UNIVERSITY OF THE WEST INDIES
Department of Computing
COMP3652–Language Processors
Semester I, 2016
Lecturer(s): Prof. Daniel Coore

# SMPL Specification with Extensions

## Introduction

SMPL is a small, but expressive toy programming language. As the name suggests, SMPL is easy to learn but, as you will discover, it is quite powerful.

This document only partly specifies SMPL. Specifically, only the core subset of SMPL is described. There are a number of extensions and modifications that can be made to SMPL, which you may choose to experiment with after implementing the core functionality.

SMPL is dynamically typed[1] and (should be) tail-recursive[2]. Its procedures are first class objects[3]. All SMPL sentences are in fact expressions in the sense that they implicitly return values. Some expressions return an "unspecified value" to indicate that the value is not useful; these expressions are essentially statements.

SMPL has integers, floating point numbers, the two boolean literals, the empty list, characters, strings, pairs and vectors as primitive data types. Storage that is dynamically allocated is automatically recovered by a garbage collector. There are no explicit pointers, all references to compound data are automatically treated as pointers to the data. All parameters are passed using the "call-by-value" convention, unless prefixed by a keyword indicating otherwise. Variables are statically scoped and may denote a value of any data type (they are dynamically typed).

## Syntax

The syntax of SMPL uses infix notation, and all binary operators must be separated from their operands by at least one whitespace character. The whitespace characters are the usual ones: space, carriage return and tab characters. (Feel free to include other control characters such as form feed and line feed if you wish). Note that it is important that operators are parsed as such only if they are surrounded by white space, because they are permitted as part of variable names. For example, `for-every?` and `q+r` are valid variable names.

### Literals and Values

SMPL can denote the following types of literals:

[1] variable types are not explicitly specified by the programmer

[2] procedure calls that are the final expressions in the calling procedure's body return their values to the calling procedure's caller (i.e. current stack frame is popped before a procedure call in tail position is executed)

[3] First class objects may be named, stored in data structures, passed as arguments and returned as values from procedures

| Escape Sequence | Character Denoted |
|---|---|
| `\\` | backslash (\) |
| `\n` | newline |
| `\t` | tab |

Table 1: Escape codes for control characters in strings

- Signed integers that can fit into 32-bit two's complement representation, strings and characters. Integers are assumed to be in decimal, unless prefixed with `#x` or `#b` in which case they are in hexadecimal or binary, respectively.
- Signed double precision floating point numbers. Note that either the whole number part or the fractional part of a floating point number is optional (but not both). For example, in SMPL, all of the following are legal floating point numbers: `1.5, 12., .317`, but the period by itself is not (obviously).
- String constants are denoted between double quotes (`"`). Within a string, certain control characters can be denoted by an escape sequence. Table 1 lists the set of escape sequences that should be recognised within SMPL strings.
- Character literals are denoted by preceding the character with `#c`. For example, the character `a` is represented as `#ca`. Common special characters have to be escaped by a backslash (just as they are in strings). Table 1 lists the codes for recognized special characters. To accomodate the remaining characters, character literals may also be specified by their unicode representation given as `#u` immediately followed by four hexadecimal digits. So the character `a` may also be denoted as `#u0061`.
- The boolean constants *true* and *false* are denoted `#t` and `#f` respectively.
- The empty list, called *nil*, is denoted by `#e`. An SMPL list is actually a sequence of pairs that terminates with the empty list. So the list operations are really a sort of *syntactic sugar* for the equivalent combinations of pairs. (This also means that testing a list with a predicate that returns true for pairs, will return true in all cases).

SMPL has two types of compound data: the vector and the pair. A vector is an array that it is not constrained to hold only one type of data (i.e. a heterogenous array). A pair contains two arbitrary objects. Lists are also supported natively, specifically as linked lists that are built up from pairs. Table 2 describes the builtin functions available for manipulating compound data in SMPL. Note that built-in functions are technically, not considered part of the language syntax. In other words, a user should be able to redefine built-in functions if he chose to do so.

Vector initialisation is quite flexible. A vector may be initialised by specifying a collection of disjoint subvectors, or the individual elements, or a combination of the two. A subvector is specified by two expressions: the first (after it has been evaluated) gives the size of the subvector, the second (after it has been evaluated) must be a procedure that when given an index less than the size of the subvector, returns the value to be stored at that position *in the subvector*. The following examples should help to clarify the description. In them, assume that the value of `x` has previously been set to 5.

| Builtin | Explanation |
|---|---|
| `pair(`⟨ $e_1$ `,` $e_2$ ⟩`)` | Create a pair containing the objects denoted by $e_1$ and $e_2$. |
| `car(`⟨ $p$ ⟩`)` | Return the first object in the pair $p$. |
| `cdr(`⟨ $p$ ⟩`)` | Return the second object in the pair $p$. |
| `pair?(`⟨ $p$ ⟩`)` | Return true if $p$ is a pair. |
| `list([`⟨ $e_1$ ⟩`, ...,` ⟨ $e_n$ ⟩`])` | Return a newly allocated list (sequence of pairs terminating. in `nil`) containing the values of the expressions $e_1$ through $e_n$ |
| `[`$e_1$`, ...,` $e_n$`]` | Return a newly allocated list, equivalent to `list(`$e_1$`, ...,` $e_n$`)`. |
| `[:[`⟨ e$_1$ ⟩`, ...,` ⟨ e$_n$ ⟩`] :]` | Return a newly allocated vector initialised with the given specifications. Each specification is either an expression or of the form ⟨ *expr*$_{size}$ ⟩ `:` ⟨ *expr*$_{init}$ ⟩. |
| ⟨ *expr*$_{vec}$ ⟩`[`⟨ $n$ ⟩`]` | Return the $n$th element of vector *vec* (indexed from 0). When on the LHS of an assignment, sets the $n$th element to the RHS. |
| `size(`⟨ *vec* ⟩`)` | Return the length of the vector *vec*. |
| `eqv?(`⟨ *expr* ⟩`,` ⟨ *expr* ⟩`)` | Return true if the two expressions evaluate to identical objects. |
| `equal?(`⟨ *expr* ⟩`,` ⟨ *expr* ⟩`)` | Return true if the two expressions evaluate to objects that are structurally identical. e.g. `eqv?(pair(1,2), pair(1,2))` ⇒ `false` because each call to `pair` returns a newly allocated object. but `equal?(pair(1,2), pair(1,2))` ⇒ `true` because each expression yields a pair containing the numbers 1 and 2 in that order. |
| `substr(`⟨ *expr* ⟩`,` ⟨ *expr* ⟩`,` ⟨ *expr* ⟩`)` | Evaluate the three expressions, the first should be a string, and the other two integers representing the start and end indexes within the string. Return the substring starting at the start index, up to, but not including the end index. If the end index is smaller than the start index, then the empty string is returned. The start index must be at least 0 and less than the length of the string, otherwise an error occurs. |

Table 2: The list of SMPL builtin functions. These are not syntactic forms of the language, although they may have specially developed methods for handling them.

```
[:  1,2,3 :]                                  ⇒  [1 2 3]
[:  1,2,x :]                                  ⇒  [1 2 5]
[:  5:  proc(i) i :]                          ⇒  [0 1 2 3 4]
[:  1, 3:  proc(n) 2 * n, 3 :]                ⇒  [1 0 2 4 3]
[:  3:  proc(n) 2 * n, 4:  proc(n) 3 * n :]   ⇒  [0 2 4 0 3 6 9]
```

### Statements and Expressions

A statement is a form that does not return a value. In SMPL, a statement is an expression whose value is not used (for example, expressions that appear as part of a sequence before the final expression) or whose value is not useful (e.g. the result of a `def`). The semicolon is used as a statement terminator. Note that *expressions* are not generally terminated by a semicolon, but if an expression is used as a statement (in a context where its result is not used) then it will need a semicolon to terminate it. The rationale for this is that a procedure call has the syntax *expr(expr)* and without a semicolon to terminate expressions, there would be no way to distinguish between a procedure call, and two consecutive expressions (being used as statements) in which the second happened to begin with an open parenthesis. Table 3 lists the keywords of SMPL and their purposes.

Identifiers in SMPL must contain at least one non-digit character, and may not begin with the character `#`. Identifiers may not contain parentheses, brackets, braces, any of the quote characters, the comma, nor the colon (although they may contain operator characters such as `-`, `+`, and `*`). The following are all legal SMPL identifiers: `foo`, `bar1`, `1bar`, `ba1r`, `foo!`, `bar?`, `fo#o`, `foo.bar`. The following are illegal identifiers: `#bar`, `12`, `(foo)`, {`bar`}, `[baz]`, `foo,bar`, `foo:bar`.

Function calls are denoted by the function expression followed immediately by a sequence of comma-separated argument expressions enclosed in parentheses. The following expressions are all legal function calls: `f(a, b)`, `g()`, `(foo(3))(a, b, c, d)`, `(proc(n) n * 4)(2)`. Functions may be declared with a variable number of arguments, but in all cases, it is an error to call a function with fewer or more arguments than is permitted by the function's declaration.

SMPL understands the following common binary operators:

- Arithmetic operators: `+, -, *, /, %, ^`
- Bitwise operators: `&, |, ~`
- Relational operators: `=, >, <, <=, >=, !=`
- Logical operators: `and, or, not`
- List concatenation: `@`

Operator precedence, from highest to lowest is as follows: $\sim \succ *, /, \% \succ +, - \succ \&, | \succ =, >, <, \leq, \geq, \neq \succ$ `not` $\succ$ `and` $\succ$ `or`

In order to apply a unary minus to an expression, the combined expression must be surrounded by parentheses. So the negative of the variable `x` is expressed as `(- x)`, not `- x`. (This will make parsing a little easier, bonus marks if you can get it to work in all cases without the parentheses).

| Keyword | Purpose |
|---|---|
| `proc([`$p_1, p_2, \ldots, p_n$`])`⟨ *body* ⟩ | return a procedure of $n$ arguments with formal parameters $p_i$ (each $p_i$ is an identifier). |
| `proc(`$p_1, \ldots, p_n$ `.` $p_{\texttt{rest}}$`)`⟨ *body* ⟩ | return a procedure that takes at least $n$ arguments. When it is called with more than $n$ arguments, the excess arguments are placed in a list which is then passed as the argument to $p_{\text{rest}}$ |
| `proc` ⟨ *id* ⟩ ⟨ *body* ⟩ | return a procedure that may take any number of arguments (including zero). When this procedure is called, all of the supplied arguments are placed in a list which is passed to the named parameter. |
| `call(`⟨ *f* ⟩`,` ⟨ *lst* ⟩`)` | `f` is an expression that evaluates to a procedure and `lst` is an expression that evaluates to a list. Apply the procedure resulting from evaluating `f` to the list of arguments resulting from evaluating `lst`. This builtin is used to invoke procedures of variable arity on an indeterminate number of arguments. e.g. `call(f, list(1, 2, 3))` is equivalent to `f(1, 2, 3)` and `call(f, x)` would apply `f` to all the elements of the list bound to `x`, and return the result. |
| `lazy(`⟨ *exp* ⟩`)` | evaluate the given expression lazily. Lazy expressions are evaluated only if their value is needed by a primitive function or operator. Lazy expressions should be evaluated at most once. |
| `let(`$b_1, b_2, \ldots, b_n$`)`⟨ *body* ⟩ | evaluate *body* in an environment extended by bindings $b_i$. The syntax of a binding is ⟨ *id* ⟩ `=` ⟨ *expr* ⟩. |
| `def` ⟨ *id* ⟩ ⟨ *expr* ⟩ | allocate storage for *id* and store in it the value of *expr* in the current environment. |
| ⟨ *id* ⟩ `:=` ⟨ *expr* ⟩ | assign the value of *expr* to variable *id*. |
| ⟨ $e_1$ ⟩`,` ⟨ $e_2$ ⟩`, ...,` ⟨ $e_n$ ⟩ | generate an $n$-valued result by evaluating each of the $n$ expressions. |
| ⟨ *id* ⟩`, ...,` ⟨ *id* ⟩ `:=` ⟨ *expr* ⟩ | assign the multiple values of *expr* to the variables on the left hand side of the assignment operator. *expr* must be a multi-valued expression returning the same number of values as the number of identifiers on the left. (e.g. `x,y := 1,2`) |
| `if` ⟨ *expr* ⟩ `then` ⟨ *expr* ⟩ `[`*else* ⟨ expr ⟩`]` | test predicate, evaluate then clause if non-false otherwise evaluate else clause, if given. |
| `case {` `[`$p_1 : c_1, \ldots, p_n : c_n$`] }` ⟨ *expr* ⟩ `:` ⟨ *expr* ⟩ | Return the value of the consequent of the first clause whose predicate is true. A clause of a case expression. If predicate is the keyword ELSE, it is regarded as true. |
| `{ ... }` | compound expression (treat the sequence of expressions enclosed within the {, } as if it were a single expression). Each expression is evaluated in order. The result of the last one is the value of the compound. |
| `(`⟨ *expr* ⟩`, ..., [`*expr*`])` | multi-valued expression. Evaluate each expression enclosed within the () in any order. The tuple of values is the result. |
| `print(`⟨ *expr* ⟩`)` | Print the value of the given expression. |
| `println(`⟨ *expr* ⟩`)` | Print the value of the given expression terminated by a newline. |
| `read()` | Read and return a string from the keyboard. |
| `readint()` | Read and return an integer from the keyboard. |
| `//` | comment to rest of line |
| `/* ... */` | block comment (nestable) |

# Extensions (Optional)

This section describes some of the extensions that you are welcome to consider implementing. It is by no means a complete list of features that you could give to a language. Indeed, it may not even be desirable for a langauge to have all of the features listed here. You should read this section as merely suggestions of extensions to SMPL.

## Extension: Lazy evaluation

You can experiment with lazy evaluation by introducing a new keyword `lazy` into SMPL to declare that the enclosed computation should be performed lazily. This extension would provide lazy evaluation within any context, not just for parameter passing. The syntax of the `lazy` form is as follows:

`lazy(`⟨ *exp* ⟩`);`

The lazy form should be permitted in any context in which an expression is expected. Here are some examples of how the `lazy` keyword affects the evaluation of ordinary SMPL programs.

```
def f proc(x, y, z) {
  if (x > 0)
  then y + 1;
  else z;
}

print f(1, 2, 1/0);
```

produces an error because dividing by zero is illegal, and in a call by value semantics, all of the arguments to a procedure have to be evaluated before the procedure's body can be entered.

On the other hand, if we evaluate the arguments lazily, we should get a different response:

```
def f proc(x, y, z) {
  if (x > 0)
  then y + 1;
  else z;
}

print f(1, 2, lazy(1/0));
```

Now, we get 3, because since the parameter `z` was never used, it was never evaluated, and we never got the error.

### Implementation of Lazy Evaluation

The discussion in class described an implementation of lazy evaluation that is dynamically scoped and therefore more dangerous than it is worth. A more useful specification of lazy evaluation is to insist that the expression is evaluated within the context in which it originally appeared, even its value is needed in another. This would require that the expression whose evalaution has been deferred must be paired with the environment in which the expression was first encountered. There are at least two different ways that you could approach implementing this.

The first way is to treat the keyword `lazy` as a kind of syntactic sugar for creating a function of no arguments that had the ability to cache the result of the expression and use it instead of re-evaluating the expression on subsequent references. For example, the form `lazy(`$E$`)` for some expression $E$ would be transformed into the anonymous function:

```
proc() if undefined(result)
       then def result E;
       result
```

In this suggested implementation, the check for the undefined result would not have to be literally generated to be parsed, but an appropriate intermediate representation for it could be created and incorporated into the body of the auto-generated anonymous function. Note that the evaluator would need to have a way to distinguish between functions generated in this way and those generated by a user actually writing a function that looked similar to the one above. The reason for the need to distinguish between the two is that when the evaluator encounters one of the auto-generated functions, it must know that it should invoke it to obtain the value of the expression that was deferred earlier. If it is a user-defined function, then a reference to the function should just return the function object itself (not the result of invoking it). (For example, if `f` had been defined to be a function like the one above, then `f` and `f()` have different meanings, so the interpreter should never return the latter for the former.)

The second approach (and more natural one) would be to create an intermediate representation for the `lazy` form and implement a special visit method for it within the `Visitor`. In the intermediate representation, you would have to capture not only the expression that was wrapped by the `lazy` keyword, but also the current environment, so that it would be available for access, when the expression's value was eventually required. You would also need a runtime data structure to represent the instance of an expression that arose from visiting the `lazy` keyword (each visit of a lazily evaluated expression will need its own cache of its result).

Think of this runtime data structure as a *memoised promise*. A *promise* is just a deferred expression along with its environment. When the evaluator tries to obtain one of the other kinds of values and encounters a promise instead, it should invoke the promise so that it can yield the value to be used. A *memoised promise* would just store that value so that subsequent references do not need to re-evaluate the expression. One potential pitfall with this approach is that *promise* objects now become special, because if you store one in a data structure, and then later retrieve it, the evaluator might invoke it, instead of returning the promise object itself. This problem can be avoided if promise objects are treated specially (not first class objects) and are restricted to arise from only certain special forms (such as `lazy`).

## Extension: Dynamic Scoping

You can experiment with an extension to SMPL that will allow the user to declare a variable to be dynamically scoped, instead of the usual static scope that it would normally have. Note that the scoping discipline only matters when we are looking up free variables in procedures. So we introduce the keyword `dynamic` to indicate that one or more variables within the current scope are to be treated as dynamically scoped variables. The syntax of the `dynamic` form is as follows:

`dynamic` ⟨ $id_1$ ⟩`, ...,` ⟨ $id_n$ ⟩`;`

The `dynamic` declaration must be the first statement in the body of a procedure, if it is being used at all. (Not all procedures will use the `dynamic` form, but those that do must place it first within their bodies.) All of the identifiers specified in the `dynamic` statement are to be looked up as dynamically scoped variables. Note that if a parameter or local variable is declared to be dynamically scoped, it will have no effect, since for static and dynamic scoping, local variables and parameters are looked up in the same way.

Here are some examples of how the `dynamic` keyword affects the evaluation of ordinary SMPL programs.

```
def y 5;
def f proc(x) x * y;
def g proc(y) y + f(y);
print(g(3));
```

Prints 18, because `g(3) = 3 + f(3) = 3 + 3 * 5 = 18`

On the other hand, if we insert a `dynamic` declaration in `f` to say that the free variable `y` in `f` is to now be dynamically scoped, then we get a different answer:

```
def y 5;
def f proc(x) dynamic y; x * y;
def g proc(y) y + f(y);
print(g(3));
```

Now, we get 12, because `g(3) = 3 + f(3) = 3 + 3 * 3 = 12`

Notice that the reference to `y` in `f` now picks up the most recent binding for `y` in the call chain, so we get the binding for `y` in the frame for `g` because `g` called `f` and its binding for `y` is more recent than the global binding for `y`.

### Usefulness of Dynamic Scoping

Although dynamic scoping is out of vogue, and perhaps will never return to vogue because it makes software more difficult to maintain, it does have some limited uses. One example is that it permits local overrides of global default settings. For example, suppose that you have a procedure, called `display`, that takes a single string as an argument and pops up a window to display that string. Normally to specify all of the properties of a window requires several parameter settings (e.g. its dimensions, the placement of the window controls, the thickness of its border, its background colour, etc. [4]). The procedure, `display` can control how the new window looks by varying these parameters, but the function calling `display` cannot, unless we add extra parameters to `display`. If we do that though, we will get ourselves into trouble, because it is not so easy to decide which parameters ought to be passed, and as soon as we start increasing the number of paramters required by `display`, the harder it becomes to use, because now its callers must supply more information.

Now, with dynamic scoping, we could have `display` declare the window controlling parameters as dynamically scoped variables. The window manager will have reasonable global defaults specified, so functions calling `display` without any special needs will have no trouble. Now, imagine that we have a procedure, called `foo` that has a particularly long message to display, so it wants to set the window size to larger than normal. It can declare local bindings for those window parameters,

[4] Typically these things are controlled globally by the window manager

and then call `display`, as usual with only one argument, and its messages will be displayed in a larger window. In the meantime, other functions calling `display` will get their messages displayed in the normally sized window, because the bindings for the larger dimensions would be local to `foo` and would not be in the scope of `display` since it was not called from `foo`. Notice that we get the neat feature of having variables that are non-local to `display` but that are not global since if one context needs to modify them for its purpose none of the other contexts using those variables need be affected.

**Suggested Implementation**

In the discussions in the lecture, the method for implementing dynamic scoping is to simply change the rule for extending environments when procedures are called. In fact, for dynamic scoping, things became somewhat simpler, because a procedure object only needed to reference its parameters and body, no closing environment. When a procedure is called, it simply extends the current frame with a new frame that contains bindings for its formal parameters.

The extension to allow dynamically scoped variables, described above, is not quite as straightforward to implement though, because we still want to have the default of static scoping. One approach is to allocate two parent environments for each new environment that we create: a dynamic one and a static one. In other words, each environment will now have both a static and a dynamic link that each point to some other environment (sometimes the two environments will be the same).

Whenever a variable is looked up, the appropriate discipline is determined for it, based on whether it is statically or dynamically scoped. If a variable is statically scoped, and not in the current frame, then we follow its static link to its parent environment. If the variable we want is not present there, then we continue to follow the static link. Likewise for a dynamically scoped variable, we follow the chain of dynamic links until we find the variable (or when we reach the global environment and still haven't found it, we return an error).

The only other issue to handle is the data structure that will manage the set of variables that are to be dynamically looked up within each context. In other words, the evaluator will need to have some way of keeping track of which variables are to be dynamically scoped, and which are not. One way is to use a static context checker that would walk a program and statically determine dynamic variable references. Once identified, those references could be replaced by specialised versions whose visit method uses a dynamic scoping version of variable lookup, while the default would remain a statical scoping version. Using this method requires two separate visitor methods in your evaluator, and a subclass of the variable reference intermediate representation that would be specific for dynamically scoped variables. A slightly less efficient method, but one that would leave the intermediate representation unmodified, would be to make the dispatch of the type of scoping rule be done at runtime.

For example, one way to do this runtime dispatch of the type of scoping to apply to a variable, is to have the intermediate representation for variable expressions have a flag within it that indicates whether the variable is statically scoped or not. Then its `visit` method can decide which visit method within the visitor to call, and each visitor can then specify a method for looking up variables with static scoping rules or dynamic scoping rules.

Alternatively, you could leave the `Visitor` interface unchanged (with only one method to visit variable expressions) and leave it up to the visitor to dispatch on the scoping rule for the variable.

To do this, you would maintain a run-time data structure for deciding how each variable is looked up. In this case, the method for visiting a variable would examine this run-time structure to determine whether a given variable had been declared dynamic, and then carry out the appropriate lookup discipline. The advantage to this method is that it might be easier to manage a run-time data structure within the evaluator since it is all confined within the evaluator, and it does not require any changes to the static structures of the SMPL processor (i.e. the intermediate representations, the visitor interface, and their implementing classes). The disadvantage of this method is that it is inefficient. Every time a variable is looked up within the same context, the interpreter must determine whether it is statically or dynamically scoped even though the result is the same every time, since the variable's scoping rule never changes while the program is running.

### Extension: Reference Parameters

Recall that a reference parameter is one that inherits the location of its argument (rather than its value). This means that mutations to the object referenced by the local parameter will be observable from the variable that was passed as an argument to the parameter. (Only a variable may be passed as an argument to a reference parameter).

Note that in many modern languages, as is the case in SMPL compound objects are already being passed by reference. For example, passing a list to a function as an argument, does not involve copying the list. Rather, the reference to the list is passed, which means that changes made to the list, by statements that are local to the function will be observable from outside of the function, through the external reference that was provided.

The difference lies in how primitive data would be passed. Currently, if a variable in the global scope contains an integer, and it is passed to a function's parameter, its value is copied to the parameter's location. A reference parameter ought to point at the location where that primitive data is stored, so that changes made locally in the function will be observable through the argument variable passed to the reference parameter. The suggested keyword to indicate a reference parameter is to place the keyword `ref` before the name of the parameter in the parameter list of a function. The example below illustrates a classic use of reference parameters:

```
def x 5;
def y 7;

def swap proc(a, b) {
  tmp = a;
  a = b;
  b = tmp;
}
swap(x, y);
```

```
def x 5;
def y 7;

def swap proc(ref a, ref b) {
  tmp = a;
  a = b;
  b = tmp;
}
swap(x, y);
```

In the implementation of `swap` on the left, after the call to `swap(x, y)`, the values of `x` and `y` remain unchanged, because they are passed by value. In contrast, after evaluating the implementation on the right, the values of `x` and `y` to be exchanged (i.e. `x` will be 7, and `y` will be 5).

### Extension: Graphics

There are a few primitives in SMPL to support graphics operations. The function `canvas` takes two integers representing the width and height of the (rectangular) drawing area to create. All drawing

operations take a canvas instance as their first argument, and render their effects upon that canvas. The coordinate system of the canvas is such that the bottom left corner is the origin (0,0) and the top right corner has the maximal $x$ and $y$ components.

The notion of a point is also natively supported in SMPL. The command `pt` takes the $x$ and $y$ coordinates (as integers) and creates a point at that location. Two point instances should always be considered equal to each other if they represent the same location, even if they are created with different calls to the `pt` command. For example, `eqv?(pt(1,2), pt(1,2))` should return true. All of the arithmetic operations should be overloaded to handle points. This means, that if `p` and `q` are two point instances, the expressions `p + q`, `p - q` are legal, and return points whose coordinates are the sum and difference, respectively, of the coordinates of `p` and `q`. Multiplication and division (`*, /`) should be overloaded to support scaling a point by an integer. The product of two points should be interpreted as the dot product, so it returns the scalar sum of the product of the corresponding coordinates of the two operand points.

Table 4 lists all of the supported graphics operations, with a brief explanation of each.

| Keyword | Description |
|---|---|
| `canvas(`⟨ *expr* ⟩`, `⟨ *expr* ⟩`)` | Create a rectangular drawing area of specified width and height (width given first). |
| `pt(`⟨ *expr* ⟩`, `⟨ *expr* ⟩`)` | Create a point to represent the given Cartesian $(x, y)$ coordinates. |
| `rect(`⟨ *expr* ⟩`, `⟨ *expr* ⟩`, `⟨ *expr* ⟩`)` | Evaluate the arguments to yield a canvas and two points. Draw a rectangle on the canvas with the bottom left corner at the first point, and the top right corner at the second point. |
| `circle(`⟨ *expr* ⟩`, `⟨ *expr* ⟩`, `⟨ *expr* ⟩`)` | Evaluate the arguments to yield a canvas, a point and a number. Draw a circle on the canvas, centred at the given point with the given radius. |
| `path(`⟨ *expr* ⟩`, `⟨ *expr* ⟩`)` | Evaluate the arguments to yield a canvas and a list of points. Draw a path of straight lines connecting each consecutive pair of points in the list. |
| `cpath(`⟨ *expr* ⟩`, `⟨ *expr* ⟩`)` | Evaluate the arguments to yield a canvas and a list of points. Draw a closed path of straight lines connecting each consecutive pair of points in the list (closed means that the last point connects to the first). |
| `clear(`⟨ *expr* ⟩`)` | Clear the canvas resulting from evaluating the given expression. |
| `setfg(`⟨ *expr* ⟩`, `⟨ *expr* ⟩`)`<br>`setbg(`⟨ *expr* ⟩`, `⟨ *expr* ⟩`)` | Evaluate the arguments to obtain a canvas and a colour (specified as a vector of 3 integers, each ranging from 0 to 255, and representing RGB intensities). Set the foreground (or background) colour on the given canvas to the given colour. |

Table 4: Graphics commands supported in SMPL

## Other Ideas for Extensions

Here are a few ideas for some more extensions to SMPL:

- arbitrary precision integer arithmetic. It would be good if SMPL were not restricted to integers that could fit within the 32-bit two's complement representation. These "big" integers could be represented by using multiple words of contiguous storage to store the bits of the number. Each of the primitive arithmetic operators would have to be redefined to accommodate these large numbers. However, the only difference the user should observe is that she is no longer restricted to small integers. (A good test case for this is to see whether your extended language can compute the factorial of 1000.)

- User-defined records. There could be a construct (similar to the `struct` construct in C) that would allow the user to define and name compound objects that were composed of several named fields. These fields could contain any value, including procedures.

- Additional control structures. Typical looping constructs such as `for`, `repeat` and `while` in Pascal could be included in SMPL.

- Exceptions and constructs for handling them. For example the `try`, `catch` and `throw` mechanism of Java and Lisp.

- Macros. A limited form of language extension can be accomplished through the use of macros. It should not be too difficult to extend SMPL to include macros that are declared and used in a similar way to procedures.

- Call by name parameter passing conventions. At the moment SMPL supports only call by value (CBV) and optionally by lazy and by reference. It could be extended to allow procedure declarations that would support parameter passing by other conventions (or you could introduce keywords that could be used in contexts expecting expressions, just like was done for lazy evaluation).

- Threads. Given that Java already has good support for threading, it is actually not too difficult to "piggy back" on Java's threads to provide threading in SMPL. If you do this, you should probably use a functional approach, where a thread is an object (much like a procedure is) and there is a special keyword to initiate a thread (i.e. start it) and continue without waiting for the thread to complete. You should provide a keyword called `join` that takes a thread, and causes the current thread to wait until the joined thread completes and returns a value (the result of the last expression evaluated) which is also the return value of the `join` special form.

## Examples

Here are some example procedures in SMPL.

```
def fact proc(n)
  // return factorial n
  if n <= 1
     then 1
     else n * fact(n - 1);

def fib proc(n)
  /* return the nth fibonacci no.*/
  if n <= 1
     then 1
     else fib(n - 1) + fib(n - 2);

def map proc(f, list)
  /* return a new list contining
     f(x) for each x in list */
  if list = #e
     then #e
     else pair(f(car(list)),
               map(f, cdr(list)));

def gcd proc(a, b)
  /* Euclidean alg. for gcd */
  case {
    a < b: gcd(b, a);
    b = 0: a;
    else : gcd(b, a % b);
  }
```

```
def foldr proc(op, base, lst)
  /* fold right LST using binary operator OP
     and base value BASE */
  if lst = #e
     then base
     else op(car(lst),
             foldr(op, base, cdr(lst)));

def append proc(l1, l2)
  /* copy l1 and then attach l2 to it
     and return the resulting list. */
  if l1 = #e
     then l2
     else cons(car(l1),
               append(cdr(l1), l2));

def vecMap proc(f, v)
  /* return new vector containing f(x)
     for each x in v */
  [: size(v): proc(i) f(v[i]) :];

def vecAppend proc(v1, v2)
  /* append v2 onto the end of v1, return
     result in a new vector */
  [: size(v1): proc(i) v1[i],
     size(v2): proc(i) v2[i] :];
```