The SPL Programming Language Reference Manual

Leonidas Fegaras
University of Texas at Arlington
Arlington, TX 76019
fegaras@cse.uta.edu

February 27, 2018

1 Introduction

The SPL language is a Small Programming Language that supports nested functions, tuples records, and arrays. This reference manual gives an informal specification for the language. Fragments of EBNF syntax are introduced at relevant points in the text; the complete grammar is given in Section 9. The SPL syntax and description are taken from the PCAT programming language by Andrew Tolmach and Jingke Li at Portland State University.

2 Lexical Specification

SPLs character set is the standard 7-bit ASCII set. SPL is case sensitive; upper and lower-case letters are not considered equivalent.

Whitespace (blank, tab, or newline characters) serves to separate tokens; otherwise it is ignored. Whitespace is needed between two adjacent keywords or identifiers, or between a keyword or identifier and a number. However, no whitespace is required between a number and a keyword, since this causes no ambiguity. Delimiters and operators do not need whitespace to separate them from their neighbors. Whitespace may not appear in any token except a string.

SPL comments are enclosed between /* and */, but they cannot be nested. Any character is legal in a comment. The first occurrence of the sequence of characters */ will terminate the comment. Comments may appear anywhere a token may appear and they are self-delimiting, i.e., they do not need to be separated from their surroundings by whitespace.

2.1 Tokens

Tokens consist of keywords, literal constants, identifiers, operators, and delimiters. The following are reserved keywords; they must be written in lower case.

```
array boolean by def else exit false float for if int loop
not print read return string to true type var while
```

A literal constant is an integer, a float, a string, or the values true or false. Integers contain only digits; they must be in the range 0 to $2^{31}1$. Floats consist of one or more digits, followed by a decimal point, followed by zero or more digits. There is no specific range constraint on reals, but the literal as a whole is limited to 255 characters in length. Note that no numeric literal can be negative, since there is no provision for a minus sign.

Strings begin and end with a double quote " and contain any sequence of printable ASCII characters (i.e., having decimal character codes in the range 32 - 126), except double quote. Note in particular that strings may not contain tabs or newlines. String literals are limited to 255 characters in length, not including the delimiting double quotes.

Identifiers are strings of letters and digits starting with a letter, excluding the reserved keywords. Identifiers are limited to 255 characters in length.

The following are the operators:

```
= + - / % == < <= > >= || &&
```

and the delimiters:

```
: ; , . ( ) { } [ ] #
```

3 Programs

A program is the unit of compilation for SPL. Programs have the following syntax:

```
program <- block
```

where block is:

```
block   <- { def ';' } stmt ';' { stmt ';' }
```

that is, it consists of a number of definitions `def` (described in Section 4), followed by at least one statement `stmt`.

A program is executed by first elaborating its definition sequence, then executing its statement sequence, and then terminating. Each file read by the compiler must consist of exactly one program. There is no facility for linking multiple programs or for separate compilation of parts of a program.

4 Definitions

All identifiers occurring in a program must be introduced by a definition. Definitions serve to specify whether the identifier represents a type, a variable, or a function definition:

```
def     <- 'var' ID [ ':' type ] '=' expr
         | 'type' ID '=' type
         | 'def' ID '(' [ ID ':' type { ',' ID ':' type } ] ')' [ ':' type ] '{' block '}'
```

Definitions may be global to the program or local to a particular procedure. The scope of a definition extends roughly from the point of definition to the end of the enclosing procedure (for local definitions) or the end of the program (for global definitions). The detailed scope rules differ for each kind of definition. A local definition of an identifier hides any outer definitions and makes them inaccessible in the inner scope. No identifier may be defined twice in the same procedure or at the global level.

Every value must have an initial value, given by expression. The type name can be omitted whenever the type can be deduced from the initial value. Procedures encompass both proper procedures, which are activated by the execution of a procedure call statement and do not return a value, and function procedures, which are activated by the evaluation of a procedure call expression and return a value which becomes the value of the call expression. Proper procedure definitions are distinguished by the lack of a return type. A procedure may have zero or more formal parameters, whose names and types are specified in the procedure definition, and whose actual values are specified when the procedure is activated. The scope of formal parameters is the body of the procedure (including its local definitions). Parameters are always passed by value. A procedure body is executed by first elaborating its definition sequence, then executing its statement sequence, and finally returning to the calling procedure. There is an implicit `return` statement at the bottom of every procedure body. Each set of procedures declared following a single `def` keyword is treated as (potentially) recursive (but cannot be mutual recursive).

For example:

```
var n = 0;                  /* the integer n is initially 0 */
def fact (i: int): int {    /* function fact is from int to int */
  if (i <= 0) return 1
     else return i*fact(i-1);
};
```

5 Types

SPL is a strongly-typed language; every expression has a unique type, and types must match at assignments, calls, etc. The only allowed implicit coercion between types is from an integer to a float.

```
type    <- 'int' | 'float' | 'string' | 'boolean'
         | ID
         | 'array' '[' type ']'
         | '{' ID ':' type { ',' ID ':' type } '}'
         | '(' [ type { ',' type } ] ')'
```

where a named type `ID` refers to a defined typed by a `type` definition.

5.1 Built-in Types

There are four built-in basic types: int, float, string, and boolean. Integer literal constants all have type `int`, floating point literal constants all have type `float`, string constants have type `string`, and the built-in values `true` and `false` have type `boolean`. `int` and `float` collectively form the numeric types. An integer value will always be implicitly coerced to a float value if necessary. The boolean type has no relation to the numeric types, and a boolean value cannot be converted to or from a numeric value.

5.2 Array Types

An array is a structure consisting of zero or more elements of the same element type. An `array[t]` type represents an array of elements of type `t`. The elements of an array can be accessed by dereferencing using an index, which ranges from 0 to the length of the array minus 1. The length of an array is not fixed by its type, but is determined when the array is created at runtime. It is a checked runtime error to dereference outside the bounds of an array.

5.3 Record Types

The `{ }` bracket syntax defines a record. A record is a structure consisting of a fixed number of named components of (possibly) different types. The record type definition specifies the name and type of each component. Component names are used to initialize and dereference components; the components for each record type form a separate namespace, so different record types may reuse the same component names. The built-in value `()` belongs to every record type. It is a checked runtime error to dereference a component from the `()` record.

5.4 Tuple Types

The parentheses syntax `(...)` defines a tuple. A tuple is a structure consisting of a fixed number of components of (possibly) different types. The tuple type definition specifies the type of each component. Tuple components can be accessed by position number using tuple projection (the `#` syntax). Note that the syntax `(...)` must have at least 2 types to be a tuple. The type `()` is the 'void' type. It has only one member: the `()` value (the null value). The syntax `(type)` is simply equivalent to `type`.

5.5 Constructed Type Values

Arrays, records, and tuples are always manipulated by value, so a value of array or record type is really a pointer to a heap object containing the array or record, though this pointer cannot be directly manipulated by the programmer. Thus, a record type that appears to contain other record types as components actually contains pointers to these types. In particular, a record type may contain (a pointer to) itself as a component, i.e., it may be recursive.

Tuples, records, and arrays have unlimited lifetimes; the heap object containing a record or array exists from the moment when its defining expression is evaluated until the end of the program. In principle, a garbage collector could be used to remove heap objects when no more pointers to them exist, but this would be invisible to the SPL programmer.

For example:

```
type tree = { left: tree, info: int, right: tree };
type env = array[(String,int)]
```

define a type for binary trees (a recursive type) using a record, and an array of pairs, where a pair is a tuple `(String,int)`.

6 L-values

An l-value is a location whose value can be either read or assigned. Variables, procedure parameters, record and tuple components, and array elements are all l-values.

```
lvalue  <- ID
         | lvalue '[' expr ']'
         | lvalue '.' ID
         | lvalue '#' INTEGER
```

The square brackets notation `[]` denotes array element dereferencing; the expression within the brackets must evaluate to an integer expression within the bounds of the array. The dot notation `.` denotes record component dereferencing; the identifier after the dot must be a component name within the record. The sharp notation `#` denotes tuple projection, where `#0#` accesses the first tuple element.

7 Expressions

Syntax of an expression:

```
expr    <- INTEGER | FLOAT | STRING | 'true' | 'false'
         | lvalue
         | unaryOp expr
         | expr binaryOp expr
         | 'array' '(' expr ',' expr ')'
         | ID '(' [ expr { ',' expr } ] ')'
         | '{' ID '=' expr { ',' ID '=' expr } '}'
         | '(' [ expr { ',' expr } ] ')'
         | '[' expr { ',' expr } ']'

unaryOp <- '-' | 'not'

binaryOp <- '+' | '-' | '*' | '/' | '%' | '&&' | '||'
         |  '==' | '<>' | '<' | '<=' | '>' | '>='
```

A number expression evaluates to the literal value specified. Note that floating point are distinguished from integers by lexical criteria. An l-value expression evaluates to the current contents of the specified location. Parentheses can be used to alter precedence in the usual way.

The operators `+`, `-`, and `*` require integer or real arguments. If both arguments are integers, an integer operation is performed and the integer result is returned; otherwise, any integer arguments are coerced to reals, a real operation is performed, and the real result is returned. The operator `/` requires integer or real arguments, coerces any integer arguments to reals, performs a real division, and always returns a real result. The operators `%` (integer remainder) take integer arguments and return an integer result. All the binary operators evaluate their left argument first.

The boolean operators `&&`, `||`, and `not` require boolean operands and return a boolean result. The `&&` and `||` are short-circuit operators; they do not evaluate the right-hand operand if the result is determined by the left-hand one.

The relational operators (eg, `<=`) return a boolean result. These operators all work on numeric arguments; if both arguments are integer, an integer comparison is made; otherwise, any integer argument is coerced to real and a real comparison is made. The operators `=` and `<>` also work on pairs of boolean arguments, or pairs of record or array arguments of the same type; for the latter, they test pointer equality

(that is, whether two records or arrays are the same instance, not whether they have the same contents). These operators all evaluate their left argument first.

The function call is evaluated by evaluating the argument expressions left-to-right to obtain actual parameter values, and then executing the function specified by ID with its formal parameters bound to the actual parameter values. The function returns by executing an explicit `return` statement (with an expression for the value to be returned). The returned value becomes the value of the function call expression.

The record construction (the `{...}` syntax) evaluates each expression left-to-right, and then creates a new record instance of type typename with named components initialized to the resulting values. The names and types of the component initializers must match those of the named type, in the same order. The tuple construction (the `(...)` syntax) works in a similar way. The array construction (the `[...]` syntax) constructs an array consisting of the elements listed inside the brackets.

Precedence and associativity: procedure call and parenthesization have the highest (most binding) precedence; followed by unary `-`; followed by `*`, `/`, and `%`; followed by `+` and `-`; followed by the relational operators; followed by `not`; followed by `&&`; followed by `||`. The binary arithmetic and logical operators are all left-associative. The relational operators are non-associative; in other words, an expression such as `a == b == c` is illegal, although one such as `(a == b) == c` is legal (presuming c has type `boolean`).

8 Statements

Statements have the following syntax:

```
stmt    <- lvalue '=' expr
         | ID '(' [ expr { ',' expr } ] ')'
         | '{' block '}'
         | 'read' '(' lvalue { ',' lvalue } ')'
         | 'print' '(' [ expr { ',' expr } ] ')'
         | 'if' '(' expr ')' stmt [ 'else' stmt ]
         | 'for' '(' ID '=' expr 'to' expr [ 'by' expr ] ')' stmt
         | 'while' '(' expr ')' stmt
         | 'loop' stmt
         | 'exit'
         | 'return' [ expr ]
```

In the assignment statement `lvalue '=' expr`, the l-value is evaluated to a location; then the expression is evaluated and its value is stored in the location.

A procedure call statement is executed by evaluating the argument expressions left-to-right to obtain actual parameter values, and then executing the proper procedure specified by ID with its formal parameters bound to the actual parameter values. The procedure returns when its final statement or an explicit `return` statement (with no expression) is executed.

A read statement is executed by evaluating the l-values to locations in left-to-right order, and then reading numeric literals from standard input, evaluating them, and assigning the resulting values into the locations. The l-values must have type integer or real, and their types guide the evaluation of the corresponding literals. Input literals are delimited by whitespace, and the last one must be followed by a carriage return.

Executing a write statement evaluates the specified expressions (which must be simple integers, reals, booleans, or string literals) in left-to-right order, and then writes the resulting values to standard output (with no separation between values), followed by a newline.

An if-then-else specifies the conditional execution of guarded statements. The expression preceding a statement sequence, which must evaluate to a boolean, is called its guard. The guards are evaluated in left-to-right order, until one evaluates to `true`, after which its associated statement sequence is executed. If no guard is satisfied, the statement sequence following the `else` (if any) is executed.

The body of a while statement is repeatedly executed as long as the expression evaluates to `true`, or until the execution of an `exit` statement within the sequence.

The body of a loop statement is repeatedly executed. The only way to terminate the iteration is by executing an `exit` statement within the sequence.

Executing the statement `for ( id = exp1 to exp2 by exp3)` stmt is equivalent to the following steps: (i) evaluate expressions `exp1`, `exp2`, and `exp3` in that order to values `v1`, `v2`, `v3` (which must be integers); (ii) if the value of `id` is less than or equal to `v2`, execute stmt; otherwise terminate the loop. (iii) set `id = id + v3` and repeat step (ii). If the `by` clause is omitted, `v3` is taken to be 1. `id` is an ordinary integer variable whose scope is the for-loop body. If an `exit` statement is executed within the body of the loop, the loop is prematurely terminated, and control passes to the statement following the `for`.

Executing `exit` causes control to pass immediately to the next statement following the nearest enclosing loop statement. If there is no such enclosing statement, the `exit` is illegal.

Executing `return` terminates execution of the current procedure and returns control to the calling context. There can be multiple `return`s within one procedure body, and there is an implicit `return` at the bottom of every procedure. A `return` from a function procedure must specify a return value expression of the return type; a `return` from a proper procedure must not. The main program body must not include a `return`.

9 Complete Concrete Syntax

```
program <- block

block   <- { def ';' } stmt ';' { stmt ';' }

def     <- 'var' ID [ ':' type ] '=' expr
         | 'type' ID '=' type
         | 'def' ID '(' [ ID ':' type { ',' ID ':' type } ] ')' [ ':' type ] '{' block '}'

type    <- 'int' | 'float' | 'string' | 'boolean'
         | ID
         | 'array' '[' type ']'
         | '{' ID ':' type { ',' ID ':' type } '}'
         | '(' [ type { ',' type } ] ')'

lvalue  <- ID
         | lvalue '[' expr ']'
         | lvalue '.' ID
         | lvalue '#' INTEGER

expr    <- INTEGER | FLOAT | STRING | 'true' | 'false'
         | lvalue
         | unaryOp expr
         | expr binaryOp expr
         | 'array' '(' expr ',' expr ')'
         | ID '(' [ expr { ',' expr } ] ')'
         | '{' ID '=' expr { ',' ID '=' expr } '}'
         | '(' [ expr { ',' expr } ] ')'
         | '[' expr { ',' expr } ']'

unaryOp <- '-' | 'not'

binaryOp <- '+' | '-' | '*' | '/' | '%' | '&&' | '||'
         |  '==' | '<>' | '<' | '<=' | '>' | '>='

stmt    <- lvalue '=' expr
         | ID '(' [ expr { ',' expr } ] ')'
         | '{' block '}'
         | 'read' '(' lvalue { ',' lvalue } ')'
         | 'print' '(' [ expr { ',' expr } ] ')'
         | 'if' '(' expr ')' stmt [ 'else' stmt ]
         | 'for' '(' ID '=' expr 'to' expr [ 'by' expr ] ')' stmt
         | 'while' '(' expr ')' stmt
         | 'loop' stmt
         | 'exit'
         | 'return' [ expr ]
```