SLIC

A Simple Language for an Instructional Compiler

Bryan Crawley

This document provides informal specifications for SLIC, the source language for our compiler project. Design of SLIC is an ongoing process. We may make modifications and additions. I will notify you of any changes that affect the implementation of your compiler. Because these are informal specifications, they may contain ambiguities that require clarification in class.

There are two sample programs attached. They illustrate many of the rules of SLIC syntax and semantics.

SLIC is an imperative language with strong similarities to Pascal, C, Basic, and many other common high-level languages. It is designed to maximize the ease of compilation. It is not intended to be a language for production programming.

SLIC documentation v.1.3
September 11, 2018

General Rules of Syntax

- A program will consist only of a main program for now. We will add subroutines later in the semester if time permits. The main program begins with:

  main;

  and ends with:

  end main;

- Each statement ends with a semicolon. Notice that **main** and **end main** are considered statements. You may put as many statements on a line as you wish, or you may begin a new line anywhere there is a space or a punctuation mark. Wherever there is a punctuation mark, there can be spaces surrounding it. Wherever there is one space, there can be multiple spaces.

- Reserved words are not case-sensitive. They can be written in either upper case or lower case or any mixture of upper and lower case. Generally, the keywords of the statements—like **main**, **end**, **if**, **while**, and so on—are reserved.

- Variable names and other identifiers defined by the programmer are case-sensitive.

- A routine is divided into two sections, the **data** section for declaring variables and the **algorithm** section for the body of the routine. Each section is identified by the corresponding reserved word followed by a colon. See the sample programs. The data section comes first, then the algorithm section.

  main;
  data:
  ...declarations...
  algorithm:
  ...body...
  end main;

- A comment begins with the pound sign (#) and extends rightward to the end of the line.

Terminating Execution

- The **exit** statement terminates execution. The syntax is:

  exit;

- An **exit** statement can appear anywhere in the algorithm section of the program. A program can contain any number of **exit** statements.

- A program will terminate execution if it "falls through the bottom" of the **main** routine by reaching **end main** without having executed an **exit** statement.

Variables, Data Types, and Declarations

- There are two data types called **integer** and **real**. The syntax for constants is the same as in Java, except that for **real** constants, when the decimal point is present, there must be at least one digit on each side of the decimal point.

- Variable names and all other identifiers are alphanumeric and begin with a letter. They are case-sensitive. All variables, both scalars and arrays, must be declared explicitly.

- A variable declaration statement is a data type name followed by a colon and a comma-delimited list of variable names. For example:

  real: X, area, Length, ROOT;

- One-dimensional arrays are permitted. An array is declared with a variable name followed by a bracketed non-negative integer constant. The integer indicates the number of elements in the array. Array indices are integers, beginning with zero as in C/C++. This declaration, for example, creates an array of 20 integers, indexed 0 through 19:

  integer: LIST[20];

- One declaration statement may declare any number of scalar and array variables. The data section of a routine may contain any number of declaration statements.

- There is no Boolean data type. Boolean operations are handled in a manner similar to C. Any numeric value may be used in a Boolean context. A zero value is a Boolean false; a non-zero value is a Boolean true. The relational and Boolean operators produce an integer zero to represent false and an integer one to represent true.

Expressions

- Expressions are in infix notation and may contain integer and real constants, integer and real variables, operators, and parentheses. Variables may be either scalars or array references. An array reference consists of the array name followed by a bracketed subscript. The subscript must be an integer expression whose value is within the range of subscripts allowed for the array, according to its declaration. A non-integer subscript is coerced to be an integer.

- The four standard arithmetic operators are provided: addition (+), subtraction (-), multiplication (*), and division (/). Both binary and unary minus are available.

- The division (/) operator with two integer operands produces an integer result. Otherwise, division produces a real result. Similarly for the other arithmetic operators—if both operands are integers, then the result is an integer. Otherwise, the result is real.

- The modulus (%) operator divides the left-hand operand by the right-hand operand and returns the remainder. It uses two integer operands and produces an integer result. A non-integer operand is coerced to be an integer before the operation is performed.

- The six standard relational operators are provided: less than (<), less than or equal to (<=), greater than (>), greater than or equal to (>=), equal to (=), and not equal to (<>). The result of these operations is an integer zero to represent a Boolean false and an integer one to represent a Boolean true.

- The three standard Boolean operators are provided: “and” (&), “or” (|), and “not” (~). Any numeric value may be used as a Boolean operand. A zero value is a Boolean *false*; a non-zero value is a Boolean *true*. The result of a Boolean operation is an integer zero to represent a Boolean *false* and an integer one to represent a Boolean *true*.

- The priority of operators is listed below. The highest priority is at the top of the list. Parentheses are available to override the priority.

    1. unary minus (-)
    2. *, /, %
    3. +, -
    4. <, <=, >, >=, =, <>
    5. &, |, ~

Assignment Statements

- The syntax of the assignment statement is:

  varref := *expression;*

- A *varref* is any valid variable reference, including both scalars and array references. An array reference consists of the array name followed by a bracketed index. The index must be an integer expression whose value is within the range of indices allowed for the array, according to its declaration.

- The *varref* is the target of the assignment. The *expression* is the source of the value that is assigned.

- Mixed-mode assignments are permitted. That is, an integer value can be assigned to a real variable, and a real value can be assigned to an integer variable. Each case will coerce the value to the date type of the variable to which it is assigned. When a real value is assigned to an integer variable, its fractional part is truncated.

Repetition Control Structures

- The syntax for conditional loops is:

  while *expression***;**
  ...body...
  end while;

- The while statement is controlled by a Boolean expression. Notice that **while** and **end while** are both considered statements and must be terminated with semicolons.

- The *expression* is evaluated before each loop iteration. If it is a Boolean *true*, the body of the loop is performed. If it is a Boolean *false*, the loop terminates. Any arithmetic expression may be used in this context. A zero value is a Boolean *false*; a non-zero value is a Boolean *true*.

- The syntax for counting loops that count upward is:

  counting *variable* **upward** *startexpr* **to** *endexpr***;**
  ...body...
  end counting;

- Notice that **counting/upward** and **end counting** are both considered statements and must be terminated with semicolons.

- The *variable* is a scalar integer variable that controls the loop. The expressions *startexpr* and *endexpr* are integer expressions that provide the starting and ending values of the counting range. The *variable* is initialized to the value of *startexpr* before the loop begins. Before each loop iteration, the *variable* is compared to *endexpr*. If it is less than or equal to *endexpr*, the body of the loop is performed. If it is greater than *endexpr*, the loop terminates. The *variable* is incremented by 1 after each loop iteration.

- The syntax for counting loops that count downward is:

  counting *variable* **downward** *startexpr* **to** *endexpr***;**
  ...body...
  end counting;

- Notice that **counting/downward** and **end counting** are both considered statements and must be terminated with semicolons.

- The *variable* is a scalar integer variable that controls the loop. The expressions *startexpr* and *endexpr* are integer expressions that provide the starting and ending values of the counting range. The *variable* is initialized to the value of *startexpr* before the loop begins. Before each loop iteration, the *variable* is compared to *endexpr*. If it is greater than or equal to *endexpr*, the body of the loop is performed. If it is less than *endexpr*, the loop terminates. The *variable* is decremented by 1 after each loop iteration.

Selection Control Structures

- The syntax used for selection is:

  if *expression***;**
  ...body...
  end if;

- Notice that **if** and **end if** are both considered statements and must be terminated with semicolons.

- The *expression* is evaluated, and if it is a Boolean *true*, the body is performed. If it is a Boolean *false*, the body is not performed. Any arithmetic expression may be used in this context. A zero value is a Boolean *false*; a non-zero value is a Boolean *true*.

- An alternative action can be specified with the if-else form:

  if *expression***;**
  ...first body...
  else;
  ...second body...
  end if;

- Notice that **if**, **else**, and **end if** are each considered statements and must be terminated with semicolons.

- The *expression* is evaluated, and if it is a Boolean *true*, the first body is performed. If it is a Boolean *false*, the second body is performed. Any arithmetic expression may be used in this context. A zero value is a Boolean *false*; a non-zero value is a Boolean *true*.

Input and Output Statements

- Keyboard input it performed by the **read** statement. The syntax is:

  read *varref***;**

- When a read statement is performed, the program pauses and waits for the user to enter a number on the keyboard. The *varref* is any scalar variable or subscripted array reference. Notice that only one variable reference is permitted on each read statement.

- Screen output is performed by the print statement. The syntax is:

  print *printlist***;**

- A *printlist* is a comma-delimited list of print items. A print item is one of:

  - An expression. The value of the expression is written to the screen.
  - A character string is enclosed in double quotation marks. The characters in the string are written to the screen. A string may include the double quotation mark itself as one of the characters in the string. Use two adjacent double quotation marks inside a string, and in their place on the screen only one double quotation mark will be displayed at run-time. A character string cannot span the end of a line in the source code; it must be contained entirely within one line.
  - An exclamation mark (!). A carriage return and line feed are performed on the screen.

  For example:

```
print "Sum is ", Sum, !;
print "Mean is ", Sum/Count, !;
print "John said ""Hello"" to Mary.";
```

- The items in the print list are written to the screen in the same order in which they appear in the statement. The print statement performs a carriage return only when the carriage return item (!) appears in the print list.

```
#================================================================
# File--------euclid.slic
# Programmer--Bryan Crawley
# Project-----Slic example
#
# This program uses Euclid's Algorithm to compute and display the
# greatest common divisor of two positive integers.
#================================================================

main;
     data:
          integer: A, B, D, Temp, Asave, Bsave;
     algorithm:
          print !,!, "                        SLIC Demonstration", !;
          print !, "This program uses Euclid's Algorithm to determine the  greatest";
          print !, "common divisor of two positive integers.  It will prompt you to";
          print !, "enter the two integers A and B, and  will display the result on";
          print !, "the screen.  The program is written in SLIC,  a new programming";
          print !, "language designed and implemented at Georgetown College.";
          print !;
          print !, "Enter A: ";
          read Asave;
          print "Enter B: ";
          read Bsave;
          A := Asave;
          B := Bsave;
          while A<>B;
               if A<B;
                    Temp := A;
                    A := B;
                    B := Temp;
               end if;
               D := A - B;
               A := B;
               B := D;
          end while;
          print !,"Greatest common divisor of ";
          print Asave, " and ", Bsave, " is ", A, ".", !, !;
          exit;
end main;
```

```
#================================================================
# File--------sort.slic
# Programmer--Bryan Crawley
# Project-----Slic example
#
# This program uses a Selection Sort to sort a list of ten
# integers into ascending order.
#================================================================

main;
     data:
          integer: List[10];  # The list of integers.
          integer: Size;      # Number of entries in the list.
          integer: Top, J;    # Counters for counting loop.
          integer: MaxLoc;    # Index of largest value in list.
          integer: Temp;      # Used in exchanging array values.

     algorithm:
          Size := 10;

          # Enter and print the initial list of numbers.
          counting J upward 0 to Size-1;
               print "Element #", J, ": ";
               read List[J];
          end counting;

          print !;

          # Sort the numbers with a Selection Sort.
          counting Top downward Size-1 to 1;
               MaxLoc := 0;
               counting J upward 1 to Top;
                    if List[J] > List[MaxLoc];
                         MaxLoc := J;
                    end if;
               end counting;
               Temp := List[Top];
               List[Top] := List[MaxLoc];
               List[MaxLoc] := Temp;
          end counting;

          # Print the sorted list.
          counting J upward 0 to Size-1;
               print "Sorted #",J,": ",List[J],!;
          end counting;
          print !,"Finished",!;
          exit;
end main;
```