countDown`, which contains a legal call statement. However, statements cannot call procedures that have not yet been declared (and thus mutual recursion is not possible).

```
begin
  var arg;
  proc countDown
  begin
    print arg;
    if arg >= 0
    then
      arg := arg - 1;
      call countDown
    end
  end;
  arg := 4;
  call countDown
end.
```

Figure 4: A SPL program with a recursive procedure, `countDown`. When run this procedure would print the numbers 4, 3, 2, 1, and then 0.

3.3 Constant Declarations

The nonterminal ⟨const-decls⟩ can be used to declare zero or more constants.

Each constant declaration has the form **const** {⟨const-defs⟩} **;**. In the list of ⟨const-defs⟩, separated by commas, each ⟨const-def⟩ has the form ⟨ident⟩ = ⟨number⟩. Such a ⟨const-def⟩ defines the name ⟨ident⟩ to be an integer constant that is initialized to the value given by ⟨number⟩.

The scope of such a constant definition is the area of the surrounding block that follows the ⟨const-def⟩.

It is an error for an ⟨ident⟩ in such a ⟨const-def⟩ to be the same as a name that is already declared (as a constant) in the same potential scope. It is also an error for the program to use the a declared constant's ⟨ident⟩ on the left hand side of an assignment statement or in a read statement.

3.4 Variable Declarations

The nonterminal ⟨var-decls⟩ can be used to declare zero or more variables.

Each variable declaration, of the form ⟨ident⟩, declares that ⟨ident⟩ is an integer variable that is initialized to the value 0.

It is an error for an ⟨ident⟩ to be declared as a variable if it has already been declared (as a constant or as a variable) in the same potential scope.

Unlike constants, variable names may appear on the left hand side of an assignment statement or in a read statement.

3.5 Procedure Declarations

The nonterminal ⟨proc-decls⟩ specifies zero or more procedure declarations.

Each procedure declaration, of the form **`proc`** ⟨ident⟩ ⟨block;⟩ declares that ⟨ident⟩ is a procedure that when run executes the ⟨block⟩; that is, it declares and initializes the constants and variables declared in the ⟨block⟩ and declares the block's procedures and then executes the statements in the ⟨block⟩ in textual order. Therefore, a procedure executes as if it were a program. Although a program has no surrounding potential scope, a procedure may use identifiers declared in its surrounding potential scope.

It is an error for an ⟨ident⟩ to be declared as a procedure if it has already been declared as a constant, variable, or procedure in the same potential scope.

Procedure names may not be used on the left hand side of an assignment statement nor may they be used in a read statement.

3.6 Statements

This section describes the semantics of each kind of statement in SPL.

Assignment Statement An assignment statement has the form ⟨ident⟩ `:=` ⟨expr⟩. It evaluates the expression ⟨expr⟩ to obtain an integer value and then it assigns that value to the variable named by ⟨ident⟩. Thus, immediately after the execution of this statement, the value of the variable ⟨ident⟩ is the value that was obtained for ⟨expr⟩.

It is an error if the left hand side ⟨ident⟩ has not been declared as a variable. (Note that the evaluation of the ⟨expr⟩ may produce also runtime errors, and any such errors become errors of the entire statement.)

Call Statement A call of the form **`call`** ⟨ident⟩ executes the ⟨block⟩ declared by the procedure named ⟨ident⟩. (Therefore, it allocates space for the constants and variables declared in that procedure's ⟨block⟩, initializes them, and then executes that ⟨block⟩'s statements in textual order.)

It is an error if the ⟨ident⟩ has not been declared as a procedure.

Since procedures in SPL do not have formal parameters and do not return results, one can only pass arguments to a procedure and return results using variables that are in a surrounding potential scope for that procedure.

Block Statement A block statement has the form ⟨block⟩. It executes as described above in subsection 3.1.

If-Statement (Note that in the concrete syntax there are no parentheses around the condition in an if-statement.)

An if-statement with the following form.

`if` C **`then`** S_1 **`else`** S_2 **`end`**

is executed by first evaluating the condition C. When C evaluates to true, then the statement list S_1 is executed in textual order; otherwise, if C evaluates to false (i.e., if it does not encounter an error), then the statement list S_2 is executed in textual order. If C encounters an error, then the entire statement encounters that error.

An if-statement with the following form.

`if` C **`then`** S_1 **`end`**

is executed by first evaluating the condition C. When C evaluates to true, then the statement list S_1 is executed in textual order, otherwise, if C evaluates to false (i.e., if it does not encounter an error), then the statement does nothing. If C encounters an error, then the entire statement encounters that error.

While Statement A while statement has the form **while** C **do** S **end** and is executed by first evaluating the condition C. If C evaluates to false, then S is not executed and the while statement finishes its execution (and does nothing). When C evaluates to true, then the statement list S is executed in textual order, followed by the execution of **while** C **do** S **end** again. Note that C is evaluated each time, not just once.

(Again, in the concrete syntax there are no parentheses around the condition.)

Read Statement A read statement of the form **read** x, where x is a declared variable identifier, reads a single ASCII character from standard input and puts its ASCII value into the variable x. The value of x will be set to -1 if an end-of-file or an error is encountered on standard input.

It is an error if x has not been previously declared as a variable.

Print Statement A print statement of the form **print** e, first evaluates the expression e, and then prints the decimal form of that (integer) value to standard output (using ASCII characters). (This is the same output as would occur for the C statement `printf("\%d", e);`, assuming that the variable `e` was an **int** variable in a C program that held the value of the expression e.)

3.7 Conditions

A ⟨condition⟩ is an expression that has a Boolean value: either true or false.

Divisible-By Condition A ⟨condition⟩ of the form **divisible** e_1 **by** e_2 first evaluates the expression e_1, then it evaluates the expression e_2. (If either evaluation encounters an error, then the condition as a whole encounters that error.) Both values must be integers; furthermore, the value of e_2 must not be 0, otherwise an error occurs. If there is no error and the value of e_1 is evenly divisible by the value of e_2 (i.e., the remainder is 0), then the value of the condition is true, otherwise the value of the condition is false. If the value of e_1 is not evenly divisible by the value of e_2 (i.e., if the remainder is not 0), then the value of the condition is false.

Relational Conditions A ⟨condition⟩ of the form $e_1\ r\ e_2$ first evaluates e_1 and then e_2, obtaining integer values v_1 and v_2, respectively. (If either evaluation encounters an error, then the condition as a whole encounters that error.) Then it compares v_1 to v_2 according to the relational operator r, as follows:

- if r is ==, then the condition's value is true when v_1 is equal to v_2, and false otherwise.
- if r is !=, then the condition's value is true when v_1 is not equal to v_2, and false when they are equal.
- if r is <, then the condition's value is true when v_1 is strictly less than v_2, and false otherwise.
- if r is <=, then the condition's value is true when v_1 is less than or equal to v_2, and false when $v_1 > v_2$.
- if r is >, then the condition's value is true when v_1 is strictly greater than v_2, and false otherwise.
- if r is >=, then the condition's value is true when v_1 is greater than or equal to v_2, and false when $v_1 < v_2$.

3.8 Expressions

A binary operator $\langle\text{expr}\rangle$ of the form $e_1\ o\ e_2$ first evaluates e_1 and then e_2, obtaining integer values v_1 and v_2, respectively. (If either evaluation encounters an error, then the expression as a whole encounters that error.) Then it combines v_1 and v_2 according to the operator o, as follows:

- An expression of the form e_1+e_2 yields the value of $v_1 + v_2$, according to the semantics of the type **`int`** in C.
- An expression of the form e_1-e_2 yields the value of $v_1 - v_2$, according to the semantics of the type **`int`** in C.
- An expression of the form e_1*e_2 yields the value of $v_1 \times v_2$, according to the semantics of the type **`int`** in C.
- An expression of the form e_1/e_2 yields the value of v_1/v_2, according to the semantics of the type **`int`** in C. However, the expression is in error if v_2 is zero.

There are also a few other cases of expressions that do not involve binary operators. These have the following semantics:

- An identifier expression, of the form x, has as its value the (current) value of the integer stored in the constant or variable named x whose declaration is found in the closest syntactically surrounding scope.

  It is an error if x has not been previously declared as a constant or variable.
- An expression of the form n, where n is a $\langle\text{number}\rangle$ yields the value of the base 10 literal n.
- An expression of the form $-e$, first evaluates e. If e does not encounter an error and has value v, then the value of the expression is the negated value of e (i.e., $-v$) according to the semantics of the type **`int`** in C.
- An expression of the form +e, has the value of the expression e, and if evaluation of e encounters an error, then so does +e.
- An expression of the form `(`e`)` yields the value of the expression e, and if evaluation of e encounters an error, then so does `(`e`)`.