

MuLE Bindings Environment Practicum¹

The SPoc Interpreter.

Assumptions

Some knowledge of BNF grammars, DrScheme and the MuLE environment.

Objective

An understanding of how symbols, letters, and values are represented in memory and how variable/value pair bindings in SPoc represent/implement the former theoretical concept. Gain an understanding of how a language program stream is checked for correctness through parsing and the relationship between a grammar which is a language generator and a parser which is a language recognizer/acceptor.

Overview of SPoc and More Generally Imperative Languages

SPoc is an example of a language from the imperative or procedural paradigm. Imperative languages including Ada, C, FORTRAN, and Pascal (and some hybrid languages such as C++) owe some of their design to the ALGOL family of languages.

Imperative or procedural languages generally “consist of a series of procedures (or subprograms or functions or subroutines) that execute when called. Each procedure consists of a sequence of statements, where each statement manipulates data that may either be local to the procedure, a parameter passed in from the calling procedure or defined globally. Local data for each procedure are stored in an activation record associated with that procedure.”² Imperative languages are simply a more English-like abstraction of the underlying machine. “Machines dictate what imperative languages deal with, and convenience, or ease of programming, motivates the programming style that imperative languages support.”³ This means “the language must allow an underlying assignment-oriented machine to be used directly and efficiently.”⁴ “Hence, the imperative model was created to mimic, as closely as possible, the actions of computers at the machine language level.”⁵

¹ This work was partially funded by National Science Foundation CCLI-DUE # 9952398.

² Pratt, Terrence W. and Marvin V. Zelkowitz, Programming Languages Design and Implementation, 3rd Edition, Prentice Hall, Engelwood Cliffs, NJ, 1996, p. 451.

³ Sethi, Ravi, Programming Languages Concepts and Constructs, Addison Wesley, Reading MA, 1989, p. 66.

⁴ *Ibid.*, p. 66.

⁵ Dershem, Herbert L. and Michael J. Jipping, Programming Languages Structures and Models, PWS Publishing Co., Boston, MA, 1995, p. 52.

Features often associated with imperative languages include but are not limited to a block structure, sequential execution of statements, assignment statements, data structures, strong scoping rules, strong typing, and statement control flow structures. SPoc only has simple primitive variables and no typing but does have a simple version of most of the features usually attributed to procedural or imperative languages.

Overview of this Laboratory Assignment

This assignment requires modification of an incomplete implementation of SPoc, a simple imperative programming language. In other words, you will undertake to complete the partial implementation of the SPoc interpreter in three phrases.

The interpreter you have been given is fairly complete but lacks several components, namely:

1. The function that will look variables up in the SPoc bindings environment. Currently all the implementation is in place except the crucial *find_pair* functions which searches a list of variable-value pairs for the value of a given variable.
2. The *parse-decl* function is incomplete and currently will only parse a single variable declaration. Your job is to expand this function so it will parse a list of declarations.
3. The *parse-expression* function is incomplete and currently only parses expressions that consist of a number. You will need to write the code for a general expression parser for SPoc expressions.

The Details

Phase 1. Finding variable values

You need to write the code for **find-pair**. This function takes two parameters: a *variable-name* and a list of (*variable-name value*) pairs and should return the value of the *variable-name* parameter, or an empty list if the *variable-name* does not appear in the list. For example, given 'a and '((a 1) (b 2) (c 3)) as parameters your function should return a 1, whereas given 'q and '((a 1) (b 2) (c 3)) as parameters your function should return ' ().

This function is used by SPoc to look up the value of a variable within a block. (Looking through different blocks is already complete and calls this function multiple times for nested blocks.)

Phase 2. Extending Declaration lists

You need to write the code for **parse-decl**. This is used by the SPoc implementation to parse SPoc variable declarations of the following form.

```
<decl>      ::= declare <id_list> $
<id_list>   ::= <id>
<id_list>   ::= <id> / <id_list>
```

This function should use already available functions `get_next_expression` and `parse-id`, to read in the declaration list. Declarations are stored as *name-value tuples* in a list with the initial value of zero. For example, if the declaration reads

```
declare a / b / c $
```

store the declarations as `((a 0) (b 0) (c 0))`. The function should consume the terminating `$` but leave the remainder of the input alone. Your function should return a Scheme "DECL" record which contains a list of *name-value tuples*, or return an "ERR" record if an error occurs during the parse of the declaration list.

The skeleton implementation provided parses variable declarations of the following form.

```
<decl>      ::= declare <id_list> $
<id_list>   ::= <id>
```

So your task is essentially to allow SPoc programs to have more than one identifier declared per block. [Hints: Notice that `parse-decl` is used to parse declarations for the main program as well as for blocks. One solution is to use a global list variable in the function to “string” together the declaration list, therefore this list had to be “nulled” before the next block.]

Phase 3. Extending SPoc arithmetic Expressions

You need to write the code for **parse-expression**. This function is used by the SPoc implementation to parse SPoc arithmetic expressions that appear on the right-hand side of assignment statements. These expressions take the form

```
<expr>      ::= <factor> + <expr> |
                <factor> - <expr> |
                <factor> * <expr> |
                <factor> / <expr> |
                <factor>
```

`<factor> ::= <id> | <no> | (<expr>)`

The `parse-expression` function should use the *parse-id* function, and built-in Scheme functions to implement a parser for SPoc arithmetic expressions. This function should return the value of the arithmetic expression. Notice this function expects its input to be either an identifier (`<id>`), a number (`<no>`) or a list containing the expression. Use Scheme's list operations (e.g. `car`, `cdr`, `null?`) in writing this parse function. The skeleton implementation provided has been crippled to only accept a number.

An example of what the bindings look like internally

The boldface type is the SPoc program. The italicized lists are the internal bindings environment.

```
program one
declare a / b $
begin
  b = 3 $
  ((one ((b 3) (a 0))) ())
  block two
  declare c / d / e $
  begin
    d = 6 $
    ((two ((e 0) (d 6) (c 0))) (one ((b 3) (a 0))) ())
    e = 8 $
    ((two ((e 8) (d 6) (c 0))) (one ((b 3) (a 0))) ())
    block three
    declare a / x $
    begin
      a = 2 $
      ((three ((x 0) (a 2))) (two ((e 8) (d 6) (c 0))) (one ((b 3) (a 0))) ())
      end $
      a = 1 $
      ((two ((e 8) (d 6) (c 0))) (one ((b 3) (a 1))) ())
    end $
  end
```

Debugging Hints

SPoc includes a print statement so that SPoc variables can be output to SPoc's transcript window. However, to aid in debugging the SPoc interpreter (or any other interpreter for that matter) two options are available. To print one or more expressions to MuLE's transcript (SPoc's interpreter's transcript) window, use a writeln of the form:

```
(writeln "stringToOutput" expression)      or  
(writeln "stringToOutput" expressionList)
```

This writeln statement is a general scheme function that is defined in the utilities file. It can be used at any time in MuLE's transcript window or in the interpreter code itself.

To print one expression to SPoc's transcript window, use a writeln (also defined in utilities.s) of the form:

```
(send win writeln expression)
```

This command is implemented in the code for the SPoc window and is a command that the window object itself understands. The command cannot be used in MuLE's transcript window, but only in the interpreter code.

Requirements

This project is due at the beginning of class on **Monday, 4th of April, 2005**.

You **may** work in a team with your **term project partner** on this practicum.

You must turn in your code on Nova (in the TurnIn/Mule2 folder).

You must hand in a **hard copy** of your code. Do not include the utilities file. All of your code must be properly commented:

The file must begin with your name and administrative information (eg, assignment number etc.).

You must describe what you did to modify the file. Indicate functions that you included and line numbers of the changed code.

You must comment all sections of changed code.