Implementing a Scheme-Like Interpreter in Perl

Bill Hails

April 15, 2006

Contents

List of Figures

1 Introduction

By the end of this chapter you should have a thorough understanding of the inner workings of a programming language interpreter. The source code is presented in full, and several iterations add more features until it could be considered pretty complete. That is quite a tall order for so short a piece as this, but the interpreter is written to be as easy to understand as possible: It has no clever optimizations that might obscure the basic ideas, and the code and the ideas will be explained without any technical jargon. It is however assumed that you has a good working knowledge of Perl (Perl5), including its object-oriented features.

The final implementation will demonstrate:

- primitive arithmetic operations;
- conditional evaluation;
- local variables;
- functions and closure;
- recursion;
- list processing;
- quote—preventing evaluation;
- a simple macro facility;
- variable assignment and side-effects;
- procedures (as opposed to functions) and sequences;
- objects and classes;
- continuations.

Having said that, time and space is not wasted fleshing the interpreter out with numerous cut'n'paste system interfaces, i/o or even much basic arithmetic (the final implementation has only multiplication and subtraction—enough for the tests and examples to work,) but by then it should be a trivial matter for anyone to add those themselves if they feel so inclined. Another point worth mentioning up front is that no claims are made that this is in any way an efficient implementation. It is just meant to be easy to understand.

My motivation for writing this is that I have always been fascinated by the idea that programming languages are just programs, but I found it very difficult in the past to work out what was actually going on in the handful of public domain implementations of programming languages that were available at the time. The temptation always seemed to be there for the authors to add all sorts of bells and whistles to their pet project, to the point that the core ideas became obscured and obfuscated. In fact it was only when I found out that the easiest implementations of Scheme to understand were written in Scheme itself that I made any real progress with that particular language. However implementing an interpreter in terms of itself (so-called "meta-circular evaluation") easily leads to confusion, and it struck me that Perl, with its very high-level constructs and high signal to noise ratio is the perfect vehicle to demonstrate the programming language concepts that I've so painfully gleaned through time, without any incestuous meta-circular issues to deal with.

2 Why Scheme?

Scheme is one of the younger members of the Lisp family of programming languages. LISP stands for "LISt Processing"[1], and this is an appropriate name since the fundamental data type in these languages is the list.

The main reason for choosing Scheme to demonstrate the internals of an interpreter is that Scheme is a very simple language, and at the same time an astonishingly powerful one. An analogy might be that if C is the "chess" of programming languages, then Scheme is more like "go". The official standard for Scheme, the "Revised(5) Report on the Algorithmic Language Scheme" or R^{5}RS (`http://www-swiss.ai.mit.edu/~jaffer/r5rs_toc.html`) as it is known, has this to say:

> Programming languages should be designed not by piling feature on top of feature, but by removing the weaknesses and restrictions that make additional features appear necessary.

Whether or not one agrees with that, and it's hard to argue, it strongly suggests that such a language might be pretty straightforward to implement.

Another interesting feature of Scheme and the Lisp family of languages is that the list data that they work with is also used to construct the internal representation of the programs themselves. Therefore Scheme programs can manipulate their own syntax trees. This makes the definition of macros (syntactic extensions) particularly easy from within the language itself, without recourse to any separate preprocessing stage. Finally, another good reason for choosing Scheme is that it is extremely easy to parse, as we shall see.

[1] Or "Lots of Irritating Single Parentheses".

3 An Introduction to PScheme

In subsequent discussions, “PScheme” means this particular implementation of a Scheme-like interpreter (Perl-Scheme). The implementation lacks a number of the features of a complete implementation, and differs from the Scheme standard at a number of points. However it could be argued that it is close enough to call itself “Scheme-like”, it’s certainly closer to a reference implementation of Scheme than it is to any other language in the Lisp family.

3.1 PScheme Syntax

PScheme has a very simple syntax. A PScheme *expression* is either a number, a string, a symbol, or a list of expressions separated by spaces and enclosed in round brackets (where an expression is either a number, a string, a symbol, or a list of...). We can write this recursive definition in a special purpose notation for describing programming language grammars called Backus-Naur Format (BNF) as follows:

$$\begin{array}{lll} \langle \textit{expression} \rangle & \texttt{::=} & \langle \textit{number} \rangle \\ & \texttt{|} & \langle \textit{string} \rangle \\ & \texttt{|} & \langle \textit{symbol} \rangle \\ & \texttt{|} & \text{'(' } \langle \textit{expression} \rangle \text{ ... ')'} \end{array}$$

Read “`::=`” as “is a”, and “`|`” as “or”.

A PScheme *number* is a sequence of digits, optionally preceded by a “`+`” or a “`-`”. PScheme does not support floating point or other more complex number types.[2]

A PScheme *string* is any sequence of characters enclosed by double quotes. Within a string, a double quote may be escaped with a backslash.

PScheme has a rather more relaxed idea of what constitutes a *symbol* than most languages, essentially it’s anything that isn’t an open or close brace and doesn’t look like a number or a string, up to the next whitespace character. So “`x`”, “`type`”, “`a2-2`”, “`this-is-a-symbol`”, “`<`”, “`*`”, “`+`”, “`&foo`”, “`=+=`” and “`$%*!!`” are all symbols.

PScheme reads an expression, then evaluates it, then prints the result. The rules for evaluation are also very simple:

- The result of evaluating a number or a string is just that number or string;
- The result of evaluating a symbol is the value that that symbol currently has, or an error if the symbol has no value;
- The result of evaluating a list of expressions is the result of evaluating each expression in turn, then applying the first evaluated expression (which should be a function) to the other evaluated expressions.

3.2 Simple Expressions

So lets take a look at PScheme in action. Firstly Scheme is typically an interactive language, it presents a prompt, and the user types in expressions. The interpreter evaluates those expressions then prints the results:

[2] A full Scheme implementation supports a large range of numeric types, from arbitrarily large integers through floating point, precision-preserving fractions, and complex numbers.

```
> 2
2
```

Here we gave the interpreter a 2, and it replied with 2, because 2 is 2 is 2 (the ">" is the PScheme prompt). Let's try something a bit more adventurous:

```
> x
Error: no binding for x in PScm::Env
```

We asked for the value of a symbol, x, and because the interpreter doesn't know what x is, we get an error. Let's try something that does work:

```
> (* 2 2)
4
```

Now that might look strange at first, but remember the first expression in the list should evaluate to a function. The multiplication symbol "*" evaluates to the internal primitive definition of how to multiply. We told PScheme to multiply 2 by 2, and it replied 4. All it has done is to:

1. evaluate the symbol `*` to get its value: the multiplication function.
2. evaluate the first `2` to get 2;
3. evaluate the second `2` to get 2;
4. applied the multiplication function to arguments 2 and 2.

Another important thing to note here is that PScheme makes no distinction between functions and operators, the operation always comes first. This has some advantages. Because the operation always comes first, it can often apply to variable numbers of arguments:

```
> (* 2 2 2 2)
16
```

A more syntax-rich language would require something like `2 * 2 * 2 * 2` to get the same result.

Now for something just a little more complex:

```
> (* (- 8 3) 2)
10
```

Here we told the interpreter to subtract 3 from 8, then multiply the result by 2. It did it by:

1. Evaluating the symbol `*` to get the multiplication function;
2. Evaluating the expression `(- 8 3)` to get 5, which it did by:
   (a) Evaluating the symbol "-" to get the subtraction function;
   (b) Evaluating `8` to get 8;
   (c) Evaluating `3` to get 3;

    (d) Applying the subtraction function to arguments 8 and 3.

3. Evaluating `2` to get 2;

4. Applying the multiplication function to arguments 5 and 2 to get 10.

Hopefully it is obvious that the interpreter is following a very simple set of rules here, albeit recursively.

This incidentally demonstrates another big simplification that PScheme makes: it is impossible for there to be any ambiguity about operator precedence, because the language forces the precedence to be explicit. In fact there is no notion of operator precedence in PScheme.

Again in a more syntax-rich language, to achieve the above result one would have to write `(8 - 3) * 2` because the equally legal `8 - 3 * 2` would be misinterpreted (a lovely expression) as `8 - (3 * 2)`.

3.3 Conditionals

The keyword *if* introduces a conditional statement. The general form of an `if` expression is:

```
(if ⟨test⟩
    ⟨true-result⟩
    ⟨false-result⟩)
```

This is simple enough, `if` expects (in this implementation at least) three arguments: a test, a true result and a false result. For example:

```
> (if 0
>     3
>     (- 8 3))
5
```

In this example since the test, `0`, is false (again, in this implementation) the false result ($8 - 3 = 5$) is returned.

Even here we can start to see some of the power of the language:

```
> ((if 0 - *) 4 5)
20
```

In the author's opinion this is a beautiful example of "removing the weaknesses and restrictions that make additional features appear necessary": because the language treats the operator position just like any other expression, any expression that evaluates to an operation is valid in that position. Furthermore because primitive operations are represented by symbols just like anything else, they can be treated just like any other variable: the `if` with a false (`0`) test argument selects the value of "`*`" to return, rather than the value of "`-`". So it's the multiplication function that gets applied to the arguments 4 and 5.

However there is a slight complication, Consider this:

```
> (if 0
>     (a-long-calculation)
>     (- 8 3))
5
```

Were `if` a normal function, the normal rules for evaluation would apply: evaluate *all* the components of the list, then apply the `if` function to the evaluated arguments. That would mean `(a-long-calculation)` and `(- 8 3)` would *both* get evaluated, then `if` would pick the result. Although the value of the whole `if` expression is unaffected, provided `(a-long-calculation)` doesn't have any side-effects, we still don't want to have that calculation executed unnecessarily. Now remember it was said that PScheme evaluates each component of the list in a list expression? Well that's not entirely the case. It always evaluates the *first* component of the list, and if the result is a simple function like multiplication, then it goes on to evaluate the other items on the list and passes the results to the function just as has already been described. However if the first component is what is called a *special form*, such as the definition of `if`, then PScheme passes the un-evaluated arguments to the special form, and that special form can do what it likes with them.

In the case of `if`, `if` evaluates its first argument (the test) and if it is true it evaluates and returns its second argument (the true result), otherwise it evaluates and returns its third argument (the false result). We can demonstrate that with a simple example:

```
> (if 1
>     10
>     x)
10
```

Because the test result was true, the `if` only evaluated the true branch of the expression, there was no error from the undefined symbol `x` on the false branch.

3.4 Global Variables

define is the way we associate values with global variables in PScheme. It has the general form:

`(define` ⟨*symbol*⟩ ⟨*expression*⟩`)`

For example:

```
> (define x 5)
x
> x
5
```

In the above example we gave `x` the value `5`, then when we asked for the value of `x`, it replied `5`. Note again that the operation (`define` in this case) always comes first. Note also that `define` must be a special form, because we didn't get an error attempting to evaluate `x` during the definition. `define` does however evaluate its second argument, so for example:

```
> (define a b)
Error: no binding for b in PScm::Env
```

causes an immediate error attempting to evaluate the undefined symbol `b` before assigning the result to `a`.

3.5 Functions

lambda, another special form, creates a function. The general layout of a `lambda` expression is:

```
(lambda (⟨symbol⟩ ...) ⟨expression⟩)
```

The `(⟨symbol⟩ ...)` part is a list of the names of the arguments to the function, and the ⟨*expression*⟩ is the body of the function. For example:

```
> (define square
>   (lambda (x) (* x x)))
square
```

Now that may also look a bit strange at first, but simply put, `lambda` creates an *anonymous* function, and that is separate from giving that function a name with `define`.

The function being defined in this example takes one argument `x` and its function body is `(* x x)`. The function body will execute when the function is invoked. This is pretty much identical to this Perl snippet:

```
our $square = sub {
    my ($x) = @_;
    $x * $x;
};
```

In fact, Perl's anonymous `sub {...}` syntax can be considered pretty much synonymous with PScheme's `(lambda ...)`. The big difference is that in PScheme that's the *only* way to create a function[3].

Having created a `square` function, it can be called:

```
> (square 4)
16
```

Although `square` was created by assignment, when it is used it is syntactically indistinguishable from any built-in function.

Anonymous functions can be called directly without giving them a name first:

```
> ((lambda (x) (* x 2)) 3)
6
```

Again this is much simpler than it might first appear. The first term of the list expression, the `lambda` expression, gets evaluated resulting in a function. That function then gets applied to its argument `3` resulting in `6`. It *is* possible to do something similar in perl, like this:

[3] There are examples of Scheme code that show things like:

```
(define (square x) (* x x))
```

This form of `define`, where the expression being defined is a list, is just syntactic sugar for the underlying form. `define` essentially re-writes it into the simpler `lambda` statement before evaluating it. The second form of `define` is certainly a little bit easier to read, but personally I find that since I have to use `lambda` in some expressions anyway, it makes sense to always use it. Plus the syntactic sugar tends to obscure what is really going on. In any case PScheme does not support this alternative form of function definition.

```
sub { shift * 2 }->(3);
```

but it's not a common idiom.

3.6 Local Variables

Moving on, how can PScheme create local variables limited (lexically) to a given scope? This is done with the *let* special form. The general form of a `let` expression is:

```
(let (⟨binding⟩ ...) ⟨expression⟩)
```

where ⟨*binding*⟩ is:

```
(⟨symbol⟩ ⟨expression⟩)
```

For example:

```
> (let ((a 10)
>       (b 20))
>   (+ a b))
30
```

That can be read aloud as "let $a = 10$ and $b = 20$ in the expression $a + b$".

`let` takes a list of bindings (name-value pairs, each pair itself a list) and a body to execute with those bindings in effect. In this example `a` is given the value `10` and `b` the value `20` while the body is evaluated. However if a later expression was to ask for the value of `a` or `b` outside of the scope (the last closing brace) of the `let`, there would be an error (assuming there wasn't a global binding of `a` or `b` in effect.)

The carefull reader will have noticed that these were described as lexically scoped variables, and yes, any functions defined in the scope of those variables are closures just like Perl closures and have access to those variables when executed even if executed outside of that scope. For example:

```
> (define times2
>   (let ((n 2))
>     (lambda (x) (* n x))))
times2
> (times2 4)
8
```

When reading this it's useful to remember that `define` *does* evaluate its second argument. That means that this expression defines `times2` to be the *result* of evaluating the `let` expression. Now that `let` expression binds `n` to `2`, then returns the result of evaluating the `lambda` expression (creating a function) with that binding in effect. It is that newly created function that gets bound to the symbol `times2`. When `times2` is later used, for example in `(times2 4)`, the body of the function, `(* x n)`, can still "see" the value of `n` that was supplied by the `let`, even though the function is executed outside of that scope. This is similar to the common Perl trick to get a private static variable:

```
{
    my $n = 2;
    sub times2 {
        my ($x) = @_;
        $n * $x;
    }
}
```

but to be truthful it's closer to the more obtuse:

```
our $times2 = do {
    my $n = 2;
    sub {
        my ($x) = @_;
        $n * $x;
    }
};
```

And that's pretty much all that is needed for now. Of course the final language has many other interesting features, but these will be introduced in later sections as the need arises. Let's take a look at our first cut at an interpreter.

4 Interpreter Version 0.0.0

This preliminary version of the interpreter supports only three operations: multiplication (`*`); subtraction (`-`); and conditional evaluation (`if`). It does however lay the groundwork for more sophisticated interpreters later in the chapter.

Scheme lisp interpreters, being interactive, are based around what is called a “read-eval-print” loop. First read an expression, then evaluate it, then print the result. In order to evaluate the expression, there must be an environment in which symbols can be given values and in which values can be looked up. That means there are five principle components to such an interpreter:

A Reader that constructs internal representations of the expressions to be evaluated;

An Evaluator that actually determines the value of the expression, using

> **A Structure** returned by the Reader, representing the expression;
>
> **An Environment** in which symbols can be associated with values and the values of symbols can be looked up.

A Print System which converts the result of evaluation back to text and displays it to the user.

The implementation we’re about to discuss takes a fairly strict object-oriented approach, with each of these components and pretty much everything else represented by classes of objects. This means that for example to evaluate an expression you call its `Eval()` method, and to print a result you call the `Print()` method of the result object. There is a good deal of scope for polymorphism with this approach, since different types of object can respond differently to the same message.

4.1 The Read-Eval-Print Loop

The top-level read-eval-print loop (repl) for the PScheme interpreter is in the package `PScm` in Listing 1. All other packages inherit from this package, although that’s mainly just a convenience.
Firstly, on lines 30-34 a global environment, `$PScm::GlobalEnv` is initialized to a new `PScm::Env` object.

```
30 our $GlobalEnv = new PScm::Env(
31     ’*’ => new PScm::Primitive::Multiply(),
32     ’-’ => new PScm::Primitive::Subtract(),
33     if  => new PScm::SpecialForm::If(),
34 );
```

There are only three things in that environment. They are the objects that will perform the primitive operations of multiplication, subtraction and conditional evaluation, and they’re bound to `"*"`, `"-"` and `"if"` respectively. We’ll see how they work presently.

`ReadEvalPrint()` on lines 36-45 is the central control routine of the whole interpreter. It takes an input file handle and an output file handle as arguments.

```
package PScm;

use strict;
use warnings;
use PScm::Read;
use PScm::Env;
use PScm::Primitive;
use PScm::SpecialForm;

require Exporter;

our @ISA    = qw(Exporter);
our @EXPORT = qw(ReadEvalPrint);

=head1 NAME

PScm - Scheme-like interpreter written in Perl

=head1 SYNOPSIS

  use PScm;
  ReadEvalPrint($in_filehandle[, $out_filehandle]);

=head1 DESCRIPTION

Just messing about, A toy lisp interpreter.

=cut

our $GlobalEnv = new PScm::Env(
    ’*’ => new PScm::Primitive::Multiply(),
    ’-’ => new PScm::Primitive::Subtract(),
    if  => new PScm::SpecialForm::If(),
);

sub ReadEvalPrint {
    my ($infh, $outfh) = @_;

    $outfh ||= \*STDOUT;
    my $reader = new PScm::Read($infh);
    while (defined(my $expr = $reader->Read)) {
        my $result = $expr->Eval();
        $result->Print($outfh);
    }
}
```

listing 1: `PScm.pm`

```
46
47 sub Print {
48     my ($self, $outfh) = @_;
49     print $outfh $self->as_string, "\n";
50 }
51
52 sub as_string { ref($_[0]); }
53
54 sub new { bless {}, $_[0] }
55
56 1;
```

listing 1: PScm.pm (Cont.)

Starting on line 39 it defaults the output file handle to `STDOUT`, then on line 40 it creates a new `PScm::Read` object on the input file handle, and on lines 41-44 it enters its main loop. The loop repeatedly collects an expression from the Reader, then evaluates the expression by calling its `Eval()` method, then prints the result by calling its `Print()` method:

```
36 sub ReadEvalPrint {
37     my ($infh, $outfh) = @_;
38
39     $outfh ||= \*STDOUT;
40     my $reader = new PScm::Read($infh);
41     while (defined(my $expr = $reader->Read)) {
42         my $result = $expr->Eval();
43         $result->Print($outfh);
44     }
45 }
```

The basis of the print system can be seen in the `Print()` and `as_string()` methods in `PScm.pm`, but we're going to leave discussion of the print system until later on. In the next section we'll look at our first, very simple, implementation of an environment.

4.2 The Environment

All an environment has to do is to return the current value for an argument symbol. Perl hashes are ideal for this task, and our implementation uses them. Our environment is implemented by `PScm::Env` in Listing 2.
It is no more than an object wrapper around a Perl hash. The `new()` method (lines 7-13) creates an object with a set of bindings (name to value mappings) that were passed in as arguments:

```
07 sub new {
08     my ($class, %bindings) = @_;
09
10     bless {
```

```
package PScm::Env;

use strict;
use warnings;
use base qw(PScm);

sub new {
    my ($class, %bindings) = @_;

    bless {
        bindings => {%bindings},
    }, $class;
}

sub LookUp {
    my ($self, $symbol) = @_;

    if (exists($self->{bindings}{ $symbol->value })) {
        return $self->{bindings}{ $symbol->value };
    } else {
        die "no binding for @{[$symbol->value]} ",
            "in @{[ref($self)]}\n";
    }
}

1;
```

listing 2: `PScm/Env.pm`

```
11          bindings => {%bindings},
12      }, $class;
13  }
```

And the `LookUp()` method on lines 15-24 looks up a symbol in the bindings, `die`-ing if the symbol does not have a binding:

```
15  sub LookUp {
16      my ($self, $symbol) = @_;
17
18      if (exists($self->{bindings}{ $symbol->value })) {
19          return $self->{bindings}{ $symbol->value };
20      } else {
21          die "no binding for @{[$symbol->value]} ",
22              "in @{[ref($self)]}\n";
23      }
24  }
```

Note that the `$symbol` passed in is an object, and `LookUp()` must call the symbol's `value()` method to get a string suitable for a hash key. The `value()` method for a symbol just returns the name of the symbol as a perl string.

Because this first version of the interpreter has no support for local variables, this class doesn't provide any methods for adding values to the environment. That will come later.

And that's all there is to our environment class. Let's move on to look at the Reader.

4.3 The Reader

The job of the Reader is to take a stream of text and convert it into a structure that the evaluator can more easily work with. So for example we want to take an expression such as `(foo ("bar" 10))` and convert it into an equivalent structure such as shown in Figure 1.
In this figure, showing the result of parsing that expression, the top-level list object has two components. Reading left to right it contains the symbol object `foo` and another list object. This sub list contains the string object `"bar"` and the number object `10`. It is apparent that that the structure is a direct representation of the text, where each list corresponds to the contents of a matching pair of braces. It should also be obvious that these structures are practically identical to Perl list references. The scheme list `(foo ("bar" 10))` corresponds directly to the nested perl listref `[$foo, ["bar", 10]]`[4].

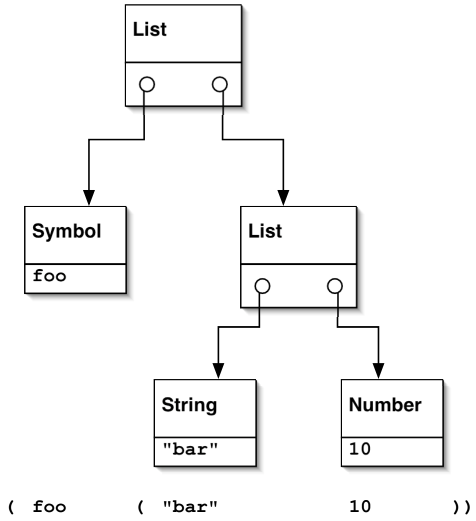
To simplify the creation of such a structure from an input stream, it is often convenient to split the process into two parts:

1. A tokeniser which recognizes and returns the basic tokens of the text (braces, symbols, numbers and strings);

2. A builder or parser which assembles those tokens into meaningful structures.

[4] but looks a lot prettier.

Figure 1: Example PScheme Expression Structure for `(foo ("bar" 10))`



That is the approach taken by the Reader described here. It was mentioned earlier that Scheme was extremely easy to parse, well here's the proof. The code for the Reader, `PScm::Read` in Listing 3 is only 63 lines long. As with the rest of the implementation, it uses an object-oriented style, so the Reader is an object that is created with an argument `FileHandle` and behaves as an iterator returning the next parsed expression from the stream on each call to `Read()`. First of all. the `new()` method (lines 9-15 simply stashes its input file handle argument along with an empty string representing the current line, and returns them in the new object.

```
09 sub new {
10     my ($class, $fh) = @_;
11     bless {
12         FileHandle => $fh,
13         Line       => '',
14     }, $class;
15 }
```

Apart from `new()` the only other publicly available method is `Read()`, which returns the next complete expression, as a structure, from the input file.

The `Read()` method calls the private `_next_token()` method (the tokeniser) for its tokens. Skipping over the `Read()` method for now, `_next_token()` on lines 38-61 simply chomps the next token off the input stream and returns it. It knows enough to skip whitespace and blank lines and to return undef at EOF (lines 41-45). If there is a line left to tokenize, then a few simple regexes are tried in turn to strip the next token from it. As soon as a token of a particular type is recognized, it is returned to the caller.

```
38 sub _next_token {
```

```
package PScm::Read;

use strict;
use warnings;
use PScm::Expr;
use PScm::Token;
use base qw(PScm);

sub new {
    my ($class, $fh) = @_;
    bless {
        FileHandle => $fh,
        Line       => ’’,
    }, $class;
}

sub Read {
    my ($self) = @_;

    my $token = $self->_next_token();
    return undef  unless defined $token;

    return $token unless $token->isa(’PScm::Token::Open’);

    my @res = ();

    while (1) {
        $token = $self->Read;
        die "unexpected EOF"
             if !defined $token;
        last if $token->isa(’PScm::Token::Close’);
        push @res, $token;
    }

    return new PScm::Expr::List(@res);
}

sub _next_token {
    my ($self) = @_;

    while (!$self->{Line}) {
        $self->{Line} = $self->{FileHandle}->getline();
        return undef unless defined $self->{Line};
        $self->{Line} =˜ s/ˆ\s+//s;
    }
```

listing 3: PScm/Read.pm

```
46
47     for ($self->{Line}) {
48         s/^\(\s*// && return PScm::Token::Open->new();
49         s/^\)\s*// && return PScm::Token::Close->new();
50         s/^([-+]?\d+)\s*//
51           && return PScm::Expr::Number->new(0 + $1);
52         s/^"((?:(?:\\.)|([^"]))*)"\s*// && do {
53             my $string = $1;
54             $string =~ s/\\//g;
55             return PScm::Expr::String->new($string);
56         };
57         s/^([^\s\(\)]+)\s*//
58           && return PScm::Expr::Symbol->new($1);
59     }
60     die "can't parse: $self->{Line}";
61 }
62
63 1;
```

listing 3: `PScm/Read.pm` (Cont.)

```
39     my ($self) = @_;
40
41     while (!$self->{Line}) {
42         $self->{Line} = $self->{FileHandle}->getline();
43         return undef unless defined $self->{Line};
44         $self->{Line} =~ s/^\s+//s;
45     }
46
47     for ($self->{Line}) {
48         s/^\(\s*// && return PScm::Token::Open->new();
49         s/^\)\s*// && return PScm::Token::Close->new();
50         s/^([-+]?\d+)\s*//
51           && return PScm::Expr::Number->new(0 + $1);
52         s/^"((?:(?:\\.)|([^"]))*)"\s*// && do {
53             my $string = $1;
54             $string =~ s/\\//g;
55             return PScm::Expr::String->new($string);
56         };
57         s/^([^\s\(\)]+)\s*//
58           && return PScm::Expr::Symbol->new($1);
59     }
60     die "can't parse: $self->{Line}";
61 }
```

Lines 47-59 do the actual tokenisation. The tokeniser only needs to distinguish open and close braces, numbers, strings and symbols, where anything that doesn't look like an open or close brace, a number or a string must be a symbol. `_next_token()` returns its data in objects, which incidentally happens to be a very convenient way of tagging the type of token returned. The objects are of two basic types: PScm::Token; and PScm::Expr.

The PScm::Token types PScm::Token::Open and PScm::Token::Close represent an open and a close brace respectively, and contain no data. The three PScm::Expr types, PScm::Expr::Number, PScm::Expr::String and PScm::Expr::Symbol (numbers, strings and symbols) contain the relevant number, string or symbol.

Now that we know how _get_token() works, we can go back and take a look at Read().

The Read() method (lines 17-36) has to return the next complete expression from the input stream. That could be a simple atom, or an arbitrarily nested list. It starts by calling _next_token() at line 20 and returning undef if _next_token() returned undef (signifying end of file).

```
17 sub Read {
18     my ($self) = @_;
19
20     my $token = $self->_next_token();
21     return undef  unless defined $token;
22
23     return $token unless $token->isa('PScm::Token::Open');
24
25     my @res = ();
26
27     while (1) {
28         $token = $self->Read;
29         die "unexpected EOF"
30              if !defined $token;
31         last if $token->isa('PScm::Token::Close');
32         push @res, $token;
33     }
34
35     return new PScm::Expr::List(@res);
36 }
```

Then, at line 23 if the token is anything other than an open brace (a PScm::Token::Open), Read() just returns it. Otherwise, the token just read is an open brace, so Read() initializes an empty result @res to hold the list it expects to accumulate then enters a loop calling itself recursively to collect the (possibly nested) components of the list. It is an error if it detects EOF while a list is unclosed, and if it detects a close brace it knows its work is done and it returns the accumulated list as a new PScm::Expr::List object.

The structure returned by Read() is completely composed of subtypes of PScm::Expr, since the PScm::Token types do not actually get entered into the structure. Let's work through the parsing of that simple expression (foo ("bar" 10)). The subscript number keeps track of which particular invocation of Read() we are talking about.

- Read$_1$ calls _next_token() and gets a '(' so it enters its loop.

- Read$_1$ calls Read$_2$ from within its loop.

  - `Read`$_2$ calls `_next_token()` and gets a `foo`, so it returns it.

- `Read`$_1$ puts the `foo` at the start of its list: `[foo`.
- `Read`$_1$ calls `Read`$_3$.
  - `Read`$_3$ calls `_next_token` and gets a `'('` so it enters its loop.
  - `Read`$_3$ calls `Read`$_4$.
    - `Read`$_4$ calls `_next_token` and gets a `"bar"` so it returns it.
  - `Read`$_3$ puts the `"bar"` at the start of its list: `["bar"`.
  - `Read`$_3$ calls `Read`$_5$.
    - `Read`$_5$ calls `_next_token` and gets a `10` so it returns it.
  - `Read`$_3$ adds the `10` to its growing list: `["bar" 10`.
  - `Read`$_3$ calls `Read`$_6$.
    - `Read`$_6$ calls `_next_token` and gets a `')'` so it returns it.
  - `Read`$_3$ gets the `')'` so it knows it has reached the end of its list and returns it: `["bar" 10]`.
- `Read`$_1$ adds the `["bar" 10]` to the end of its own growing list: `[foo ["bar" 10]`.
- `Read`$_1$ calls `Read`$_7$.
  - `Read`$_7$ calls `_next_token` and gets a `')'` so it returns it.
- `Read`$_1$ gets the `')'` so it knows it has reached the end of its list and returns it: `[foo ["bar" 10]]`.

So the Reader does indeed return the structure expected.

The `PScm::Token` and `PScm::Expr` classes are in their eponymous files. The `PScm::Token` classes in Listing 4 are purely parse-related. As mentioned earlier, they are returned by the tokeniser to indicate open and close braces. These tokens are used to guide the parser, but it does not actually include them in the result. `PScm::Token::Open` and `PScm::Token::Close` both inherit from `PScm::Token`. There are no methods in any of those three classes. `PScm::Token` inherits a stub `new()` method from the `PScm` class that just blesses an empty hash with the argument class.

As for the `PScm::Expr` objects that `Read()` accumulates and returns, as noted `Read()` has done all of the work in constructing a tree of them for us, so they are more properly discussed in the next section where we look at expressions.

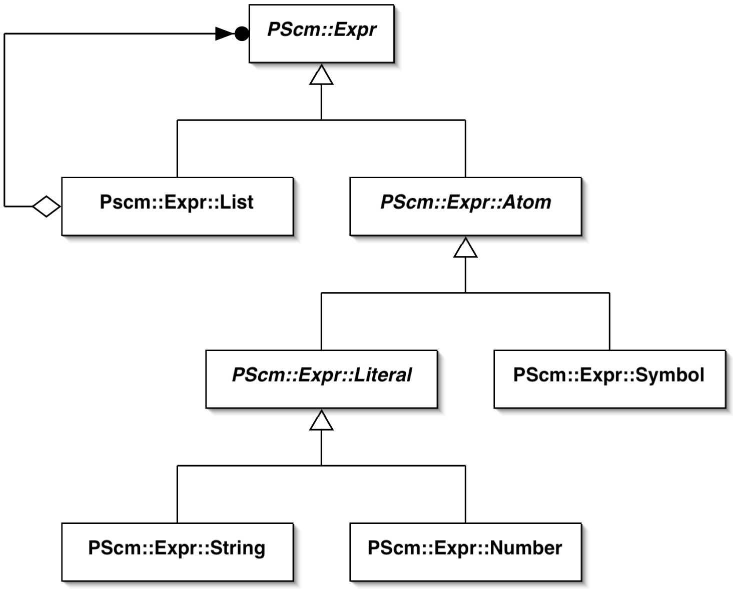
4.4 PScheme Expressions

The various `PScm::Expr` objects are defined in `PScm/Expr.pm`. These objects represent the basic data types that are visible to the user: strings; numbers; symbols; and lists (expressions). They are the types returned by the Reader and printed by the print system. It would be premature to go into all the details of the `PScm::Expr` package right now, but it is worth pointing out a few salient features about it.

```
package PScm::Token;

use strict;
use warnings;
use base qw(PScm);

###########################
package PScm::Token::Open;

use base qw(PScm::Token);

#########################
package PScm::Token::Close;

use base qw(PScm::Token);

1;
```

listing 4: `PScm/Token.pm`

Figure 2: `PScm::Expr` classes



Firstly the classes arrange themselves in a Composite Pattern according to the hierarchy of PScheme types as in Figure 2.
This figure is drawn using a standard set of conventions for diagramming the relationships between classes in an object-oriented design, called “the Unified Modelling Language”, or UML.

For those who don't know UML, the white triangular shape means "inherits from" or "is a subclass of", and the black arrow and circle coming from the white diamond means "aggregates one or more of". The classes with names in italics are "abstract" classes. As far as Perl is concerned, calling a class "abstract" just means that we promise not to create any actual object instances of that particular class.

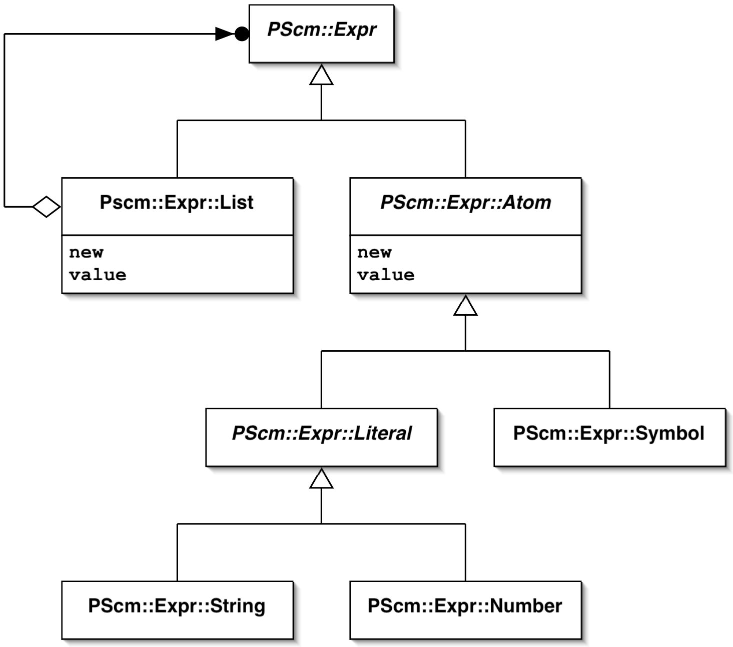
The root of the hierarchy is `PScm::Expr`, representing any and all expressions. That divides into lists (`PScm::Expr::List`) and atoms (`PScm::Expr::Atom`).

Lists are composed of expressions (the aggregation relationship.)

Atoms represent any data type that cannot be trivially taken apart, anything that's not a list in other words. Atoms are subclassed into literals (`PScm::Expr::Literal`) and symbols (`PScm::Expr::Symbol`), and literals are subclassed into strings (`PScm::Expr::String`) and numbers (`PScm::Expr::Number`).

We'll see a lot of this diagram in various guises as we progress. Here's the same diagram, with the location of the `new()` and `value()` methods added.

Figure 3: `PScm::Expr` `new()` and `value()` methods



This brings us to the second salient point. There are only two `new()` methods in the whole class structure. The `PScm::Expr::Atom` abstract class is the parent class for strings and numbers (via `PScm::Expr::Literal`) and for symbols. Since all of these types are simple scalars, the `new()` method in `PScm::Expr::Atom` does for all of them: it just blesses a reference to the scalar into the appropriate class.

```
sub new {
    my ($class, $value) = @_;
    bless \$value, $class;
}
```

The `PScm::Expr::List` class has the other `new()` method that simply bundles up its argument Perl list in a new object:

```
sub new {
    my ($class, @list) = @_;

    $class = ref($class) || $class;
    bless [@list], $class;
}
```

Both of these methods have already been seen in action in the Reader.

Alongside each `new()` method is an analogous `value()` method that does the exact reverse of `new()` and retrieves the underlying value from the object. In the case of atoms, it dereferences the scalar value:

```
sub value { ${ $_[0] } }
```

and in the case of lists, it dereferences the list:

```
sub value { @{ $_[0] } }
```

We've seen that the various PScheme expression types (lists, numbers, strings and symbols) arrange themselves naturally into a hierachy of types and also form a recognised design pattern called "Composite". Next we're going to look at how those expressions are evaluated by the interpreter.

4.5 Evaluation

To evaluate a `PScm::Expr` structure, as mentioned earlier, the top level `ReadEvalPrint()` loop just calls the structure's `Eval()` method. The `Eval()` methods of `PScm::Expr` are located in three of its subclasses as shown in Figure 4.

The figure shows that there is a separate `Eval()` method for lists, literals and symbols. Let's look first at the `Eval()` methods for literals. `PScm::Expr::String` and `PScm::Expr::Number` are both subclasses of `PScm::Expr::Literal` and the `Eval()` method there, which they share, just returns `$self`:

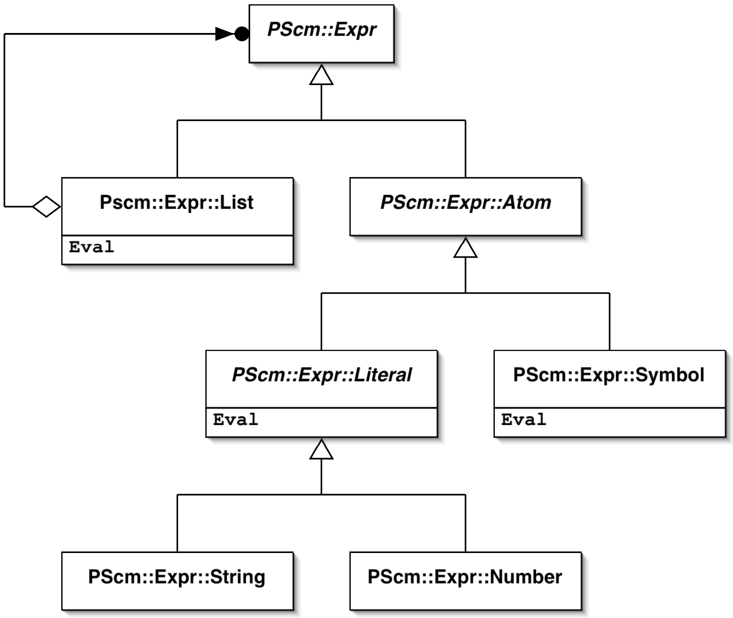
```
sub Eval {
    my ($self) = @_;
    return $self;
}
```

This means that numbers and strings evaluate to themselves, as they should.

Evaluation of a symbol is only slightly more complex. The `Eval()` method in `PScm::Expr::Symbol` looks up its value in the global environment `$PScm::GlobalEnv`:

```
sub Eval {
    my ($self) = @_;
    return $PScm::GlobalEnv->LookUp($self);
}
```

Figure 4: `PScm::Expr` `Eval()` Methods



Remember that `LookUp()` expects a symbol object as argument and calls its `value()` method to get a string that it can then use to retrieve the actual value from the environment.

Before showing how `PScm::Expr::List` objects are evaluated, we need to consider a couple of support methods for lists, `first()` and `rest()`.

The `first()` method of `PScm::Expr::List` just returns the first component of the list:

```
sub first { $_[0][0] }
```

The `rest()` method of `PScm::Expr::List` returns all but the first component of the list as a new `PScm::Expr::List` object:

```
sub rest {
    my ($self) = @_;

    my @value = $self->value;
    shift @value;
    return $self->new(@value);
}
```

Now we can look at the evaluation of list expressions. Here's `PScm::Expr::List::Eval()`:

```
sub Eval {
    my ($self) = @_;
    my $op = $self->first()->Eval();
58      return $op->Apply($self->rest);
59  }
```

It's surprisingly simple. a PScm::Expr::List just evaluates its first element (line 57). That should return one of PScm::Primitive::Multiply, PScm::Primitive::Subtract or PScm::SpecialForm::If, which gets assigned to $op. Of course because we're not doing any error checking, first() could return anything, so we're assuming valid input.

Because Eval() does not know or care whether the operation $op it derived on line 57 is a simple primitive or a special form, on line 58 it passes the rest of itself unevaluated to that operations Apply() method which decides whether or not to evaluate the arguments, and what to do with them afterwards[5].

So we've seen how PScm::Expr objects evaluate themselves. In particular we've seen how a list evaluates itself by evaluating its first component to get a primitive operation or special form, then calling that object's Apply() method with the rest of the list, unevaluated, as argument. Next we're going to look at how one of those types, the PScm::Primitive type, implements that Apply() method.

4.6 Primitive Operations

The primitive built-in functions all live in PScm/Primitive.pm, shown in Listing 5
This class holds all of the code for simple functions that can be passed already evaluated arguments. It would be fairly trivial to add a lot of functionality in here, but it is deliberately kept to only a bare minimum of functions so that the code is short and easy to digest.

The base PScm::Primitive class provides the Apply() method:

```
sub Apply {
    my ($self, $form) = @_;

    my @unevaluated_args = $form->value;
    my @evaluated_args = map { $_->Eval() } @unevaluated_args;
    return $self->_apply(@evaluated_args);
}
```

This extracts the arguments to the operation from the $form by calling the $form's value() method. $form is a PScm::Expr::List and we've already seen that the value() method for a list object dereferences and returns the underlying list. Apply() evaluates each argument by mapping a call to each one's Eval() method and then passes the resulting list to a private _apply() method.

_apply() is implemented differently by each primitive operation. So each subclass of PScm::Primitive only needs an _apply() method which will be called with a list of already evaluated arguments.

[5] Lisp purists might raise an eyebrow at this point, because Eval() is *supposed* to know what kind of form it is evaluating and decide whether or not to evaluate the arguments. But this is an object-oriented application, and it makes much more sense to leave that decision to the objects that need to know.

```perl
package PScm::Primitive;

use strict;
use warnings;
use base qw(PScm);
use PScm::Expr;

sub Apply {
    my ($self, $form) = @_;

    my @unevaluated_args = $form->value;
    my @evaluated_args = map { $_->Eval() } @unevaluated_args;
    return $self->_apply(@evaluated_args);
}

sub _check_type {
    my ($self, $thing, $type) = @_;

    die "wrong type argument(", ref($thing),
        ") to ", ref($self), "\n"
            unless $thing->isa($type);
}

##################################
package PScm::Primitive::Multiply;

use base qw(PScm::Primitive);

sub _apply {
    my ($self, @args) = @_;

    my $result = 1;

    while (@args) {
        my $arg = shift @args;
        $self->_check_type($arg, 'PScm::Expr::Number');
        $result *= $arg->value;
    }

    return new PScm::Expr::Number($result);
}

##################################
package PScm::Primitive::Subtract;

```

listing 5: PScm/Primitive.pm

```
use base qw(PScm::Primitive);

sub _apply {
    my ($self, @args) = @_;

    unshift @args, PScm::Expr::Number->new(0) if @args < 2;

    my $arg = shift @args;
    $self->_check_type($arg, 'PScm::Expr::Number');

    my $result = $arg->value;

    while (@args) {
        $arg = shift @args;
        $self->_check_type($arg, 'PScm::Expr::Number');
        $result -= $arg->value;
    }

    return new PScm::Expr::Number($result);
}

1;
```

listing 5: PScm/Primitive.pm (Cont.)

The _apply() in PScm::Primitive::Multiply is very straightforward. It simply multiplies its arguments together and returns the result as a new PScm::Expr::Number. Note that, somewhat accidentally, if only given one argument it will simply return it, and if given no arguments it will return 1.

```
sub _apply {
    my ($self, @args) = @_;

    my $result = 1;

    while (@args) {
        my $arg = shift @args;
        $self->_check_type($arg, 'PScm::Expr::Number');
        $result *= $arg->value;
    }

    return new PScm::Expr::Number($result);
}
```

The _check_type() method in the base class just saves us some typing, since we are checking the type of argument to the primitive:

```
sub _check_type {
    my ($self, $thing, $type) = @_;
18
19      die "wrong type argument(", ref($thing),
20          ") to ", ref($self), "\n"
21              unless $thing->isa($type);
22  }
```

PScm::Primitive::Subtract's _apply() method is more complicated only because it distinguishes between unary negation (- x) and subtraction. If it gets only one argument it returns its negation, otherwise it subtracts subsequent arguments from the first one. It will return 0 if called with no arguments.

```
48  sub _apply {
49      my ($self, @args) = @_;
50
51      unshift @args, PScm::Expr::Number->new(0) if @args < 2;
52
53      my $arg = shift @args;
54      $self->_check_type($arg, 'PScm::Expr::Number');
55
56      my $result = $arg->value;
57
58      while (@args) {
59          $arg = shift @args;
60          $self->_check_type($arg, 'PScm::Expr::Number');
61          $result -= $arg->value;
62      }
63
64      return new PScm::Expr::Number($result);
65  }
```

That's all the primitive operations we support. There are a whole host of others that could trivially be added here and it might be entertaining to add them, but the really interesting stuff will all be happening over in the special forms, discussed next.

4.7 Special Forms

All the code for special forms is in PScm/SpecialForm.pm in Listing 6.
At the moment there is only one special form, if, so the listing is short. It will get longer in subsequent versions though.

For special forms, the Apply() method is in the individual operation's class. On line 16 PScm::SpecialForm::If's Apply() method extracts the condition, the expression to evaluate if the condition is true, and the expression to evaluate if the condition is false, from the argument $form. Then on line 18 it evaluates the condition, and calls the result's isTrue() method to determine which branch to evaluate:

```
12  sub Apply {
13      my ($self, $form) = @_;
14
15      my ($condition, $true_branch, $false_branch) =
package PScm::SpecialForm;

use strict;
use warnings;
use base qw(PScm);

##############################
package PScm::SpecialForm::If;

use base qw(PScm::SpecialForm);

sub Apply {
    my ($self, $form) = @_;

    my ($condition, $true_branch, $false_branch) =
                                        $form->value;

    if ($condition->Eval()->isTrue) {
        return $true_branch->Eval();
    } else {
        return $false_branch->Eval();
    }
}

1;
```

listing 6: PScm/SpecialForm.pm

```
16                                         $form->value;
17
18     if ($condition->Eval()->isTrue) {
19         return $true_branch->Eval();
20     } else {
21         return $false_branch->Eval();
22     }
23 }
```

If the condition is true, PScm::SpecialForm::If::Apply() evaluates and returns the true branch (line 19), otherwise it evaluates and returns the false branch (line 21). The decision of what is true or false is delegated to an isTrue() method. The one and only isTrue() method is defined in PScm/Expr.pm right at the top of the data type hierarchy, in the PScm::Expr class as:

```
sub isTrue {
    my ($self) = @_;
    scalar($self->value);
}
```

Remembering that value() just dereferences the underlying list or scalar, isTrue() then pretty much agrees with Perl's idea of truth, namely that zero, the empty string, and the empty list are false, everything else is true[6].

4.8 Output

Now we're going to take a look at the print system.

After Eval() returns the result to the repl, ReadEvalPrint() calls the result's Print() method with the output handle as argument. That method is defined in PScm.pm

```
sub Print {
    my ($self, $outfh) = @_;
    print $outfh $self->as_string, "\n";
}
```

All it does is print the string representation of the object obtained by calling its as_string() method. A fallback as_string() method is provided in this class at line 52.

```
52 sub as_string { ref($_[0]); }
```

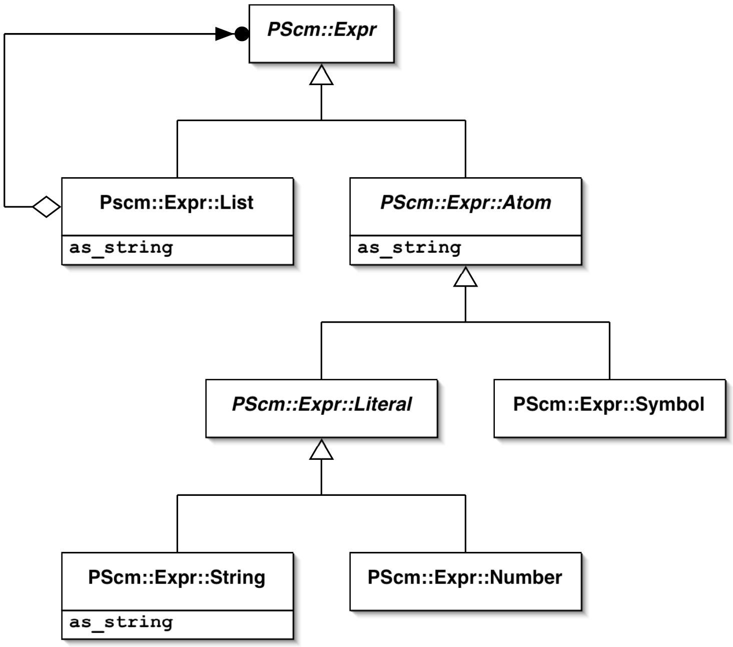
It just returns the class name of the object. This is needed occasionally in the case where internals such as primitive operations might be returned by the evaluator, for example:

```
> *
PScm::Primitive::Multiply
```

But that is an unusual and usually unintentional situation. The main as_string() methods are strategically placed around the by now familiar PScm::Expr hierarchy, as shown in Figure 5.
The as_string() method in PScm::Expr::Atom is just a call to value():

[6] This differs from a true Scheme implementation where special boolean values #t and #f represent truth and falsehood, and *everything* else is true.

Figure 5: `PScm::Expr` `as_string()` methods



```
23 sub as_string { $_[0]->value }
```

That method works for both symbols and numbers.

`PScm::Expr::List`'s `as_string()` method returns a string representation of the list by recursively calling `as_string()` on each of its components and concatenating the result, separated by spaces and wrapped in braces[7].

```
48 sub as_string {
49     my ($self) = @_;
50     return '(' .
51            join(' ', map { $_->as_string } $self->value) .
52            ')';
53 }
```

Finally, `PScm::Expr::String`'s `as_string()` method at lines 87-94 overrides the one in `PScm::Expr::Atom` because it needs to put back any backslashes that the parser took out, and wrap itself in double quotes.

```
87 sub as_string {
88     my ($self) = @_;
89 
90     my $copy = $self->value;
91     $copy =~ s/\\/\\\\/sg;
```

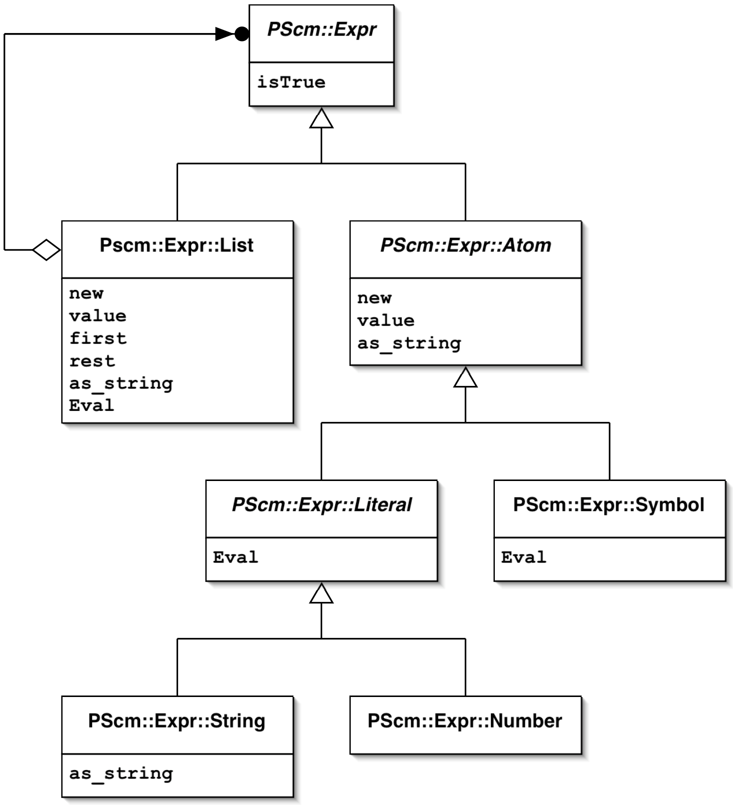
[7] We haven't seen anything yet that might, when evaluated, return a list for printing. That's for later.

```
    $copy =~ s/"/\\"/sg;
    return qq'"$copy"';
}
```

4.9 Summary

We're finally in a position to understand the whole of `PScm::Expr` as shown in Listing 7.
The final version of our diagram, with all of the methods from `PScm::Expr` in place is shown in Figure 6.

Figure 6: `PScm::Expr` methods



That may seem like a lot of code for what is effectively just a pocket calculator, but what has been done is to lay the groundwork for a much more powerful set of language constructs that can be added in subsequent sections. Let's recap with an overview of the whole thing.

- A global environment is set up in the PScm package, containing bindings for defined operations.

- The top-level read-eval-print loop (repl) in the PScm package creates a `PScm::Read` object and calls its `Read()` method.

```
package PScm::Expr;

use strict;
use warnings;
use base qw(PScm);

sub isTrue {
    my ($self) = @_;
    scalar($self->value);
}

##########################
package PScm::Expr::Atom;
use base qw(PScm::Expr);

sub new {
    my ($class, $value) = @_;
    bless \$value, $class;
}

sub value { ${ $_[0] } }

sub as_string { $_[0]->value }

##########################
package PScm::Expr::List;
use base qw(PScm::Expr);

sub new {
    my ($class, @list) = @_;

    $class = ref($class) || $class;
    bless [@list], $class;
}

sub value { @{ $_[0] } }

sub first { $_[0][0] }

sub rest {
    my ($self) = @_;

    my @value = $self->value;
    shift @value;
    return $self->new(@value);
```

listing 7: PScm/Expr.pm

```
}

sub as_string {
    my ($self) = @_;
    return '(' .
           join(' ', map { $_->as_string } $self->value) .
           ')';
}

sub Eval {
    my ($self) = @_;
    my $op = $self->first()->Eval();
    return $op->Apply($self->rest);
}

############################
package PScm::Expr::Symbol;
use base qw(PScm::Expr::Atom);

sub Eval {
    my ($self) = @_;
    return $PScm::GlobalEnv->LookUp($self);
}

#############################
package PScm::Expr::Literal;
use base qw(PScm::Expr::Atom);

sub Eval {
    my ($self) = @_;
    return $self;
}

############################
package PScm::Expr::Number;
use base qw(PScm::Expr::Literal);

############################
package PScm::Expr::String;
use base qw(PScm::Expr::Literal);

sub as_string {
    my ($self) = @_;

```

listing 7: PScm/Expr.pm (Cont.)

```
    my $copy = $self->value;
    $copy =~ s/\\/\\\\/sg;
    $copy =~ s/"/\\"/sg;
    return qq'"$copy"';
}

1;
```

listing 7: `PScm/Expr.pm` (Cont.)

- That `Read()` method returns `PScm::Expr` objects which the repl evaluates. It evaluates them by calling their `Eval()` method.
  - `PScm::Expr::Number` and `PScm::Expr::String` objects both share an `Eval()` method that just returns the object unevaluated.
  - `PScm::Expr::Symbol` objects have an `Eval()` method that looks up the value of the symbol in the environment.
  - `PScm::Expr::List` objects have an `Eval()` method that evaluates the first component of the list, which should return a primitive operation, then calls that operations `Apply()` method with the remaining unevaluated components of the list as argument. What happens next depends on the type of the operation.
    - `PScm::Primitive` objects share an `Apply()` method that evaluates each of the arguments and then passes them to the individual primitive's private `_apply()` method.
    - `PScm::SpecialForm` objects each have their own `Apply()` method that decides whether, and how, to evaluate the arguments.
- The repl then takes the result of the evaluation and calls its `Print()` method, which is defined in the `PScm` base class.
  - That `Print()` method just calls `$self->as_string()` and prints the result. The `PScm::Expr::Atom` class has an `as_string()` method that returns the underlying scalar, but `PScm::Expr::String` provides an override that wraps the result in double quotes. The `PScm::Expr::List` class has an `as_string()` method that recursively calls `as_string()` on its components and returns the result wrapped in braces.

At the heart of the whole interpreter is the dynamic between `Eval()` which evaluates expressions, and `Apply()` which applies operations to their arguments.

4.10 Tests

Here's a test module for our first version of the interpreter in Listing 8. The evaluate sub just takes a string expression, writes it out to a file, and calls `ReadEvalPrint()` on it, with the output redirected to another file. It then reads that

```
use strict;
use warnings;
use Test::More tests => 9;
use FileHandle;

BEGIN { use_ok(’PScm’) };

sub evaluate {
    my ($expression) = @_;

    my $fh = new FileHandle("> junk");
    $fh->print($expression);
    $fh = new FileHandle(’< junk’);
    my $outfh = new FileHandle("> junk2");
    ReadEvalPrint($fh, $outfh);
    $fh = 0;
    $outfh = 0;
    my $res = ‘cat junk2‘;
    chomp $res;
    unlink(’junk’);
    unlink(’junk2’);
    # warn "# [$res]\n";
    return $res;
}

ok(evaluate(’1’) eq ’1’, ’numbers’);
ok(evaluate(’+1’) eq ’1’, ’explicit positive numbers’);
ok(evaluate(’-1’) eq ’-1’, ’negative numbers’);
ok(evaluate(’"hello"’) eq ’"hello"’, ’strings’);
ok(evaluate(’(* 2 3 4)’) eq ’24’, ’multiplication’);
ok(evaluate(’(- 10 2 3)’) eq ’5’, ’subtraction’);
ok(evaluate(’(- 10)’) eq ’-10’, ’negation’);
ok(evaluate(’(if (* 0 1) 10 20)’) eq ’20’, ’simple conditional’);

# vim: ft=perl
```

listing 8: `t/PScm.t`

output back in and returns it as a string. The various simple tests just exercise the system.

To allow users to play a little more with the interpreter, there’s a tiny interactive shell that requires `Term::ReadLine::Gnu` and the `libreadline` library. It’s in `t/interactive` and can be run, without installing the interpreter, by doing:

```
$ perl -Ilib ./t/interactive
```

from the root of any version of the distribution. It’s short enough to show here in its entirety.

```
use PScm;

package GetLine;

use Term::ReadLine;
sub new {
    my ($class) = @_;
    bless {
        term => new Term::ReadLine(’PScheme’),
    }, $class;
}

sub getline {
    my ($self) = @_;
    $self->{term}->readline(’> ’);
}

package main;

my $in = new GetLine();

ReadEvalPrint($in);

# vim: ft=perl
```

listing 9: `t/interactive`

5 Interpreter Version 0.0.1—Implementing `let`

`let` allows the extension of the environment, temporarily, to include new bindings of symbols to data. Of course the Environment that has been described so far is not extensible, so the first thing to do is to look at how we might change the environment package to allow extension.

5.1 The Environment

Remember the original environment implementation from Listing 2, where we just created an object wrapper around a Perl hash. We can build on this idea, but we need to think a bit harder about what it means to extend an environment. It would be a good idea to keep the extensions separate from what is already in the environment, so that they can be easily undone when the time comes. It's a really bad idea to just poke more key-value pairs into that hash; the cost of working out what, and how, to undo those changes could be prohibitive. Therefore, each extension should have its own object hash.

A useful distinction to make at this point is between any individual hash, and the environment as a whole. When the text refers to the environment as a whole It'll just say "the environment", but when It's talking about a particular object hash component, It'll say "environment frame", or just "frame".

A simple extension then, and one which a number of programming languages do in fact implement, is a stack of environment frames. A new frame containing the new bindings is pushed on top of the old, and the `LookUp()` method starts at the top of the stack and works its way down until it either finds a binding for the argument symbol, or hits the bottom of the stack and signals an error. To restore the previous environment, the top frame is simply popped off the stack again.

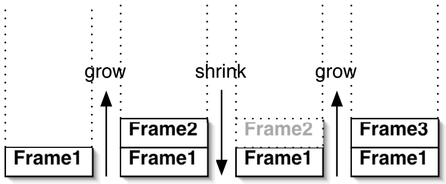
Perl lists have `push` and `pop` operations, so we could easily use those. Alternatively we could keep a current "top of stack" index, and increment that to push, or decrement it to pop, something like:

```
sub push {
    my ($self, $frame) = @_;
    $self->{stack}[$self->{index}] = $frame;
    ++$self->{index};
}

sub pop {
    my ($self) = @_;
    --$self->{index};
    die "stack underflow" if $self->{index} < 0;
    return $self->{stack}[$self->{index}];
}
```

This has the minor advantage of not immediately loosing what was previously on the top of the stack after a `pop()`.

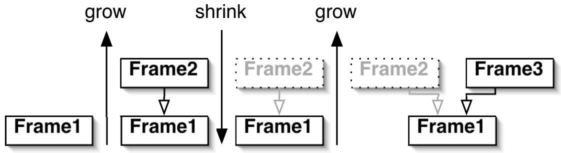
The major drawback of a stack is that it is a linear structure, and extending the stack again necessarily obliterates what was previously there, see Figure 7. If we plan at a later stage to support closure, where functions hang on to their environments after control has left them, then a stack is obviously inadequate unless some potentially complex additional code protects and copies those vunerable environment frames.

Figure 7: Stacks Destroy Old Environment Frames



Enter the linked list. A linked list is just a collection of objects, hashes or whatever, where each one contains a reference to the next one on the list. If an environment, rather than being a stack of frames, was a linked list of frames, then just as with a stack, `PScm::Env::LookUp()` need only walk through the chain until it finds the first (most local) occurrence of the symbol and return that. The advantage of a linked list is that many environment frames can share the same parent, and creating a new child frame does not destroy the previous frame, see Figure 8.

Figure 8: Linked Lists Don't Destroy Old Environment Frames



As long as something continues to hold a reference to the old **Frame2** in this figure, then it will not be garbage collected and remains as valid as any other environment.

Here's `PScm::Env::LookUp()` modified to use a linked list.

```
29 sub LookUp {
30     my ($self, $symbol) = @_;
31
32     if (exists($self->{bindings}{ $symbol->value })) {
33         return $self->{bindings}{ $symbol->value };
34     } elsif ($self->{parent}) {
35         return $self->{parent}->LookUp($symbol);
36     } else {
37         die "no binding for @{[$symbol->value]} ", "in @{[ref($self)]}\n";
38     }
39 }
```

The only change is on lines 34–35 where if `LookUp()` can't find the symbol in the current environment frame it looks in its parent frame, if it has one.

The `PScm::Env::new()` method is little changed, it additionally checks the argument class in case `new()` is being called as an object method, and adds a parent field to the object, with an initial zero value meaning “no parent”.

```
07 sub new {
08     my ($class, %bindings) = @_;
09
10     $class = ref($class) || $class;
11     bless {
12         bindings => {%bindings},
13         parent   => 0,
14     }, $class;
15 }
```

Finally we need an `Extend()` method that will create a new environment from an existing one by creating a new frame with the new bindings, and setting the new frame’s parent to be the original environment.

```
17 sub Extend {
18     my ($self, $ra_symbols, $ra_values) = @_;
19
20     my %bindings = ();
21     my @names = map { $_->value } @$ra_symbols;
22     my @values = map { $_->Eval($self) } @$ra_values;
23     @bindings{ @names } = @values;
24     my $new = $self->new(%bindings);
25     $new->{parent} = $self;
26     return $new;
27 }
```

Because the `Extend()` method will be used by `let` it takes a reference to an array of symbols and a reference to an array of values, rather than the simple `%initial` hash that `new()` takes. It maps the symbols to a list of strings (line 21) and the values to a list of their evaluated equivalents (line 22). Then on line 23 it populates a hash with these and on line 24, creates a new environment with that hash. Finally on line 25 it sets that new environment’s parent to be the original environment `$self` and returns the new environment.

We’ll se what that passing of `$self` to `Eval()` on line 22 is about in the next section.

5.2 Global Environments have a Problem

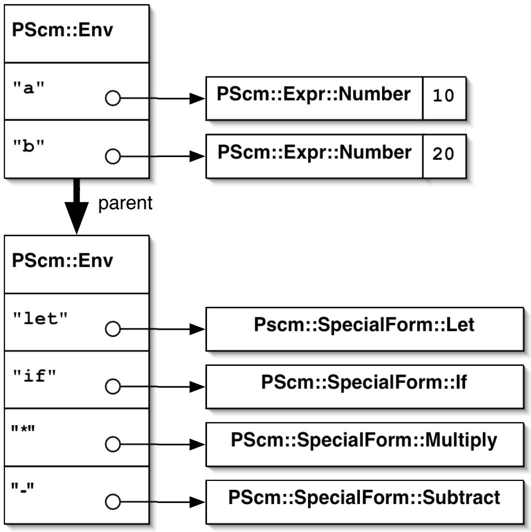
Assume our interpreter already has `let` installed as `PScm::SpecialForm::Let`, and is about to evaluate the `(+ a b)` part of the expression

```
(let ((a 10)
      (b 20))
     (+ a b))
```

It will have already extended the environment with the bindings for `a` and `b` so the global environment at that point will look like Figure 9.

Now, consider what `let` might have had to do to extend the environment and might have to do to restore it again afterwards:

Figure 9: Environment during evaluation of example "`let`"



1. Save the current value of `$PScm::GlobalEnv`;

2. Call `Extend()` on `$PScm::GlobalEnv` to get a new one with `a` and `b` appropriately bound;

3. Assign that new environment to `$PScm::GlobalEnv`;

4. Call `Eval()` on the expression `(+ a b)` and save the result;

5. Restore the previous value of `$PScm::GlobalEnv`;

6. Return the result of evaluating the body.

There's something not quite right there, something ugly. We've made it the responsibility of `let` to restore that previous environment, and if we go down that road, all of the other operations that extend environments will similarly be required to restore the environment for their callers. There's another bit of ugliness too, the simple existence of a global variable. It's the only one in our application. Does it have to be there? What could replace it?

5.3 Environment Passing

You are asked to take a leap of faith here when it is suggested that a better mechanism is to pass the environment around between `Eval()` and `Apply()` within the interpreter. Just suppose that `Eval()` was given the current environment as argument along with the expression to evaluate. Furthermore suppose that it passed the environment to `Apply()`. Now we've already seen that Special Forms (of which `let` is one,) each have their own `Apply()` method, so how would `let`'s `Apply()` look if the environment were passed in? It would:

1. Call `Extend()` on the argument environment to get a new one with `a` and `b` appropriately bound;
2. Return the result of calling `Eval()` on the expression `(+ a b)` with the new environment as argument.

Isn't that better![8]

Because environments would be local (`my`) variables, and because `PScm::Env::Extend()` does not alter the original environment in any way, we can rely Perl's own garbage collection take care of old unwanted environments for us.

The changes to our interpreter to make this happen are in fact quite limited. Fist of all, because `Eval()` now expects an environment as argument, the top-level `PScm::ReadEvalPrint()` must create one and pass it in. Note that it is the same as the `$PScm::GlobalEnv` from our previous interpreter, with the addition of a binding for `let`.

```
sub ReadEvalPrint {
    my ($infh, $outfh) = @_;

    $outfh ||= \*STDOUT;
    my $reader = new PScm::Read($infh);
    while (defined(my $expr = $reader->Read)) {
        my $result = $expr->Eval(
            new PScm::Env(
                let => new PScm::SpecialForm::Let(),
                '*' => new PScm::Primitive::Multiply(),
                '-' => new PScm::Primitive::Subtract(),
                if  => new PScm::SpecialForm::If(),
            )
        );
        $result->Print($outfh);
    }
}
```

The `Eval()` method for literal atoms (strings and numbers) is functionally unchanged. It ignores any argument environment because literals evaluate to themselves.

```
package PScm::Expr::Literal;
use base qw(PScm::Expr::Atom);

sub Eval {
    my ($self, $env) = @_;
    return $self;
}

############################
```

[8] If the improvement to the design of `let` does not warrant such a drastic change, it should be noted that closure is much easier to implement with this model and could be egregiously difficult to implement otherwise.

The Eval() method for symbols now uses the argument environment rather than a global one in which to lookup its value:

```
package PScm::Expr::Symbol;
use base qw(PScm::Expr::Atom);

sub Eval {
    my ($self, $env) = @_;
    return $env->LookUp($self);
}

##############################
```

The Eval() method for lists (expressions) is little changed either, it evaluates the operation in the current (argument) environment then calls the operation's Apply() method, passing the current environment as an additional argument.

```
sub Eval {
    my ($self, $env) = @_;
    my $op = $self->first()->Eval($env);
    return $op->Apply($self->rest, $env);
}
```

Apply() must change too. As shown earlier, There is one Apply() method for all PScm::Primitive classes, which evaluates all of the arguments to the primitive operation then calls the operation's private _apply() method with its pre-evaluated arguments. That needs to change only to evaluate those arguments in the current (argument) environment:

```
sub Apply {
    my ($self, $form, $env) = @_;

    my @unevaluated_args = $form->value;
    my @evaluated_args = map { $_->Eval($env) } @unevaluated_args;
    return $self->_apply(@evaluated_args);
}
```

Note particularly that there is no need to pass the environment to the private _apply() method: since all its arguments are already evaluated it has no need of an environment to evaluate anything in. Therefore the primitive multiply and subtract operations are unchanged from the previous version of the interpreter.

The Apply() method for special forms is separately implemented by each special form. In our previous interpreter there was only one special form: if, so let's take a look at how that has changed.

```
package PScm::SpecialForm::If;

use base qw(PScm::SpecialForm);

sub Apply {
    my ($self, $form, $env) = @_;

    my ($condition, $true_branch, $false_branch) =
                                        $form->value;

    if ($condition->Eval($env)->isTrue) {
        return $true_branch->Eval($env);
    } else {
        return $false_branch->Eval($env);
    }
}

1;
```

Pretty simple, The only change is that `PScm::SpecialForm::If::Apply()` passes its additional argument `$env` to each call to `Eval()`.

Finally, as we saw in the previous section, the `Extend()` method of `PScm::Env` passes the current environment, `$self`, to each call to `Eval()`:

```
sub Extend {
    my ($self, $ra_symbols, $ra_values) = @_;

    my %bindings = ();
    my @names = map { $_->value } @$ra_symbols;
    my @values = map { $_->Eval($self) } @$ra_values;
    @bindings{ @names } = @values;
    my $new = $self->new(%bindings);
    $new->{parent} = $self;
    return $new;
}
```

5.4 `let` Itself

Now we're done, we can look at that implementation of `PScm::SpecialForm::Let::Apply()` in our environment passing interpreter.

Remember that `let` has the general form:

`(let (`⟨*binding*⟩` ...) `⟨*expression*⟩`)`

where ⟨*binding*⟩ is:

`(`⟨*symbol*⟩ ⟨*expression*⟩`)`

So, with that in mind, here's the `Apply()` method.

```
package PScm::SpecialForm::Let;

use base qw(PScm::SpecialForm);

sub Apply {
```

```
13      my ($self, $form, $env) = @_;
14
15      my ($bindings, $body) = $form->value;
16      my (@symbols, @values);
17
18      foreach my $binding ($bindings->value) {
19          my ($symbol, $value) = $binding->value;
20          push @symbols, $symbol;
21          push @values,  $value;
22      }
23
24      return $body->Eval($env->Extend(\@symbols, \@values));
25  }
26
27  ##############################
```

It starts off at line 15 extracting the bindings (first list) and body (second list) from the argument form. Then it sets up two empty lists to collect the symbols (line 16) and the values (line 16) separately. Then in the loop on lines 18-22 it iterates over each binding collecting the unevaluated symbol in one list and the unevaluated argument in the other (remember that the environment's `Extend` method will evaluate the expressions in the `@values` list). Finally on line 24 it calls the body's `Eval()` method with an extended environment where those symbols are bound to those values.

Line 24 encapsulates our new simple definition for `let` quite concisely. The environment created by `Extend()` is passed directly to `Eval()` and the result of calling `Eval()` is returned directly.

5.5 Summary

In order to get `let` working easily we had to make two changes to the original implementation. Firstly adding an Extend method to the `PScm::Env` class to create new environments, and secondly altering the various `Eval()` and `Apply()` methods to pass the environment as argument rather than using a global environment. Having done that, the actual implementation of `let` was trivial.

5.6 Tests

Tests added to `PScm.t` from the 0.0.1 interpreter are here.

```
use strict;
use warnings;
use Test::More tests => 11;
use FileHandle;

BEGIN { use_ok(’PScm’) };

sub evaluate {
    my ($expression) = @_;

    my $fh = new FileHandle("> junk");
    $fh->print($expression);
    $fh = new FileHandle(’< junk’);
    my $outfh = new FileHandle("> junk2");
    ReadEvalPrint($fh, $outfh);
    $fh = 0;
    $outfh = 0;
    my $res = ‘cat junk2‘;
    chomp $res;
    unlink(’junk’);
    unlink(’junk2’);
    # warn "# [$res]\n";
    return $res;
}

ok(evaluate(’1’) eq ’1’, ’numbers’);
ok(evaluate(’+1’) eq ’1’, ’positive numbers’);
ok(evaluate(’-1’) eq ’-1’, ’negative numbers’);

ok(evaluate(’"hello"’) eq ’"hello"’, ’strings’);

ok(evaluate(’(* 2 3 4)’) eq ’24’, ’multiplication’);
ok(evaluate(’(- 10 2 3)’) eq ’5’, ’subtraction’);
ok(evaluate(’(- 10)’) eq ’-10’, ’negation’);

ok(evaluate(’(let ((x 2)) x)’) eq ’2’, ’simple let’);

ok(evaluate(’(if 0 10 20)’) eq ’20’, ’simple conditional’);

ok(evaluate(<<EOF) eq ’20’, ’conditional evaluation’);
(let ((a 00)
      (b 10)
      (c 20))
     (if a b c))
EOF
```

listing 10: t/PScm.t

```

# vim: ft=perl
```

listing 10: `t/PScm.t` (Cont.)

6 Interpreter Version 0.0.2—Implementing `lambda`

Having derived an environment passing interpreter in version 0.0.1, the addition of functions, specifically closures, becomes much more tractable.

So far the text has been pretty relaxed about the uses of the words *function* and *closure*, which is ok because in PScheme all functions are in fact closures. But before we go any further we'd better have a clearer definition of what a closure is, and what the difference between a function and a closure might be.

First of all what precisely is a function? On consideration, functions are a lot like `let` expressions: they both extend an environment then execute an expression in the extension. A `let` expression extends an environment with key-value pairs, then evaluates its body in that new environment. A function extends an environment by binding its formal arguments to its actual arguments[9] then evaluates its body in that new environment. For example, assuming the definition:

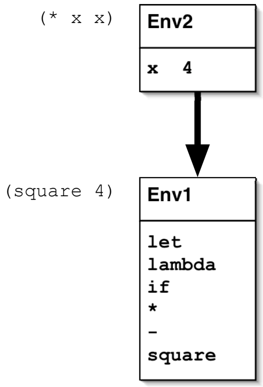
```
(define square (lambda (x) (* x x)))
```

then while executing the expression:

```
(square 4)
```

the global environment would be extended with a binding of `x` to `4`, and the body of the function, `(* x x)` would be evaluated in that new environment, as in Figure 10.

Figure 10: Functions extend an environment



Now, a *closure* is simply a function that when executed will extend the environment that was current at the time the closure was created. Consider an example we've seen before.

```
> (define times2
>   (let ((n 2))
```

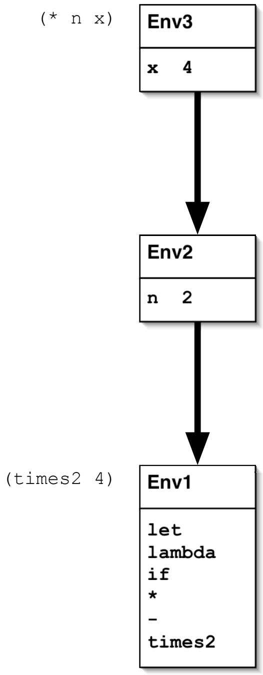
[9] Formal arguments are the names that a function gives to its variables. Actual arguments are the values passed to a function.

```
>       (lambda (x) (* n x))))
times2
> (times2 4)
8
```

The `lambda` expression is being executed in an environment where `n` is bound to `2`. The result of that `lambda` expression, a closure, is also the result of the `let` expression and therefore gets bound to the symbol `times2` in the global environment.

Now when `times2` is called, the closure body `(* n x)` must execute in an environment where `n` is still bound to `2`, as in Figure 11.

Figure 11: Closures extend the lexical environment



A function which is not a closure would have to pick a different environment to extend. It could choose the environment it is being executed in but that would cause confusion, or it could extend the global environment. The latter choice is the standard one for non-closure implementations, but as already noted all functions in PScheme are closures (and there are no advantages to them not being closures) so we don't have to worry about that.

So we're going to continue to use the words *function* and *closure* pretty much interchangeably, but when we use the word *function* we're emphasizing the functional aspects of the object under discussion, and when we use the word *closure*, we're emphasizing its environmental behaviour.

When considering functions (closures) there are two parts to the story, the first part is how `lambda` creates a function, and the second is how the function gets evaluated when it is called. In the next section we'll look at the first part, how `lambda` creates a function.

6.1 lambda

We need a good, simple example of closures in action. The following example fits our purposes, but is a bit more complicated than the examples we've seen so far:

```
> (let ((a 4)
>       (times2
>         (let ((n 2))
>           (lambda (x) (* x n)))))
>   (times2 a))
8
```

This example is just a little tricky, so in detail:

- The outer `let` binds `a` to `4` and `times2` to the result of evaluating the inner `let`, then evaluates `(times2 a)` in that new environment.
- The inner `let` binds `n` to `2` then evaluates the `lambda` expression.
- The value of that inner `let` is the result of evaluating the `lambda` expression, and that, a closure, is what gets bound to the symbol `times2` by the outer `let`.
- When `(times2 a)` is evaluated, the closure bound to `times2` can still "see" the value of `n` and so `(* x n)` produces the expected result (`8`).

Just to be absolutely sure that semantics of that expression are well understood, here is an equivalent in Perl:

```
{
    my $a = 4;
    my $times2 = do {
        my $n = 2;
        sub {
            my ($x) = @_;
            $x * $n;
        }
    };
    $times2->($a);
}
```

Now we're going to walk through the execution of the PScheme statement in a lot more detail, considering what the interpreter is actually doing to produce the final result.

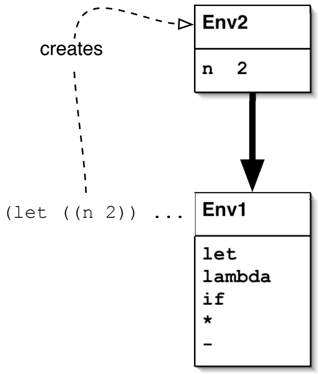
The very first thing that happens when evaluating our PScheme example is that the outer `let` evaluates the number `4` in the global environment. It does not yet bind that value to `a`, it first must evaluate the expression that will be bound to `times2`.

```
> (let ((a 4)
>       (times2
>         (let ((n 2))
>           (lambda (x) (* x n)))))
>   (times2 a))
8
```

The next thing that happens is the outer `let` initiates the evaluation of the inner `let`. The inner `let` extends the global environment with a binding of `n` to 2, as hilighted in the following code and shown in Figure 12.

```
> (let ((a 4)
>       (times2
>         (let ((n 2))
>           (lambda (x) (* x n)))))
>   (times2 a))
8
```

Figure 12: `let` binds `n` to 2



Then, in that new environment, labelled **Env2**, the `let` evaluates the `lambda` expression:

```
> (let ((a 4)
>       (times2
>         (let ((n 2))
>           (lambda (x) (* x n)))))
>   (times2 a))
8
```

Evaluating a `lambda` expression is just the same as evaluating any other list expression, it's (unevaluated) arguments are passed to its `Apply()` method, along with the current environment. In our example the arguments to the `lambda`'s `Apply()` would be:

1. A list of the unevaluated arguments containing

    (a) the formal arguments to the function: `(x)`
    (b) the body of the function: `(* x n)`

2. the current environment, **Env2** that the `let` just created, with a binding of `n` to `2`.

To start to make this happen we first need to add a new subclass of `PScm::SpecialForm`, rather unsurprisingly called `PScm::SpecialForm::Lambda`, and we need to add a binding from the symbol `lambda` to an object of that class in the initial environment. Firstly, here's `ReadEvalPrint()` with the additional binding:

```
30 sub ReadEvalPrint {
31     my ($infh, $outfh) = @_;
32
33     $outfh ||= \*STDOUT;
34     my $reader = new PScm::Read($infh);
35     while (defined(my $expr = $reader->Read)) {
36         my $result = $expr->Eval(
37             new PScm::Env(
38                 let    => new PScm::SpecialForm::Let(),
39                 '*'    => new PScm::Primitive::Multiply(),
40                 '-'    => new PScm::Primitive::Subtract(),
41                 if     => new PScm::SpecialForm::If(),
42                 lambda => new PScm::SpecialForm::Lambda(),
43             )
44         );
45         $result->Print($outfh);
46     }
47 }
```

The only change is the addition of line 42 with the new binding for `lambda`.

Now we can look at that new package `PScm::SpecialForm::Lambda`. All it's `Apply()` method has to do is to store the details of the function definition and the current environment in another new type of object representing the closure:

```
package PScm::SpecialForm::Lambda;

use base qw(PScm::SpecialForm);
use PScm::Closure;

sub Apply {
    my ($self, $form, $env) = @_;

    my ($args, $body) = $form->value;
    return PScm::Closure::Function->new($args, $body, $env);
}

1;
```

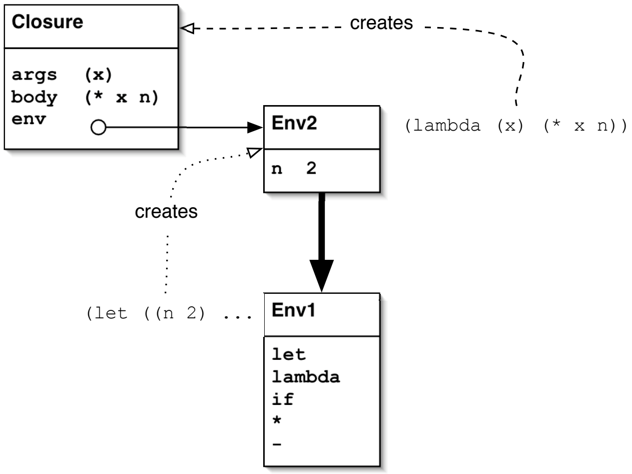
At line 54 it unpacks the formal arguments (i.e. `(x)`) and body (`(* x n)`) of its argument `$form` (the arguments to the `lambda` expression) and on line 55 it returns a new `PScm::Closure::Function` object containing those values and, most importantly, also containing the current environment (**Env2** in our example.)

That `new()` method (actually in `PScm::Closure::Function`'s parent class `PScm::Closure`) does no more than bundle its arguments:

```
sub new {
    my ($class, $args, $body, $env) = @_;

    bless {
        args => $args,
        body => $body,
        env  => $env,
    }, $class;
}
```

So in our example it is **Env2** that is captured, along with the arguments and body of the function, in the resulting closure. This is shown in Figure 13.

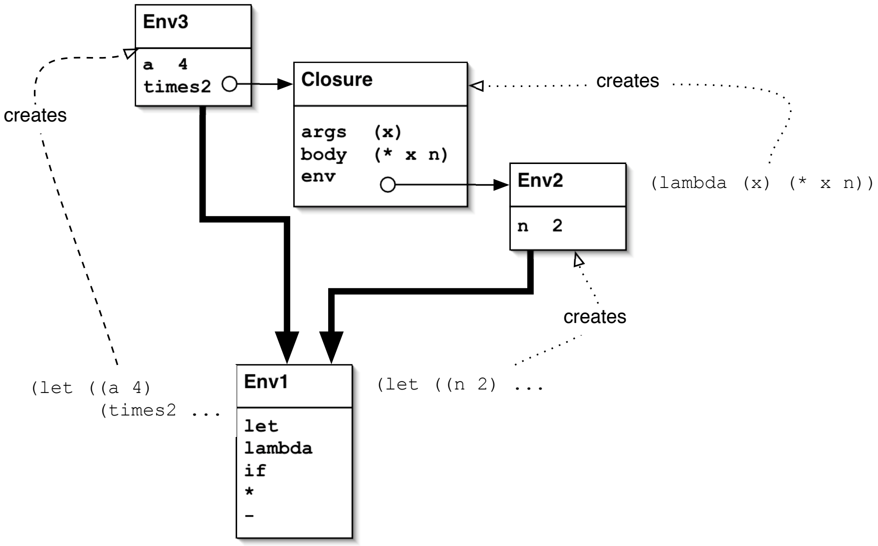
Figure 13: Closure Captures the Local Environment



As we've noted, the value of the inner `let` expression is that new **Closure** object, and next the outer `let` recieves the value of the inner `let`, and extends the global environment with a binding of `times2` to that. It also binds `a` to `4`:

```
> (let ((a 4)
>       (times2
>         (let ((n 2))
>           (lambda (x) (* x n)))))
>   (times2 a))
8
```

Figure 14: `let` binds `times2`



The resulting environment is labelled **Env3** in Figure 14.
Now at this point the only thing hanging on to the old **Env2**, where `n` has a value, is that **Closure**, and the only thing hanging on to the **Closure** is the binding for `times2` in **Env3** (the code for the `Apply()` method of the outer `let` is currently holding on to **Env3**.)

Having created **Env3**, the outer `let` evaluates its body, `(times2 a)` in that environment.

```
> (let ((a 4)
>       (times2
>         (let ((n 2))
>           (lambda (x) (* x n)))))
>   (times2 a))
8
```

That brings us to the second part of our story, how a function (a closure) gets evaluated.

6.2 Closure

To recap, we've reached the stage where the subexpression `(times2 a)` is about to be evaluated. It will be evaluated in the context of **Env3** from Figure 14 which the outer `let` has just set up with a binding of `a` to 2 and `times2` to the closure.

Since `(times2 a)` is a list, the `Eval()` method for lists comes in to play again. It evaluates the first component of the list, the symbol `times2`, in the context of **Env3** resulting in the **Closure**. Then it passes the rest of the form (a list containing the symbol `a`) unevaluated, along with the current environment **Env3**, to the closure's `Apply()` method. Closures, being operations, have to have an `Apply()` method, and here it is:

```
29 package PScm::Closure::Function;
30
31 use base qw(PScm::Closure);
32
33 sub Apply {
34     my ($self, $form, $env) = @_;
35
36     my @evaluated_args = map { $_->Eval($env) } $form->value;
37     return $self->_apply(@evaluated_args);
38 }
39
40 1;
```

First of all, on line 36 it evaluates each component of the form (each argument to the function) with `map`, passing the argument `$env` (**Env3**) to each call to `Eval()`. After all, closures are functions, and functions take their arguments evaluated.

At line 37 our closure's `Apply()` returns the result of calling a separate `_apply()` method on those evaluated arguments, much as primitive operations do. Note particularily that it does not pass its argument `$env` to the private `_apply()` method.

The private `_apply()` method is in the parent `PScm::Closure` class[10]:

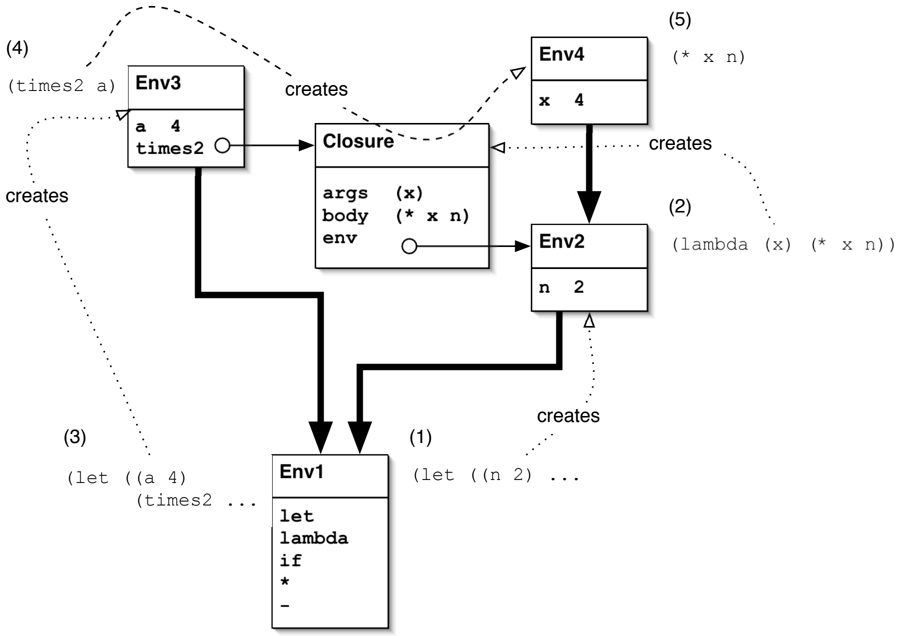
```
21 sub _apply {
22     my ($self, @args) = @_;
23
24     my $extended_env = $self->env->ExtendUnevaluated([$self->args], [@args]);
25     return $self->body->Eval($extended_env);
26 }
```

This `_apply()` method does not need an argument environment because the correct environment to extend is the one that was current when the closure was created, and that was captured as part of the closure object. On line 24 It extends that environment with bindings from its formal arguments, also collected when the closure was created (i.e. `x`,) to the actual arguments it was passed (i.e. `4`, already evaluated, hence the call to `ExtendUnevaluated()` which we'll look at presently.) Then it evaluates its body (the of the function) passing that extended environment as argument and returns the result (line 25).

Returning to our example, we're still considering the evaluation of the subexpression `(times2 a)`. As we've said the closure's `Apply()` method evaluates its argument `a` in the environment it was passed, **Env3**, resulting in `4`. But it is the captured environment, **Env2**, that the closure extends with a binding of `x` to `4`, resulting in **Env4** (Figure 15). It is in **Env4**, with `x` bound to `4` and `n` still bound to `2`, that the closure executes the body of the function `(* x n)`.
Figure 15 pretty much tells the whole story. Here's our example one last time so it can be walked through referring to the figure:

[10] Why? Because a later version of the interpreter will support more than one type of closure.

Figure 15: Closure Extends Captured Env



```
(let ((a 4)
      (times2
         (let ((n 2))
           (lambda (x) (* x n)))))
   (times2 a))
```

- At (1) in the figure, the inner `let` extends the global env **Env1** with a binding of `n` to `2` producing **Env2**.
- At (2) the inner `let` then evaluates the `lambda` expression in the context of **Env2**, creating a **Closure** that captures **Env2**.
- At (3), the outer `let` extends the global environment **Env1** with bindings of `a` to `4` and `times2` to the value of the inner `let`, the **Closure**.
- At (4) the outer `let` evaluates the subexpression `(times2 a)` in the context of **Env3**. In this environment `times2` evaluates to the closure, and its `Apply()` evaluates `a` in the same environment **Env3** where it evaluates to `4`.
- Finally, at (5), the closure extends the originally captured environment **Env2** with a binding of `x` to `4` producing **Env4** and evaluates its body, `(* x n)`, in this environment.

That just leaves a couple of loose ends to tidy up. If you remember, the `Extend()` method in `PScm::Env` evaluates the values it is passed, and since the arguments to _apply in PScm::Closure are already evaluated, _apply() needed to call a variant of Extend(), namely ExtendUnevaluated(). Here's that method:

```
sub ExtendUnevaluated {
    my ($self, $ra_symbols, $ra_values) = @_;

    my %newbindings;
    @newbindings{ map { $_->value } @$ra_symbols } = @$ra_values;
    my $newenv = $self->new(%newbindings);
    $newenv->{parent} = $self;
    return $newenv;
}
```

You can see that it does pretty much exactly what the old Extend() did, except that it doesn't call Eval() on the values. In fact Extend() can now be re-written and simplified to use it:

```
sub Extend {
    my ($self, $ra_symbols, $ra_values) = @_;

    return $self->ExtendUnevaluated($ra_symbols,
        [ map { $_->Eval($self) } @$ra_values ]);
}
```

6.3 Summary

Hopefully the power, flexibility and elegance of an environment-passing interpreter combined with a linked-list environment implementation is becoming apparent. The enormous advantage over a stack discipline is that individual environments need not go away just because a particular construct returns. They can hang around as long as they are needed and garbage collection will remove them when the time comes. Without further ado then, here's the full source for our new PScm::Closure package in Listing 11.

6.4 Tests

Here's the additions to PScm.t for the interpreter version 0.0.2.
We have added three more tests since the previous version. The first new test, starting on line 48, tests the simple creation of a lambda expression, its binding to a symbol, and its application to arguments. The second addition starting on line 54 just exercises pretty much exactly the example we've been working through. The third addition starting on line 61 starts to flex the muscles of our nascent interpreter a little more. It creates a local makemultiplier function that when called with an argument n will return another function that will multiply n by its argument. It then binds the result of calling (makemultiplier 3) to times3 and calls (times3 5), confirming that the result is 15, as expected.

We could rewrite the body of that last test in Perl as follows:

```
{
    my $times3 = do {
```

```perl
package PScm::Closure;

use strict;
use warnings;
use base qw(PScm);

sub new {
    my ($class, $args, $body, $env) = @_;

    bless {
        args => $args,
        body => $body,
        env  => $env,
    }, $class;
}

sub args { $_[0]->{args}->value }
sub body { $_[0]->{body} }
sub env  { $_[0]->{env} }

sub _apply {
    my ($self, @args) = @_;

    my $extended_env = $self->env->ExtendUnevaluated([$self->args], [@args]);
    return $self->body->Eval($extended_env);
}

################################
package PScm::Closure::Function;

use base qw(PScm::Closure);

sub Apply {
    my ($self, $form, $env) = @_;

    my @evaluated_args = map { $_->Eval($env) } $form->value;
    return $self->_apply(@evaluated_args);
}

1;
```

listing 11: PScm/Closure.pm

```
use strict;
use warnings;
use Test::More tests => 14;
use FileHandle;

BEGIN { use_ok(’PScm’) }

sub evaluate {
    my ($expression) = @_;

    my $fh = new FileHandle("> junk");
    $fh->print($expression);
    $fh = new FileHandle(’< junk’);
    my $outfh = new FileHandle("> junk2");
    ReadEvalPrint($fh, $outfh);
    $fh    = 0;
    $outfh = 0;
    my $res = ‘cat junk2‘;
    chomp $res;
    unlink(’junk’);
    unlink(’junk2’);

    # warn "# [$res]\n";
    return $res;
}

ok(evaluate(’1’)  eq ’1’,  ’numbers’);
ok(evaluate(’+1’) eq ’1’,  ’positive numbers’);
ok(evaluate(’-1’) eq ’-1’, ’negative numbers’);

ok(evaluate(’"hello"’) eq ’"hello"’, ’strings’);

ok(evaluate(’(* 2 3 4)’)  eq ’24’,  ’multiplication’);
ok(evaluate(’(- 10 2 3)’) eq ’5’,   ’subtraction’);
ok(evaluate(’(- 10)’)     eq ’-10’, ’negation’);

ok(evaluate(’(let ((x 2)) x)’) eq ’2’, ’simple let’);

ok(evaluate(’(if 0 10 20)’) eq ’20’, ’simple conditional’);

ok(evaluate(<<EOF) eq ’20’, ’conditional evaluation’);
(let ((a 00)
      (b 10)
      (c 20))
     (if a b c))
```

listing 12: t/PScm.t

```
EOF

ok(evaluate(<<EOF) eq '16', 'lambda');
(let ((square
       (lambda (x) (* x x))))
     (square 4))
EOF

ok(evaluate(<<EOF) eq '12', 'closure');
(let ((times3
      (let ((n 3))
           (lambda (x) (* n x)))))
     (times3 4))
EOF

ok(evaluate(<<EOF) eq '15', 'higher order functions');
(let ((times3
      (let ((makemultiplier
             (lambda (n)
                     (lambda (x) (* n x)))))
           (makemultiplier 3))))
     (times3 5))
EOF

# vim: ft=perl
```

listing 12: t/PScm.t (Cont.)

```
        my $makemultiplier = sub {
            my ($n) = @_;
            return sub {
                my ($x) = @_;
                return $n * $x;
            }
        };
        $makemultiplier->(3);
    };
    $times3->(5);
}
```

7 Interpreter Version 0.0.3—Recursion and `letrec`

Let's try a little experiment with the interpreter version 0.0.2. We'll try to use `let` to define a recursive function, the perennial factorial function[11].

```
> (let ((factorial
>       (lambda (n)
>               (if n
>                   (* n (factorial (- n 1)))
>                   1))))
>      (factorial 3))
Error: no binding for factorial in PScm::Env
```

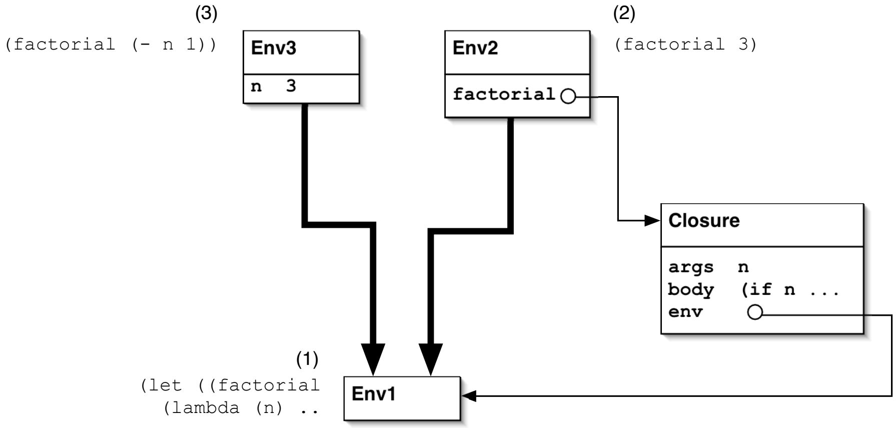
It didn't work. The reason that it didn't work is obvious, considering how `let` works.

`let` evaluates its bindings in the enclosing environment, so it is the enclosing environment (the global environment in this case) that the `lambda` captures. Now that environment doesn't have a binding for `factorial` so any recursive call to `factorial` from the body of the function (closure) is bound to fail.

`let` will create a binding for `factorial`, but only by extending the global environment *after* the `lambda` expression has been evaluated.

So the error is not coming from the call to `(factorial 3)`, it's coming from the attempted recursive call to `(factorial (- n 1))` inside the body of the `factorial` definition. The environment diagram in Figure 16 should help to make that clear.

Figure 16: Why recursion doesn't work



The `let` evaluates the `lambda` expression in the initial environment, **Env1** at (1), so that's the environment that gets captured by the Closure. Then the `let` binds the closure to the symbol `factorial` in an extended environment **Env2**, and that's where the body of the `let`, `(factorial 3)`, gets evaluated at (2). Now after evaluating its argument `3` in **Env2**, the closure proceeds to extend

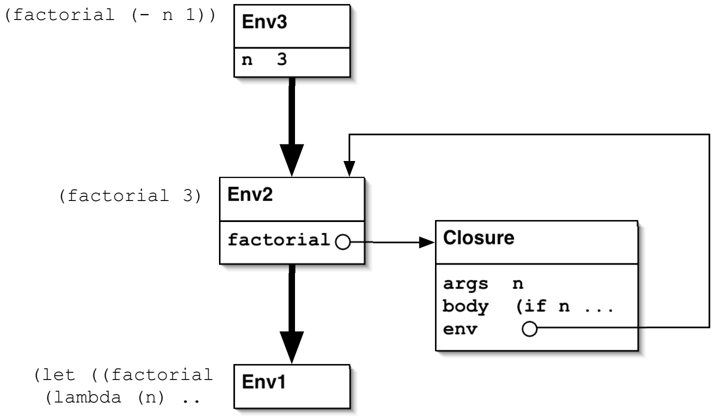
[11] Factorial(n), often written $n!$, is $n \times (n-1) \times (n-2) \times \cdots \times 1$.

the environment it captured, the global environment **Env1**, with a binding of `n` to `3` producing **Env3**. It's in **Env3** that the body of the factorial function gets evaluated at (3). Now `n` has a binding in that environment, but unfortunately `factorial` doesn't, so the recursive call fails.

7.1 `letrec`

What we need is a variation of `let` that arranges to evaluate the values for its bindings in an environment where the bindings are already in place. Essentially the environments would appear as in Figure 17.

Figure 17: Recursive environments



In this figure the closure has been persuaded to capture an environment **Env2** containing a binding that refers back to the closure itself (a circular reference in effect.) In this circumstance any recursive call to `factorial` from the body of the closure *would* work because the closure would have extended **Env2** and its body would execute in a context (**Env3**) where `factorial` did have a value.

The special form we're looking for is called `letrec` (short for "`let` recursive") and it isn't too tricky, although a bit of a hack. Let's first remind ourselves how `let` works.

1. Evaluate the values in the current, passed in environment;
2. Create a new environment as an extension of the current one, with those values bound to their symbols;
3. Evaluate the body of the `let` in that new environment.

Our variant, `letrec`, isn't all that different. What it does is:

1. Create a new extended environment first, with dummy values bound to the symbols;
2. Evaluate the values in that new environment;
3. Assign the values to their symbols in that new environment;

4. Evaluate the body in that new environment.

Obviously if any of the values in a letrec are expressions other than lambda expressions, and they make reference to other letrec values in the same scope, then there will be problems. Remember that all lambda does is to capture the current environment along with formal arguments and function body. It does not immediately evaluate anything in that captured environment. For that reason real letrec implementations may typically only allow lambda expressions as values. PScheme doesn't bother making that check, simply to keep the code as concise as possible[12].

7.2 Assignment

To implement letrec then, we first need to add a method to the environment class PScm::Env that will allow assignment to existing bindings. Here is that method.

```
sub Assign {
    my ($self, $symbol, $value) = @_;

    if (defined(my $ref = $self->_lookup_ref($symbol))) {
        $$ref = $value;
    } else {
        die "no binding for @{[$symbol->value]}",
            " in @{[ref($self)]}\n";
    }
}
```

Assign() uses a helper function _lookup_ref() to actually do the symbol lookup. If _lookup_ref() finds a binding, then Assign() puts the new value in place through the reference that _lookup_ref() returns. It is an error if there's not currently such a symbol in the environment. This makes sense because it keeps the distinction between variable binding and assignment clear: variable binding creates a new binding; assignment changes an existing one.

_lookup_ref() is simple enough, it does pretty much what LookUp() does, except it returns a reference to what it finds, and returns undef, rather than die()-ing if it doesn't find a value:

```
sub _lookup_ref {
    my ($self, $symbol) = @_;

    if (exists($self->{bindings}{ $symbol->value })) {
        return \$self->{bindings}{ $symbol->value };
    } elsif ($self->{parent}) {
        return $self->{parent}->_lookup_ref($symbol);
    } else {
        return undef;
    }
}
```

[12] One possible way to detect this type of error would be to bind dummy objects to the symbols. These dummy objects would have an Eval() method that would die() with an informative error message if it was ever called.

Incidentally, `LookUp()` itself has been modified (and simplified) to make use of it.

```
sub LookUp {
    my ($self, $symbol) = @_;

    if (defined(my $ref = $self->_lookup_ref($symbol))) {
        return $$ref;
    } else {
        die "no binding for @{[$symbol->value]}",
            " in @{[ref($self)]}\n";
    }
}
```

All that remains to be done is to add a `LetRec` subclass of `PScm::SpecialForm` with an `Apply()` method that implements the algorithm we've discussed, then add a binding in the initial environment from "`letrec`" to an instance of that class.

We can simplify things a bit by subclassing `PScm::SpecialForm::Let` instead of `PScm::SpecialForm`, and factoring common code out of `PScm::SpecialForm::Let`'s `Apply()` method into a new `UnPack` method in that class. So first here's the new version of `PScm::SpecialForm::Let::Apply()`

```
sub Apply {
    my ($self, $form, $env) = @_;

    my ($ra_symbols, $ra_values, $body) = $self->UnPack($form);

    return $body->Eval($env->Extend($ra_symbols, $ra_values));
}
```

The common code in `PScm::SpecialForm::Let::UnPack()` just unpacks the symbols, bindings and body from the argument `$form` and returns them:

```
sub UnPack {
    my ($self, $form) = @_;

    my ($bindings, $body) = $form->value;
    my (@symbols, @values);

    foreach my $binding ($bindings->value) {
        my ($symbol, $value) = $binding->value;
        push @symbols, $symbol;
        push @values,  $value;
    }

    return (\@symbols, \@values, $body);
}
```

Now, our new `Apply()` in `PScm::SpecialForm::LetRec` makes use of that same `UnPack()` method (by inheriting from `PScm::SpecialForm::Let`). It differs from the original `Apply()` only in that it calls the environment's `ExtendRecursively()` method, rather than `Extend()`.

```
package PScm::SpecialForm::LetRec;

use base qw(PScm::SpecialForm::Let);

sub Apply {
    my ($self, $form, $env) = @_;

    my ($ra_symbols, $ra_values, $body) = $self->UnPack($form);

    return $body->Eval($env->ExtendRecursively($ra_symbols, $ra_values));
}

##############################
```

So we need to take a look at that `ExtendRecursively()` method in `PScm::Env`.

```
sub ExtendRecursively {
    my ($self, $ra_symbols, $ra_values) = @_;

    my $newenv = $self->ExtendUnevaluated($ra_symbols, $ra_values);
    $newenv->_eval_values();
    return $newenv;
}
```

It creates a new environment by extending the current environment, `$self` with the symbols bound to their unevaluated values. Then it calls a new, private method `_eval_values()` on the new environment. Here's that method:

```
sub _eval_values {
    my ($self) = @_;
    map { $self->{bindings}{$_} = $self->{bindings}{$_}->Eval($self) }
        keys %{$self->{bindings}};
}
```

All *that* does is to loop over all of its bindings, replacing the unevaluated expression with the result of evaluating the expression in the current environment. Since all of those expressions are expected to be lambda expressions, the resulting closures capture the environment that they are themselves bound in. QED.

A careful reader may have realised that a valid alternative implementation of `letrec` would just create an empty environment extension, then populate the environment afterwards with an alternative version of `Assign()` which did not require the symbols to pre-exist in the environment. The main reason it is not done that way is that the current behaviour of `Assign()` is more appropriate for later extensions to the interpreter.

Just to be complete, here's the new version of `PScm::ReadEvalPrint()` with the binding for `letrec`.

```
sub ReadEvalPrint {
    my ($infh, $outfh) = @_;

    $outfh ||= \*STDOUT;
    my $reader = new PScm::Read($infh);
    while (defined(my $expr = $reader->Read)) {
        my $result = $expr->Eval(
            new PScm::Env(
                let    => new PScm::SpecialForm::Let(),
                ’*’    => new PScm::Primitive::Multiply(),
                ’-’    => new PScm::Primitive::Subtract(),
                if     => new PScm::SpecialForm::If(),
                lambda => new PScm::SpecialForm::Lambda(),
                letrec => new PScm::SpecialForm::LetRec(),
            )
        );
        $result->Print($outfh);
    }
}
```

7.3 Summary

Let evaluates the values of its bindings in the enclosing environment. Then it creates an extended environment in which to evaluate the body of the `let` expression. This means that recursive `lambda` expressions defined by `let` won’t work, because there’s no binding for the recursive function when the `lambda` expression is evaluated. In order to get recursion to work, we needed to create a variant of `let`, called `letrec` (`let` recursive) which sets up a dummy environment with stub values for the symbols in which to evaluate the `lambda` expressions, so that the `lambda` expressions could capture that environment in their resulting closures. Having evaluated those expressions, `letrec` assigns their values to the existing bindings in the new environment, replacing the dummy values. Thus when the closure executes later, the environment it has captured, and which it will extend with its formal arguments bound to actual values, will contain a reference to the closure itself, so a recursive call is successful.

7.4 Tests

Here’s the additions to `PScm.t` for version 0.0.3 of the interpreter.
There are three new tests since 0.0.2, the first two, starting at line 70 just prove what we already know, that `let` does not (and should not) support recursion. The other new test starting at line 80 replaces `let` with `letrec` and proves that `letrec` on the other hand does support recursive function definitions.

```
use strict;
use warnings;
use Test::More tests => 17;
use FileHandle;

BEGIN { use_ok(’PScm’) }

sub evaluate {
    my ($expression) = @_;

    my $fh = new FileHandle("> junk");
    $fh->print($expression);
    $fh = new FileHandle(’< junk’);
    my $outfh = new FileHandle("> junk2");
    ReadEvalPrint($fh, $outfh);
    $fh    = 0;
    $outfh = 0;
    my $res = ‘cat junk2‘;
    chomp $res;
    unlink(’junk’);
    unlink(’junk2’);

    # warn "# [$res]\n";
    return $res;
}

ok(evaluate(’1’)  eq ’1’,  ’numbers’);
ok(evaluate(’+1’) eq ’1’,  ’positive numbers’);
ok(evaluate(’-1’) eq ’-1’, ’negative numbers’);

ok(evaluate(’"hello"’) eq ’"hello"’, ’strings’);

ok(evaluate(’(* 2 3 4)’)  eq ’24’,  ’multiplication’);
ok(evaluate(’(- 10 2 3)’) eq ’5’,   ’subtraction’);
ok(evaluate(’(- 10)’)     eq ’-10’, ’negation’);

ok(evaluate(’(let ((x 2)) x)’) eq ’2’, ’simple let’);

ok(evaluate(’(if 0 10 20)’) eq ’20’, ’simple conditional’);

ok(evaluate(<<EOF) eq ’20’, ’conditional evaluation’);
(let ((a 00)
      (b 10)
      (c 20))
     (if a b c))
```

listing 13: `t/PScm.t`

```
EOF

ok(evaluate(<<EOF) eq ’16’, ’lambda’);
(let ((square
       (lambda (x) (* x x))))
     (square 4))
EOF

ok(evaluate(<<EOF) eq ’12’, ’closure’);
(let ((times3
      (let ((n 3))
           (lambda (x) (* n x)))))
     (times3 4))
EOF

ok(evaluate(<<EOF) eq ’15’, ’higher order functions’);
(let ((times3
      (let ((makemultiplier
             (lambda (n)
                     (lambda (x) (* n x)))))
           (makemultiplier 3))))
     (times3 5))
EOF

ok(!defined(eval { evaluate(<<EOF) }), ’let does not support recursion’);
(let ((factorial
        (lambda (n)
          (if n (* n (factorial (- n 1)))
                1))))
  (factorial 4))
EOF

ok($@ eq "no binding for factorial in PScm::Env\n", ’let does not support recursion [2]’);

ok(evaluate(<<EOF) eq "24", ’letrec and recursion’);
(letrec ((factorial
           (lambda (n)
             (if n (* n (factorial (- n 1)))
                   1))))
  (factorial 4))
EOF

# vim: ft=perl
```

listing 13: t/PScm.t (Cont.)

8 Interpreter Version 0.0.4—Another Variation on `let`

Suppose we have a fairly complicated calculation to make. We'd like to compute intermediate values in order to simplify our code. For example:

```
> (let ((a 5)
>       (b (* a 2))
>       (c (- b 3)))
>      c)
Error: no binding for a in PScm::Env
```

It didn't work. The error occurs when attempting to evaluate `(* a 2)`. Why? Well in much the same way as `let` fails for recursive definitions: because `let` binds its arguments in parallel, at the point that it is trying to evaluate `(* a 2)`, it is still doing so in the environment prior to binding `a`.

`letrec` can't help here, because it sets up an environment with dummy values to evaluate its values in, which is OK if those values are closures that just capture that environment for later, but very bad if they're actually going to try to do any immediate evaluations with those dummy values.

8.1 Sequential Binding

Of course the solution is quite simple even using our existing `let` form: we just nest the `let` expressions so that each value expression gets evaluated in an environment where the previous value is already bound to its symbol:

```
> (let ((a 5))
>      (let ((b (* a 2)))
>           (let ((c (- b 3)))
>                c)))
7
```

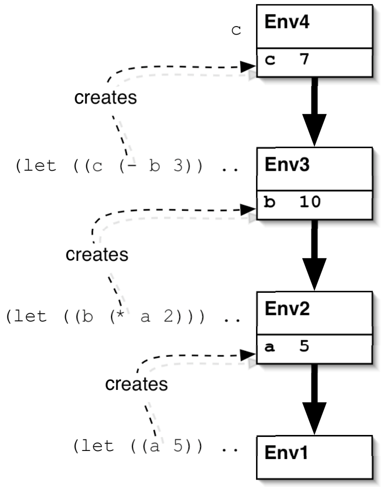
That would give us a set of environments as in Figure 18.
The outer `let` binds `a` to 5 to create **Env2**. The next `let` evaluates `(* a 2)` in the context of **Env2** and creates **Env3** with a binding of `b` to the result `10`. The innermost `let` evaluates `(- b 3)` in **Env3** and binds `c` to the result, creating **Env4**. In **Env4** the final evaluation of `c` results in 7, which is the result of the expression.

While that works fine, it's rather ugly and verbose code. Wouldn't it be better if there was a variant of `let` that did all that for us, binding its variables sequentially? This variant of `let` is called `let*` (let-star) and is found in most lisp implementations.

8.2 `let*`

To implement `let*`, in the same way as we implemented `letrec`, we create a new sub-class of `PScm::SpecialForm::Let` and give it an `Apply()` method, then bind a symbol (`let*`) to an instance of that class in the initial environment. Our new class will be called `PScm::SpecialForm::LetStar` and here's that `Apply()` method.

Figure 18: Nested environments



```
package PScm::SpecialForm::LetStar;

use base qw(PScm::SpecialForm::Let);

sub Apply {
    my ($self, $form, $env) = @_;

    my ($ra_symbols, $ra_values, $body) = $self->UnPack($form);

    return $body->Eval($env->ExtendIteratively($ra_symbols, $ra_values));
}

##############################
```

Again it only differs from the `let` and `letrec` implementations of `Apply()` in the way it extends the environment it passes to the `Eval()` of its body. In this case it calls the new `PScm::Env` method `ExtendIteratively()`.

```
sub ExtendIteratively {
    my ($self, $ra_symbols, $ra_values) = @_;

    my @symbols = @$ra_symbols;
    my @values = @$ra_values;
    my $newenv = $self;

    while (@symbols) {
        my $symbol = shift @symbols;
        my $value = shift @values;
        $newenv = $newenv->Extend([$symbol], [$value]);
    }

    return $newenv;
}
```

This method implements the algorithm we discussed earlier, creating a new environment frame for each individual binding, and evaluating the value part in the context of the previous environment frame. The last environment frame, the head of the list of frames rooted in the original environment, is returned by the method[13].

8.3 Summary

This may all seem a bit academic, but let's remember that Perl supports both types of variable binding, `let` and `let*`, in the following way.

Parallel assignment is done in a list context:

```
my ($a2, $b2) = ($a * $a, $b * $b);
```

Sequential binding is done by sequential assignment:

```
my $a = 5;
my $b = $a * 2;
my $c = $b - 3;
```

`let` has its uses, just as assignment in a list context does. For instance with parallel assignment it becomes possible to swap the values of variables without needing an additional temporary variable. In Perl:

```
($b, $a) = ($a, $b);
...
```

and in PScheme:

```
(let ((a b)
      (b a))
     ...)
```

Again, just for completeness, here's our 0.0.4 version of `ReadEvalPrint()` with the additional `let*` binding.

```
sub ReadEvalPrint {
    my ($infh, $outfh) = @_;

    $outfh ||= \*STDOUT;
    my $reader = new PScm::Read($infh);
    while (defined(my $expr = $reader->Read)) {
        my $result = $expr->Eval(
            new PScm::Env(
                let    => new PScm::SpecialForm::Let(),
```

[13] An alternative implementation would be to only create one new environment frame, then iteratively evaluate and bind each value in turn, in the context of that new environment.

```
        '*'    => new PScm::Primitive::Multiply(),
        '-'    => new PScm::Primitive::Subtract(),
        if     => new PScm::SpecialForm::If(),
        lambda => new PScm::SpecialForm::Lambda(),
        letrec => new PScm::SpecialForm::LetRec(),
        'let*' => new PScm::SpecialForm::LetStar(),
      )
    );
    $result->Print($outfh);
  }
}
```

8.4 Tests

The additional tests for 0.0.4 are here.

The first new test, lines 88-94, proves that ordinary `let` binds in parallel, by doing the variable swapping trick. The second new test, lines 96-101, demonstrates `let*` binding sequentially since the innermost binding of b to a sees the immediately previous binding of a to the outer b.

```
use strict;
use warnings;
use Test::More tests => 19;
use FileHandle;

BEGIN { use_ok(’PScm’) }

sub evaluate {
    my ($expression) = @_;

    my $fh = new FileHandle("> junk");
    $fh->print($expression);
    $fh = new FileHandle(’< junk’);
    my $outfh = new FileHandle("> junk2");
    ReadEvalPrint($fh, $outfh);
    $fh    = 0;
    $outfh = 0;
    my $res = ‘cat junk2‘;
    chomp $res;
    unlink(’junk’);
    unlink(’junk2’);

    # warn "# [$res]\n";
    return $res;
}

ok(evaluate(’1’)  eq ’1’,  ’numbers’);
ok(evaluate(’+1’) eq ’1’,  ’positive numbers’);
ok(evaluate(’-1’) eq ’-1’, ’negative numbers’);

ok(evaluate(’"hello"’) eq ’"hello"’, ’strings’);

ok(evaluate(’(* 2 3 4)’)  eq ’24’,  ’multiplication’);
ok(evaluate(’(- 10 2 3)’) eq ’5’,   ’subtraction’);
ok(evaluate(’(- 10)’)     eq ’-10’, ’negation’);

ok(evaluate(’(let ((x 2)) x)’) eq ’2’, ’simple let’);

ok(evaluate(’(if 0 10 20)’) eq ’20’, ’simple conditional’);

ok(evaluate(<<EOF) eq ’20’, ’conditional evaluation’);
(let ((a 00)
      (b 10)
      (c 20))
     (if a b c))
```

listing 14: `t/PScm.t`

```
EOF

ok(evaluate(<<EOF) eq ’16’, ’lambda’);
(let ((square
       (lambda (x) (* x x))))
     (square 4))
EOF

ok(evaluate(<<EOF) eq ’12’, ’closure’);
(let ((times3
      (let ((n 3))
           (lambda (x) (* n x)))))
     (times3 4))
EOF

ok(evaluate(<<EOF) eq ’15’, ’higher order functions’);
(let ((times3
      (let ((makemultiplier
             (lambda (n)
                     (lambda (x) (* n x)))))
           (makemultiplier 3))))
     (times3 5))
EOF

ok(!defined(eval { evaluate(<<EOF) }), ’let does not support recursion’);
(let ((factorial
        (lambda (n)
          (if n (* n (factorial (- n 1)))
                1))))
  (factorial 4))
EOF

ok($@ eq "no binding for factorial in PScm::Env\n", ’let does not support recursion [2]’);

ok(evaluate(<<EOF) eq "24", ’letrec and recursion’);
(letrec ((factorial
           (lambda (n)
             (if n (* n (factorial (- n 1)))
                   1))))
  (factorial 4))
EOF

ok(evaluate(<<EOF) eq ’1’, ’let binds in parallel’);
(let ((a 1)
```

listing 14: t/PScm.t (Cont.)

```
      (b 2))
     (let ((a b)
           (b a))
          b))
EOF

ok(evaluate(<<EOF) eq '2', 'let* binds sequentially');
(let ((a 1)
      (b 2))
  (let* ((a b)
         (b a))
    b))
EOF

# vim: ft=perl
```

listing 14: `t/PScm.t` (Cont.)

9 Interpreter Version 0.0.5—List Processing

It was mentioned in the introduction to this chapter that one of the great strengths of the Lisp family of languages is their ability to treat programs as data: to manipulate the same list structures that their expressions are composed of. So far we haven't seen any of that functionality implemented in our interpreter.

Those list structures are the ones constructed by the Reader, and the Reader can be considered a general purpose data input package: all it does is categorize and collect input into strings, numbers, symbols and lists. That's a very useful structure for organizing any kind of information, not just PScheme programs. The read-eval-print loop will, however, attempt to evaluate any such structure read in, so we need a way of stopping that.

9.1 `quote`

The appropriate form is called `quote` and is a `PScm::SpecialForm`. It takes a single argument and returns it unevaluated:

```
> (quote a)
a
> (quote (+ 1 2))
(+ 1 2)
```

The implementation is rather trivial: Quote is used to turn off evaluation. Since special forms don't have their arguments evaluated for them, all that the `Apply()` method in `PScm::SpecialForm::Quote` need do is to return its first argument, still unevaluated.

Here's `PScm::SpecialForm::Quote`.

```
package PScm::SpecialForm::Quote;

use base qw(PScm::SpecialForm);

sub Apply {
    my ($self, $form, $env) = @_;
    return $form->first;
}

1;
```

9.2 `list`

Another useful operation is called `list`. It takes a list of arguments and constructs a new list from them. It is just a `PScm::Primitive` and so it's arguments are evaluated:

```
> (list (- 8 1) "hello")
(7 "hello")
```

It does nothing itself but return the list of its evaluated arguments to the caller as a new PScm::Expr::List, so it's also trivial to implement. To recap, all PScm::Primitive classes share a common Apply() method that evaluates the arguments then calls the class-specific _apply() method. So all we have to do is to subclass PScm::Primitive to PScm::Primitive::List and give that new subclass an appropriate _apply() method.

```
package PScm::Primitive::List;

use base qw(PScm::Primitive);

sub _apply {
    my ($self, @args) = @_;

    return new PScm::Expr::List(@args);
}

##############################
```

As has been said, it's trivial because it just returns its arguments as a new PScm::Expr::List.

9.3 car and cdr

So we can create lists, but what can we do with them? We already have two primitive internal operations on lists, namely the PScm::Expr::List::first() and rest() methods. They are something like the complement of the list primitive in that they take apart a list into its components. Bowing to historical precedent however, Scheme, and hence PScheme, doesn't call them "first" and "rest", instead they are called car and cdr[14].

Again their implementation is simple, we add a new subclass of PScm::Primitive for each of them, and give each new class an _apply() method that calls the relevant internal method. Here's PScm::Primitive::Car.

```
package PScm::Primitive::Car;

use base qw(PScm::Primitive);

sub _apply {
    my ($self, $arg) = @_;

    $self->_check_type($arg, 'PScm::Expr::List');
    return $arg->first;
}

##############################
```

[14] Obligatory footnote. CAR stands for "the Contents of the Address part of the Register" and CDR stands for "the Contents of the Decrement part of the Register." This is a reference to the original hardware on which the first Lisp system was implemented, and means nothing now but the names have stuck.

It uses `_check_type()` to verify that its argument is a list, then calls its `first()` method, returning the result.

Here's the equivalent `PScm::Primitive::Cdr` class.

```
package PScm::Primitive::Cdr;

use base qw(PScm::Primitive);

sub _apply {
    my ($self, $arg) = @_;

    $self->_check_type($arg, 'PScm::Expr::List');
    return $arg->rest;
}

##############################
```

9.4 `cons`

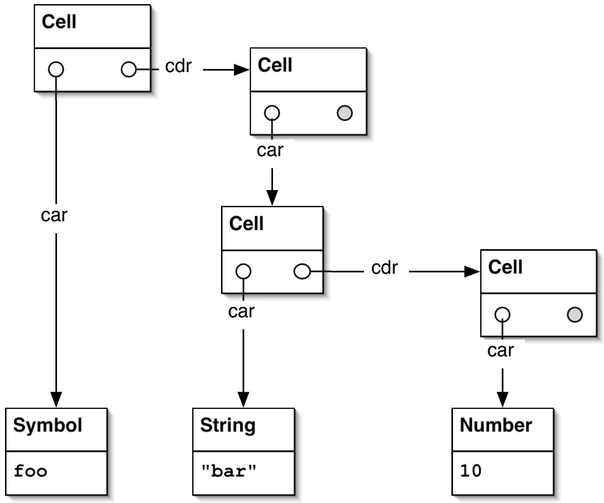
We're only missing one piece from our set of basic list operations now, but before adding that it is necessary to explain a significant deviation that PScheme has so far made from other Lisp and Scheme implementations. In our PScheme implementation lists have so far been implemented as object wrappers around Perl lists. This has the advantage that the implementation is as simple as can be. However real Lisp systems implement lists as chains of what are called *cons cells*. A cons cell is a structure with two components, both references to other data types. For a true list, the first component points at the current list element and the second component points at the rest of the list. The first component is called the `car` and the second component the `cdr`, hence the eponymous functions that access those components. So for example the lisp expression `(foo ("bar" 10))` Is not normally implemented as in Figure 1, but as shown in Figure 19.

This means that a true Lisp list is in fact a linked list. A primary advantage of this is that the internal `first()` and `rest()` (`car` and `cdr`) operations are equally efficient: there is no need for `rest()` to construct a new list object. A second advantage is that cons cells are more flexible than lists. A true list is a chain of cons cells linked by their cdr component, ending in a cons cell with an empty cdr. In general the cdr need not point to another cons cell, it could equally well point to any other data type.

Let's make the change to use that alternative implementation. Since the `PScm::Expr::List` class hides its internal structure and provides accessor methods, that should be the only package that needs to change. Here's the new implementation:

```
package PScm::Expr::List;
use base qw(PScm::Expr);

sub new {
```

Figure 19: Cons Cell Representation of a nested list



```
    my ($class, @list) = @_;

    $class = ref($class) || $class;
    if (@list) {
        my $first = shift @list;
        $class->Cons($first, $class->new(@list));
    } else {
        new PScm::Expr::List::Null();
    }
}

sub value {
    my ($self) = @_;
    my @value = ($self->first, $self->rest->value);
    return @value;
}

sub first {
    $_[0]->{first};
}

sub rest {
    $_[0]->{rest};
}

sub as_string {
    my ($self) = @_;
    return ’(’ . join(’ ’, map { $_->as_string } $self->value) . ’)’;
}
```

```
59
60 sub Eval {
61     my ($self, $env) = @_;
62     my $op = $self->first()->Eval($env);
63     return $op->Apply($self->rest, $env);
64 }
65
66 sub Cons {
67     my ($class, $first, $rest) = @_;
68     bless {
69         first => $first,
70         rest  => $rest,
71     }, $class;
72 }
73
74 ###############################
```

The `new()` method on lines 29-39 is a little more complicated than it was, because it has to recurse on its argument list building a linked list. If the list is not empty then on line 35 it calls an ancilliary method `Cons()` (defined on lines 66-72) to actually construct a new `PScm::Expr::List` node (a cons cell). The empty list is represented by a new class `PScm::Expr::List::Null` which gets tagged on to the end of the newly constructed list.

if you remember the old implementation just wrapped its argument list:

```
29 sub new {
30     my ($class, @list) = @_;
31
32     $class = ref($class) || $class;
33     bless [@list], $class;
34 }
```

The other methods are fairly straightforward.

- The `value()` method on lines 41-45 converts the linked list back into a Perl list[15].
- The `first()` and `rest()` are simplified, they are now just accessors to the equivalent fields.
- The `as_string()` and `Eval()` methods are in fact unchanged from the previous implementations, as they rely entirely on the accessor methods.
- The new `Cons()` method abstracts the creation of a single cons cell, and will be used by our nascent `cons` primitive as well as by `new()` above.

[15] The reason for the temporary `@value` variable is not redundant clarification of the code, it is a workaround for a rather obscure piece of Perl behaviour. If the code had simply said:

`return (`$self- > first$`,self->rest->value);`

Then the scalar context imposed by the `isTrue` method in `PScm::Expr` would cause Perl to apply the comma operator, throwing away the left hand side and returning only the right, recursively, so all lists would end up being treated as false.

The other part of our alternative implementation of lists is that new `PScm::Expr::List::Null` class. It represents the PScheme empty list, and quite reasonably descends from the list class. It provides a simple `new()` method with no arguments, and overrides the `value()` method to just return an empty perl list.

```
package PScm::Expr::List::Null;
use base qw(PScm::Expr::List);

sub new {
    my ($class) = @_;

    $class = ref($class) || $class;
    bless {}, $class;
}

sub value { (); }

sub first { $_[0] }

sub rest { $_[0] }

sub Eval { $_[0] }

############################
```

Interestingly, it also overrides `first()` and `rest()` to return `$self`, so the `car` or `cdr` of the empty list is the empty list, and it overrides `Eval()` to just return `$self` too, so an empty list evaluates to itself.

Back to our `cons` function. Scheme implementations have a `cons` operation that takes two arguments and creates a cons cell with its car referencing the first argument, and its cdr referencing the second.

```
> (cons 10 (list 20 30))
(10 20 30)
```

Thus the `car` and `cdr` operations are the precise complement of `cons`: `cons` constructs a cell, and `car` and `cdr` take the cell apart.

```
> (car (cons 10 (list 20 30)))
10
> (cdr (cons 10 (list 20 30)))
(20 30)
```

Provided the second argument to `cons` is a list, the result will also be a list, so this implementation provides a `cons` function that requires its second argument to be a list and builds a new list from it with its first argument prepended. It's implemented in the normal way, by subclassing `PScm::Primitive` and giving the new class, `PScm::Primitive::Cons` in this case, an `_apply()` method. Here's that method.

```
sub _apply {
    my ($self, $car, $cdr) = @_;

    $self->_check_type($cdr, ’PScm::Expr::List’);
    return PScm::Expr::List->Cons($car, $cdr);
}
```

It can be seen that all it does is to check that it’s second argument is a list, then call the `PScm::Expr::List`’s `Cons` method.

9.5 Summary

In interpreter 0.0.5 five related operations for creating and manipulating list structures have been added. We’ll put those to good use in the next version of the interpreter when we look at macros. In the process of implementing one of those operations, `cons`, the basic list implementation was changed to be closer to a “standard” scheme implementation.

Just for the sake of completeness Here’s the changes to our top-level `PScm::ReadEvalPrint()` method, where we add the new bindings for these functions in the initial environment.

```
sub ReadEvalPrint {
    my ($infh, $outfh) = @_;

    $outfh ||= \*STDOUT;
    my $reader = new PScm::Read($infh);
    while (defined(my $expr = $reader->Read)) {
        my $result = $expr->Eval(
            new PScm::Env(
                let    => new PScm::SpecialForm::Let(),
                ’*’    => new PScm::Primitive::Multiply(),
                ’-’    => new PScm::Primitive::Subtract(),
                if     => new PScm::SpecialForm::If(),
                lambda => new PScm::SpecialForm::Lambda(),
                list   => new PScm::Primitive::List(),
                car    => new PScm::Primitive::Car(),
                cdr    => new PScm::Primitive::Cdr(),
                cons   => new PScm::Primitive::Cons(),
                letrec => new PScm::SpecialForm::LetRec(),
                ’let*’ => new PScm::SpecialForm::LetStar(),
                quote  => new PScm::SpecialForm::Quote(),
            )
        );
        $result->Print($outfh);
    }
}
```

9.6 Tests

These tests exercise our new list functions.

```
use strict;
use warnings;
use Test::More tests => 24;
use FileHandle;

BEGIN { use_ok(’PScm’) }

sub evaluate {
    my ($expression) = @_;

    my $fh = new FileHandle("> junk");
    $fh->print($expression);
    $fh = new FileHandle(’< junk’);
    my $outfh = new FileHandle("> junk2");
    ReadEvalPrint($fh, $outfh);
    $fh    = 0;
    $outfh = 0;
    my $res = ‘cat junk2‘;
    chomp $res;
    unlink(’junk’);
    unlink(’junk2’);

    # warn "# [$res]\n";
    return $res;
}

ok(evaluate(’1’)  eq ’1’,  ’numbers’);
ok(evaluate(’+1’) eq ’1’,  ’positive numbers’);
ok(evaluate(’-1’) eq ’-1’, ’negative numbers’);

ok(evaluate(’"hello"’) eq ’"hello"’, ’strings’);

ok(evaluate(’(* 2 3 4)’)  eq ’24’,  ’multiplication’);
ok(evaluate(’(- 10 2 3)’) eq ’5’,   ’subtraction’);
ok(evaluate(’(- 10)’)     eq ’-10’, ’negation’);

ok(evaluate(’(let ((x 2)) x)’) eq ’2’, ’simple let’);

ok(evaluate(’(if 0 10 20)’) eq ’20’, ’simple conditional’);

ok(evaluate(<<EOF) eq ’20’, ’conditional evaluation’);
(let ((a 00)
      (b 10)
      (c 20))
     (if a b c))
```

listing 15: t/PScm.t

```
EOF

ok(evaluate(<<EOF) eq '16', 'lambda');
(let ((square
       (lambda (x) (* x x))))
     (square 4))
EOF

ok(evaluate(<<EOF) eq '12', 'closure');
(let ((times3
      (let ((n 3))
           (lambda (x) (* n x)))))
     (times3 4))
EOF

ok(evaluate(<<EOF) eq '15', 'higher order functions');
(let ((times3
      (let ((makemultiplier
             (lambda (n)
                     (lambda (x) (* n x)))))
           (makemultiplier 3))))
     (times3 5))
EOF

ok(!defined(eval { evaluate(<<EOF) }), 'let does not support recursion');
(let ((factorial
        (lambda (n)
          (if n (* n (factorial (- n 1)))
                1))))
  (factorial 4))
EOF

ok($@ eq "no binding for factorial in PScm::Env\n", 'let does not support recursion [2]');

ok(evaluate(<<EOF) eq "24", 'letrec and recursion');
(letrec ((factorial
           (lambda (n)
             (if n (* n (factorial (- n 1)))
                   1))))
  (factorial 4))
EOF

ok(evaluate(<<EOF) eq '1', 'let binds in parallel');
(let ((a 1)
```

listing 15: t/PScm.t (Cont.)

```
      (b 2))
     (let ((a b)
           (b a))
          b))
EOF

ok(evaluate(<<EOF) eq '2', 'let* binds sequentially');
(let ((a 1)
      (b 2))
  (let* ((a b)
         (b a))
    b))
EOF

ok(evaluate(<<EOF) eq '(1 2 3)', 'list primitive');
(let ((a 1)
      (b 2)
      (c 3))
  (list a b c))
EOF

ok(evaluate(<<EOF) eq '1', 'car primitive');
(let ((a (list 1 2 3)))
     (car a))
EOF

ok(evaluate(<<EOF) eq '(2 3)', 'cdr primitive');
(let ((a (list 1 2 3)))
     (cdr a))
EOF

ok(evaluate(<<EOF) eq '(1 2 3)', 'cons primitive');
(cons 1 (list 2 3))
EOF

ok(evaluate('(quote (list 1 2 3))') eq '(list 1 2 3)', 'quote special form');

# vim: ft=perl
```

listing 15: `t/PScm.t` (Cont.)

10 Interpreter Version 0.0.6—Macros

What is a macro? People familiar with the C programming language will probably think of macros as being purely a textual substitution mechanism done in some sort of preprocessing step before the compiler proper gets to look at the code. However that's a somewhat limited perspective, perfectly adequate for languages like C but constraining from our point of view. A better definition of a macro is any sort of substitution that can happen before the final code is executed.

The real importance of macros is their potential to allow syntactic extensions to their language. In the case of PScheme, each special form is a syntactic extension to the language, and so our working definition of a PScheme macro could be something that allows us to define our own special forms within the language itself. Here's an example. Suppose the language lacked the `let` special form. as was mentioned in a previous section, `let` shares a good deal in common with `lambda`. In fact any `let` expression, say

```
(let ((a 10)
      (b 20))
     (- b a))
```

has an equivalent `lambda` expression, in this case

```
((lambda (a b) (- b a)) 10 20)
```

The body of the `let` is the same as the body of the `lambda`, and the bindings of the `let` are split between the formal and actual arguments to the `lambda` expression. In general any `let` expression:

```
(let ((⟨var1⟩ ⟨val1⟩)
      (⟨var2⟩ ⟨val2⟩)
      ...)
     ⟨expression⟩)
```

has an equivalent lambda form:

```
((lambda (⟨var1⟩ ⟨var2⟩ ...)
         ⟨expression⟩)
    ⟨val1⟩ ⟨val2⟩ ...)
```

Of course internally `let` doesn't make use of closures, but in the case of the `lambda` equivalent to `let`, the `lambda` expression is evaluated immediately in the same environment as it was defined, so closure is immaterial. All that our purported `let` macro need do then, is to rewrite its arguments into an equivalent `lambda` form and have that executed in its place. We developed all of the tools we will need to do that in the 0.0.5 interpreter in a previous section. All we need to do now is to think of a way to allow us to define macros, how hard can it be? Macros will obviously share a great deal in common with functions. They will have a separate declaration and use. They will also take arguments, and have a body that is evaluated in some way. In fact the first part of their implementation, that of parsing their declaration will be virtually identical to that of `lambda` expressions, except that the `lambda` keyword is already taken. Tke keyword "`macro`" can be used in its place.

10.1 **macro**

As before then, we subclass PScm::SpecialForm and give the new class an Apply() method. The new class is called PScm::SpecialForm::Macro after its eponymous symbol. Here's the Apply() method for PScm::SpecialForm::Macro.

```
package PScm::SpecialForm::Macro;

use base qw(PScm::SpecialForm);
use PScm::Closure;

sub Apply {
    my ($self, $form, $env) = @_;
    my ($args, $body) = $form->value;
    return PScm::Closure::Macro->new($args, $body, $env);
}

#################################
```

It's virtually identical to PScm::SpecialForm::Lambda except that it creates a new PScm::Closure::Macro (not to be confused with this PScm::SpecialForm::Macro class) instead of a PScm::Closure::Function. So we've left the problem of how to make a macro actually work until last, in the PScm::Closure::Macro's Apply() method.

10.2 Evaluating Macros

Consider how PScm::Closure::Function works. It evaluates its arguments in the passed-in environment then gives the results to its parent _apply() method which extends the environment that was captured when the closure was created with bindings of those actual arguments to its formal arguments. Then it evaluates its body in that extended environment and returns the result. Here again is PScm::Closure::Function's Apply() method:

```
sub Apply {
    my ($self, $form, $env) = @_;

    my @evaluated_args = map { $_->Eval($env) } $form->value;
    return $self->_apply(@evaluated_args);
}
```

And here's the private _apply() method in the base PScm::Closure class:

```
sub _apply {
    my ($self, @args) = @_;

    my $extended_env = $self->env->ExtendUnevaluated([$self->args], [@args]);
    return $self->body->Eval($extended_env);
}
```

Any implementation of macros will share something in common with this implementation of functions, but there will be differences. Obviously a macro should be passed its arguments unevaluated. That way it can perform whatever (list) operations it likes on that structure. Then when it returns a new form, it is that form that gets evaluated. It's as simple as that, and here's the `Apply()` method for `PScm::Closure::Macro`:

```
package PScm::Closure::Macro;

use base qw(PScm::Closure);

sub Apply {
    my ($self, $form, $env) = @_;

    my @unevaluated_args = $form->value;
    my $new_form         = $self->_apply(@unevaluated_args);
    return $new_form->Eval($env);
}

1;
```

Compare that with the `Apply()` method from `PScm::Closure::Function` above.

Functions evaluate their arguments, then evaluate their body with those arguments bound. Macros don't evaluate their arguments, they evaluate their body with their unevaluated arguments bound, then re-evaluate the result.

Just for fun, let's take a look at how we might attack the problem which introduced this section: implementing `let` in terms of `lambda`.

```
(let* ((mylet
         (macro (bindings body)
                (let* ((names (cars bindings))
                       (values (cadrs bindings)))
                  (cons (list (quote lambda) names body) values)))))
  (mylet ((a 1)
          (b 2))
         (list a b)))
```

This code uses `let*` (remember we're pretending that we don't have `let`) to bind mylet to a `macro` definition, then it uses mylet in the body of the `let*`. It makes use of some supporting functions that we'll define presently, but first let's try to get a feel for what it is doing. As stated above, the symbol `macro` introduces a macro definition. The arguments to `mylet` will be the same as those to `let`, namely a list of bindings and a body to execute with those bindings in place. It has to separate the bindings (symbol-value pairs) into two lists, one of the symbols and one of the values. It uses a function cars to extract the car of each binding (the symbol) into the list called names.

Here's the definition of cars:

```
(letrec (...
          (cars
            (lambda (lst)
              (if lst
                (cons (car (car lst))
                      (cars (cdr lst)))
                (quote ()))))
          ...)
  ...)
```

It's a recursive function, hence the need for `letrec` to bind it. Passed a list of zero or more bindings, it collects the car of each into a new list. It does this by checking the argument list. If the list is empty it returns the empty list, otherwise it conses the car of the first component of the list with the result of calling itself on the rest of the list.

`cadrs`[16] is very similar. It walks the list collecting the second component of each sublist (the values of the bindings).

```
(letrec (...
          (cadrs
            (lambda (lst)
              (if lst
                (cons (car (cdr (car lst)))
                      (cadrs (cdr lst)))
                (quote ()))))
          ...)
  ...)
```

Here's the whole thing.

```
(let* ((mylet
         (letrec ((cars
                    (lambda (lst)
                      (if lst
                        (cons (car (car lst))
                              (cars (cdr lst)))
                        (quote ()))))
                  (cadrs
                    (lambda (lst)
                      (if lst
                        (cons (car (cdr (car lst)))
                              (cadrs (cdr lst)))
                        (quote ())))))
           (syntax (bindings body)
                   (let ((names (cars bindings))
                         (values (cadrs bindings)))
                     (cons (list (quote lambda) names body) values))))))
  (mylet ((a 1)
          (b 2))
         (list a b)))
```

[16] The term `cadr` is a contraction of "`car` of the `cdr`" e.g. `(cadr x) == (car (cdr x))`. this sort of contraction is often seen in scheme code, sometimes nested as much as four or five levels deep, i.e. `cadadr`.

After collecting the names into one list and the values into another, the `mylet` macro builds:

```
((lambda (<names>) (<body>)) <values>)
```

Where ¡names¿, ¡body¿ and ¡values¿ are expanded. Note the use of list to construct lists of lists, contrasted with the use of cons to prepend the body to the argument list, making it the first form. That means we get

```
((lambda (a b) (list a b)) 1 2)
```

instead of the incorrect

```
((lambda (a b) (list a b)) (1 2)).
```

Another point to note is that the constructed `mylet` macro is a true closure, since it has captured the definitions of the `cars` and `cadrs` and executes in an outer environment (the `let*`) where those functions are no longer defined.

10.3 Summary

The macro substitution system demonstrated here is pretty crude, after all it requires the programmer to directly manipulate low-level list structures, rather than just supplying an "example" of how the transformation is to be performed. In fact the topic of macro expansion as provided by a full Scheme implementation is deserving of a book to itself. Apart from the templating ability, there are also issues of avoiding variable collision (so-called *hygenic macros*) so that full scheme macros are much closer to the idea of C++'s `inline` functions than they are to C's `#define`.

Here's the additions to `ReadEvalPrint()` which bind our new macro feature to the initial environment:

```
sub ReadEvalPrint {
    my ($infh, $outfh) = @_;

    $outfh ||= \*STDOUT;
    my $reader = new PScm::Read($infh);
    while (defined(my $expr = $reader->Read)) {
        my $result = $expr->Eval(
            new PScm::Env(
                let    => new PScm::SpecialForm::Let(),
                '*'    => new PScm::Primitive::Multiply(),
                '-'    => new PScm::Primitive::Subtract(),
                if     => new PScm::SpecialForm::If(),
                lambda => new PScm::SpecialForm::Lambda(),
                list   => new PScm::Primitive::List(),
                car    => new PScm::Primitive::Car(),
                cdr    => new PScm::Primitive::Cdr(),
                cons   => new PScm::Primitive::Cons(),
                letrec => new PScm::SpecialForm::LetRec(),
                'let*' => new PScm::SpecialForm::LetStar(),
                macro  => new PScm::SpecialForm::Macro(),
                quote  => new PScm::SpecialForm::Quote(),
            )
        );
        $result->Print($outfh);
    }
}
```

11 Interpreter Version 0.0.7—Side Effects

The question arises as to why the implementation of the `define` special form has beed deferred for so long, when it would have made so much of the previous discussion easier, particularily the Scheme examples. Well there are good reasons. Consider what the language so far has got.

11.1 The Beauty of Functional Languages

So far, the language is fairly complete. It is however a purely functional language, in the sense that its expressions are like mathematical functions: the value of a variable is fixed for the duration of the expression. There is no sequential execution (though there is recursion) and most importantly there are no *side effects*. That means that one part of a program cannot affect the execution of another part that it does not contain, except by returning values that get passed in to that other part.

While this might seem to be a limitation of the language, it does in fact have some very important advantages. Essentially it makes large-scale expressions in the language very simple to analyse: since one part of an expression cannot affect another structurally independant part except through its inputs (arguments,) different components of a program (including the current environment) could be fed to separate processes, perhaps even separate machines on a network.

To make this clear, suppose we had a networked version of the language. There would be a pool of processes available to do work, and each process could delegate sub-parts of their work to other processes. A simple rule might be that for parallel binding, e.g. `let` bindings and the arguments to closures and primitive functions, the evaluation of the arguments could be performed by "outsourcing" the evaluation of each subexpression to a different process. For example consider the following fragment:

```
...
(let ((a (func1 x))
      (b (func2 y))
      (c (func3 z)))
  (func4 (func5 a) (func6 b) (func7 c)))
...
```

The `let` could elect to send the subexpression `(func1 x)` (plus the environment where `func1` and `x` have a value) to one process, and the subexpression `(func2 y)` to another. While those two expressions are being evaluated the `let` could get on with evaluating `(func3 z)` itself. Then when it had finished that evaluation it would collect the result of the other two evaluations and proceed to evaluate its body. Similarily the evaluation of the body (the call to `func4`) could outsource the evaluation of the arguments `(func5 a)` and `(func6 b)` and get on with evaluating `(func7 c)`, collecting the other results when it finished, and then proceeding to evaluate the `func4` call with those evaluated arguments. It probably shouldn't outsource something a simple as variable lookup.

The implementation of this networked programming language is left to you. Some sort of central ticketing server would be needed, with a queue where requestors could post their requests in exchange for a ticket, and a pool where clients could post their results so that they need not wait for the requestor to get back to them... quite an interesting project.

Beyond this point that kind of application becomes nearly impossible because we are about to introduce *side effects*, in particular *variable assignment* and, to make that useful, *sequences* of expressions. That's just the proper name for something we're all very familiar with—one *statement* following another.

In fact, the primary difference between a *statement* and an *expression* is that a statement is executed primarily for its side effects. There are only a couple of different side effects that we need to consider. The first is *variable assignment*, that is to say the changing of the existing value of a variable binding. The second is *global definition*, the creation of new entries in the initial environment.

11.2 Variable Assignment

This version of the interpreter will only add variable assignment (and sequences,) global definition is left for later. Variable assignment should be a special form, so that the variable being assigned to does not get evaluated. In Scheme, and PScheme, the symbol bound to the variable assignment special form is `set!`, the exclaimation point signifying that this operation has a side effect that might upset an otherwise purely functional program[17]. The syntax is simple, but there is a gotcha:

```
(set! a 10)
Error: no binding for a in PScm::Env
```

This may sound unnecessarily picky, but for variable assignment to work, there must already be a binding in place that the assignment can affect. This works:

```
> (let ((a 5))
>      (let (dummy (set! a 10))
>           a))
10
```

This is a bit contrived, we use a dummy variable to allow the `set!` expression to be evaluated before the body of the inner `let` returns the new value of `a`. However it should be clear from the example that the `set!` did take effect.

So how do we go about implementing `set!`? Well as luck would have it, we already have a method for changing existing bindings in an environment, we have the `Assign()` method that was developed for the `letrec` special form in a previous incarnation. It has precisely the right semantics too (what a coincidence!) It searches through the chain of environment frames until it finds an appropriate binding, assigning the value if it does and `die()`-ing if it doesn't. Here it is again:

```
sub Assign {
    my ($self, $symbol, $value) = @_;

    if (defined(my $ref = $self->_lookup_ref($symbol))) {
        $$ref = $value;
```

[17] the exclaimation point is used to suffix all such side-effecting operations, with the exception of `define`, for some reason

```
    } else {
        die "no binding for @{[$symbol->value]}",
            " in @{[ref($self)]}\n";
    }
}
```

So all we have to do is wire it up. The process of creating new special forms should be familiar by now. Firstly we subclass `PScm::SpecialForm` and give it an `Apply()` method. The new class in this case is `PScm::SpecialForm::Set`. Its `Apply()` method as usual takes a form and the current environment as arguments. In this case the form should contain a symbol and a value expression. This `Apply()` will evaluate the expression and call the environment's `Assign()` method with the symbol and resulting value as arguments:

```
package PScm::SpecialForm::Set;

use base qw(PScm::SpecialForm);

sub Apply {
    my ($self, $form, $env) = @_;
    my ($symbol, $expr) = $form->value;
    $env->Assign($symbol, $expr->Eval($env));
}

#################################
```

And that's all there is to it, apart from adding a `PScm::SpecialForm::Set` object to the initial environment, bound to the symbol `set!`.

11.3 Sequences

Sequences are a fairly trivial addition to the language. Rather than a single expression, a *sequence* contains one or more expressions. Each expression is evaluated in order, and the value of the last expression is the value of the sequence. This is such a common thing in many other languages (such as Perl) that it goes without notice, but thinking about it, sequences only become relevant and useful in the presence of side effects. Since only the value of the last expression is returned, preceding expressions can only affect the computation if they have side effects.

The keyword introducing a sequence in PScm is `begin`. `begin` takes a list of one or more expressions, evaluates each one, and returns the value of the last expression. It functions something like a block in Perl, except that a Perl block also encloses a variable scope like `let` does, but `begin` does not imply any scope.

```
(begin <expression1> <expression2> ...)
```

`begin` could, in fact, be implemented as a function, provided that functions are guaranteed to evaluate their arguments in left-to-right order, as this implementation does. However it is safer not to make that assumption (remember the networked language where evaluation was envisaged in parallel,) so `begin` is better implemented as a special form which can guarantee that left to right behaviour.

As might be imagined the code is quite trivial: evaluate each component of the form and return the last value. We pick as an initial value the empty list, so that if the body of the `begin` is empty, that is the result. As usual we subclass `PScm::SpecialForm`, in this case to `PScm::SpecialForm::Begin`, and give the new class an `Apply()` method:

```
126 package PScm::SpecialForm::Begin;
127
128 use base qw(PScm::SpecialForm);
129
130 sub Apply {
131     my ($self, $form, $env) = @_;
132     my (@expressions) = $form->value;
133
134     my $ret = new PScm::Expr::List();
135
136     while (@expressions) {
137         $ret = shift(@expressions)->Eval($env);
138     }
139
140     return $ret;
141 }
142
143 1;
```

On line 132 it extracts the expressions from the argument `$form`. Then on line 134 it initialises the return value `$ret` to an initial value. On lines 136-138 it loops over each expression, evaluating it in the current environment, and assigning the result to `$ret`, replacing the previous value. Lastly on line 140 it returns the final value.

11.4 Summary

In this section we've seen variable assignment added to the language, but also taken some time to consider some drawbacks of that feature. We've also looked at how sequences become useful in the presence of variable binding.

As usual we need to add our new forms to the interpreter by binding them in the initial environment. Here's `ReadEvalPrint()` with the additional bindings.

```
sub ReadEvalPrint {
    my ($infh, $outfh) = @_;

    $outfh ||= \*STDOUT;
    my $reader = new PScm::Read($infh);
    while (defined(my $expr = $reader->Read)) {
        my $result = $expr->Eval(
            new PScm::Env(
                let    => new PScm::SpecialForm::Let(),
                '*'    => new PScm::Primitive::Multiply(),
                '-'    => new PScm::Primitive::Subtract(),
                if     => new PScm::SpecialForm::If(),
                lambda => new PScm::SpecialForm::Lambda(),
                list   => new PScm::Primitive::List(),
                car    => new PScm::Primitive::Car(),
                cdr    => new PScm::Primitive::Cdr(),
                cons   => new PScm::Primitive::Cons(),
                letrec => new PScm::SpecialForm::LetRec(),
                'let*' => new PScm::SpecialForm::LetStar(),
                macro  => new PScm::SpecialForm::Macro(),
                quote  => new PScm::SpecialForm::Quote(),
                'set!' => new PScm::SpecialForm::Set(),
                begin  => new PScm::SpecialForm::Begin(),
            )
        );
        $result->Print($outfh);
    }
}
```

12 Interpreter Version 0.0.8—`define`

And so to `define`. `define` is another type of side effect. It differs from `set!` in that it is an error if the symbol already has a binding, and it differs also in that the binding is always installed in the current environment.

12.1 Environment Changes

Code that manipulates environments is best defined in the environment package `PScm::Env`, and that's what we do. The additional method in `PScm::Env` is called, unsurprisingly, `Define()` and it's quite simple.

```
98  sub Define {
99      my ($self, $symbol, $value) = @_;
100
101     if (exists($self->{bindings}{ $symbol->value })) {
102         die "attempt to redefine @{[$symbol->value]}\n";
103     } else {
104         $self->{bindings}{ $symbol->value } = $value;
105         return $symbol;
106     }
107 }
```

It takes a symbol and a value (already evaluated) as arguments. On line 101 it checks that a binding for the symbol does not exist in the current frame. If it does it dies with an informative message. Otherwise, on line 104 it directly adds the binding from the symbol value (i.e. string) to the value, and on line 105 it returns the symbol being defined, to give the print system something sensible to print.

12.2 The `define` Special Form

Now we need to follow the usual procedure to add another special form to the language: we subclass `PScm::SpecialForm` and give our new class an `Apply()` method. In this case the new class is called `PScm::SpecialForm::Define`.

```
144 package PScm::SpecialForm::Define;
145
146 use base qw(PScm::SpecialForm);
147
148 sub Apply {
149     my ($self, $form, $env) = @_;
150     my ($symbol, $expr) = $form->value;
151     $env->Define($symbol, $expr->Eval($env));
152 }
153
154 1;
```

All it does is on line 150 it extracts the symbol and the expression from the argument `$form` then on line 151 it calls the `Define()` environment method described above with the symbol and evaluated expression (value) as argument.

12.3 Persistant Environments

There is one more change to make. In order for `define` to be effective from one expression to another, it no longer makes sense to create a fresh environment for each expression to be evaluated in, as `ReadEvalPrint` has done so far, because that would eradicate the effect of any `define` performed by a prior expression. The solution is of course trivial, we create the initial environment outside of the read-eval-print loop itself, and pass it to each top-level `Eval()`:

```
sub ReadEvalPrint {
    my ($infh, $outfh) = @_;

    $outfh ||= \*STDOUT;
    my $reader      = new PScm::Read($infh);
    my $initial_env = new PScm::Env(
        let    => new PScm::SpecialForm::Let(),
        '*'    => new PScm::Primitive::Multiply(),
        '-'    => new PScm::Primitive::Subtract(),
        if     => new PScm::SpecialForm::If(),
        lambda => new PScm::SpecialForm::Lambda(),
        list   => new PScm::Primitive::List(),
        car    => new PScm::Primitive::Car(),
        cdr    => new PScm::Primitive::Cdr(),
        cons   => new PScm::Primitive::Cons(),
        letrec => new PScm::SpecialForm::LetRec(),
        'let*' => new PScm::SpecialForm::LetStar(),
        macro  => new PScm::SpecialForm::Macro(),
        quote  => new PScm::SpecialForm::Quote(),
        'set!' => new PScm::SpecialForm::Set(),
        begin  => new PScm::SpecialForm::Begin(),
        define => new PScm::SpecialForm::Define(),
    );

    while (defined(my $expr = $reader->Read)) {
        my $result = $expr->Eval($initial_env);
        $result->Print($outfh);
    }
}
```

Now we can actually write some of the earliest examples from this chapter in the language at hand.

```
> (define factorial
>        (lambda (x)
>            (if x
>                (* x (factorial (- x 1)))
>                1)))
factorial
> (factorial 4)
24
```

The factorial function was chosen to demonstrate that functions created by define can call themselves recursively. After all, the environment they are executed in must, by virtue of how define operates, contain a binding for the function itself.

define can be used for other things too. Because of the simple semantics of the PScheme language, define is perfectly suited for creating aliases to existing functions. For instance if a programmer doesn't like the rather obscure names for the functions car and cdr, they can provide aliases:

```
> (define first car)
first
> (define rest cdr)
rest
```

These are completely equivalent to the original functions except in name. The primitive definitions are bound to symbols in the initial environment exactly as they are (still) bound to their original symbols.

13 Interpreter Version 0.0.9—Classes and Objects

Almost every modern programming language has an object-oriented extension or variant available. Some languages, such as SmallTalk and Ruby are "pure" object-oriented languages in that *everything* in the language is an object[18]. Others, such as Perl, add object-oriented features to what is essentially a procedural core.

Practically every implementation of an object-oriented language or language extension has its peculiarities. There are a lot of trade-offs and choices to be made. Most of these differences come down to issues of *visibility* of components of the objects to other parts of the program being written in the language: Should the value fields of an object be visible outside of that object? Should an object be able to see the fields in an object it inherits from? Should certain methods of an object be hidden from the outside world? from its descendants?

The implementation discussed here makes choices in order to leverage existing code. Those choices result in a particular object-oriented "style".

13.1 Features of this implementation

- fields are only visible to methods
- fields behave like normal variables
- fields are not visible to methods in descendant classes

classes objects methods private protected public messages fields inheritance single multiple

PScm::SpecialForm::Class PScm::SpecialForm::DummyMethod PScm::Method PScm::BuiltinMethod PScm::Env::Object PScm::Env::Super PScm::Class PScm::Class::Root PScm::Closure::Method

[18] If you don't know SmallTalk, you might be surprised at how far that statement goes. Not only are the simple numeric and string data types objects, but arrays, hashes (called Dictionaries), booleans, code blocks and even classes are objects in SmallTalk

14 Interpreter Version 0.0.10—Continuations

15 Summary

In this chapter we've watched the evolution of a programming language from humble beginnings to a powerful and complete implementation. Starting from a global environment model with basic arithmetic and conditional evaluation, we introduced an environment passing model which made possible the implementation of local variables and much else besides. We also reasoned that trees are a much better structure for combining environment frames than stacks are, especially in the next section where we introduced function definition and closure. We then went on to introduce recursive functions, and showed that a different kind of binding is necessary to get recursive functions to work. Moving on, we introduced another type of binding which is performed sequentially. In the next section we looked at adding list processing to the language, allowing it to manipulate directly the structures that the language is composed of. Then in posession of that new set of functions we added a macro facility that allowed the program to rewrite parts of its own structure.

Before adding other desirable features to the language we paused to describe the benefits of a language without such features, and noted that such a pure functional language was amenable to parallel evaluation. Brushing aside those concerns we lastly moved on to add side effects, (both definition and assignment) and sequences.

That's all for now.