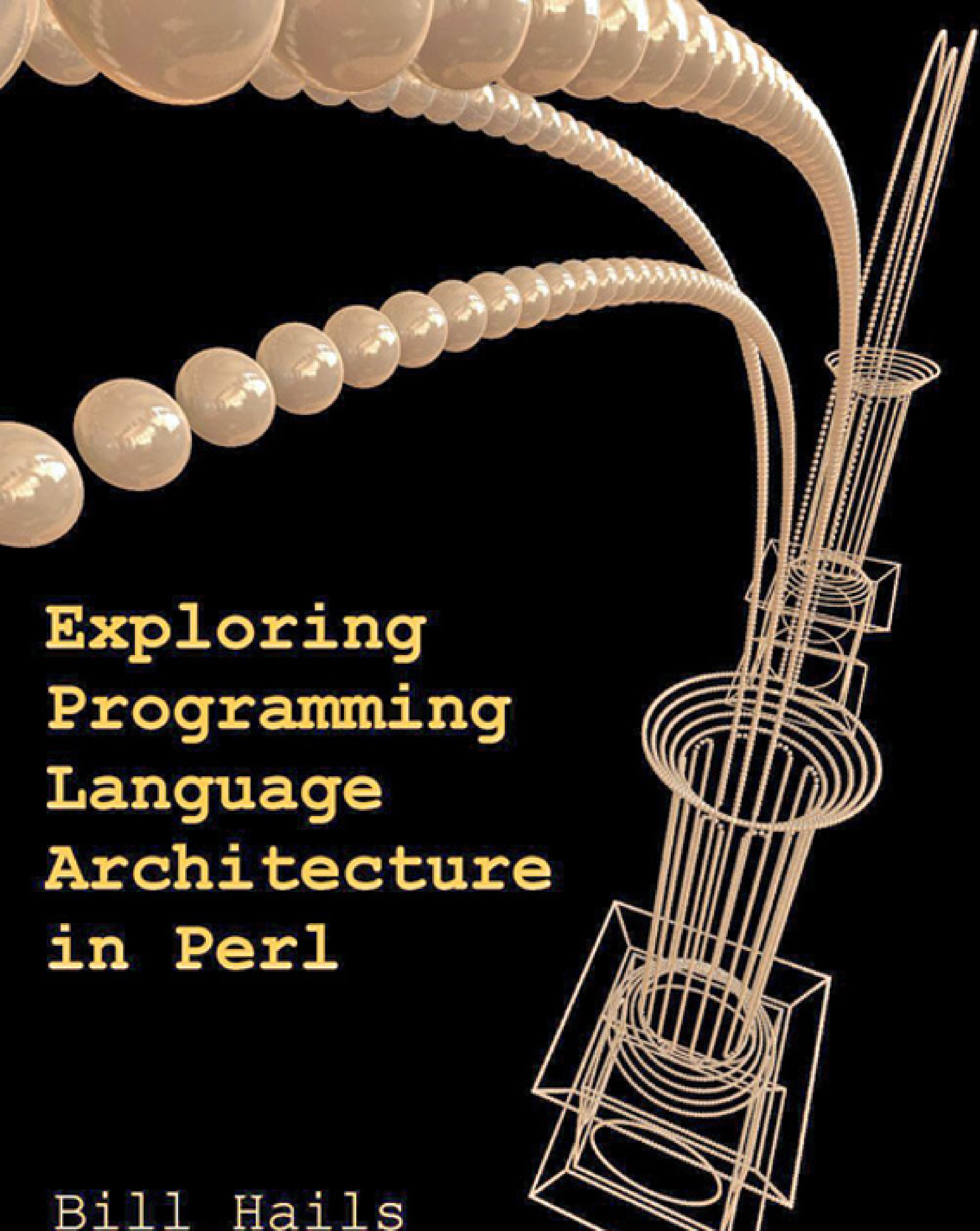


Exploring Programming Language Architecture in Perl

Bill Hails

Exploring Programming Language Architecture in Perl

Bill Hails

Sun Mar 14 10:43:13 2010 UST

Author may be contacted at me@billhails.net.

Typeset by the author in XML and translated into $\LaTeX2\varepsilon$ by custom software.
Cover image written in POVRay by the author.

Description: an online book using the Perl programming language to explore various aspects of programming language architecture.

Keywords: perl, scheme, interpreter, pscheme
ISBN: 978-1-4452-2592-0

Contents

List of Figures

Part I

Implementing a Scheme-Like Interpreter

Chapter 1

Introduction

Why would anyone want to write a Scheme interpreter in Perl?
– *Felix Lee*

Madness.
– *Larry Wall*

By the end of this book 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. 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 described to the best of my ability without any unexplained technical jargon. It is however assumed that you have 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;

- threads;
- exceptions;
- non-determinism and chronological backtracking;
- logic programming.

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, addition 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 a production-quality, or even an efficient implementation. It is just meant to be easy to understand.

Putting it another way, if you've come here looking for an off-the shelf scheme-like interpreter that you can *use*, you've come to the wrong place: there are many and better freely available implementations on the net. On the other hand if you're more interested in how such interpreters might *work*, I'd like to think that you might find what you're looking for here.

1.1 Why Perl?

My motivation for writing this book is that I have always been fascinated by the fact 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"[1] implementations of programming languages 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 until the core ideas became obscured and obfuscated. The fact that they were invariably implemented in the low-level system language C didn't help matters. 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.

Another reason for my wanting to write about programming languages is the wonderful *gestalt* inherant in their construction: how such an apparently small amount of code could achieve so much. This is of course due to the deeply recursive nature of their design, and this small implementation in Perl attempts to be as concise and recursive as possible.

1.2 Why Scheme?

Scheme is one of the younger members of the *Lisp* family of programming languages. LISP stands for "LISt Processing"[2], 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

[1] yes, I'm older than your grandfather

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

that if C is the “chess” of programming languages, then Scheme is more like “go”. The official standard for Scheme, the “Revised(6) Report on the Algorithmic Language Scheme” or R^{6}RS [12] 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 consistent 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 directly 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.

1.3 References

I provide and refer to a select bibliography. Almost all of the concepts in this book are well known in academic circles and it would be disingenuous of me to try to pass them off as my own. The bibliography should provide you with a small collection of useful jumping off points should you wish to investigate any of these topics further.

1.4 Typography

All of the source code listings and extracts from the source are shown in `fixed-width type` with line numbers, and are pulled directly from the code of the working interpreter. Furthermore, when displaying a newer version of an individual method or package, the differences bethween that version and the previous one are calculated automatically and displayed in bold. Package names are displayed **Like::This** and methods like `this()`. Scheme code looks `(like this)`.

Other in-line code, such as Scheme and occasionally Perl examples are unfortunately not so rigorously constrained. The possibility exists that even though I have manually tested all of those examples there could be an error or two in there, for which I can only apologise.

1.5 A Note on the Interpreter Versions

You won’t find this interpreter on CPAN. The reason for this is that each version of the interpreter, while building on the features of previous versions, is a thing of itself and exists to demonstrate specific pedagogical points. I couldn’t publish only the final version to CPAN since it is fit for no purpose other than this book and too complex for casual perousal without having first digested the earlier versions.

However each version is available online for you to download and play with, they are linked to from the text at the end of each chapter.

Chapter 2

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.

2.1 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:

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

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.[1]

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 or round bracket. So “`x`”, “`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;

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

- 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.

2.2 Simple Expressions

PScheme is an interactive language, it presents a prompt, and the user types in expressions. The interpreter evaluates those expressions then prints the results:

```
> 2
2
```

The ">" is the PScheme prompt. We gave the interpreter a `2`, and it replied with 2, because 2 is 2 is 2.

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.

Here's something that does work:

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

Now that might look strange at first, but remember the first subexpression in a list should evaluate to a function. The multiplication symbol "`*`" does indeed evaluate to the internal primitive definition of how to multiply; we told PScheme to multiply 2 by 2, and it replied 4. In detail what it has done is:

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;
5. printed the result: `4`.

One 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;
5. Printing the result: `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. 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)`.

2.3 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 *consequent* (true result) and an *alternative* (false result). For example:

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

In this example since the test, `0`, is false (again, in this implementation) the alternative $(8 - 3 = 5)$ 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, it then 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`, 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 the result is true it evaluates and returns its second argument (the consequent), otherwise it evaluates and returns its third argument (the alternative). We can demonstrate that with a simple example:

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

Because the test result was true, the `if` only evaluated the consequent expression, there was no error from the undefined symbol `x` in the alternative.

2.4 Global Variables

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

(define ⟨*symbol*⟩ ⟨*expression*⟩)

where ⟨*symbol*⟩ is being bound to the value of ⟨*expression*⟩.

For example:

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

In the above example we defined `x` to be `5`. Then when we asked for the value of `x` PScheme 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:

```
> (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`.

2.5 Functions

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

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

The (⟨*symbol*⟩ ...) part is the names of the arguments to the function, and the ⟨*expression*⟩ is the body of the function.

Here's an 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 more or less equivalent 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[2].

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

[2] 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. Since the definition here mimics the intended usage of the function it 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 syntax for function definition.

```
> (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 also be called directly without giving them a name first:

```
> ((lambda (x) (* x x)) 3)
9
```

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 which will square it's argument. That function then immediately gets applied to `3` resulting in `9`. It *is* possible to do something similar in perl, like this:

```
sub { my ($x) = @_; $x * $x }->(3);
```

but it's not a common idiom.

As an aside, you may be wondering what the eleventh letter of the Greek alphabet has to do with the creation of a function. The term comes from a branch of mathemetics called the *lambda calculus* which is concerned with describing and reasoning about the behaviour of mathematical functions in general. Even though the lambda calculus was devised before the creation of the first computer, it turns out that it provides a sound theoretical basis for the implementation of programming languages, and Lisp was the first programming language to exploit that fact. There is a good introduction to the lambda calculus in [10], and a more detailed and rigorous treatment in [11].

2.6 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⟩)
```

`let` takes a list of *bindings* (symbol-value pairs) and a body to execute with those bindings in effect.

For example:

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

That can be read aloud as "let $a = 10$ and $b = 10 + 10$ in the expression $a + b$". Symbol a is given the value 10 and symbol 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 weren't global bindings of a and b in effect.)

The careful 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 (* n x) 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[3].

[3] If you want more of an introduction to Scheme in general, you could do worse than look at [6].

Chapter 3

Interpreter Version 0.0.0

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

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, then *loop*. This long-winded term is often abbreviated to *repl*. In order for the repl to evaluate the expression, there must additionally be an environment in which symbols can be given values and in which values can be looked up. All this means that there are six 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 (and incidentally returned by the Evaluator, representing the result);
- **An *Environment*** in which symbols can be associated with values and the values of symbols can be looked up.
- **A Set of *Primitive Operations*** bound to symbols in the initial environment, which implement all of the individual built in commands.

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 OO approach, with each of these components and pretty much everything else represented by classes of objects. As a consequence of this the Evaluator and the Print system are distributed throughout the Structure component. This means that for example to evaluate an expression you call its `Eval` method, and to print a result you call the `Print` method on 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.

3.1 The Read-Eval-Print Loop

The top-level read-eval-print loop (repl) for the PScheme interpreter is in the package **PScm** in Listing 3.11.1 on page 34. All other packages inherit from this package, although that's mainly just a convenience.

Firstly, on Lines 31-35 a global environment, `$PScm::GlobalEnv` is initialised to a new **PScm::Env** object.

```
031 our $GlobalEnv = new PScm::Env(
032     ’*’ => new PScm::Primitive::Multiply(),
033     ’-’ => new PScm::Primitive::Subtract(),
034     if  => new PScm::SpecialForm::If(),
035 );
```

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 37-46 is the central control routine of the whole interpreter. It takes an input file handle and an output file handle as arguments. Starting on Line 40 it defaults the output file handle to stdout, then on Line 41 it creates a new **PScm::Read** object on the input file handle, and on Lines 42-45 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:

```
037 sub ReadEvalPrint {
038     my ($infh, $outfh) = @_;
039
040     $outfh ||= new FileHandle(">-");
041     my $reader = new PScm::Read($infh);
042     while (defined(my $expr = $reader->Read)) {
043         my $result = $expr->Eval();
044         $result->Print($outfh);
045     }
046 }
```

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.

3.2 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 3.11.2 on page 36.

It is no more than an object wrapper around a Perl hash. The `new()` method (Lines 7-11) creates an object with a set of bindings (name to value mappings) that were passed in as arguments:

```
007 sub new {
008     my ($class, %bindings) = @_;
009
010     bless { bindings => {%bindings}, }, $class;
011 }
```

The `LookUp()` method on Lines 13-22 looks up a symbol in the bindings, `die`-ing if the symbol does not have a binding:

```
013 sub LookUp {
014     my ($self, $symbol) = @_;
015
016     if (exists($self->{bindings}{ $symbol->value })) {
017         return $self->{bindings}{ $symbol->value };
018     } else {
019         die "no binding for @{[$symbol->value]} ",
020           "in @{[ref($self)]}\n";
021     }
022 }
```

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.

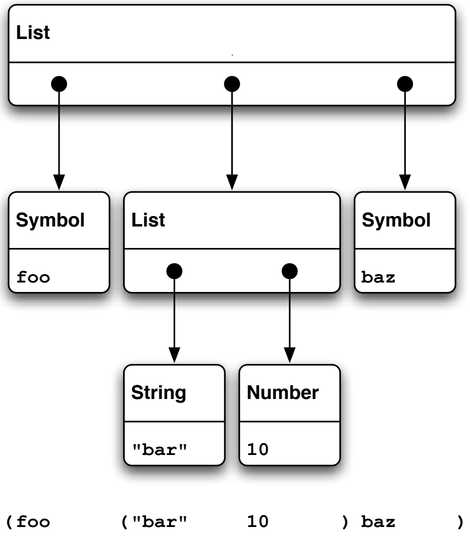
Because this first version of the interpreter has no support for local variables, this class doesn't provide any methods for adding values to the environment. That will come later.

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

3.3 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) baz)` and convert it into an equivalent structure such as shown in Figure 3.1.

Figure 3.1: Example PScheme Structure for `(foo ("bar" 10) baz)`



In this figure, showing the result of parsing that expression, the top-level list object has three components. Reading left to right it contains the symbol object foo, another list object and the symbol object baz. The 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) baz) corresponds directly to the nested perl listref [$foo, ["bar", 10], $baz][1].

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

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

A builder or parser which assembles those tokens into meaningful structures (lists).

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 3.11.3 on page 37 is only 63 lines long.

As with the rest of the implementation, it uses an OO 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(). 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.

```
009 sub new {
010     my ($class, $fh) = @_;
011     bless {
012         FileHandle => $fh,
013         Line       => '',
014     }, $class;
015 }
```

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 tokenise, 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 recognised, it is returned to the caller.

```
038 sub _next_token {
039     my ($self) = @_;
040
041     while (!$self->{Line}) {
042         $self->{Line} = $self->{FileHandle}->getline();
043         return undef unless defined $self->{Line};
```

[1] but looks a lot prettier.

```
044         $self->{Line} =~ s/^\s+//s;
045     }
046
047     for ($self->{Line}) {
048         s/^\(\s*// && return PScm::Token::Open->new();
049         s/^\)\s*// && return PScm::Token::Close->new();
050         s/^([-+]?\d+)\s*//
051           && return PScm::Expr::Number->new($1);
052         s/^"((?:(?:\\.)|([^"]))*)"\s*// && do {
053             my $string = $1;
054             $string =~ s/\\//g;
055             return PScm::Expr::String->new($string);
056         };
057         s/^([^\s\(\)]+)\s*//
058           && return PScm::Expr::Symbol->new($1);
059     }
060     die "can't parse: $self->{Line}";
061 }
```

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** contain the relevant number, string or symbol.

Now that we know how `_next_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 symbol, string or number, 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).

```
017 sub Read {
018     my ($self) = @_;
019
020     my $token = $self->_next_token();
021     return undef unless defined $token;
022
023     return $token unless $token->is_open_token;
024
025     my @res = ();
026
027     while (1) {
028         $token = $self->Read;
029         die "unexpected EOF"
030           if !defined $token;
031         last if $token->is_close_token;
032         push @res, $token;
033     }
034
035     return new PScm::Expr::List(@res);
036 }
```

Then, at Line 23 if the token is anything other than an open brace (determined by the call to `is_open_token()`[2]), `Read()` just returns it. Otherwise, the token just read is an open brace, so `Read()` initialises 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 (`is_close_token()`) 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) baz)`. In the following, 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 the result: `("bar" 10)`.
- `Read`$_1$ adds the `("bar" 10)` to the end of its own growing list: `(foo ("bar" 10)`.

[2] It could have just said

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

but I always think it's a bit rude to peep into the implementation like that, much better to ask it what it thinks it is, not forcibly extract its data type.

- `Read`$_1$ calls `Read`$_7$.
  - `Read`$_7$ calls `_next_token()` and gets a `baz` so it returns it.
- `Read`$_1$ adds the `baz` to the end of its own growing list: `(foo ("bar" 10) baz`.
- `Read`$_1$ calls `Read`$_8$.
  - `Read`$_8$ 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 the result: `(foo ("bar" 10) baz)`.

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 3.11.4 on page 39 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**. **PScm::Token** defines default implementations for `is_open_token()` and `is_close_token()`, which the two derived classes override appropriately. **PScm::Token** is just:

```
package PScm::Token;

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

sub is_open_token { 0 }
sub is_close_token { 0 }
```

PScm::Token::Open overrides `is_open_token()`:

```
package PScm::Token::Open;

use base qw(PScm::Token);

sub is_open_token { 1 }
```

and **PScm::Token::Close** overrides `is_close_token()`:

```
package PScm::Token::Close;

use base qw(PScm::Token);

sub is_close_token { 1 }

1;
```

PScm::Token inherits a stub `new()` method from the **PScm** class that just blesses an empty hash with the argument class.

——— • ———

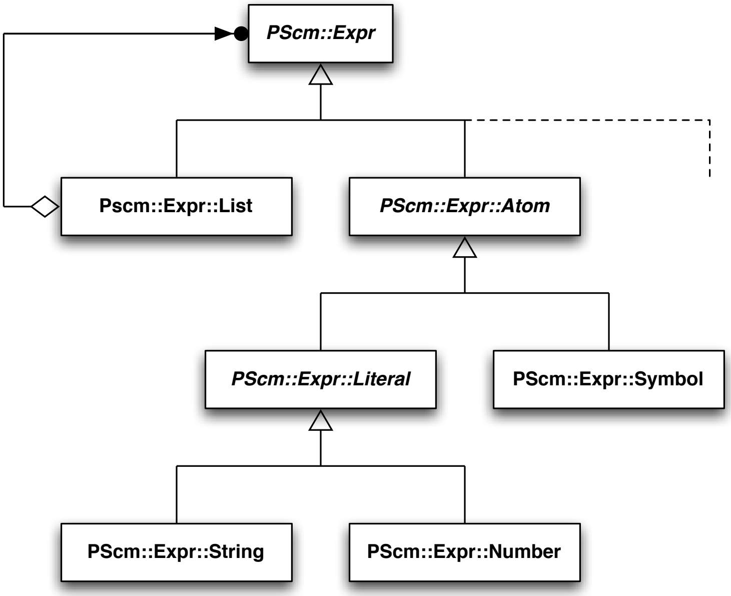
As for the **PScm::Expr** objects that `Read()` accumulates and returns, as noted `Read()` has done all of the work in constructing a tree of them for us, so they are more properly discussed in the next section where we look at expressions.

3.4 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. 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.

Firstly the classes arrange themselves in a Composite Pattern [8, pp163–173] according to the hierarchy of PScheme types as in Figure 3.2.

Figure 3.2: **PScm::Expr** classes



This figure is drawn using a standard set of conventions for diagramming the relationships between classes in an OO design, called "the Unified Modelling Language", or UML. [5]

For those who don't know UML, the triangular shape means "inherits from" or "is a subclass of", and the black arrow and circle coming from the white diamond means "aggregates zero 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 unterminated dotted line simply implies that we will be deriving other classes from **PScm::Expr** later on.

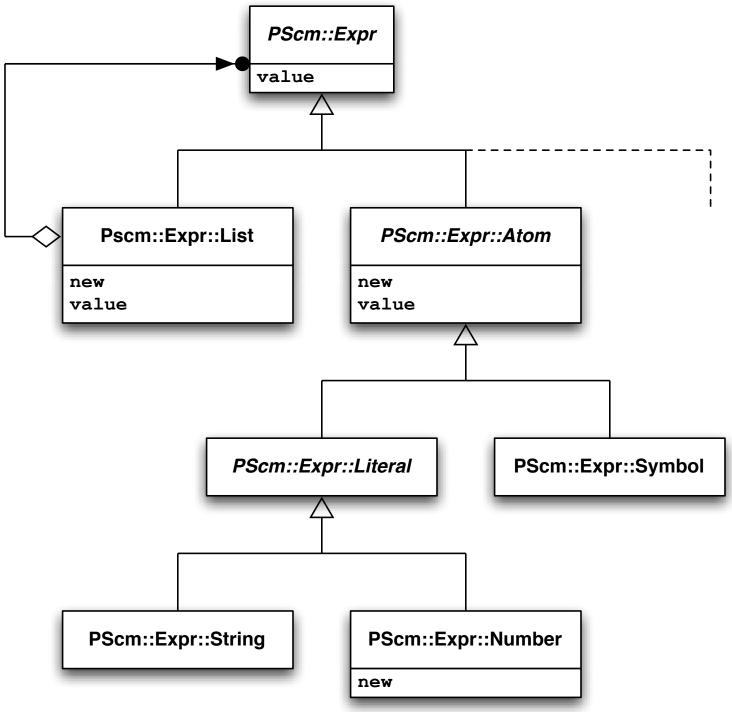
The root of the hierarchy is **PScm::Expr**, representing any and all expressions. That divides into lists (**PScm::Expr::List**) and atoms (**PScm::Expr::Atom**).

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, in Figure 3.3 with the location of the `new()` and `value()` methods added.

Figure 3.3: **PScm::Expr** `new()` and `value()` methods



As you can see, there are three `new()` methods in the 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 most of them: it blesses a reference to the scalar into the appropriate class.

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

However the **PScm::Expr::Number** package supplies its own `new()` method, because we avail ourselves of the core **Math::BigInt** package for our integers. While it is nice to have arbitrary sized integers by default, the main reason for doing this is to avoid the embarrassment of Perl's automatic type conversion to floating point on integer overflow when implementing a language that is only supposed to support integer arithmetic.

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

use Math::BigInt;

sub new {
    my ($class, $value) = @_;
    $value = new Math::BigInt($value) unless ref($value);
    $class->SUPER::new($value);
}
```

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

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

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

All three of these `new()` methods have already been seen in action in the Reader.

Alongside most of the `new()` methods is a `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:

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

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

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

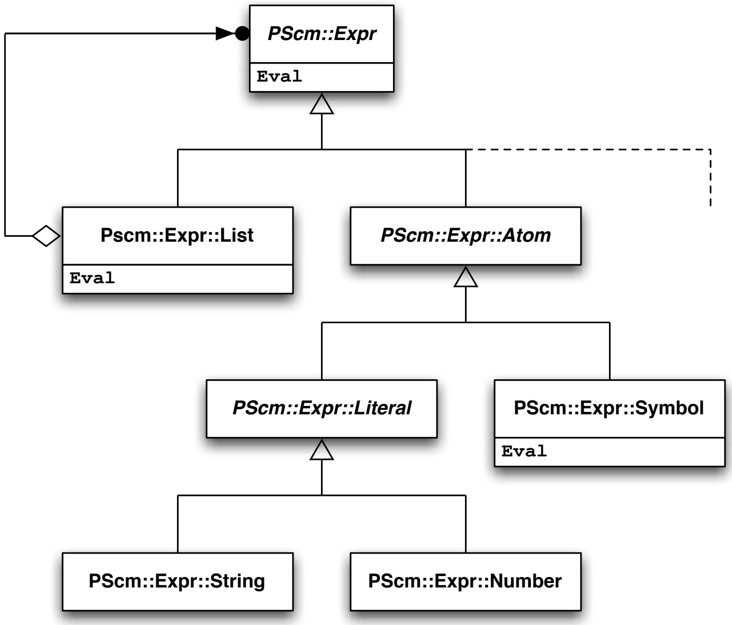
Even though **PScm::Expr::Number** has its own `new()` method, we don't need a separate `value()` method for numbers, we never need to retrieve the actual perl number from the **Math::BigInt** object so we just inherit `value()` from **PScm::Expr::Atom**. We do however provide a default `value()` method in **PScm::Expr**. This default method just returns `$self`.

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

This is solely for the benefit of those as-yet undescribed additional **PScm::Expr** subclasses, which will all stand for their own values.

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.

Figure 3.4: **PScm::Expr** `Eval()` Methods



3.5 Evaluation

To evaluate a **PScm::Expr**, as mentioned earlier, the top level `ReadEvalPrint()` loop just calls the expression's `Eval()` method. The `Eval()` methods of **PScm::Expr** are located in three of its subclasses as shown in Figure 3.4.

The figure shows that there is a separate `Eval()` method for lists and for symbols, and a default method for all other **PScm::Expr**.

3.5.1 Evaluation of Literals

Let's look first at the default `Eval()` method in **PScm::Expr** which currently applies to literals. **PScm::Expr::String** and **PScm::Expr::Number** share this default `Eval()` method, which just returns `$self`:

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

This means that numbers and strings evaluate to themselves, as they should, and if we were to add other types of expression later on, they too would by default evaluate to themselves.

3.5.2 Evaluation of Symbols

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);
}
```

Remember that `LookUp()` from **PScm::Env** 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 hash representing the environment.

3.5.3 Evaluation of Lists

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()`:

```
062 sub Eval {
063     my ($self) = @_;
064     my $op = $self->first()->Eval();
065     return $op->Apply($self->rest);
066 }
```

It's surprisingly simple. a **PScm::Expr::List** just evaluates its first element (Line 64). 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 **PScm::Expr::List**'s `Eval()` does not know or care whether the operation `$op` it derived on Line 64 is a simple primitive or a special form, on Line 65 it passes the rest of itself (the list of arguments) unevaluated to that operations `Apply()` method which applies itself to those arguments. Each individual operation's `Apply()` method will decide whether or not to evaluate its arguments, and what to do with them afterwards[3].

[3] 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.

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 one of those `Apply()` methods, the **PScm::Primitive** `Apply()` method.

3.6 Primitive Operations

The primitive built-in functions all live in `PScm/Primitive.pm`, shown in Listing 3.11.5 on page 40. This class holds all of the code for simple functions that can be passed already evaluated arguments. You can see that it in fact inherits from **PScm::Expr** rather than directly from **PScm**, which explains the dotted line in the various **PScm::Expr** figures.

This base **PScm::Primitive** class provides the `Apply()` method for all simple functions:

```
007 sub Apply {
008     my ($self, $form) = @_;
009
010     my @unevaluated_args = $form->value;
011     my @evaluated_args = map { $_->Eval() } @unevaluated_args;
012     return $self->_apply(@evaluated_args);
013 }
```

On Line 10 it 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. Then, on Line 11, `Apply()` evaluates each argument by mapping a call to each one's `Eval()` method. Finally, on Line 12, it passes the resulting list of evaluated arguments to a private `_apply()` method and returns the result.

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

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 = PScm::Expr::Number->new(1)->value();

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

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

On Line 31 the rather convoluted trick to get an initial value will work whether or not the underlying implementation of **PScm::Expr::Number** uses **Math::BigInt** or not.

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

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

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

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.

```perl
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);
}
```

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 all the really interesting stuff is happening over in the special forms, discussed next.

3.7 Special Forms

All the code for special forms is in `PScm/SpecialForm.pm` in Listing 3.11.6 on page 42. Like **PScm::Primitive** it descends from **PScm::Expr**.

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 15 **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 17 it evaluates the condition, and calls the result's `isTrue()` method to determine which branch to evaluate:

```
012 sub Apply {
013     my ($self, $form) = @_;
014
015     my ($condition, $true_branch, $false_branch) = $form->value;
016
017     if ($condition->Eval()->isTrue) {
018         return $true_branch->Eval();
019     } else {
020         return $false_branch->Eval();
021     }
022 }
```

If the condition is true, `PScm::SpecialForm::If::Apply()` evaluates and returns the true branch (Line 18), otherwise it evaluates and returns the false branch (Line 20). 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:

```
007 sub isTrue {
008     my ($self) = @_;
009     scalar($self->value);
010 }
```

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[4].

That really is all there is to evaluation. Next we're going to take a look at the print system.

3.8 Output

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`

```
048 sub Print {
049     my ($self, $outfh) = @_;
050     print $outfh $self->as_string, "\n";
051 }
```

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 53.

[4] This differs from a true Scheme implementation where special boolean values `#t` and `#f` represent truth and falsehood, and *everything* else is true. The reason for having an `isTrue()` is to encapsulate the chosen behaviour. If we wanted to change the meaning of truth, we need only do so here.

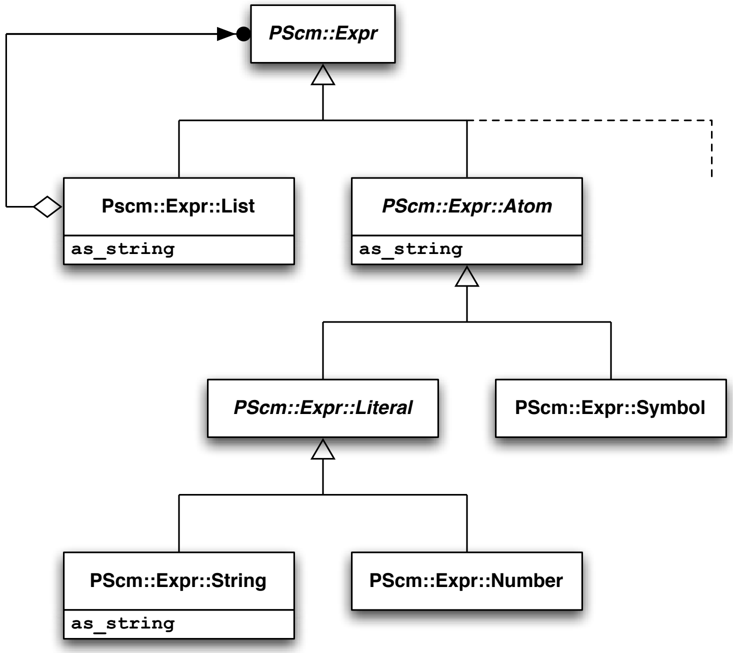
```
sub as_string { ref($_[0]); }
```

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 3.5.

Figure 3.5: **PScm::Expr** `as_string()` methods



The `as_string()` method in **PScm::Expr::Atom** is just a call to `value()`:

```
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[5].

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

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

```
058       . join(’ ’, map { $_->as_string } $self->value)
059       . ’)’;
060 }
```

Finally, **PScm::Expr::String**’s `as_string()` method at Lines 97-104 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.

```
097 sub as_string {
098     my ($self) = @_;
099 
100     my $copy = $self->value;
101     $copy =~ s/\\/\\\\/sg;
102     $copy =~ s/"/\\"/sg;
103     return qq’"$copy"’;
104 }
```

3.9 Summary

We’re finally in a position to understand the whole of **PScm::Expr** as shown in Listing 3.11.7 on page 43. The final version of our diagram, with all of the methods from **PScm::Expr** in place is shown in Figure 3.6 on the following page.

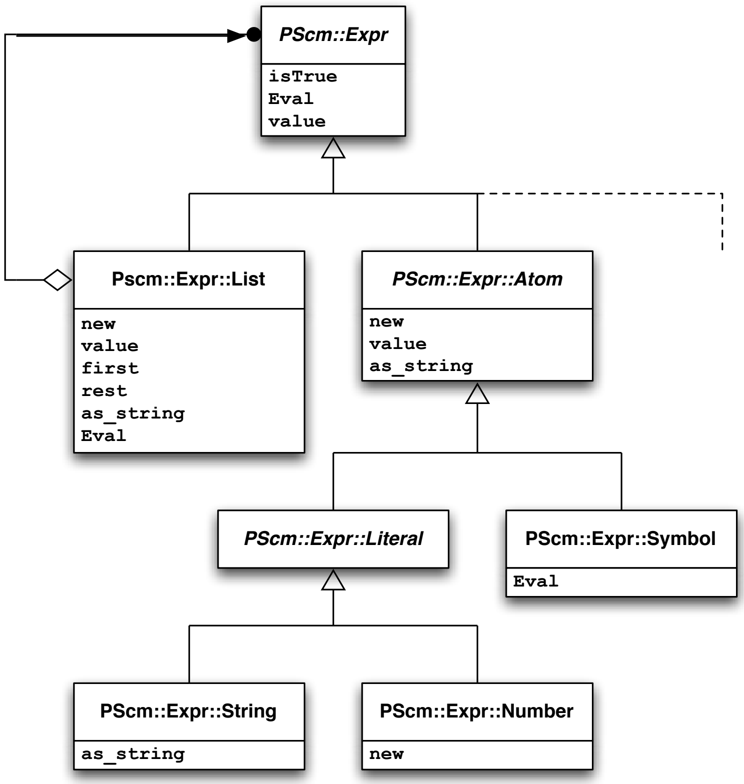
—— • ——

That may seem like a lot of code for what is effectively just a pocket calculator[6], but what has been done is to lay the groundwork for a much more powerful set of language constructs that will be added in subsequent chapters. 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 repeatedly calls its `Read()` method.
- 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 global environment.
  - **PScm::Expr::List** objects have an `Eval()` method that evaluates the first component of the list, which should return a primitive operation or special form, 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.

[6] especially one where the `+` and `/` keys don’t work.

Figure 3.6: **PScm::Expr** methods



- **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.

3.10 Tests

The test module for our first version of the interpreter is in Listing 3.11.8 on page 46. The **PScm::Test** package shown in Listing 3.11.9 on page 47 provides an `eval_ok()` sub which 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 output back in and compares it to its second argument[7]. 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, in Listing 3.11.10 on page 48.

[7] Ok, I should have used **IO::String**, so sue me.

3.11 Listings

3.11.1 PScm.pm

```
package PScm;

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

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

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

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

sub new { bless {}, $_[0] }

1;
```

3.11.2 PScm/Env.pm

```
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;
```

3.11.3 PScm/Read.pm

```
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->is_open_token;

    my @res = ();

    while (1) {
        $token = $self->Read;
        die "unexpected EOF"
          if !defined $token;
        last if $token->is_close_token;
        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/^\s+//s;
    }

    for ($self->{Line}) {
        s/^\(\s*// && return PScm::Token::Open->new();
        s/^\)\s*// && return PScm::Token::Close->new();
        s/^([-+]?\d+)\s*//
          && return PScm::Expr::Number->new($1);
        s/^"((?:(?:\\.)|([^"]))*)"\s*// && do {
            my $string = $1;
            $string =~ s/\\//g;
            return PScm::Expr::String->new($string);
        };
        s/^([^\s\(\)]+)\s*//
          && return PScm::Expr::Symbol->new($1);
    }
    die "can't parse: $self->{Line}";
}

1;
```

3.11.4 PScm/Token.pm

```
package PScm::Token;

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

sub is_open_token { 0 }
sub is_close_token { 0 }

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

use base qw(PScm::Token);

sub is_open_token { 1 }

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

use base qw(PScm::Token);

sub is_close_token { 1 }

1;
```

3.11.5 PScm/Primitive.pm

```
package PScm::Primitive;

use strict;
use warnings;
use base qw(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 = PScm::Expr::Number->new(1)->value();

    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;

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;
```

3.11.6 PScm/SpecialForm.pm

```
package PScm::SpecialForm;

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

##############################
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;
```

3.11.7 PScm/Expr.pm

```
package PScm::Expr;

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

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

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

sub value { $_[0] }

#########################
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);
}

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);

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

use Math::BigInt;

sub new {
    my ($class, $value) = @_;
    $value = new Math::BigInt($value) unless ref($value);
    $class->SUPER::new($value);
}

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

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

    my $copy = $self->value;
    $copy =˜ s/\\/\\\\/sg;
    $copy =˜ s/"/\\"/sg;
    return qq’"$copy"’;
```

```
}

1;
```

3.11.8 t/PScm.t

```
use strict;
use warnings;
use Test::More;
use lib ’./t/lib’;
use PScm::Test tests => 10;

BEGIN { use_ok(’PScm’) }

eval_ok(’1’,                  ’1’,       ’numbers’);
eval_ok(’+1’,                 ’1’,       ’explicit positive numbers’);
eval_ok(’-1’,                 ’-1’,      ’negative numbers’);
eval_ok(’"hello"’,            ’"hello"’, ’strings’);
eval_ok(’(* 2 3 4)’,          ’24’,      ’multiplication’);
eval_ok(’(- 10 2 3)’,         ’5’,       ’subtraction’);
eval_ok(’(- 10)’,             ’-10’,     ’negation’);
eval_ok(’(if (* 0 1) 10 20)’, ’20’,      ’simple conditional’);
eval_ok(<<EOT,                <<EOR,     ’no overflow’);
(* 1234567890987654321 1234567890987654321)
EOT
1524157877457704723228166437789971041
EOR

# vim: ft=perl
```

3.11.9 t/lib/PScm/Test.pm

```
package PScm::Test;
use strict;
use warnings;
use FileHandle;
require Exporter;

our @ISA = qw(Exporter);
our @EXPORT = qw(eval_ok evaluate);

my $Test = Test::Builder->new;

sub import {
    my ($self) = shift;
    my $pack = caller;
    $Test->exported_to($pack);
    $Test->plan(@_);

    $self->export_to_level(1, $self, ’eval_ok’);
    $self->export_to_level(1, $self, ’evaluate’);
}

sub eval_ok {
    my ($expr, $expected, $name) = @_;
    my $result = evaluate($expr);
    $result .= "\n" if $expected =~ /\n/;
    $Test->is_eq($result, $expected, $name);
}

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

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

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

1;
```

3.11.10 t/interactive

```
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
```

Full source code for this version of the interpreter is available at
`http://billhails.net/Book/releases/PScm-0.0.0.tgz`

Chapter 4

Implementing `let`

`let` allows the extension of the environment, temporarily, to include new bindings of symbols to data. `let` was introduced in Section 2.6 on page 12 but as a quick reminder, here it is in action:

```
> (let ((x 10)
>       (y 20))
>      (* x y))
200
> x
Error: no binding for x in PScm::Env
```

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.

4.1 The Environment

Remember the original environment implementation from Section 3.2 on page 16, 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 environment extension actually means. It would be a good idea to keep the environment 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 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".

4.1.1 A Stack-based Environment

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.

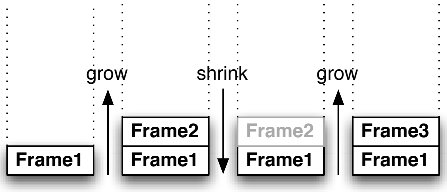
Perl lists have push and pop operations, so we could easily use those with a simple array representing the stack. Alternatively we could keep a current "top of stack" index, and increment that to push, or decrement it to pop, something like:

```
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 4.1. 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 4.1: Stacks Destroy Old Environment Frames

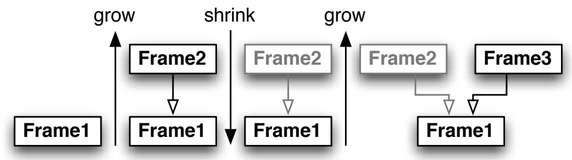


4.1.2 A Linked List Environment

Enter the linked list. A linked list is just a collection of objects, hashes or whatever, where each one contains a reference to the previous 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 advantages of a linked list are that many environment frames can share the same parent, and therefore creating a new extension frame does not destroy the previous extension, see Figure 4.2 on the next page. 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.

Figure 4.2: Linked Lists Don't Destroy Old Environment Frames



```
025 sub LookUp {
026     my ($self, $symbol) = @_;
027
028     if (exists($self->{bindings}{ $symbol->value })) {
029         return $self->{bindings}{ $symbol->value };
030     } elsif ($self->{parent}) {
031         return $self->{parent}->LookUp($symbol);
032     } else {
033         die "no binding for @{[$symbol->value]} ",
034           "in @{[ref($self)]}\n";
035     }
036 }
```

The only change is on Lines 30–31 (in bold) 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 (which it will be), and adds a parent field to the object, with an initial zero value meaning "no parent".

```
007 sub new {
008     my ($class, %bindings) = @_;
009
010     $class = ref($class) || $class;
011     bless { bindings => {%bindings}, parent => 0 }, $class;
012 }
```

Finally we need an `Extend()` method of **PScm::Env** 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.

```
014 sub Extend {
015     my ($self, $ra_symbols, $ra_values) = @_;
016
017     my %bindings = ();
018     my @names    = map { $_->value } @$ra_symbols;
019     @bindings{@names} = map { $_->Eval($self) } @$ra_values;
020     my $new = $self->new(%bindings);
021     $new->{parent} = $self;
022     return $new;
023 }
```

Because the `Extend()` method will be used by `let` and other constructs later, 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. On Line 18 It maps the symbols to a list of strings, then on Line 19 it uses those strings as keys in a hash mapping them to their equivalent values. On Line 20, creates a new environment with that hash. Finally on Line 21 it sets that new environment's parent to be the original environment `$self` and returns the new environment.

4.2 Global Environments have a Problem

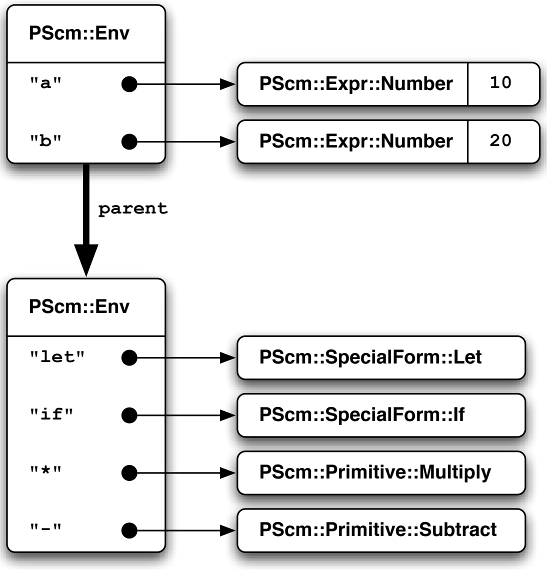
Before proceeding to implement `let`, we need to address an incipient problem with the global environment.

To understand this, 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 4.3.

Figure 4.3: Environment during evaluation of example "`let`"



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

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?

4.3 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![1]

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 ||= new FileHandle(">-");
    my $reader = new PScm::Read($infh);
```

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

```
    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);
    }
}
```

Now you should remember from Section 3.5 on page 25 that there are three implementations of `Eval()`, one in **PScm::Expr**, one in **PScm::Expr::Symbol** and one in **PScm::Expr::List**. Each of these must deal with the extra environment argument they are now being passed.

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

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

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);
}
```

So `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 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()`.

4.4 `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.

```
008 package PScm::SpecialForm::Let;
009
010 use base qw(PScm::SpecialForm);
011
012 sub Apply {
013     my ($self, $form, $env) = @_;
014
015     my ($bindings, $body) = $form->value;
016     my (@symbols, @values);
017
018     foreach my $binding ($bindings->value) {
019         my ($symbol, $value) = $binding->value;
020         push @symbols, $symbol;
021         push @values,  $value;
022     }
023
024     return $body->Eval($env->Extend(\@symbols, \@values));
025 }
```

It starts off at Line 15 extracting the bindings and body from the argument form. Then it sets up two empty lists to collect the symbols 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 evaluated argument in the other. 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.

4.5 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.

4.6 Tests

Rather than adding more tests to `t/PScm.t`, There's a new `t/PScm_Let.t` which you can see in Listing 4.7.1 on the facing page. It adds two tests, the first just tests that a value bound by a `let` expression is available in the body of the `let`, and the second proves that the body of the `let` can be an arbitrary expression.

4.7 Listings

4.7.1 t/PScm_Let.t

```
use strict;
use warnings;
use Test::More;
use lib ’./t/lib’;
use PScm::Test tests => 3;

BEGIN { use_ok(’PScm’) }

eval_ok(’(let ((x 2)) x)’, ’2’, ’simple let’);

eval_ok(<<EOF, ’20’, ’conditional evaluation’);
(let ((a 00)
      (b 10)
      (c 20))
     (if a b c))
EOF

# vim: ft=perl
```

Full source code for this version of the interpreter is available at
`http://billhails.net/Book/releases/PScm-0.0.1.tgz`

Chapter 5

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 and Perl 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[1] 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)
```

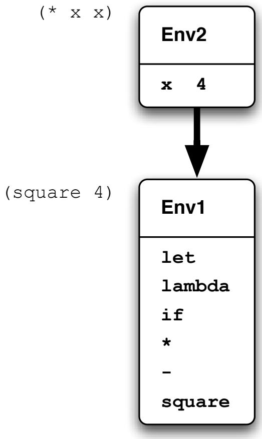
the global environment would be extended with a binding of `x` to `4`, and the body of the function, `(* x x)` would be evaluated in that new environment, as in Figure 5.1 on the next page.
Now, a *closure* is simply a function that when executed will extend the environment that was current at the time the closure was created. Consider an example we've seen before.

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

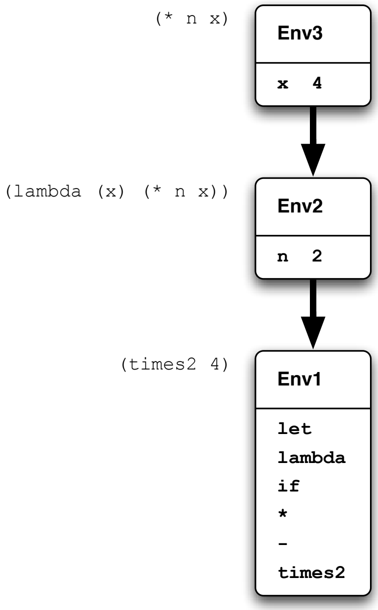
[1] Formal arguments are the names that a function gives to its arguments. Actual arguments are the values passed to a function.

Figure 5.1: Functions extend an environment just like `let` does



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 5.2.

Figure 5.2: Closures extend the lexical environment



So referring to that figure: `let` extended the global environment to **Env2** with a binding of `n` to `2`. Then the closure, when it was created by `lambda` in **Env2**, must have somehow held on to, or "captured" **Env2**, so that when the closure is later executed **Env2** is the one that it extends to **Env3** with its own argument `x`.

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 horrendous confusion: any variables referred to in the function body that were not declared by the function might pick up values randomly from the callers environment. Alternatively 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 the actual implementation of closures (functions) there are two parts to the story. The first part is how `lambda` creates a closure, and the second is how the closure gets evaluated when it is called. In the next section we'll look at the first part, how `lambda` creates a closure.

5.1 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 not much different from our earlier `times2` example, except that an outer `let` provides bindings for both `times2` and a variable `a` that will be argument to `times2`. It is however just a little tricky, so in detail:

- The outer `let` reads: "let `a` be `4` and `times2` be the result of evaluating the inner `let` in the expression `(times2 a)`."
- The inner `let` reads "let `n` be `2` in the expression `(lambda ...)`."
- The value of that inner `let` is the result of evaluating that `lambda` expression and thus a closure, and that 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 variable `n` from the environment that was current when it was created, and so the body of the closure, `(* x n)`, wilth `x` bound to `4` and `n` bound to `2`, 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 5.3 on the next page.

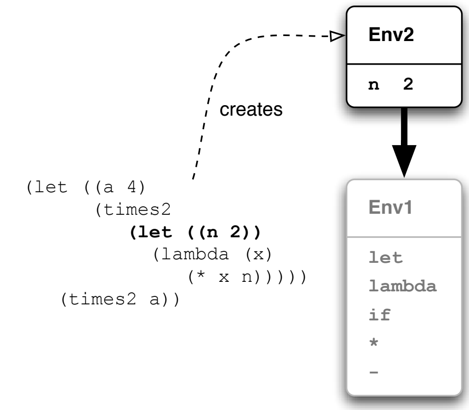
```
> (let ((a 4)
>       (times2
>         (let ((n 2))
>           (lambda (x)
>              (* x n)))))
>   (times2 a))
8
```

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, its (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:

Figure 5.3: `let` binds `n` to `2`



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

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:

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

    $outfh ||= new FileHandle(">-");
    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(),
            )
        );
        $result->Print($outfh);
    }
}
```

The only change is the addition of Line 43 with the new binding for lambda.

Now we can look at that new package **PScm::SpecialForm::Lambda**. All its 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:

```
045 package PScm::SpecialForm::Lambda;
046
047 use base qw(PScm::SpecialForm);
048 use PScm::Closure;
049
050 sub Apply {
051     my ($self, $form, $env) = @_;
052
053     my ($args, $body) = $form->value;
054     return PScm::Closure::Function->new($args, $body, $env);
055 }
056
057 1;
```

On Line 53 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 54 it returns a new **PScm::Closure::Function** object containing those values and, most importantly, also containing the current environment (**Env2** in our example.)

That PScm::Closure::Function::new() method (actually in **PScm::Closure**) does no more than bundle its arguments:

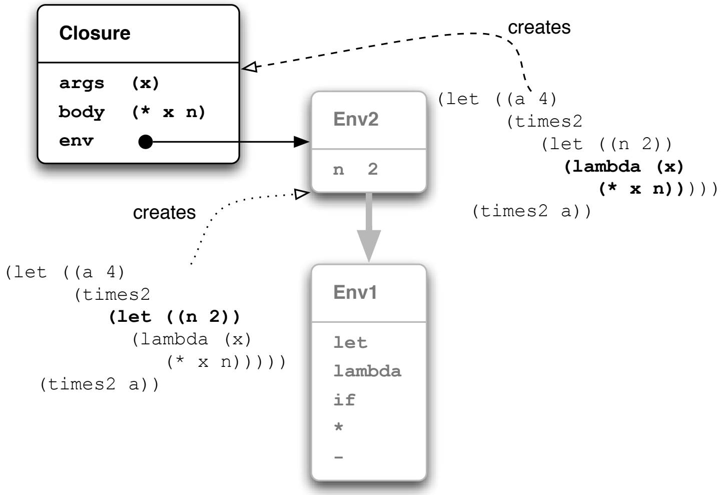
```
sub new {
    my ($class, $args, $body, $env) = @_;

    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 5.4 on the facing page.

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:

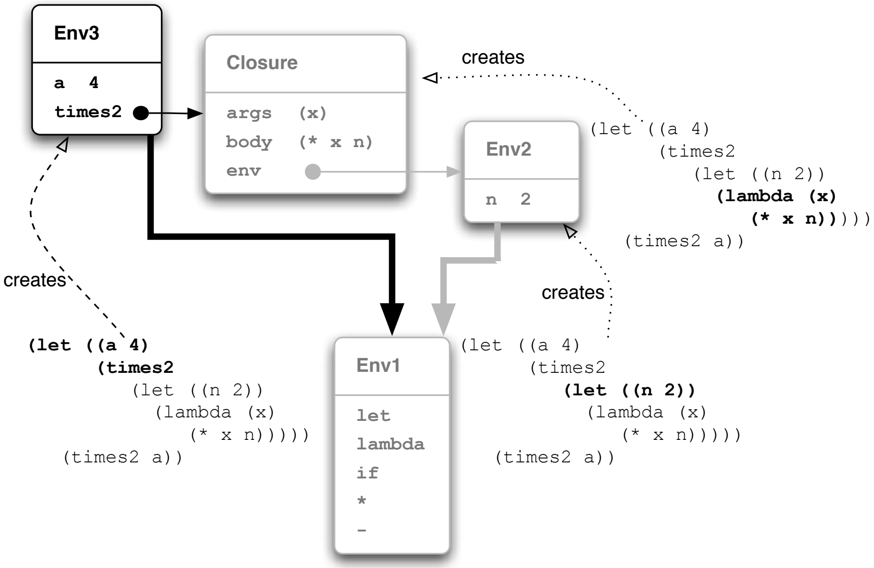
```
> (let ((a 4)
>       (times2
>         (let ((n 2))
>           (lambda (x)
>              (* x n)))))
>   (times2 a))
8
```

Figure 5.4: Closure Captures the Local Environment



The resulting environment is labelled **Env3** in Figure 5.5.

Figure 5.5: `let` binds `times2` and `a`



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.

5.2 Evaluating a 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 5.5 on the preceding page which the outer `let` has just set up with a binding of `a` to 4 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:

```
043 sub Apply {
044     my ($self, $form, $env) = @_;
045
046     my @evaluated_args = map { $_->Eval($env) } $form->value;
047     return $self->_apply(@evaluated_args);
048 }
```

First of all, on Line 46 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 47 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[2]:

```
021 sub _apply {
022     my ($self, @args) = @_;
023
024     my $extended_env =
025       $self->env->ExtendUnevaluated([$self->args], [@args]);
026     return $self->body->Eval($extended_env);
027 }
```

This `_apply()` method does not need an argument environment because the correct environment to extend is the one that was captured when the the closure object was created. On Line 24 It extends that

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

previously captured environment with bindings from its formal arguments, also collected when the closure object was created (i.e. `x`), to the actual arguments it was passed (i.e. `4`, already evaluated). Because the arguments are already evaluated, it must call a new variant of `PScm::Env::Extend()` called `ExtendUnevaluated()`, which does just that. Lastly `_apply()` evaluates its body (the body of the function, `(* x n)`) passing that extended environment as argument and returns the result (Line 26).

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 5.6). 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 5.6: Closure Extends Captured Env

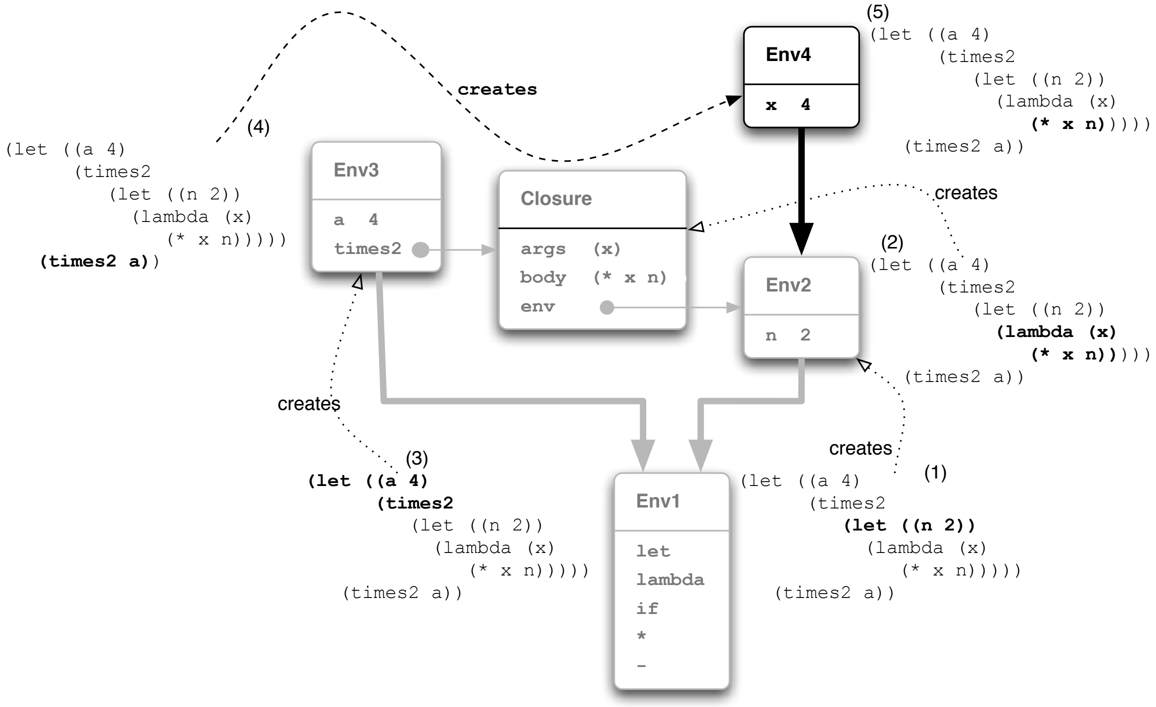


Figure 5.6 pretty much tells the whole story. Here's our example one last time so it can be walked through referring to the figure:

```
(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.

5.3 Printing a Closure

It would be nice if, when given a closure to print, PScheme could produce something a bit more informative than the unhelpful "`PScm::Closure::Function`" that results from the default `as_string()` method in the top-level **PScm** package. We could instead print a representation of the `lambda` expression that created the closure. For example.

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

This is trivially accomplished by overriding the default `as_string()` method in **PScm::Closure**. Here's that override.

```
sub as_string {
    my ($self) = @_;
    return PScm::Expr::List->new(
        $self->_symbol,
        $self->{args},
        $self->{body}
    )->as_string;
}
```

All it does is to construct a new **PScm::Expr::List** containing the symbol that constructed the closure (`lambda`) the formal arguments to the closure and the body of the closure. It then calls that list's `as_string()` method and returns the result. The aquisition of the `lambda` symbol is deferred to a separate method `_symbol()` in **PScm::Closure::Function** (again because later later versions of the interpreter will have different kinds of closures). Here's `_symbol()`.

```
sub _symbol {
    PScm::Expr::Symbol->new('lambda');
}
```

Job done. Closures, when printed, will now produce a useful representation of the function they perform.

5.4 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 5.6.1 on the next page.

5.5 Tests

You can see the tests for the `lambda` form in a new file, `t/PScm_Lambda.t`, in Listing 5.6.2 on page 72. The first test exercizes the simple creation of a `lambda` expression, its binding to a symbol, and its application to arguments. The second works through pretty much exactly the example we've been working through. The third 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. Incidentally, this demonstrates that the environment created by a `lambda` expression is equally ameanable to capture by a closure.

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

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

5.6 Listings

5.6.1 PScm/Closure.pm

```
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);
}

sub as_string {
    my ($self) = @_;
    return PScm::Expr::List->new(
        $self->_symbol,
        $self->{args},
        $self->{body}
    )->as_string;
}

################################
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);
}

```

```
sub _symbol {
    PScm::Expr::Symbol->new(’lambda’);
}

1;
```

5.6.2 t/PScm_Lambda.t

```
use strict;
use warnings;
use Test::More;
use lib './t/lib';
use PScm::Test tests => 5;
use FileHandle;

BEGIN { use_ok('PScm') }

eval_ok(<<EOF, '16', 'lambda');
(let ((square
       (lambda (x) (* x x))))
     (square 4))
EOF

eval_ok(<<EOF, '(lambda (x) (* x x))', 'lambda to string');
(let ((square
       (lambda (x) (* x x))))
     square)
EOF

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

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

# vim: ft=perl
```

Full source code for this version of the interpreter is available at
http://billhails.net/Book/releases/PScm-0.0.2.tgz

Chapter 6

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[1].

```
> (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 the expression half of its bindings in the enclosing environment, *before* it binds the values to the symbols in a new 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`, `factorial` is only visible within the body of the `let`, so any recursive call to `factorial` from the body of the function (closure) is bound to fail.

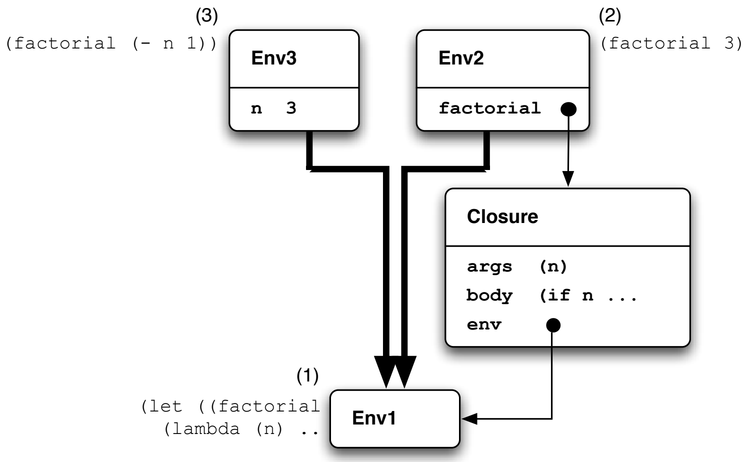
Putting it another way, `let` will create a binding for `factorial`, but only by extending the global environment *after* the `lambda` expression has been evaluated, in the global environment, therefore capturing the global environment.

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 6.1 on the next page should help to make that clear.

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 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.

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

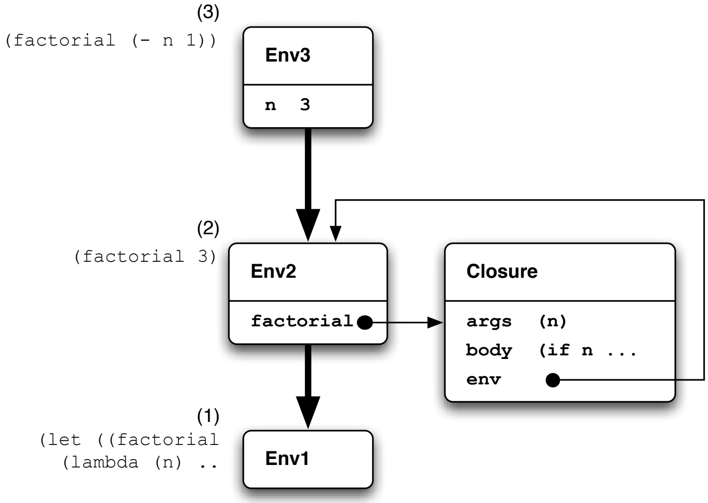
Figure 6.1: Why recursion doesn't work



6.1 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 6.2.

Figure 6.2: 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 value component of each binding 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[2].

6.1.1 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:

[2] 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.

```
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;
    }
}
```

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";
    }
}
```

6.1.2 PSCM::LetRec itself

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, $env);

    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, $env) = @_;

    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, $env);

    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 ||= new FileHandle(">-");
    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);
    }
}
```

6.2 Summary

Let evaluates the values of its bindings in the enclosing environment. Then it creates an extended environment with each symbol bound to its value, 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 not yet a binding for the recursive function when the `lambda` expression is evaluated to create the closure. 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.

6.3 Tests

The tests for the letrec form are in t/PScm_Letrec.t which you can see in Listing 6.4.1 on the next page.

There are three tests. The first two, just prove what we already know, that let does not (and should not) support recursion. The other new test replaces let with letrec and proves that letrec on the other hand does support recursive function definitions.

6.4 Listings

6.4.1 t/PScm_Letrec.t

```
use strict;
use warnings;
use Test::More;
use lib ’./t/lib’;
use PScm::Test tests => 4;

BEGIN { use_ok(’PScm’) }

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

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

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

# vim: ft=perl
```

Full source code for this version of the interpreter is available at
`http://billhails.net/Book/releases/PScm-0.0.3.tgz`

Chapter 7

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.

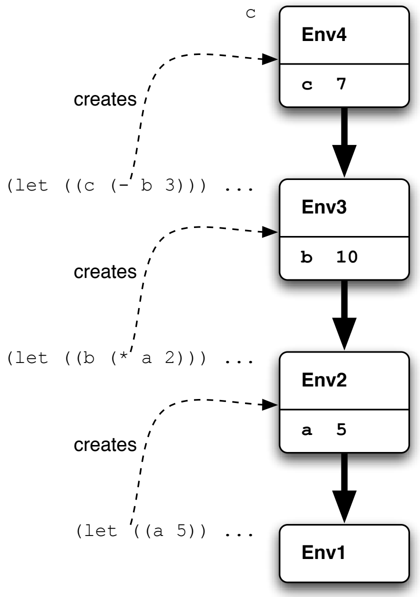
7.1 Sequential Binding

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 7.1 on the following page.
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.

Figure 7.1: Nested environments



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.

7.2 `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.

```
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[1].

7.3 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 like `let` is done in a list context:

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

Sequential binding like `let*` 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:

[1] 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.

```
(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 ||= new FileHandle(">-");
    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(),
                'let*' => new PScm::SpecialForm::LetStar(),
            )
        );
        $result->Print($outfh);
    }
}
```

7.4 Tests

The additional tests for 0.0.4 are in `t/PScm_LetStar.t` which you can see in Listing 7.5.1 on the next page.

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

7.5 Listings

7.5.1 **t/PScm_LetStar.t**

```
use strict;
use warnings;
use Test::More;
use lib ’t/lib’;
use PScm::Test tests => 3;

BEGIN { use_ok(’PScm’) }

eval_ok(<<EOF, ’1’, ’let binds in parallel’);
(let ((a 1)
      (b 2))
     (let ((a b)
           (b a))
          b))
EOF

eval_ok(<<EOF, ’2’, ’let* binds sequentially’);
(let ((a 1)
      (b 2))
  (let* ((a b)
         (b a))
    b))
EOF

# vim: ft=perl
```

Full source code for this version of the interpreter is available at
`http://billhails.net/Book/releases/PScm-0.0.4.tgz`

Chapter 8

List Processing

It was mentioned in Section 1.2 on page 4 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 categorise 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.

8.1 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;
```

8.2 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 its 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**.

8.3 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`[1].

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) = @_;

```

[1] 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.

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

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;
}
```

8.4 cons

We're only missing one piece from our set of basic list operations now, but before adding that it is necessary to explain and rectify a significant deviation that PScheme has so far made from other Lisp and Scheme implementations. In our PScheme implementation lists have been implemented as object wrappers around Perl lists. This had the advantage that the Perl implementation was as simple as it could be. However real Lisp systems implement lists as chains of what are called *cons cells*, or more commonly *pairs*. 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) baz)` Is *not* properly implemented as in Figure 3.1 on page 17, but as shown in Figure 8.1 on the following page.

The unfilled `cdr` pointers in the figure represent null pointers and terminate the list structure.

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, it just returns it's `rest` component. 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. A cons cell is constructed with the primitive operator `cons`.

8.4.1 Dot Notation

Since we can now build structures that cannot be represented in simple "`(a b c)`" list notation, we need to extend that notation to cope. The extension is thankfully very simple: if the last component of a list is prepended by a period (separated by a space) for example `(a . b)` it is taken to be the `cdr` of the list. This is called *dot notation*. See Figure 8.2 on the next page.

Dot notation will be supported for both input and output.

Figure 8.1: Cons Cell Representation of a nested list `(foo ("bar" 10) baz)`

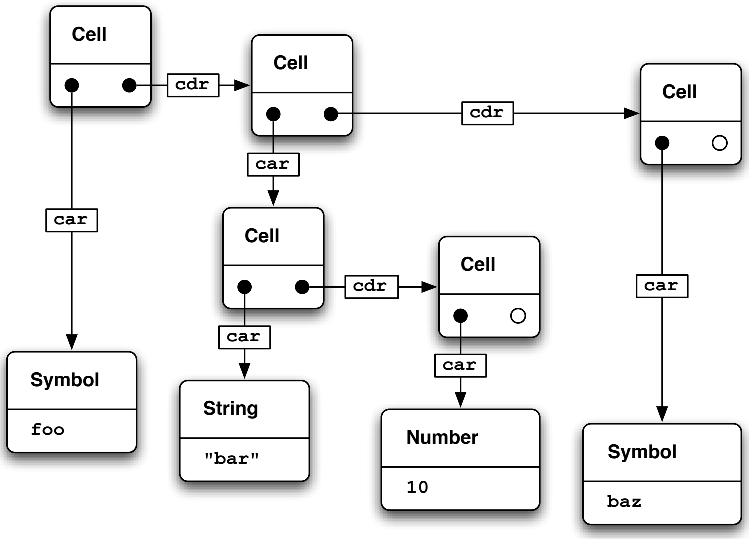
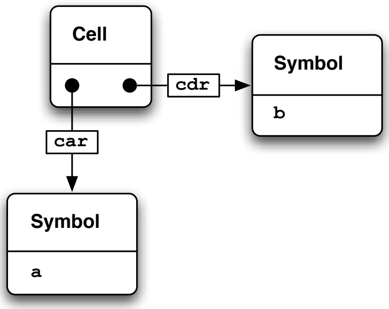


Figure 8.2: The pair `(a . b)`



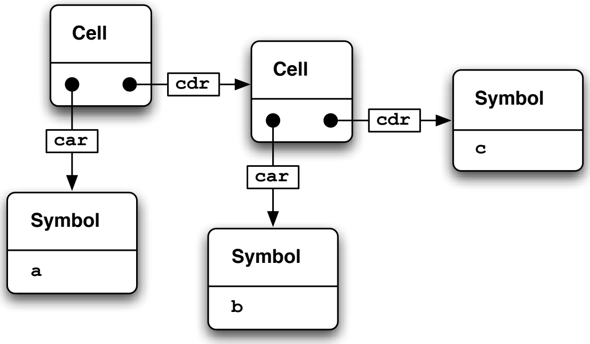
Dot notation is not limited to just dotted pairs, for example in Figure 8.3 on the facing page you can see that more complex structures can also be represented.

Dot notation is actually capable of representing any structure we can envisage[2]. In fact it is reasonable to think of the normal list notation we have been using so far as merely a convenient shorthand for dot notation. For example the list `(a)` is the same as the pair `(a . ())` (because `()` is the empty list.) Likewise the list `(a b c)` can be represented as `(a . (b . (c . ())))`. Obviously this unwieldy notation is to be avoided unless necessary, but you should at least be aware of it.

Dot notation has a number of uses. Most importantly it allows us to easily specify variable numbers of arguments to our lambda expressions: if the formal arguments in a `lambda` expression are dotted, then

[2] Well, actually it is possible to imagine circular structures that would defeat any notation. A true Scheme will even allow the creation of such circular lists, with such dubious expressions as `(set-cdr! x x)`, but that's a world of pain that we will stay well away from.

Figure 8.3: The structure (a b . c)



that can be taken to mean the dotted symbol is to be bound to a list of the remaining actual arguments. For Example:

```
> (let ((test
>         (lambda (a b . c)
>           (list a b c))))
>   (test 1 2 3 4 5))
(1 2 (3 4 5))
```

The a and b are bound to 1 and 2 as always, but the c, because it occupies the entirity of the rest of the formal argument list gets bound to the remaining list of additional arguments.

Interestingly this also allows entirely arbitrary lists of arguments. If you think about it the dotted pair notation (. a) can be made perfectly legal for input, and is equivalent to the symbol a: the opening brace implies a list, but the first symbol encountered is the cdr of a list that has not started to form yet, so the result is just that symbol. Since we must accept lambda expressions with such an argument declaration, we must also accept lambda expressions with a single symbol instead of a list of arguments. For example we could define our list primitive in PScheme like:

```
> (let ((list (lambda args args)))
>   (list 1 2 3 4 5))
(1 2 3 4 5)
```

list takes any number of arguments as a single list args and returns them.

8.5 Implementation

Let's make the change to use that alternative implementation. Since the **PScm::Expr::List** class hides its internal structure and provides accessor methods, technically that should be the only package that needs to change. However it is worthwhile making use of the new list structure in other parts of the PScheme system. The language is highly recursive, and this new linked list structure lends itself to recursion much more naturally than a plain perl @list does.

8.5.1 Changes to Expressions

To get us started, here’s the new implementation of **PScm::Expr::List**:

```
059 package PScm::Expr::List;
060 use base qw(PScm::Expr);
061
062 sub new {
063     my ($class, @list) = @_;
064
065     $class = ref($class) || $class;
066     if (@list) {
067         my $first = shift @list;
068         $class->Cons($first, $class->new(@list));
069     } else {
070         new PScm::Expr::List::Null();
071     }
072 }
073
074 sub Cons {
075     my ($class, $first, $rest) = @_;
076     return PScm::Expr::List::Pair->new($first, $rest);
077 }
078
079 sub as_string {
080     my ($self) = @_;
081     return ’(’ . join(’ ’, $self->strings) . ’)’;
082 }
```

The `new()` method on Lines 62-72 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 68 it calls an ancilliary method `Cons()` (defined on Lines 74-77) to actually construct a new **PScm::Expr::List::Pair** node (a cons cell). So the **PScm::Expr::List** class is now in fact abstract. Although it has a `new()` method, that method actually returns instances of either **PScm::Expr::List::Pair** or another new object, **PScm::Expr::List::Null**, which represents the empty list.

if you remember the old implementation of `new()` just wrapped its argument list:

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

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

The `as_string()` method of **PScm::Expr::List** has changed too. Rather than just mapping `as_string()` over the components of the list, it calls a separate `strings()` method that will return an array of strings, and joins and wraps that. The main reason for that is to cope with dotted pair notation. We’ll see how `strings()` works soon.

Much of the functionality that was in **PScm::Expr::List** has been moved out into **PScm::Expr::List::Pair**, shown next:

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

use constant {
    FIRST => 0,
    REST => 1,
};

sub new {
    my ($class, $first, $rest) = @_;
    bless [$first, $rest], $class;
}

sub value {
    my ($self) = @_;
    my @value = ($self->[FIRST], $self->[REST]->value);
    return @value;
}

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

sub rest { $_[0][REST] }

sub strings {
    my ($self) = @_;
    return ($self->[FIRST]->as_string,
            $self->[REST]->strings);
}

sub Eval {
    my ($self, $env) = @_;
    my $op = $self->[FIRST]->Eval($env);
    return $op->Apply($self->[REST], $env);
}

sub map_eval {
    my ($self, $env) = @_;
    return $self->Cons($self->[FIRST]->Eval($env),
                       $self->[REST]->map_eval($env));
}

sub is_pair { 1 }
```

As a minor optimization, **PScm::Expr::List::Pair** will store its `first` and `rest` components in an array ref rather than a hash. For that reason it declares two constants `FIRST` and `REST` to act as indexes into that structure. **PScm::Expr::List::Pair** has its own `new()` method which we've already seen being called by `Cons()`. It just collects its two arguments into a new object.

The other methods in **PScm::Expr::List::Pair** are fairly straightforward.

- The `value()` method on Lines 98-102 converts the linked list back into a Perl list[3]. Because the structure is no longer necessarily a true list we must supply an alternative `value()` method in **PScm::Expr** which just returns `$self` (ignoring the dot notation). There is another `value()` method in **PScm::Expr::List::Null** that returns the empty (Perl) list and likewise terminates the recursion of `PScm::Expr::List::value()`.

- The `first()` and `rest()` methods are simplified, they are now just accessors to their equivalent fields.

- As mentioned above, the `as_string()` method from **PScm::Expr::List** has to deal with dot notation, so cannot simply `map` an `as_string()` over the list's `value()`. Instead it calls a helper method `strings()`.

- `strings()`, on Lines 108-112, returns a perl list of the string representation of the first item on the list, plus the result of calling itself on the rest of the list. There is an implementation of `strings()` in **PScm::Expr::List::Null** that just returns the empty (Perl) list:

  ```
  sub strings { (); }
  ```

  and another at the root of the heirarchy in **PScm::Expr** which catches the situation where a type other than a list or null is the `cdr` of a list:

  ```
  sub strings {
      my ($self) = @_;
      return ('.', $self->as_string);
  }
  ```

  It returns a list of the string `'.'` plus the result of calling `as_string()` on itself. Since it knows it must be terminating a list, it need not recurse.

- Back to **PScm::Expr::List::Pair**. The `Eval()` method on Lines 114-118 is functionally unchanged from the previous implementation, but it directly accesses the `FIRST` and `REST` fields rather than using the `first()` and `rest()` method calls for a slight performance improvement.

[3] 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 treat the comma as 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.

- A new `map_eval()` method on Lines 120-124 will come in very handy later. It takes an environment as argument and builds a copy of itself with each component replaced by the result of evaluating that component in the argument environment. Because, like `strings()` it must deal with the possibility that the structure is not a true list, an additional `map_eval()` method is provided in the base **PScm::Expr** class that just returns the result of calling `Eval()` on `$self`.
- Finally, an identifying `is_pair()` method is defined to be true in this class only. It is defined false by default in **PScm::Expr**.

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

```
129 package PScm::Expr::List::Null;
130 use base qw(PScm::Expr::List);
131
132 sub new {
133     my ($class) = @_;
134
135     $class = ref($class) || $class;
136     bless {}, $class;
137 }
138
139 sub value { (); }
140
141 sub strings { (); }
142
143 sub first { $_[0] }
144
145 sub rest { $_[0] }
146
147 sub is_null { 1 }
```

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[4].

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.

[4] This interesting trick of having an object to represent the absence of another object is a well-known design pattern called the *Null Object Pattern*.

```
> (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, but there is no requirement for the second argument to be a list. That makes `cons` a second way to create dotted pairs, other than inputting them directly:

```
> (quote (a . b))
(a . b)
> (cons (quote a) (quote b))
(a . b)
> (cons (quote a) (quote (b)))
(a b)
```

`cons` is 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.

```
package PScm::Primitive::Cons;

use base qw(PScm::Primitive);

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

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

1;
```

It can be seen that all it does is to call **PScm::Expr::List**'s `Cons()` method.

8.5.2 Changes to Primitives and Special Forms

As I've already said, It will pay to make use of this new list structure wherever possible throughout the PScheme implementation. The first place we shall do so is in `PScm::Primitive::Apply()`, which can now make use of that new `map_eval()` method:

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

    my @evaluated_args = $form->map_eval($env)->value();
    return $self->_apply(@evaluated_args);
}
```

The question arises as to why I'm then just calling `value()` on the result and passing an ordinary perl list to the individual primitives, after I just said that it was worthwhile passing around the new linked list structures. It's just a personal choice, but I feel that the individual primitives should present an "abstraction layer" such that anything below that layer is pure Perl and does not depend on the details of the PScheme implementation above that layer. Anyway, that's how I see it.

Special forms, on the other hand, are very much part of the PScheme implementation and do make full use of the new linked lists.

First up is **PScm::SpecialForm::Let**. If you remember `let` and friends make use of a shared `UnPack()` method which used to return a list of two references to perl lists, one for the symbols and one for the values, plus the body of the `let` expression. Now it will return PScheme lists instead, and those lists will be passed to the various `Extend*()` methods in the environment implementation, which will also have to change.

anyway here's the new `PScm::SpecialForm::Let::Apply()`:

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

    my ($symbols, $values, $body) = $self->UnPack($form);

    return $body->Eval($env->Extend($symbols, $values));
}
```

Only the variable names have changed: `$symbols` and `$values` used to be called `$ra_symbols` and `$ra_values` to indicate that they were references to arrays. This is no longer the case.

The `UnPack()` still just calls `value()` on the `$form` to get the bindings and the body, but now it makes use of an additional `unpack_bindings()` method to build the new PScheme lists:

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

    my ($bindings, $body) = $form->value;
    my ($symbols, $values) = $self->unpack_bindings($bindings);
    return ($symbols, $values, $body);
}
```

`unpack_bindings()` itself is where the real work gets done:

```
sub unpack_bindings {
    my ($self, $bindings) = @_;
    if ($bindings->is_null) {
        my $null = new PScm::Expr::List::Null();
        return ($null, $null);
    } else {
        my ($symbols, $values) =
            $self->unpack_bindings($bindings->rest);
        return (
            PScm::Expr::List->Cons($bindings->first->first,
                                   $symbols),
            PScm::Expr::List->Cons($bindings->first->rest->first,
                                   $values)
        );
    }
}
```

If it has reached the end of the bindings it creates a new null object and returns two of it. otherwise it calls itself on the rest of the bindings, collects the results, then prepends the symbol from the current binding onto the first list and the value from the current binding onto the second list. Finally it returns those two new lists. Essentially it recurses to the end of the bindings then builds a pair of lists on the way back up. This guarantees that the symbols and values are in the same order in the results as they were in the bindings.

Given the new `UnPack()` method, the `Apply()` methods for **PScm::SpecialForm::LetRec**:

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

    my ($symbols, $values, $body) = $self->UnPack($form);

    return $body->Eval(
        $env->ExtendRecursively($symbols, $values));
}
```

and **PScm::SpecialForm::LetStar**:

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

    my ($symbols, $values, $body) = $self->UnPack($form);

    return $body->Eval(
        $env->ExtendIteratively($symbols, $values));
}
```

are similarily unchanged except for variable renaming.

I've taken the opportunity to make a small but significant change to **PScm::SpecialForm::If**:

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

    my $condition = $form->first;
    my $true_branch = $form->rest->first;
    my $false_branch = $form->rest->rest->first;

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

Because it extracts the condition, true and false branches from the `$form` by using combinations of `first()` and `rest()` rather than just using `value()`, and because the `first()` and `rest()` of the empty list is the empty list, and because the empty list evaluates to itself, our new `if` no longer requires a third argument. If the test fails and a false branch is not supplied the result will be the empty list.

`PScm::SpecialForm::Lambda::Apply()` is unchanged, but the closure that it constructs will make use of the new lists when it interacts with the changed environment implementation.

The only remaining special form is the new **PScm::SpecialForm::Quote**, but we've seen that already.

8.5.3 Changes to Closures

In fact there is very little change in this package. The differences are that the specific `PScm::Closure::Function::Apply()` method (which applies a `lambda` closure to its arguments) now uses `map_eval()` to construct a PScheme list of evaluated arguments and pass that to the shared `_apply()` rather than a perl list:

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

    my $evaluated_args = $form->map_eval($env);
    return $self->_apply($evaluated_args);
}
```

The shared `_apply()` method recieves a PScheme list of actual arguments rather than a reference to an array and it passes that, plus its PScheme list of formal arguments directly to the environment's `ExtendUnevaluated()` method:

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

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

8.5.4 Changes to the Environment

The various `Extend*()` methods of **PScm::Env** now take **PScm::Expr::List** objects as arguments rather than perl listrefs. This makes them the right place to implement the new variable arguments functionality of `lambda`. First of all `Extend()` itself:

```
sub Extend {
    my ($self, $symbols, $values) = @_;

    return $self->ExtendUnevaluated($symbols,
                                    $values->map_eval($self));
}
```

Variable names have changed to reflect the fact that they are no longer references to arrays, and `Extend()` uses `map_eval()` to evaluate the list of values before passing them to `ExtendUnevaluated()`.

`ExtendUnevaluated()` is similarily changed:

```
sub ExtendUnevaluated {
    my ($self, $symbols, $values) = @_;

    my %bindings;
    $self->_populate_bindings(\%bindings, $symbols, $values);
    my $newenv = $self->new(%bindings);
    $newenv->{parent} = $self;
    return $newenv;
}
```

It uses a new private `_populate_bindings()` method to populate a hash of bindings from the `$symbols` and `$values` lists. After that it does what it always did, creating a new **PScm::Env** and setting its `parent` field to `$self` before returning it.

The private `_populate_bindings()` method actually does the “parsing” of the argument list of symbols and their binding to equivalent values.

```
sub _populate_bindings {
    my ($self, $bindings, $symbols, $values) = @_;
    if ($symbols->is_null) {
        die "too many arguments\n" unless $values->is_null;
    } elsif ($symbols->is_pair) {
        if ($values->is_pair) {
            my $symbol = $symbols->first;
            if ($symbol->is_symbol) {
                $bindings->{$symbol->value} = $values->first;
                $self->_populate_bindings($bindings,
                                          $symbols->rest,
                                          $values->rest);
            } else {
                die "bad formal arguments[1]:",
                    $symbol->as_string(), "\n";
            }
        } else {
            die "not enough arguments\n";
        }
    } elsif ($symbols->is_symbol) {
        $bindings->{$symbols->value} = $values;
    } else {
        die "bad formal arguments[2]:",
            $symbols->as_string(), "\n";
    }
}
```

On Line 33 it checks to see if it has reached the end of the list of symbols. If it has, then it throws an error if it has not also reached the end of the list of values (too many arguments).

If `$symbols` is not null, then on Line 35 it checks to see if it is a pair. If it is, then it gets the current symbol, checks that it *is* a symbol, binds it to the equivalent value, and recurses on the rest of the two lists.

If `$symbols` is not a pair, then on Line 38 it checks to see if it is itself a symbol. This would correspond to either a single `args` in a `lambda` statement like `(lambda args ...)`, or a dotted pair terminating a list of arguments. In either case if `$symbols` is a symbol, it binds it to the entirity of the rest of the `$values` list and terminates the recursion.

If `$symbols` is none of the empty list, a pair or a symbol then something is seriously wrong and `_populate_bindings()` throws an exception.

Next up is `ExtendRecursively()`. It is unchanged except that, as in other cases, its arguments have been renamed because they are no longer array references:

```
sub ExtendRecursively {
    my ($self, $symbols, $values) = @_;

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

Finally, `ExtendIteratively()` benefits from the new lists too. It returns `$self` if the list of symbols is empty, otherwise it calls `Extend()` on the first of each list then calls itself on the extended result with the rest of the list:

```
sub ExtendIteratively {
    my ($self, $symbols, $values) = @_;

    if ($symbols->is_null) {
        return $self;
    } else {
        return $self->Extend($symbols->first, $values->first)
                    ->ExtendIteratively($symbols->rest,
                                        $values->rest);
    }
}
```

Note that `ExtendIteratively()` is only used by `let*`, and so does not need to deal with non-lists.

8.5.5 Changes to the Reader

The changes in thr Reader support the input of dot notation, and they make direct use of `PScm::Expr::List::Cons()` to facilitate this. First of all `Read()` itself has been somewhat simplified:

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

020     my $token = $self->_next_token();
021     return undef unless defined $token;
022
023     if ($token->is_open_token) {
024         return $self->read_list();
025     } else {
026         return $token;
027     }
028 }
```

It may look very different but all that has really happened is that the old looping code which collects a list has been moved out into a separate method, and that method, `read_list()`, is now recursive instead of iterative:

```
030 sub read_list {
031     my ($self) = @_;
032     my $token = $self->read_list_element();
033     if ($token->is_close_token) {
034         return new PScm::Expr::List::Null();
035     } elsif ($token->is_dot_token) {
036         $token = $self->read_list_element();
037         die "syntax error" unless $token->is_expr;
038         my $close = $self->read_list_element();
039         die "syntax error" unless $close->is_close_token;
040         return $token;
041     } else {
042         return PScm::Expr::List->Cons($token, $self->read_list());
043     }
044 }
```

`read_list()` collects its components using another new method `read_list_element()`. On Line 33 if it detects a closing brace it returns the empty list. If the token is not a closing brace, then on Line 35 if the token is a dot, it reads the next element, checks that it is an expression, checks that the element after that is a closing brace, and returns the element just read as the entire list. In the final case, if the token is neither a closing brace or a dot, it uses `Cons()` to construct a list with the current token as the first element and the result of calling `read_list()` as the rest. Hopefully you can convince yourself that this will deal correctly with both ordinary lists and dotted pair notation.

`read_list_element()` just centralises the otherwise tedious and repetitive checking for EOF which is a syntax error (unterminated list) while collecting list elements:

```
046 sub read_list_element {
047     my ($self) = @_;
048     my $token = $self->Read();
049     die "unexpected EOF"
050       if !defined $token;
051     return $token;
052 }
```

Finally `_next_token()` has an extra clause to detect a standalone period (dot) and if it does it returns an instance of a new token class **PScm::Token::Dot**:

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

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

    for ($self->{Line}) {
        s/^\(\s*// && return PScm::Token::Open->new();
        s/^\)\s*// && return PScm::Token::Close->new();
        s/^\.\s*// && return PScm::Token::Dot->new();
        s/^([-+]?\d+)\s*//
          && return PScm::Expr::Number->new($1);
        s/^"((?:(?:\\.)|([^"]))*)"\s*// && do {
            my $string = $1;
            $string =~ s/\\//g;
            return PScm::Expr::String->new($string);
        };
        s/^([^\s\(\)]+)\s*//
          && return PScm::Expr::Symbol->new($1);
    }
    die "can't parse: $self->{Line}";
}
```

The new class is in `PScm/Token.pm` alongside the others:

```
package PScm::Token::Dot;

use base qw(PScm::Token);

sub is_dot_token { 1 }

1;
```

There is a default `is_dot_token()` method in **PScm::Token** that returns false.

8.6 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, and out of that we won the ability to construct dotted pairs, and from that we got variable arguments to `lambda` expressions.

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 ||= new FileHandle(">-");
    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);
    }
}
```

8.7 Tests

Tests for version 0.0.5 of the interpreter are in `t/PScm_List.t` and `t/PScm_dot.t`, which you can see in Listing 8.8.1 on the next page and Listing 8.8.2 on page 106.

The tests in `t/PScm_List.t` exercise our new list functions. The first test shows that `list` evaluates its arguments then returns a list of them. The second test proves that the `car` of the list `(1 2 3)` is the value `1`. The third test proves that the `cdr` of the list `(1 2 3)` is the list `(2 3)`. The fourth test proves that `(cons 1 (list 2 3))` is `(1 2 3)`. The fifth test proves that `quote` protects its argument from evaluation: `(quote (list 1 2 3))` is `(list 1 2 3)`. Lastly, three tests verify that dot notation on input and output behaves as expected.

The tests in Listing 8.8.2 on page 106 verify the new variable arguments to lambda expressions, much as our previous examples demonstrated.

8.8 Listings

8.8.1 t/PScm_List.t

```
use strict;
use warnings;
use Test::More;
use lib ’t/lib’;
use PScm::Test tests => 9;

BEGIN { use_ok(’PScm’) }

eval_ok(<<EOF, ’(1 2 3)’, ’list primitive’);
(let ((a 1)
      (b 2)
      (c 3))
  (list a b c))
EOF

eval_ok(<<EOF, ’1’, ’car primitive’);
(let ((a (list 1 2 3)))
     (car a))
EOF

eval_ok(<<EOF, ’(2 3)’, ’cdr primitive’);
(let ((a (list 1 2 3)))
     (cdr a))
EOF

eval_ok(<<EOF, ’(1 2 3)’, ’cons primitive’);
(cons 1 (list 2 3))
EOF

eval_ok(’(quote (list 1 2 3))’, ’(list 1 2 3)’,
    ’quote special form’);

eval_ok(’(quote (1 2 . 3))’, ’(1 2 . 3)’, ’dot in’);

eval_ok(’(quote (1 2 . (3)))’, ’(1 2 3)’, ’dot out’);

eval_ok(’(quote (1 . (2 . (3 .()))))’,
        ’(1 2 3)’, ’complex dot out’);

# vim: ft=perl
```

8.8.2 t/PScm_Dot.t

```
use strict;
use warnings;
use Test::More;
use lib './t/lib';
use PScm::Test tests => 3;
use FileHandle;

BEGIN { use_ok('PScm') }

eval_ok(<<EOF, <<EOR, 'lambda args');
(let ((test (lambda args args)))
  (test 1 2 3))
EOF
(1 2 3)
EOR

eval_ok(<<EOF, <<EOR, 'lambda args');
(let ((test (lambda (a b . rest)
               (list a b rest))))
  (test 1 2 3 4))
EOF
(1 2 (3 4))
EOR

# vim: ft=perl
```

Full source code for this version of the interpreter is available at
`http://billhails.net/Book/releases/PScm-0.0.5.tgz`

Chapter 9

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 or replacement 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 Chapter 5 on page 59, `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 list manipulation tools we will need to do that in the 0.0.5 version of the interpreter from Chapter 8 on page 87 (remember that code and data are the same list objects so list functions can operate on both). All we need to do now is to think of a way to allow us to define macros.

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. We'll use "`macro`" in its place.

9.1 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**.

```
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** 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.

9.2 Evaluating Macros

Consider how `PScm::Closure::Function::Apply()` works. It evaluates its arguments in the passed-in environment then gives the results to its parent `PScm::Closure::_apply()` method. That `_apply()` method 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 = $form->map_eval($env);
    return $self->_apply($evaluated_args);
}
```

And here again is 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.

In fact it's as simple as that, and here's the `Apply()` method for **PScm::Closure::Macro**:

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

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

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 they re-evaluate the result. This is quite subtle. Macros perform substitutions on their arguments, but the result of those substitutions must be subsequently evaluated for the macro to have had the desired effect.

To finish off this part of the implementation, we must remember that in Section 5.3 on page 68 we made closures printable, and since macros are a new kind of closure, we must supply the supporting `_symbol()` method in **PScm::Closure::Macro** for the `PScm::Closure::as_string()` method to find. This `_symbol()` method returns the symbol `macro` so that if a macro is printed it will display as `(macro` ⟨*args*⟩ ⟨*body*⟩`)`. Here's the new `PScm::Closure::Macro::_symbol()` method.

```
sub _symbol {
    PScm::Expr::Symbol->new('macro');
}
```

9.2.1 Trying it out

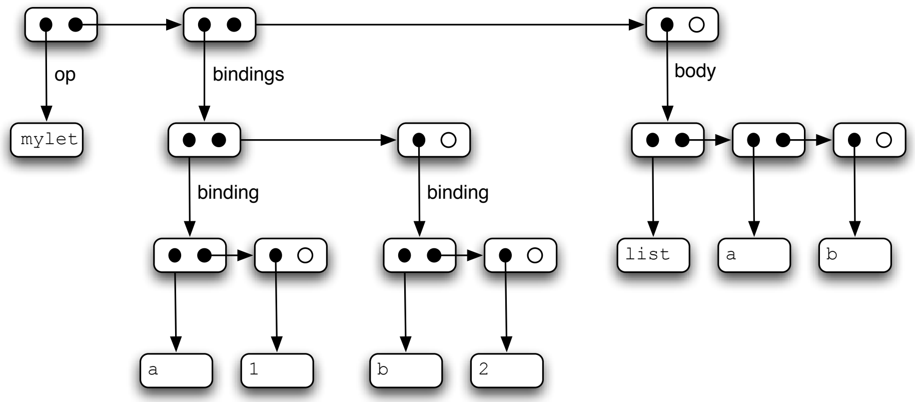
Just for fun, let's take a look at how we might attack the problem which introduced this section: implementing `let` in terms of `lambda`. Remember that any `let` expression has an equivalent `lambda` form, so here's a use of `macro` that translates one into the other:

```
(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 might be useful in the following discussion to refer to Figure 9.1 which shows the internal structure of the `mylet` form that we'll be rearranging.

Figure 9.1: Example `mylet` internal structure



The `mylet` macro 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)
              (map car lst)))
          ...)
  ...)
```

It uses another yet to be defined function `map`, which does the same as Perl's built in `map`: it applies a function to each element of a list and returns a new list of the results[1]. `map` is surprisingly easy to implement in PScheme:

```
(letrec ((map
           (lambda (op lst)
```

[1] Perl actually borrows its map function from Lisp, which has had one for many years.

```
             (if lst
               (cons (op (car lst))
                     (map op (cdr lst)))
               ()))))
         ...)
```

It's a recursive function, hence the need for `letrec` to bind it. Passed a function and list of zero or more bindings, if the list is empty it returns the empty list, otherwise it `cons`-es the result of calling the function on the `car` of the list with to the result of calling itself on the rest (`cdr`) of the list. So for example if `lst` is `((a 1) (b 2))`, then `(map car lst)` would return the list `(a b)`, and that is exactly what the `cars` function does.

`cadrs`[2] is very similar. It walks the list collecting the second component of each sublist (the values of the bindings). So for example given the list `((a 1) (b 2))`, `cadrs` will return the list `(1 2)`.

```
(letrec (...
          (cadrs
            (lambda (lst)
              (map (lambda (x) (car (cdr x))) lst)))
          ...)
  ...)
```

Again it makes use of `map` this time passing it an anonymous function that will take the `car` of the `cdr` of its argument. This is very much in the style of real Scheme programming now: constructing lambda expressions on the fly and passing them to other functions as arguments, I hope you are aquiring a taste for it. Anyway here's the whole `mylet` definition plus some code that calls it.

```
(let* ((mylet
        (letrec ((map
                   (lambda (op lst)
                     (if lst
                       (cons (op (car lst))
                             (map op (cdr lst)))
                       ()))))
                 (cars
                   (lambda (lst)
                     (map car lst)))
                 (cadrs
                   (lambda (lst)
                     (map (lambda (x) (car (cdr x))) lst))))
              (macro (bindings body)
                      (let* ((names (cars bindings))
                             (values (cadrs bindings)))
                        (cons (list (quote lambda) names body)
                              values))))))
    (mylet ((a 1)
```

[2] 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`.

```
           (b 2))
          (list a b)))
```

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 using the appropriate magic:

```
(cons (list (quote lambda) names body) values)
```

A point worth noting is that the constructed mylet macro is a true closure, since it has captured the definitions of the cars and cadrs functions and executes in an outer environment (the let*) where those functions are not visible.

9.2.2 An Improvement

The macro substitution system demonstrated so far 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[3].

However there is one simple addition that we can make, which will greatly improve the usefulness of macros, and that involves an extension to the quote special form that we introduced in Section 8.1 on page 87. If you remember quote just returns its argument, preventing unwanted evaluation. This already has proved useful in the construction of macros, as we have seen above.

Now one perfect use of quote would be to provide *templates* for macros, if we could arrange that parts of the quoted template could be substituted before the quoted template is returned. To that purpose we introduce a keyword unquote which marks a section of a quoted form for evaluation. Perhaps an example might make this clear:

```
> (let ((x "rain")
>       (y "spain")
>       (z "plain"))
>      (quote
>         (the (unquote x)
>          in (unquote y)
>          falls mainly on the (unquote z))))
(the "rain" in "spain" falls mainly on the "plain")
```

[3] A full scheme implementation provides an extend-syntax special form. Using extend-syntax, defining mylet is as simple as:

```
(extend-syntax (mylet)
    (mylet ((var val) ...) body)
    ((lambda (var ...) body) val ...))
```

The `let` bindings bind `x` to the string `"rain"` etc. That is not the important part. The important part is the body of the `let` where the use of the `unquote` keyword allows evaluation of the contained expressions (`x` etc.) despite their being inside a `quote`.

How can this help us with macro definitions? Well in a big way! consider this macro definition of a `while` loop:

```
(define while
  (macro (test body)
         (quote (letrec
                  ((loop
                     (lambda ()
                       (if (unquote test)
                         (begin
                           (unquote body)
                           (loop))
                         ())))))
                  (loop)))))
```

It uses a few features that aren't available yet, like `define` and `begin` (which just executes one expression after another), and it would seem to be in danger of running out of stack, but I hope you can see that essentially the `quote` and `unquote` are doing all of the work building the body of the macro. The quoted result is shown in bold, with the internal substitutions unbolded again.

Implementing `unquote` is easy, but it's a little different from the normal special forms and primitives we've seen up to now. I've been careful to only refer to it as a "keyword", because it means nothing special outside of a quoted expression.

We'll obviously have to change the way `quote` works to make this happen, so lets start by looking at the changed `PScm::SpecialForm::Quote::Apply()`.

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

Rather than just returning its first argument, it now calls a new method `Quote()` on it, passing `Quote()` the current environment. `Quote()` essentially just makes a copy of the expressions concerned, but it keeps an eye out for `unquote` symbols. Now this method will be implemented in the **PScm::Expr** classes as follows:

The default `Quote()` in **PScm::Expr** just returns `$self`:

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

The `Quote()` in **PScm::Expr::List::Pair** is where most of the decision making happens.

```
sub Quote {
    my ($self, $env) = @_;
    if ($self->[FIRST]->is_unquote) {
        return $self->[REST]->first->Eval($env);
    } else {
```

```
138         return $self->quote_rest($env);
139     }
140 }
```

On Line 135 it checks to see if the first element of the list is the symbol `unquote` (`is_unquote`.) If it is then it *evaluates* the second element in the current environment and returns it. If the first element is not `unquote` then it hands over control to a helper routine `quote_rest()`.

Here's `quote_rest()`.

```
142 sub quote_rest {
143     my ($self, $env) = @_;
144     return $self->Cons(
145         $self->[FIRST]->Quote($env),
146         $self->[REST]->quote_rest($env)
147     );
148 }
```

It just walks the list, recursively, constructing a copy as it goes by calling `Quote()` on each element and calling `Cons()` on the quoted subexpression and the result of the recursive call[4].

The **PScm::Expr::List::Null** package inherits `Quote()` from **PScm::Expr**, which just returns `$self`, and **PScm::Expr** also has a `quote_rest()` method which also just returns `$self` and usefully terminates the recursion of the non-empty **PScm::Expr::List** `quote_rest()` method.

```
043 sub quote_rest { $_[0] }
```

That just leaves that `is_unquote()` method. Well since only a symbol could possibly be `unquote`, we can put a default `is_unquote()` method at the top of the expression type hierachy, in **PScm::Expr**, which just returns false:

```
012 sub is_unquote { 0 }
```

Then for **PScm::Expr::Symbol** only, we override that with a method that checks to see if its `value()` is the string `"unquote"`:

```
182 sub is_unquote {
183     my ($self) = @_;
184     return $self->value eq "unquote";
185 }
```

That completes our re-implementation of `quote` to allow the recognition of the `unquote` keyword, but we're not quite done yet.

`quote` and `unquote` turn out to be so useful in the definition of macros that PScheme provides shorthand syntactic sugar for these forms. The construct `'`⟨*expression*⟩ (note the single quote) gets expanded to `(quote` ⟨*expression*⟩`)`, and similarily the construct `,`⟨*expression*⟩ with a leading comma gets expanded to `(unquote` ⟨*expression*⟩`)`. This is fairly easy to do, so let's see what changes we need to make to the reader to make this happen.

First here's the changes to `PScm::Read::_next_token()`.

[4] Note the similarity between this method and the definition of `map` in Pscheme above.

```
066 sub _next_token {
067     my ($self) = @_;
068
069     while (!$self->{Line}) {
070         $self->{Line} = $self->{FileHandle}->getline();
071         return undef unless defined $self->{Line};
072         $self->{Line} =~ s/^\s+//s;
073     }
074
075     for ($self->{Line}) {
076         s/^\(\s*// && return PScm::Token::Open->new();
077         s/^\)\s*// && return PScm::Token::Close->new();
078         s/^\'\s*// && return PScm::Token::Quote->new();
079         s/^\,\s*// && return PScm::Token::Unquote->new();
080         s/^\.\s*// && return PScm::Token::Dot->new();
081         s/^([-+]?\d+)\s*//
082           && return PScm::Expr::Number->new($1);
083         s/^"((?:(?:\\.)|([^"]))*)"\s*// && do {
084             my $string = $1;
085             $string =~ s/\\//g;
086             return PScm::Expr::String->new($string);
087         };
088         s/^([^\s\(\)]+)\s*//
089           && return PScm::Expr::Symbol->new($1);
090     }
091     die "can't parse: $self->{Line}";
092 }
```

The change is very simple. You can see that on Lines 78–79 if it strips a leading quote or comma, it returns an equivalent token object. Those new token types are both in `PScm/Token.pm`, here's **PScm::Token::Quote**.

```
package PScm::Token::Quote;

use base qw(PScm::Token);

sub is_quote_token { 1 }
```

It inherits from **PScm::Token** and a default `is_quote_token()` there returns false.

The equivalent **PScm::Token::Unquote** deliberately inherits from **PScm::Token::Quote** rather than just **PScm::Token** so it gets the overridden `is_quote_token()` method, and supplies an additional `is_unquote_token()` method returning true. Again a default `is_unquote_token()` in **PScm::Token** returns false.

```
package PScm::Token::Unquote;

use base qw(PScm::Token::Quote);
```

```

sub is_unquote_token { 1 }
```

The upshot of this is that both **PScm::Token::Quote** and **PScm::Token::Unquote** return true for `is_quote_token()`, but only **PScm::Token::Unquote** returns true for `is_unquote_token()`. Finally, let's see how the reader `PScm::Read::Read()` makes use of these new token objects.

```
017 sub Read {
018     my ($self) = @_;
019
020     my $token = $self->_next_token();
021     return undef unless defined $token;
022
023     if ($token->is_quote_token) {
024         my $expr = $self->Read;
025         die "syntax Error"
026             unless defined($expr) && $expr->is_expr;
027         return new PScm::Expr::List(
028             $token->is_unquote_token
029                 ? new PScm::Expr::Symbol('unquote')
030                 : new PScm::Expr::Symbol('quote'),
031             $expr
032         );
033     }
034
035     if ($token->is_open_token) {
036         return $self->read_list();
037     } else {
038         return $token;
039     }
040 }
```

The additional code on Lines 23–33 checks to see if the token is a quote or unquote token, and if so reads the next expression, checks that it is valid and returns a new **PScm::Expr::List** containing the appropriate `quote` or `unquote` symbol and the expression read afterwards. The `is_expr()` method is defined to be true in **PScm::Expr** and false in **PScm::Token**, and its use here stops dubious constructs like “`')`”.

So we now have a convenient shorthand for `quote` and `unquote`. To demonstrate it in action, here's that `while` macro again, this time using the new tokens.

```
(define while
  (macro (test body)
         '(letrec
            ((loop
               (lambda ()
                 (if ,test
                   (begin
                     ,body
                     (loop))
                   ())))
             (loop))))
```

We’ll be making significant use of `macro`, `quote` and `unquote` in subsequent chapters, so it’s worth familiarizing yourself now with this new idiom[5].

9.2.3 One Last Addition

Sometimes given a quoted expression, you’d really just like to have it evaluated. It may have been passed as an argument, or it may have been constructed in some other way. What you need is another round of evaluation. This is supplied by the special form `eval`. `eval` really does do just that. For example:

```
> (eval '(* 2 2))
4
```

The quote stopped the first round of evaluation, but `eval` got another try at it. Here’s another example:

```
> (eval (list '* 4 4))
16
```

`eval` is quite simple. It is a special form because it needs access to an environment in which to perform the evaluation (remember primitives have their arguments evaluated for them and so don’t need an environment.) It evaluates its first argument in the current environment (special forms don’t have their arguments evaluated for them,) then it evaluates the result a second time, this time in the top-level environment. Here’s **PScm::SpecialForm::Eval**:

```
package PScm::SpecialForm::Eval;

use base qw(PScm::SpecialForm);

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

You can see that the second round of evaluation is done in the context of the top-level environment obtained by calling a new method `top()` on the current environment. That `top()` method is also very simple:

[5] The quote and unquote described here are done differently in true Scheme. A true Scheme implementation distinguishes between a simple `quote` which does not recognize `unquote`, and an alternative `quasiquote` which does. This means `quote` is as efficient as our original implementation, but we still have access to an `unquote` mechanism. The `quote` form still has the “`'`” syntactic sugar, and `quasiquote` uses the alternative “`‘`” (backtick) shorthand. Additionally a full Scheme provides an `unquote-splicing` (“`,@`”) which expects a list and splices it into the existing form at that point.

```
sub top {
    my ($self) = @_;
    if ($self->{parent}) {
        return $self->{parent}->top;
    } else {
        return $self;
    }
}
```

It just checks to see if it has a parent, calling `top()` recursively on that if it has, and returning itself if it hasn’t.

One thing to watch out for with `eval`: the code that is evaluated is *not* a closure. Any variables in that code will be looked up in the top-level environment, not the one where the expression was constructed, nor the one that is current when `eval` is called. For example:

```
> (let ((* -))
>   (* 3 3))
0
> (let ((* -))
>   (eval ’(* 3 3)))
9
```

Nonetheless `eval` is a useful tool in your kit, we’ll see it in action in later chapters.

9.3 Summary

Here’s the additions to `ReadEvalPrint()` which bind our new macro feature and `eval` in the initial environment. The `quote` binding was already there, and as shown above, `unquote` is only a keyword and does not need a binding:

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

    $outfh ||= new FileHandle(">-");
    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(),
                eval   => new PScm::SpecialForm::Eval(),
                macro  => new PScm::SpecialForm::Macro(),
                quote  => new PScm::SpecialForm::Quote(),
            )
        );
        $result->Print($outfh);
    }
}
```

9.4 Tests

The tests for `macro` and `unquote` are in Listing 9.5.1 on the next page.
The first test just implements and tests the `mylet` example we worked through in the text, and the second test shows `unquote` in action with a variation on another example we've already seen. The third test exercises the syntax extensions in the reader, and the fourth test demonstrates that macros, like closures, produce a textual representation of themselves when printed.

The tests for `eval` are in Listing 9.5.2 on page 122. This just does a simple evaluation of a quoted form.

9.5 Listings

9.5.1 t/PScm_Macro.t

```
use strict;
use warnings;
use Test::More;
use lib ’t/lib’;
use PScm::Test tests => 5;

BEGIN { use_ok(’PScm’) }

eval_ok(<<EOF, ’(1 2)’, ’macros’);
(let* ((mylet
        (letrec ((map
                   (lambda (op lst)
                     (if lst
                       (cons (op (car lst))
                             (map op (cdr lst)))
                       ()))))
                 (cars
                   (lambda (lst)
                     (map car lst)))
                 (cadrs
                   (lambda (lst)
                     (map (lambda (x) (car (cdr x))) lst))))
              (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)))
EOF

eval_ok(<<EOF, <<EOR, ’unquote’);
(let ((x (quote rain))
      (y (quote spain))
      (z (quote plain)))
     (quote (the (unquote x)
             in (unquote y)
             falls mainly on the
             (unquote z))))
EOF
(the rain in spain falls mainly on the plain)
EOR

eval_ok(<<EOF, <<EOR, ’quote and unquote syntactic sugar’);
(let ((x ’rain)
      (y ’spain)
      (z ’plain))
     ’(the ,x
           in ,y
           falls mainly on the
           ,z))
EOF
(the rain in spain falls mainly on the plain)
EOR

eval_ok(<<EOF, <<EOR, ’macro to string’);
(macro (x)
   ’(a ,x))
EOF
(macro (x) (quote (a (unquote x))))
EOR

# vim: ft=perl
```

9.5.2 t/PScm_Eval.t

```
use strict;
use warnings;
use Test::More;
use lib 't/lib';
use PScm::Test tests => 3;

BEGIN { use_ok('PScm') }

eval_ok(<<EOT, <<EOR, 'eval');
(eval '(* 2 2))
EOT
4
EOR

eval_ok(<<EOT, <<EOR, 'eval operates in the top-level environment');
(let ((* -))
  (eval '(* 3 3)))
EOT
9
EOR

# vim: ft=perl
```

Full source code for this version of the interpreter is available at
`http://billhails.net/Book/releases/PScm-0.0.6.tgz`

Chapter 10

Side Effects

The question arises as to why the implementation of the `define` special form, described back in Section 2.4 on page 10 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.

10.1 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, and it makes no difference in what order the arguments to a function are evaluated. 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 as arguments 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 independant part, different components of a program (including the current environment) could be fed to separate processes, perhaps even separate machines on a network.

For example 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 in parallel 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, and there'd have to be some way of telling the server that a particular ticket depends on other tickets, so that a client would never block waiting for the server to return an outstanding ticket... quite an interesting project. The real point though is that both `let` and `lambda` not only conceptually evaluate and bind their arguments in parallel, they could actually do so *without disturbing the sense of the program.*

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 purely 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 *definition*, the creation of new entries in the current environment. Note that this is very different from what the various `let` special forms do. They always create a new environment, they never modify an existing one. Even `letrec` which may appear to be modifying an existing environment by changing bindings is not really doing so: nothing actually gets to *use* that environment until after `letrec` has finished building it, the creation of that new environment is atomic as far as the PScheme language is concerned.

Just to reiterate before we move on. In a functional language it makes no difference in what order the arguments to a function are evaluated, but in a language with side-effects, if those arguments cause side-effects during their evaluation, then the order of evaluation is significant and must be taken into account when designing a program.

10.2 Variable Assignment

This version of the interpreter will only add variable assignment (and sequences,) 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[1]. 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. The reasoning is that the variable being assigned to need not be

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

in the current environment frame: it must be searched for, and if we allow `set!` to create new variables then they would probably be installed in the current top frame. So the scope of a variable that was `set!` would depend on whether the variable already existed or not, which is inconsistent at best.

Anyway, this works:

```
> (let* ((a 5)
>        (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 `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 co-incidence!) 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 = $value;
    } else {
        die "no binding for @{[$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 our new class 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!`.

10.3 Sequences

Sequences are another fairly trivial addition to the language. Rather than a single expression, a *sequence* contains zero or more expressions. Each expression is evaluated in order, and the value of the last expression is the value of the sequence. The value of an empty sequence is null, the empty list. 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 zero or more expressions, evaluates each one, and returns the value of the last expression, or null if there are no expressions to evaluate. 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> ...)
```

With `begin`, we can write that rather awkward `set!` example from the previous section a lot more clearly:

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

`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:

```
package PScm::SpecialForm::Begin;

use base qw(PScm::SpecialForm);

sub Apply {
    my ($self, $form, $env) = @_;
    my (@expressions) = $form->value;

    my $ret = new PScm::Expr::List();

    while (@expressions) {
        $ret = shift(@expressions)->Eval($env);
    }

163     return $ret;
164 }
165
166 1;
```

On Line 155 it extracts the expressions from the argument `$form`. Then on Line 157 it initialises the return value `$ret` to an initial value. On Lines 159-161 it loops over each expression, evaluating it in the current environment, and assigning the result to `$ret`, replacing the previous value. Lastly on Line 163 it returns the final value.

10.4 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 assignment. In fact sequences serve no purpose without side-effects and side-effects are difficult to do without sequences.

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 ||= new FileHandle(">-");
    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(),
                eval   => new PScm::SpecialForm::Eval(),
                macro  => new PScm::SpecialForm::Macro(),
                quote  => new PScm::SpecialForm::Quote(),
                'set!' => new PScm::SpecialForm::Set(),
                begin  => new PScm::SpecialForm::Begin(),
            )
        );
        $result->Print($outfh);
    }
}
```

It's worth pointing out a major difference between PScheme and standard scheme implementations here. In standard scheme, function bodies and the bodies of `let` expressions and their variants are all implicit sequences. So in fact in a real scheme implementation you could just write:

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

In PScheme function and `let` bodies are single statements and require an explicit `begin` to create a sequence, because I wanted to keep the distinction between single expressions and sequences clear to the reader.

10.5 Tests

A test of `set!` and `begin` can be seen in Listing 10.6.1 on the facing page. The test binds `a` to `1` in a `let`, then in the body of the `let` it performs a `begin` which `set!`s `a` to `2` then returns the new value of `a`.

10.6 Listings

10.6.1 t/PScm_SideEffects.t

```
use strict;
use warnings;
use Test::More;
use lib ’t/lib’;
use PScm::Test tests => 2;

BEGIN { use_ok(’PScm’) }

eval_ok(<<EOF, ’2’, ’begin and assignment’);
(let ((a 1))
    (begin (set! a 2)
           a))
EOF

# vim: ft=perl
```

Full source code for this version of the interpreter is available at `http://billhails.net/Book/releases/PScm-0.0.7.tgz`

Chapter 11

define

And so to `define`. `define` is another type of side effect. It differs from `set!` in that it is not an error if the symbol does not exist, and it differs also in that the binding is always installed in the current environment. Therefore executing `define` at the top level prompt will install the binding in the global environment.

11.1 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 very simple.

```
131 sub Define {
132     my ($self, $symbol, $value) = @_;
133
134     $self->{bindings}{ $symbol->value } = $value;
135     return $symbol;
136 }
```

It takes a symbol and a value (already evaluated) as arguments. On Line 134 it directly adds the binding from the symbol to the value, reguardless of any previous value, and on Line 135 it returns the symbol being defined, to give the print system something sensible to print.

11.2 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**.

```
167 package PScm::SpecialForm::Define;
168
169 use base qw(PScm::SpecialForm);
170
171 sub Apply {
```

```
172     my ($self, $form, $env) = @_;
173     my ($symbol, $expr) = $form->value;
174     $env->Define($symbol, $expr->Eval($env));
175 }
176
177 1;
```

All it does is on Line 173 it extracts the symbol and the expression from the argument `$form` then on Line 174 it calls the `Define()` environment method described above with the symbol and evaluated expression (value) as argument.

11.3 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 ||= new FileHandle(">-");
    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(),
        eval   => new PScm::SpecialForm::Eval(),
        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 book 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 closures created by define can call themselves recursively. After all, the environment they capture must, by virtue of how define operates, contain a binding for the closure 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
> (first (list 1 2 3 4))
1
> (rest (list 1 2 3 4))
(2 3 4)
```

These are completely equivalent to the original functions except in name. The primitive definitions are bound to the new symbols first and rest in the top level environment exactly as they are still bound to their original symbols.

11.4 Tests

Tests for define can be seen in Listing 11.5.1 on the next page. The first test demonstrates global definition, and also that closures bound by define are naturally capable of recursion since define assigns them in the current environment. The second test shows that define can be used in any environmental context, even in the body of a closure, to create new bindings. This second test is quite interesting because it demonstrates a function that creates a “helper” function (h) that is only visible to the containing times2 closure.

11.5 Listings

11.5.1 t/PScm_Define.t

```
use strict;
use warnings;
use Test::More;
use lib 't/lib';
use PScm::Test tests => 3;

BEGIN { use_ok('PScm') }

eval_ok(<<EOF, <<EOX, 'global definition');
(define square
        (lambda (x) (* x x)))
(square 4)
EOF
square
16
EOX

eval_ok(<<EOF, <<EOX, 'local definition');
(define times2
    (lambda (x)
        (begin
            (define h
                (lambda (k)
                    (- k (- k))))
            (h x))))
(times2 5)
EOF
times2
10
EOX

# vim: ft=perl
```

Full source code for this version of the interpreter is available at http://billhails.net/Book/releases/PScm-0.0.8.tgz

Chapter 12

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" OO languages in that *everything* in the language is an object[1]. Other languages such as Perl and this PScheme implementation add OO features to what is essentially a procedural core.

Every object implementation 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 object components from other parts of a program: should the fields of an object be visible at all outside of that object? Should an object be able to see the fields in an object it inherits from? Should an object of a particular class be able to see fields of another object of the same class? 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 OO "style". I've also decided, somewhat perversely, to be as *different* from the Perl 5 object implementation as possible within the constraints imposed, in order to give the reader a sense of the different choices that are available.

12.1 Features of this implementation

Rather than listing the features up front, let's start off with some examples of the extension in action, showing its syntax etc. We can pick up the semantics as we go along.

First of all a new special form called `make-class` creates a new class. It returns that new class as its value. The syntax of `make-class` is:

```
(make-class ⟨parent-expression⟩ (⟨field⟩ ...) ⟨method⟩ ...)
```

⟨*parent-expression*⟩ is an expression evaluating to another class. Each ⟨*field*⟩ is a symbol naming one of the object's fields. Each ⟨*method*⟩ has the form:

[1] 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, exceptions and even classes are objects in SmallTalk. Furthermore even the simplest operations such as addition are methods: adding `2 + 2` involves sending the object `2` the message `+` with argument `2`, and conditional expressions like `if` are implemented by sending a boolean object representing the condition a message `ifTrue` with argument the code block to execute if the condition is true. See [9] if you want more information.

```
(⟨name⟩ (⟨arg⟩ ...) ⟨body⟩)
```

where ⟨*name*⟩ is the name of the method, the ⟨*arg*⟩s are the arguments, and ⟨*body*⟩ is the body, much like `lambda` expressions. Also, somewhat like `lambda` expressions, but not identically, method bodies capture the lexical environment current when the *class* is created.

The system provides a pre-built root class to act as a starting point for any class hierachy. That class is bound to the symbol `root`.

So here is how we might create a crude "bank account" class:

```
(define Account
    (make-class
        root
        (balance)
        (init (amount) (set! balance amount))
        (deposit (amount) (set! balance (+ balance amount)))
        (withdraw (amount) (set! balance (- balance amount)))
        (balance () balance)
        (clone () (class balance))))
```

`make-class` returns the new class, and we bind that to the symbol `Account` with `define`.

Our new `Account` descends directly from the `root` class. It has a single `balance` field, and five methods called `init`, `deposit`, `withdraw`, `balance` and `clone`. Note that there is no conflict between the field called `balance` and the method of the same name: methods exist in a separate namespace.

The `init` method is special. It will get called whenever a new object is created. It should normally assign values to the object's fields, since they initially all have a value of zero.

We'll come back to that `clone` method in a bit.

Creating instances of `Account` simply involves invoking the `Account` class with whatever arguments its `init` method takes. It will return an object of class `Account`, suitably initialised:

```
(define my-account (Account 20))
```

This creates an object of class `Account` with an initial `balance` of `20`, since `init` assigns the argument `20` to the `balance` field. `define` binds the new object to the symbol `my-account`.

This starts to explain that mysterious `clone` method. All methods have access to a special variable called `class`, that refers to the class of the object; this has some parallels with the Perl `__PACKAGE__` identifier. So `clone` need only call `(class balance)` to create a copy of the current object.

To call a method on an object you invoke the object, with the method name as the first argument and arguments to the method itself as the remaining arguments:

```
(my-account deposit 10)
```

The `deposit` method takes the argument `10` and adds it to the existing `balance`.

12.1.1 Inheritance

Classes and objects wouldn't be much fun without inheritance, so here's an example of a derived class:

```
(define InterestAccount
    (make-class
        Account
        (rate)
        (init (interest amount)
            (begin (super init amount)
                   (set! rate interest)))
        (accumulate ()
            (this deposit (* (this balance) rate)))))
```

Note a few things in particular.

- The parent class in this case is `Account`, the class we created previously.
- The `InterestAccount` class adds an extra field, `rate`.
- The `InterestAccount`'s `init` method, before setting the new object's `rate` to `interest`, invokes the parent's `init` method with the call `(super init amount)` to set the `balance`. This is more or less equivalent to the Perl `SUPER` method qualifier:

  ```
  $self->SUPER::init($amount);
  ```

  The `super` object is an implicit field of the class, and is automatically initialised when an object is created. It represents the parent object.
- The special variable `this` is an implicit argument to methods, it represents the object on which the method was originally invoked, just as `$self` conventionally does for perl methods[2].
- The `InterestAccount` class cannot see its parent's fields, only its methods. It has to call `(this balance)` to get the value of `balance` and `(this deposit` ⟨*arg*⟩`)` to change it.
- Methods are always called from an object. There are no shorthands. Even within a method body it is necessary to use one of the special objects `this` or `super` to call a method on the current object.

The `InterestAccount` class can be used as follows:

```
> (define my-account (InterestAccount 20 2))
my-account
> (my-account deposit 10)
30
> (my-account balance)
30
> (my-account accumulate)
90
> (my-account balance)
90
```

Quite a nice rate of interest that is.

[2] We could have chosen the name `self` instead, to make the examples easier for perl programmers to read, but the perl under the hood might start to get ugly.

12.1.2 Class Variables and Methods

Although there is no direct support for class variables and methods, the fact that classes, and hence methods, capture the current environment allows us to fake them. Here's a variant on the `Account` class that keeps track of the total amount of money in all account objects:

```
(define Account
    (let ((total 0))
        (make-class
            root
            (balance)
            (set-balance (op amount)
                (begin (set! balance (op balance amount))
                       (set! total (op total amount))))
            (init (amount) (this set-balance + amount))
            (deposit (amount) (this set-balance + amount))
            (withdraw (amount) (this set-balance - amount))
            (balance () balance)
            (total () total))))
```

The `let` binds `total` to an initial value of `0` then evaluates the `make-class` construct in this new environment. The newly created class captures that environment. That new class is then returned by the `let` expression and bound to `Account`.

Every instance of `Account` will share the lexically scoped variable `total`. Rather than change each of `init`, `deposit` and `withdraw` to individually maintain both the value of `total` and `balance`, a new method `set-balance` has been added. It takes an operation `op` (`+` or `-`) and an `amount` and applies the operation with the amount separately to both the `balance` and the `total`. The `init`, `deposit` and `withdraw` methods have been modified to use this new method, and another new method, `total`, provides read-only access to the value of `total`.

12.1.3 `super` Calls

We're not quite finished with the details. There's one last wrinkle to deal with reguarding that `super` object mentioned earlier. Although on the surface it may appear to behave just like any other object, a call to a method on the `super` object cannot just be a simple method invocation. If it was, then `this` in the called method would be the object referred to by `super` in the calling method. Instead the `this` in the called method should continue to refer to the `this` that was current before the `super` call. To make that clear, consider a rather contrived example:

```
> (define cat
>    (make-class
>        root
>        ()
>        (poke () (this respond))
>        (respond () 'purr)
>    ))
cat
> (define lion
>    (make-class
>        cat
>        ()
>        (poke () (super poke))
>        (respond () 'roar)
>    ))
lion
> (define leo (lion))
leo
> (leo poke)
roar
```

- `leo` is an instance of `lion` and the call `(leo poke)` invokes `lion`'s `poke` method.
- `lion`'s `poke` method does `(super poke)`, which invokes `cat`'s `poke` method.
- `cat`'s `poke` method does `(this respond)` but since, even after the `super` call, `this` is still the originating object `leo` of class `lion`, it is `lion`'s `respond` method that gets invoked, resulting in "`roar`" rather than the "`purr`" from `cat`'s `respond` method.

12.1.4 Feature Summary

Here's a summary, then, of the main features of our implementation

- There is no multiple inheritance.
- There are no class variables or methods, but the capture of the lexical environment by a class definition allows us to fake them.
- Only methods are inherited: fields are visible only to methods of the class in which they are defined.
- Fields behave like normal variables within method bodies.
- The special variable `this` is always available as an implicit argument to each method and refers to the object that the method was originally invoked on.
- The special variable `super` is always available as an implicit field of every object and refers to the parent object of the object that owns the method.
- Calling a method on the `super` object passes `this`, not `super`, as the implicit object argument to the called method.
- The special variable `class` is always available as an implicit field in every object and refers to the class of the object.

12.2 Implementation

So how do we go about implementing this extension? Well, to be frank, in a fairly ad hoc way. It should be obvious that PScheme objects have a lot in common with environments, namely that they store the values of variables. Our existing environment implementation can easilly be pressed in to service as a PScheme object. In fact in this implementation objects *are* environments, or rather, chains of environments linked via a `super` field where each individual environment represents an instance of the equivalent class in the class hierachy.

We need only add a couple of methods to our existing environment implementation to get all we need.

- The first of those new **PScm::Env** methods would of course be an `Apply()`, since now environments are exposed in the language as objects and are invoked as operators: `(`⟨*object*⟩ ⟨*method*⟩ ⟨*arg*⟩`...)`.
- The second method we'll need is some sort of `lookup_method()` method because we've said that PScheme methods live in a separate namespace from normal fields. We can always recognise PScheme method invocation and hence the PScheme method name by context: it is always the first "argument" to an object. We cheat egregiously here and just use the `class` binding in each PScheme object to locate the PScheme class, and check to see if the PScheme method is in it. If not found then `lookup_method()` recurses down the chain of objects via the `super` binding and tries again.

Of course we need an object to represent PScheme classes, but that is just going to contain the parent PScheme class, the fields and methods of the class, and the environment that was current at the point of its creation. That nascent PScheme Class package will need an `Apply()` method that will create PScheme objects on demand, passing any arguments to that object's nearest `init` method.

That is pretty much all we need to do. Methods can act just like closures but will extend the environment representing the object when they are called. There are a few fiddly details around method invocation on a `super` object, but we'll deal with that later.

12.2.1 Class Creation with make-class

Starting from the top, the special form `(make-class` ⟨*parent-expr*⟩ `(`⟨*field*⟩`...)` ⟨*method*⟩`...)` is simple to implement, it just returns an instance of that yet to be described **PScm::Class** object. Hopefully it's not too counterintuitive that PScheme classes are in fact Perl objects. Here's the implementation of `make-class` in a new **PScm::SpecialForm::MakeClass** package:

```
package PScm::SpecialForm::MakeClass;

use base qw(PScm::SpecialForm);

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

    my $parent_expr = $form->first;
    my $fields = $form->rest->first;
    my $methods = $form->rest->rest;
    my $parent_class = $parent_expr->Eval($env);
    return PScm::Class->new($parent_class,
                            $fields,
                            $methods,
                            $env);
193 }
194
195 1;
```

As is usual for a special form it just has an `Apply()` method. On Line 185 that unpacks the parent expression, fields and methods from the argument `$form`, and on Line 188 it evaluates the parent expression in the current environment to get an actual parent class (**PScm::Class**) object. Finally on Line 189 it returns a new instance of **PScm::Class** capturing those values and the current environment.

The `new()` method in **PScm::Class** doesn't do anything too clever either, On Line 10 it declares a hashref `$rh_methods`, and then on Line 11 it calls a helper static method `_populate_methods_hash()` which will chop up the PScheme methods into names, arguments and bodies, storing each pair of args and body in the hash keyed on the PScheme method name. Then starting on Line 13 it returns a new instance containing that hash along with the parent PScheme class, fields and current environment:

```
007 sub new {
008     my ($class, $parent, $fields, $methods, $env) = @_;
009
010     my $rh_methods = {};
011     $class->_populate_methods_hash($rh_methods, $methods);
012
013     return bless {
014         parent  => $parent,
015         fields  => [$fields->value],
016         methods => $rh_methods,
017         env     => $env,
018     }, $class;
019 }
```

Here's `_populate_methods_hash()`:

```
021 sub _populate_methods_hash {
022     my ($class, $rh_methods, $methods) = @_;
023     if ($methods->is_pair) {
024         my $method = $methods->first;
025         my ($name, $args, $body) = $method->value;
026         $rh_methods->{ $name->value } =
027                 { args => $args, body => $body };
028         $class->_populate_methods_hash($rh_methods, $methods->rest);
029     }
030 }
```

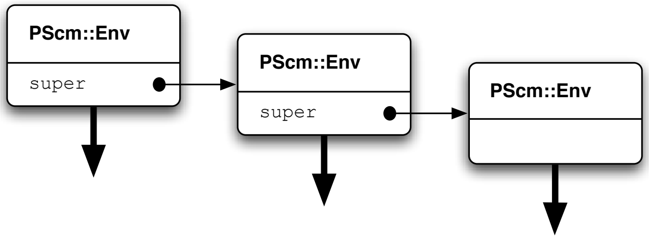
That's it for `make-class`. Following our previous course, we next need to look at PScheme object creation, which occurs when a PScheme class is invoked with arguments intended for its `init` method: (⟨*class*⟩ ⟨*arg*⟩...).

12.2.2 Object Creation

It's probably worth stepping back at this stage to clarify how our environments will be arranged into objects. As I've already mentioned each PScheme object (environment) extends the environment that was captured by its class. Furthermore each PScheme object (environment) has a `super` field referring to the anonymous PScheme object created by its parent PScheme class.

That means we end up with a situation something like Figure 12.1

Figure 12.1: Notional object structure



Although this figure does not tell the whole story, it at least emphasizes that a PScheme object consists of a number of environment frames, one for each class in the equivalent PScheme class hierachy, but those environment frames are connected not by a direct parent/child relationship but via an ordinary variable in each frame called `super`. The environment frames representing each object extend the environment that their respective classes captured. This is implied, but not shown, by the unterminated arrows in the figure.

So we were about to take a look at how PScheme classes create PScheme objects. Classes create objects when they are invoked as `(`⟨*class*⟩ ⟨*arg*⟩`...)`. To make anything invokeable we just need give it an `Apply()` method, and here's the one for **PScm::Class**:

```
032 sub Apply {
033     my ($self, $form, $env) = @_;
034
035     my $new_object = $self->make_instance();
036     $new_object->call_method($new_object, "init", $form, $env);
037     return $new_object;
038 }
```

On Line 35 it calls a `make_instance()` method to create a new PScheme object (really a **PScm::Env**). Then on Line 36 it calls the PScm `init` method of the new object. This is done using a `call_method()` method of **PScm::Env**. This takes the PScheme object on which the method is being invoked (`$new_object`, which will be passed to the PScheme method as `this`), the name of the PScheme method (`"init"`), and the arguments to the method itself (`$form` and `$env`.) We'll look at `call_method()` later.

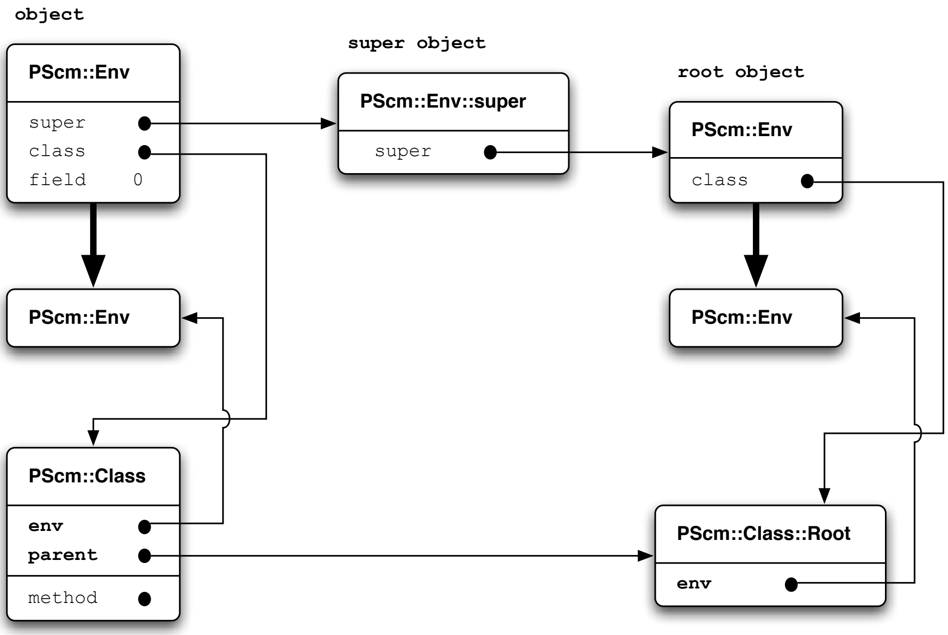
The `make_instance()` method of **PScm::Class** must recurse down to the root of the PScheme class hierachy, creating a chain of anonymous PScheme objects on the way back up, each linked to its parent by a `super` field. Here it is:

```
040 sub make_instance {
041     my ($self) = @_;
042
043     my $parent_instance = $self->{parent}->make_instance();
044
045     return $self->{env}->ExtendUnevaluated(
046         new PScm::Expr::List(
047             PScm::Expr::Symbol->new("class"), # $self
048             PScm::Expr::Symbol->new("super"), # $parent_instance
049             @{ $self->{fields} },             # 0...
050         ),
051         new PScm::Expr::List(
052             $self,                                                 # "class"
053             PScm::Env::Super->new(super => $parent_instance),      # "super"
054             ((PScm::Expr::Number->new(0)) x @{ $self->{fields} }), # field...
055         )
056     );
057 }
```

The first thing it does, on Line 43 is to call its parent PScheme class' `make_instance()` method to get an instance of its parent class. Then starting on Line 45 the rest of the method extends the environment that the **PScm::Class** object captured when it was created, with appropriate bindings from `class` to `$self`, `super` to the `$parent_instance` and from each of the PScheme classes `fields` to initial values of zero. It is this new environment that is returned by `make_instance()`.

If you were reading the above code carefully, you'd have noticed that the `super` field does not link directly to the parent instance, but via a derivative of **PScm::Env** called **PScm::Env::Super**. This is so that the `super` object can have a separate `Apply()` method. That gives the lie to our simple picture of environments-as-objects in Figure 12.1 on the preceding page. In fact the true situation is shown in Figure 12.2.

Figure 12.2: Real object structure



To keep things simple this figure only shows an object whose immediate parent is `root`. You can see that the PScheme object is joined to its parent via a **PScm::Env::Super** object bound to its `super` field, and that the **PScm::Env::Super** object also has a `super` field providing the link to the real parent. Additionally each PScheme object has a `class` binding referring to the PScheme class that created it. That is a **PScm::Class** for all but the root object, which has no `super` binding and has a `class` binding that refers to a **PScm::Class::Root** object. **PScm::Class::Root** is a derivative of **PScm::Class**, and it is a **PScm::Class::Root** instance that will be bound to `root` in the initial environment.

That conveniently brings us back round to the `make_instance()` method, and how that recursive call to the parent PScheme class' `make_instance()` is terminated. That happens when it hits the `make_instance()` method of the **PScm::Class::Root** package, shown next.

```
070 package PScm::Class::Root;
071
072 use base qw(PScm::Class);
073
074 sub new {
075     my ($class, $env) = @_;
076
077     return bless {
078         parent  => 0,
079         fields  => [],
080         methods => {},
081         env     => $env,
082     }, $class;
083 }
084
085 sub make_instance {
086     my ($self) = @_;
087
088     return $self->{env}
089       ->ExtendUnevaluated(
090           new PScm::Expr::Symbol("class"),
091           $self
092       );
093 }
094
095 1;
```

The `new()` method on Lines 74-83 is just meant to be easy to call from the repl where the `root` class will be initialised. It creates a PScheme class with no parent, no fields, no methods, and whatever `env` is passed in[3].

The `make_instance()` method on Lines 85-93 is not recursive, it just extends the captured environment with a binding of `class` to `$self` (the **PScm::Class::Root** object,) returning the result. Note

[3] It's actually redundant for that root environment to have a `class` binding or a parent environment, since the root class currently has no methods. However if we did want to extend the implementation to add generic methods to the root class then all the pieces we need are in place, so we can accept that redundancy for now.

that it takes advantage of the fact that `ExtendUnevaluated()` can cope with a single symbol and value as well as lists of the same.

12.2.3 `init` Method Invocation

So far we've looked at how PScheme classes are created and how they in turn create PScheme objects. Next we're going to look at how PScheme methods are invoked, starting with the `init` method.

If you remember, the PScheme `init` method is called by `PScm::Class::Apply()` when creating a new object, by calling `call_method()` on the instance `$new_object` returned by `PScm::Class::make_instance()`. Since `$new_object` is a **PScm::Env**, `call_method()` must be a method of **PScm::Env**, and here it is.

```
168 sub call_method {
169     my ($self, $this, $method_name, $args, $env) = @_;
170
171     if (my $method = $self->_lookup_method($method_name)) {
172         return $method->ApplyMethod($this, $args, $env);
173     }
174 }
```

`call_method()` is passed both the "real" perl object `$self` and the object representing PScheme's idea of the current object, `$this`, that the PScheme method is being invoked on. Normally these are one and the same. Additionally it is passed the method name, arguments and another environment in which the arguments are to be evaluated if a method is found. On Line 171 it uses `_lookup_method()` (discussed next) to find the method, and if found then on Line 172 it invokes the PScheme method by calling its `ApplyMethod()` and returns the result. If no method can be found it returns `undef`, and since in the case of **PScm::Class**' `Apply()` the result of calling `init` is discarded anyway, it is not fatal if an `init` method is not found.

`_lookup_method()` employs a simple strategy to locate a method. First it checks in the current PScheme object's `class`, and if it can't find the method there, it recurses to its `super` object. That leads to the equally simple definition below.

```
sub _lookup_method {
    my ($self, $method_name) = @_;

    return $self->_lookup_method_here($method_name)
      || $self->_lookup_method_in_super($method_name);
}
```

So `_lookup_method()` breaks down into two simpler methods: `_lookup_method_here()` and `_lookup_method_in_super()`, which it tries in turn. `_lookup_method_here()` is similarily simple.

```
sub _lookup_method_here {
    my ($self, $method_name) = @_;

    if (exists $self->{bindings}{class}) {
        return $self->{bindings}{class}
          ->get_method($method_name, $self);
    }
}
```

It checks to see if the current object has a `class` binding, and if so it calls `get_method()` on the class, returning the result. `get_method()` will return `undef` if it can't find the method in the class, and `_lookup_method_here()` returns `undef` if there is no `class` binding.

`_lookup_method_in_super()` is equally simple.

```
sub _lookup_method_in_super {
    my ($self, $method_name) = @_;

    if (exists $self->{bindings}{super}) {
        return $self->{bindings}{super}
          ->_lookup_method($method_name);
    }
}
```

It checks to see if the current PScheme object has a `super`, and if so it calls `_lookup_method()` on it. Otherwise it returns `undef`.

Since `_lookup_method()`, `_lookup_method_here()` and `_lookup_method_in_super()` are all methods of **PScm::Env**, they are all available to **PScm::Env::Super** where they work without modification: `super` objects have a `super` field but no `class` field.

Going back to `_lookup_method_here()`, if that found a `class` binding, it called `get_method()` on the PScheme class, passing it the method name to look for, and perhaps less obviously, `$self` as well. Here's what `get_method()` back in **PScm::Class** does with those arguments.

```
059 sub get_method {
060     my ($self, $method_name, $object) = @_;
061
062     if (exists $self->{methods}{$method_name}) {
063         return PScm::Closure::Method->new(
064             $self->{methods}{$method_name}{args},
065             $self->{methods}{$method_name}{body}, $object);
066     }
067 }
```

On Line 62 it looks in its `methods` subhash for a key matching the string `$method_name`. If it finds one it knows it has found the method and returns a new instance of **PScm::Closure::Method**, a closure just like a lambda expression, containing the relevant method args and method body from the subhash, and most importantly capturing the environment `$object`. Reasoning backwards, this is correct, `$object` (the `$self` from `_lookup_method_here()`) is the environment in which the method was found, (via `class`) and that is the environment that the method should extend when it executes, so that the method body can "see" the fields of the object.

I'd just like to emphasize a point here, the object `$object` passed to `get_method()` is *not* necessarily the same as the `this` that will be passed to the method when it executes. That would only be true if the method was found in the first PScheme object that `_lookup_method()` looked in.

There's very little left to cover now. We just need to take a look at **PScm::Closure::Method**. This is a subclass of **PScm::Closure**, as you can see.

```
067 package PScm::Closure::Method;
068
069 use base qw(PScm::Closure);
070
071 sub new {
072     my ($class, $args, $body, $env) = @_;
073
074     bless {
075         args => PScm::Expr::List->Cons(
076             PScm::Expr::Symbol->new("this"), $args
077         ),
078         body => $body,
079         env  => $env,
080     }, $class;
081 }
082
083 sub ApplyMethod {
084     my ($self, $this, $form, $env) = @_;
085     my $evaluated_args = $form->map_eval($env);
086     return $self->_apply(PScm::Expr::List->Cons($this, $evaluated_args));
087 }
088
089 1;
```

The `new()` method on Lines 71-81 is the one we just saw being called by `get_method()`. What differentiates it from the normal **PScm::Closure::Function** `new()` method is that on Line 75 it prepends the symbol `this` to the argument list as it constructs the closure. That "implicit" argument will be supplied by `ApplyMethod()` which you can also see in this package.

The defining feature of a closure is that it captures an environment when it is created and extends it when it is executed. These method closures are no different, but the environment that they capture is the object in whose class the method was found. Hence method bodies can see the fields of the object as normal variables: they *are* normal variables.

`ApplyMethod()` also behaves pretty much like the normal closure's `Apply()`, but it differs in having an extra `$this` argument. On Line 85 it calls `map_eval()` on the argument `$form` with the current environment to get a list of evaluated arguments, just as the normal closure's `Apply()` does. But then on Line 86 it prepends `$this` (the PScheme `this`) to those actual arguments when calling the generic **PScm::Closure** `_apply()` method. This ties in with the `new()` method having supplied an extra symbol `this` to the list of formal arguments.

We've now covered everything to do with PScheme object creation and initialisation in PScheme. Along the way we've seen, by following the process of calling an object's `init` method, most of the machinery behind method invocation. There are only two remaining details to fill in.

12.2.4 General Method Invocation

The first of those is normal PScheme method invocation. That is done by invoking the object with the method name and arguments, for example:

```
(my-account deposit 10)
```

Since objects (environments) are now directly invokeable, they too must have an `Apply()` method, shown here:

```
176 sub Apply {
177     my ($self, $form, $env) = @_;
178 
179     my ($method_symbol, $args) = ($form->first, $form->rest);
180     my $res =
181       $self->call_method($self, $method_symbol->value, $args, $env);
182 
183     if (defined $res) {
184         return $res;
185     } else {
186         die "method ", $method_symbol->value, " not found\n";
187     }
188 }
```

On Line 179 it splits the argument `$form` into a method name (a symbol) and a list of arguments to the method. Then on Line 181 it attempts to call the method, and collects the result. Now the result will only be undefined if a method could not be found, in which case an exception is raised. Otherwise the result is returned.

12.2.5 `super` Method Invocation

The only remaining piece is invocation of a PScheme method on a `super` object. `super` is bound to a **PScm::Env::Super** instance, which in turn has a `super` binding referring to the next object in the chain. Here's **PScm::Env::Super**:

```
191 package PScm::Env::Super;
192 
193 use base qw(PScm::Env);
194 
195 sub Apply {
196     my ($self, $form, $env) = @_;
197 
198     my ($method, $args) = ($form->first, $form->rest);
199     my $this = $env->LookUp(PScm::Expr::Symbol->new("this"));
200     my $res  =
201       $self->call_method($this, $method->value, $args, $env);
202 
203     if (defined $res) {
```

```
204         return $res;
205     } else {
206         die "method ", $method->value, " not found in super\n";
207     }
208 }
209
210 1;
```

If you compare the `Apply()` method here with the one in **PScm::Env** above, you can see they differ in that on Line 199 the `Apply()` looks up `this` in the current environment. Then on Line 201 it passes `$this` instead of `$self` to `call_method()`. The upshot of that is the variable `$this` will be the one that gets bound to the implicit `this` argument to the PScheme method when it is invoked.

12.2.6 Wiring it up

And finally, we just need to see how the new object code is wired into the repl. Here's `ReadEvalPrint()` for version 0.0.9 of our interpreter.

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

    $outfh ||= new FileHandle(">-");
    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(),
        '+'          => new PScm::Primitive::Add(),
        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(),
        eval         => new PScm::SpecialForm::Eval(),
        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(),
        'make-class' => new PScm::SpecialForm::MakeClass(),
    );

    $initial_env->Define(
        PScm::Expr::Symbol->new("root"),
061         PScm::Class::Root->new($initial_env)
062     );
063
064     while (defined(my $expr = $reader->Read)) {
065         my $result = $expr->Eval($initial_env);
066         $result->Print($outfh);
067     }
068 }
```

The changes are in bold. On Line 41 I finally caved in and added primitive addition as a builtin. I leave it to the reader to do the same. On Line 56 you can see the additional binding of `make-class` to a **PScm::SpecialForm::MakeClass** object, and on Line 59 we attach a new **PScm::Class::Root** to the symbol `root` in the initial environment. That needs to be done using `Define()` because we need to pass the value of `$initial_env` to the `new()` method of **PScm::Class::Root**

12.3 Summary and Variations

This object extension added new methods to **PScm::Env** to allow environments to behave as operators within the language. It added new classes **PScm::SpecialForm::MakeClass** to implement the `make-class` special form, **PScm::Class** and **PScm::Class::Root** to host our PScheme class code, **PScm::Env::Super** to provide an alternative `Apply()` method for `super` method invocation, and **PScm::Closure::Method** for the modified closure behaviour of PScheme methods. To summarise:

- A **PScm::SpecialForm::MakeClass** object bound to `make-class` in the initial environment creates instances of **PScm::Class** when called with arguments `parent`, `fields` and `methods`.

- A **PScm::Class::Root** object bound to `root` in the initial environment provides a base class in which other classes can be rooted.

- **PScm::Class** has an `Apply()` method, and when a **PScm::Class** is invoked with arguments, that `Apply()` method first creates a PScm object, which is in fact just an instance of a **PScm::Env**, then calls that new object's `init` method with the arguments that were passed to the class.

- To create a new object (environment) **PScm::Class**'s `Apply()` calls `make_instance()` which recurses down the chain of PScheme classes to the `root` and creates a chain of objects (environments) on the way back up

  - Each element of this chain has a `class` binding referring to the **PScm::Class** instance that created it.
  - Each element is joined indirectly to the previous by a `super` binding referring to a **PScm::Env::Super** object which itself has a `super` binding referring to the actual parent object.
  - Each object in the chain extends the environment that its class captured when it was created.

- To call a PScheme method, `init` or otherwise, the `call_method()` method of **PScm::Env** is used. This uses `_lookup_method()` to locate the method and create an instance of **PScm::Closure::Method** from it. If a method is found `_call_method()` invokes the method's `ApplyMethod()`.

- `_lookup_method()` looks first in the current environment for a `class` binding and if found checks the class for the method, otherwise it recurses on the `super` field.
- `Apply()` in **PScm::Env** passes `$self` (the object on which the method is being invoked) as the value of `this` to `call_method()`.
- The `Apply()` method of **PScm::Env::Super** instead looks up the value of `this` in the current environment and passes that as the value of `this` to `call_method`.
- When a PScheme method is found in a PScheme class, the **PScm::Class** method `get_method()` creates an instance of **PScm::Closure::Method**, a closure which captures the environment (object) in whose class the method was found, and which has an additional implicit `self` argument.
- When the method body is executed by **PScm::Closure::Method**'s `ApplyMethod()` the value of `this` is passed as an additional argument.

Since the new **PScm::Class** has a file to itself there's a full listing in Listing 12.5.1 on page 153.

To recap, let's consider our original example classes: `Account` and `InterestAccount`
Figure 12.3 on the following page shows the situation after the creation of the `Account` and `InterestAccount` classes, and the `my-account` instance of an `InterestAccount` that was discussed in the examples in Section 12.1 on page 135.

You can see that the `my-account` object is really just a **PScm::Env** and its parent env is the global environment (implied by the unterninated heavy arrows.) The `my-account` object's parent environment is the global environment because that is the environment that the `InterestAccount` class was created in. If the `InterestAccount` class had captured a different environment, then that would have been the one that instances of that class extended.

Note the three bindings in the `my-account` object. The `rate` variable is the one supplied by the class definition, the other two, `super` and `class` are automatically provided by the implementation when new objects are created.

The `super` variable refers to a **PScm::Env::Super** object, derived from **PScm::Env**, which in turn has a `super` variable, and differs from **PScm::Env** only in its `Apply()` method, which arranges to forward the current value of `this` (rather than the `super` object itself) to the called method.

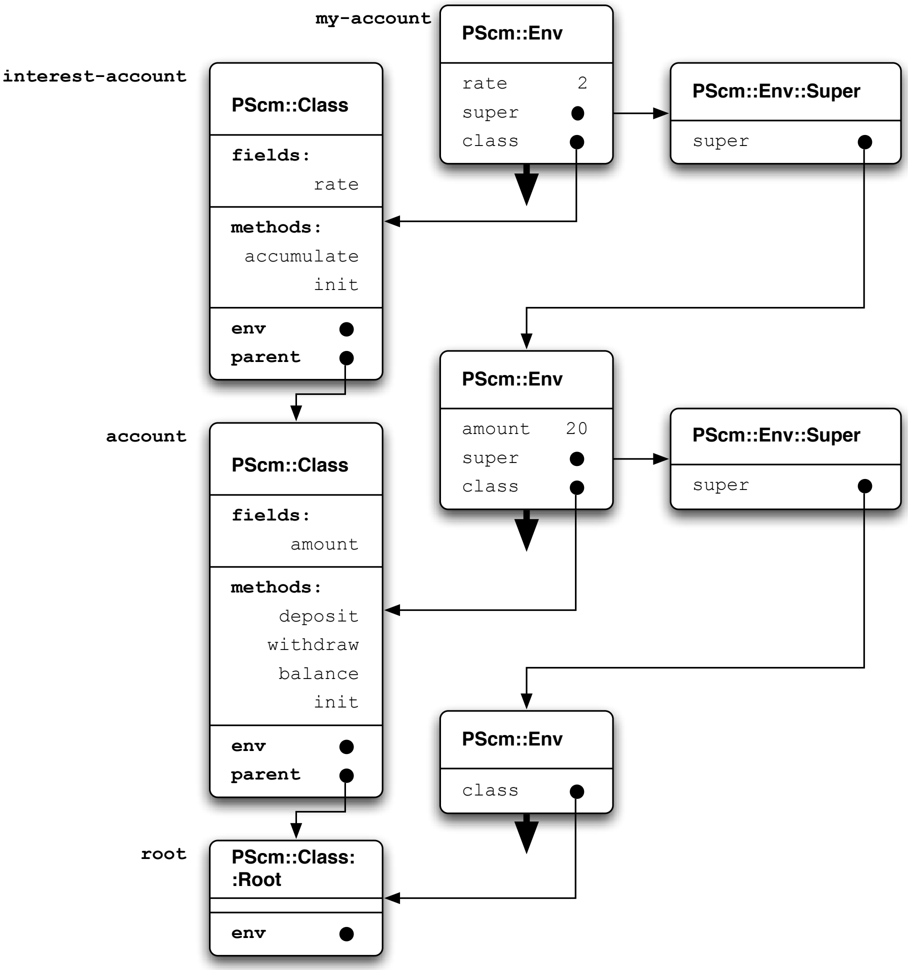
The `class` variable refers to a **PScm::Class** object which contains field (variable) names, method names along with their definitions, the environment that was current at the time of the creation of the **PScm::Class**, and a `parent` field pointing at the parent PScheme class.

12.4 Tests

Tests for our OO extension are in Listing 12.5.2 on page 155.

The first test exercizes the creation of a class. The second test creates a class (our `account` class from the examples above) then creates an object from it and calls a couple of its methods. The third test uses the `interest-account` example that we've looked at to test inheritance. The fourth test demonstrates that lexical variables outside of a class are visible to its methods and can therefore be used as class variables. Finally, the fifth test uses an abstract form of that "`leo`" example to demonstrate that method calls on a `super` object persist the current value of `this`.

Figure 12.3: Example classes and objects



12.5 Listings

12.5.1 PScm/Class.pm

```
package PScm::Class;

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

sub new {
    my ($class, $parent, $fields, $methods, $env) = @_;

    my $rh_methods = {};
    $class->_populate_methods_hash($rh_methods, $methods);

    return bless {
        parent  => $parent,
        fields  => [$fields->value],
        methods => $rh_methods,
        env     => $env,
    }, $class;
}

sub _populate_methods_hash {
    my ($class, $rh_methods, $methods) = @_;
    if ($methods->is_pair) {
        my $method = $methods->first;
        my ($name, $args, $body) = $method->value;
        $rh_methods->{ $name->value } =
                { args => $args, body => $body };
        $class->_populate_methods_hash($rh_methods, $methods->rest);
    }
}

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

    my $new_object = $self->make_instance();
    $new_object->call_method($new_object, "init", $form, $env);
    return $new_object;
}

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

    my $parent_instance = $self->{parent}->make_instance();

    return $self->{env}->ExtendUnevaluated(
        new PScm::Expr::List(
            PScm::Expr::Symbol->new("class"), # $self
            PScm::Expr::Symbol->new("super"), # $parent_instance
            @{ $self->{fields} },             # 0...
        ),
        new PScm::Expr::List(
            $self,                                                 # "class"
            PScm::Env::Super->new(super => $parent_instance),      # "super"
            ((PScm::Expr::Number->new(0)) x @{ $self->{fields} }), # field...
        )
    );
}

sub get_method {
    my ($self, $method_name, $object) = @_;

    if (exists $self->{methods}{$method_name}) {
        return PScm::Closure::Method->new(
            $self->{methods}{$method_name}{args},
            $self->{methods}{$method_name}{body}, $object);
    }
}

##########################
package PScm::Class::Root;

use base qw(PScm::Class);

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

    return bless {
        parent  => 0,
        fields  => [],
        methods => {},
        env     => $env,
    }, $class;
}

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

    return $self->{env}
      ->ExtendUnevaluated(
          new PScm::Expr::Symbol("class"),
          $self
      );
}

1;
```

12.5.2 t/PScm_OO.t

```
use strict;
use warnings;
use Test::More;
use lib ’t/lib’;
use PScm::Test tests => 6;

BEGIN { use_ok(’PScm’) }


eval_ok(<<EOF, "account", ’classes’);
(define account
    (make-class root (amount)
           (init (x) (set! amount x))
    ))
EOF

eval_ok(<<EOF, <<EOR, ’objects’);
(define account
    (make-class
           root
           (balance)
           (init (x) (set! balance x))
           (balance () balance)
           (withdraw (x) (set! balance (- balance x)))
    ))
(define myaccount (account 10))
(myaccount balance)
(myaccount withdraw 2)
(myaccount balance)
EOF
account
myaccount
10
8
8
EOR

eval_ok(<<EOF, <<EOR, ’inheritance’);
(define account
    (make-class
           root
           (balance)
           (init (x) (set! balance x))
           (balance () balance)
           (withdraw (x) (set! balance (- balance x)))
           (deposit (x) (set! balance (+ balance x)))
    ))
(define interest-account
    (make-class
           account
           (rate)
           (init (x r)
                (begin
                    (super init x)
                    (set! rate r)))
           (accumulate ()
                (this deposit (* (this balance) rate)))
    ))
(define myaccount (interest-account 10 2))
(myaccount balance)
(myaccount withdraw 2)
(myaccount balance)
(myaccount accumulate)
(myaccount balance)
EOF
account
interest-account
myaccount
10
8
8
24
24
EOR

eval_ok(<<EOF, <<EOR, ’class variables’);
(define counter-class
    (let ((count 0))
        (make-class
            root
            ()
            (init () (set! count (+ count 1)))
            (count () count)
        )))
(define o1 (counter-class))
(o1 count)
(let ((o2 (counter-class))
      (o3 (counter-class)))
     (o1 count))
EOF
counter-class
o1
1
3
EOR

eval_ok(<<EOF, <<EOR, ’super calls’);
(define c1
   (make-class
       root
       ()
       (ma () (this mb))
       (mb () 0)
   ))
(define c2
   (make-class
       c1
       ()
       (ma () (super ma))
       (mb () 1)
   ))
((c2) ma)
EOF
c1
c2
1
EOR

# vim: ft=perl
```

Full source code for this version of the interpreter is available at
`http://billhails.net/Book/releases/PScm-0.0.9.tgz`

Chapter 13

Continuations

What are continuations? Why should you want to know about them? The rest of this chapter is devoted to answering the first of those questions, but the second question deserves some sort of an answer early on, if only to encourage you to pursue the answer to the first.

I hope you can remember (I certainly do) that wonderful *eureka* moment when you first "got" recursion, and all its implications. Grasping the concept of continuations is an even more rewarding and dare I say transcendental experience, and well worth the effort.

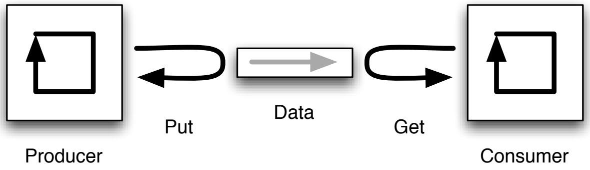
Continuations are an advanced control-flow technique that can be used to implement any and all standard control-flow mechanisms including but not limited to conditional branching, loops (with break statements), goto, return etc. Beyond the standard control-flow mechanisms, continuations also promise an almost limitless potential for new types of control flow that might be difficult or near impossible to achieve in any other way, for example

- co-routines;
- threads;
- exceptions;
- logic programming

Let's talk a bit about co-routines. Co-routines are groups of two or more functions or methods that interact with one another in a much more even-handed way than just "`A()` calls `B()`." A classic example is a producer-consumer pair of routines, which pass data, possibly via some intermediate structure such as a list. The producer produces data, pushing it on to the list, and the consumer consumes it, shifting it off the list again, something like Figure 13.1 on the next page.

Think of the producer using some complex algorithm to generate a stream of data while the consumer uses an equally complex algorithm to parse it. Both the producer and consumer are independant loops, so on the face of it, if the producer was called it would never relinquish control to give the consumer a look-in, it would just continue to push data onto that list. Likewise the consumer, if it were running, would just consume data until the list was exhausted. Both loops could have extensive internal logic and state such that even if the producer could simply call the consumer when it had produced something, the consumer would have great difficulty returning control to the producer without loosing all of its internal state. Reversing the roles, so that the consumer called the producer would still have exactly the same issues.

Figure 13.1: Producer/consumer pair



The only apparent solution would be to implement the producer and the consumer as separate threads, or even as completely separate processes and have some IPC mechanism to pass the data between them. But this pair of co-routines might only be part of a much larger system and the division into separate threads or processes might be inelegant or inappropriate. Continuations provide a solution by allowing a sort of "procedural goto" wherby control passes directly from the center of one routine to the heart of the other, and then back again, resuming exactly where the "goto" left off!

—— • ——

Threads are a common enough concept nowadays, but you might be surprised to hear that continuations make it almost trivial to implement so called "green" threads (application as opposed to operating system threads). We'll actually do this in a later chapter.

—— • ——

Exceptions are a simple application of continuations, where control, rather than unwinding down a stack, proceeds immediately to some handler routine. Perl's `eval{die}` construct is an example of this sort of thing. We'll demonstrate a very simple `error` construct towards the end of this chapter.

—— • ——

Logic Programming, as demonstrated by languages such as *Prolog* [3], is a completely different paradigm; it has more in common with recursive database search and large-scale pattern matching than the mostly functional style of programming presented by PScheme. However a more advanced application of continuations makes it possible to implement the basis of such a language, and we will attempt that later on too.

—— • ——

I hope that I have whet your appetite for the potential of continuations, however the topic of continuations is somewhat difficult, and this chapter is a long one.

Before diving in, it would be a good idea to discuss a couple of related topics, namely *tail recursion* and *tail call optimization*. Then with those under our belts, we can progress to continuations themselves. We will talk about continuations by discussing *continuation passing style*, a programming technique available to many languages, including Perl. Then we proceed to re-write the interpreter from Chapter 12 on page 135 in continuation passing style, and by exposing the underlying continuations in the PScheme language, we show what an incredibly powerful tool they are.

Much of the above may not make much sense on first reading, but hopefully the rest of the chapter will make it clear, so let's get started.

13.1 Tail Recursion and Tail Call Optimization

Consider the following definition of `factorial()` in Perl:

```
sub factorial {
    my ($n) = @_;
    if ($n == 0) {
        return 1;
    } else {
        return $n * factorial($n - 1);
    }
}

print factorial(5), "\n";  # prints 120
```

This is the classic recursive definition of the factorial function: the factorial of 0 is 1, and the factorial of any other positive number is that number times the factorial of one less than that number (factorial is not defined for negative numbers). You will be getting very familiar with this function in various guises from here on in so it is probably worth taking a good long look at it now in its simplest form before we start to change things.

To start off, consider the behaviour of this function when called with a positive numeric argument. The evolution on the stack of the call to `factorial(5)` would proceed as follows:

```
factorial(5)
5 * factorial(4)
5 * 4 * factorial(3)
5 * 4 * 3 * factorial(2)
5 * 4 * 3 * 2 * factorial(1)
5 * 4 * 3 * 2 * 1 * factorial(0)
5 * 4 * 3 * 2 * 1 * 1
5 * 4 * 3 * 2 * 1
5 * 4 * 3 * 2
5 * 4 * 6
5 * 24
120
```

Although this picture omits many details, it is obvious that the stack grows (to the right in the example) and that there are deferred multiplications that only get performed as the calls to `factorial()` return and the stack is unwound again.

We can rewrite that factorial function in a different form with the addition of a helper function, like this:

```
sub factorial {
    my ($n) = @_;
    return factorial_helper($n, 1);
}

sub factorial_helper {
    my ($n, $result) = @_;
    if ($n == 0) {
        return $result;
    } else {
        return factorial_helper($n - 1, $n * $result);
    }
}

print factorial(5), "\n";  # still prints 120
```

This version works by moving the body of the factorial function into the helper function and passing it an additional value, an *accumulator* with an initial value of `1`. This means that the helper function can calculate the result as it proceeds up the stack rather than having to wait for `$n` to reach zero and calculating the result on the way back down.

This is still a recursive definition, but because of the way the result is calculated it differs from the original `factorial()` in one absolutely crucial detail: the *last* thing it does is to call itself recursively and it *immediately* returns the result. In the original definition the result of the recursive call to `factorial()` had to be multiplied by the current value of `$n` before it could be returned.

A function which is called and its result immediately returned is said to be in *tail position* and the code making the call is said to be making a *tail call*. A recursive function which calls itself in tail position is said to be *tail recursive*. Tail calls are special because the stack setup to make the call and teardown afterwards is essentially redundant: the result of the function making the tail call is the result of the function being called, and the caller's stack frame is destroyed immediatly after the called function's stack frame is. If we could overwrite the caller's arguments with the arguments to the called function, then `goto` the called function, then when that function does a `return` it will return not to the caller, but to the caller of the caller. This is called *tail-call optimization* (TCO).

Figure 13.2 shows a normal procedure call in tail postion. You can see that the stack is extended by the called function's frame (which includes the return address), then that extension is discarded as the called function returns, then the calling function's frame is subsequently discarded as the calling function returns to the previous caller (we are talking about the Perl stack here, not PScheme environments).

Figure 13.2: Tail call without TCO

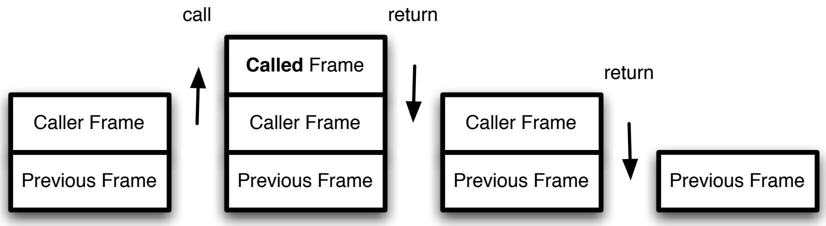
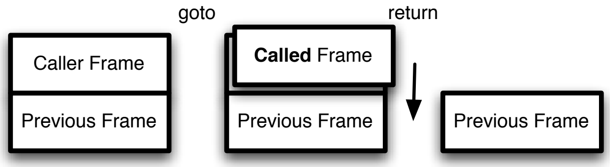


Figure 13.3 on the facing page shows the effect of tail call optimization. The caller's frame is replaced by the called function's frame, then the caller jumps to the called function. When the called function returns it does so directly to the previous caller.

Perl allows us to do precisely this, by means of assignment to `@_` and the special `goto &sub` syntax. Here's our `factorial_helper()` again, this time with TCO:

Figure 13.3: Tail call with TCO



```
sub factorial_helper {
    my ($n, $result) = @_;
    if ($n == 0) {
        return $result;
    } else {
        @_ = ($n - 1, $n * $result);
        goto \&factorial_helper;
    }
}
```

This function, although written in a recursive style, operates in a constant space and consumes no stack. In fact it is pretty much equivalent to this iterative definition:

```
sub factorial_helper {
    my ($n, $result) = @_;
  REPEAT:
    if ($n == 0) {
        return $result;
    } else {
        --$n; $result *= $n;
        goto REPEAT;
    }
}
```

which just emphasizes the point that TCO'd tail-calls are really just `goto`s with arguments.

Many language implementations (`gcc` springs to mind) can perform *implicit* TCO, detecting calls in tail position and replacing the call with a `goto`, and that's *all* calls in tail position, not just recursive ones. Furthermore, some languages such as Scheme *require* this behaviour of their implementations[1]. Our PScheme implementation, through the use of continuations, will by the end of this chapter support something equivalent.

That's really all there is to tail recursion and TCO[2]. I've already said they have a direct bearing on continuations, but there is a lot more to continuations than that, so lets take a look at using continuations in perl.

[1] R^{6}RS requires that tail calls consume no resources, not necessarily that they perform precisely the TCO demonstrated here. For now if you can just accept that implicit TCO or an equivalent is a desirable thing, all will become clear later.

[2] For an alternative exposition of TCO (sometimes called *Tail Call Elimination*) see [4, pp229–234]

13.2 Continuation Passing Style

This section discusses continuation passing style (CPS). It contains a number of exclamation marks; I hope you will agree that they are justified.

One way of thinking about a procedural computation is by decomposing it into just two main operations:

1. Calling a function with arguments;

2. Returning a result from a function.

Continuation passing style eliminates the second of these operations; in pure continuation passing style no function you call ever returns!

This page deliberately left blank to allow time for reflection on the enormity of the previous statement.

...That being the case, you need to figure out how to tell a CPS function what to do with its result. So, since a CPS function can't return a result, it is instead passed an additional procedure as argument: a *continuation*, and it passes its result to that.

The continuation represents the remainder of the computation after a function "returns". Since calling a continuation is equivalent to returning a result in non-CPS, you can also think of a continuation as a reference to your function's `return` statement.

As you might imagine, a computation which never returns will simply consume stack indefinately, until it completes, but I hope the discussion of TCO above has addressed some of your reservations on that score, and as I've said, continuations themselves, when fully realised, provide an alternative mechanism for dealing with the same issue.

——•——

So what does continuation passing style look like in Perl? Well since continuations are procedures, closures are an obvious and easy way to implement them. So our continuations can be created by `sub { ... }` and called by `$continuation->(...)`.

For a first example of CPS transformation, we'll go back to our original `factorial()` function from Section 13.1 on page 161 and re-write it in CPS. To save you having to refer back to it, here it is again.

```
sub factorial {
    my ($n) = @_;
    if ($n == 0) {
        return 1;
    } else {
        return $n * factorial($n - 1);
    }
}

print factorial(5), "\n";  # prints 120
```

Now as we have said, all CPS functions take an additional continuation argument. The continuation we pass it depends on what we want to do with the result. Our original example printed the result, so let's just pass a continuation to do that:

```
factorial(5, sub { print shift, "\n" });
```

The additional continuation argument `sub { print shift, "\n" }` just takes one argument and prints it.

Next up is `factorial()` itself. This CPS `factorial()` takes an additional continuation as argument, so the first couple of lines are easy:

```
sub factorial {
    my ($n, $cont) = @_;
```

Next remember that whenever the function used to return a result, it must now call its continuation on that result, so the next couple of lines are also pretty easy: wheras the original returned `1` if `$n` was `0`, the CPS version calls its continuation on the value `1` instead.

```
if ($n == 0) {
    $cont->(1);
```

This works for `factorial(0, sub { print shift, "\n" } )`: the continuation will get and print a `1`.

That leaves the tricky bit. The original function reads:

```
} else {
    return $n * factorial($n - 1);
}
```

You can see that recursive call to `factorial()` has some deferred computation, namely the multiplication by `$n` to be done when the call returns. But as we've said a CPS function never returns so we must somehow wrap that deferred computation up in a new continuation and pass it to `factorial()`.

If you get stuck on a difficult CPS transform, it almost always pays to break the expression into a sequence of simpler operations first. We can do that here easily enough:

```
} else {
    my $factorial_result = factorial($n - 1);
    my $result = $n * $factorial_result;
    return $result;
}
```

So this is much easier now. You can see that the first thing that happens is that `factorial()` calls itself. Then the result is multiplied by `$n`, and finally it is returned. So our new continuation is just the code that now follows the call to `factorial()`, wrapped in a function:

```
sub {
    my ($factorial_result) = @_;
    my $result = $n * $factorial_result;
    return $result;
}
```

Since `factorial()` will call this continuation with its result, `$factorial_result` is the argument to the continuation.

There is one additional change that we need to make. Where the original code did a `return $result`, our new continuation must call the original continuation on the `$result` instead.

```
sub {
    my ($factorial_result) = @_;
    my $result = $n * $factorial_result;
    $cont->($result);
}
```

This is our new continuation. All that remains is to pass it to our recursive factorial call:

```
} else {
    factorial($n - 1, sub {
            my ($factorial_result) = @_;
            my $result = $n * $factorial_result;
            $cont->($result);
        }
    );
}
```

We can now shorten this considerably by eliminating the temporary variables:

```
} else {
    factorial($n - 1, sub { $cont->($n * shift) });
}
```

The new continuation is `sub { $cont->($n * shift) }`. It takes one argument: the result so far (this is the value that our non-CPS `factorial()` would have returned). It multiplies the result by the current value of `$n` then calls the current continuation `$cont` on that value[3].

That completes our initial CPS re-implementation of `factorial()`:

```
sub factorial {
    my ($n, $cont) = @_;
    if ($n == 0) {
        $cont->(1);
    } else {
        factorial($n - 1, sub { $cont->($n * shift) });
    }
}

factorial(5, sub { print shift, "\n" });  # still prints 120
```

At the risk of labouring a point, consider the call:

```
factorial(3, sub { print shift, "\n" });
```

The evolution of the *continuation* will proceed as follows:

[3] Actually, if we were implementing in rabid, pure CPS, then even the primitives such as multiplication would take a continuation. We could simulate that here by re-writing our continuation as:

```
sub { times($n, shift, $cont) }
```

where `times()` is

```
sub times {
    my ($x, $y, $cont) = @_;
    $cont->($x * $y);
}
```

which at least emphasizes that the continuation is still being passed, though it is overkill for our purposes.

```
sub {
    print shift, "\n"
}

sub {
    sub {
        print shift, "\n"
    }->(3 * shift)
}

sub {
    sub {
        sub {
            print shift, "\n"
        }->(3 * shift)
    }->(2 * shift)
}

sub {
    sub {
        sub {
            sub {
                print shift, "\n"
            }->(3 * shift)
        }->(2 * shift)
    }->(1 * shift)
}

sub {
    sub {
        sub {
            sub {
                print shift, "\n"
            }->(3 * shift)
        }->(2 * shift)
    }->(1 * shift)
}->(1)  # factorial 0

sub {
    sub {
        sub {
            print shift, "\n"
        }->(3 * shift)
    }->(2 * shift)
}->(1)
```

```
sub {
    sub {
        print shift, "\n"
    }->(3 * shift)
}->(2)

sub {
    print shift, "\n"
}->(6)

print 6, "\n"
```

The deferred multiplications accumulate until we reach the point where the entire accumulated continuation is finally called with argument `1`, then they unwind from the outside in until the original continuation gets invoked on the argument `6` and `6` is printed. If you think about it, this evolution is functionally identical with the implicit deferred computations on the stack in our original `factorial()`, the only difference being that now we have a variable `$cont` that explicitly refers to the continuation.

Still sticking with our CPS `factorial()`, there is more that we can do. Because in CPS no function ever returns, all function calls must be in tail position![4]. As you can see our recursive call to `factorial()` is now in tail position, so we can use TCO to remove the spurious use of stack:

```
sub factorial {
    my ($n, $cont) = @_;
    if ($n == 0) {
        $cont->(1);
    } else {
        @_ = ($n - 1, sub { $cont->($n * shift) });
        goto \&factorial;
    }
}
```

This is still a "recursive" definition of `factorial()`, but now it is not the stack which is growing, but the continuation itself which consumes more and more space as our computation proceeds.

An astute reader will have realised that, in fact, we are still using stack when the continuations actually get triggered: those calls to `$cont->($n * shift)` will of course use just as much stack as the original did. However note that the continuations themselves must be called in tail position, so with a little more work we can eliminate that stack overhead too:

```
sub factorial {
    my ($n, $cont) = @_;
    if ($n == 0) {
        @_ = (1);
        goto $cont;
    } else {
        @_ = ($n - 1, sub { @_ = ($n * shift); goto $cont; });
```

[4] It's obvious really: Since no CPS function ever returns, any deferred computation must have been moved into the continuation, and a function call without deferred computation is by definition in tail position.

```
        goto \&factorial;
    }
}
```

This is very messy, but it works as advertised: it consumes absolutely no stack at any point; all deferred computations are in the continuations. Just bear in mind that in a language that provided implicit TCO, we wouldn't need any of those assignments to `@_` or the `goto`s, and I've promised that continuations themselves will allow an alternative and cleaner solution in our interpreter.

Moving on, what about that iterative/recursive definition of `factorial()` with a helper function from Section 13.1 on page 161? We can re-write that in CPS too. How does it compare? Well first here's a non-CPS variation on the original again, thoroughly TCO'd this time:

```
sub factorial {
    my ($n) = @_;
    @_ = ($n, 1);
    goto \&factorial_helper;
}

sub factorial_helper {
    my ($n, $result) = @_;
    if ($n == 0) {
        return ($result);
    } else {
        @_ = ($n - 1, $n * $result);
        goto \&factorial_helper;
    }
}

print factorial(5), "\n";  # prints 120
```

and here it is re-written in CPS:

```
sub factorial {
    my ($n, $cont) = @_;
    @_ = ($n, 1, $cont);
    goto \&factorial_helper;
}

sub factorial_helper {
    my ($n, $result, $cont) = @_;
    if ($n == 0) {
        @_ = ($result);
        goto $cont;
    } else {
        @_ = ($n - 1, $n * $result, $cont);
        goto \&factorial_helper;
    }
}

factorial(5, sub { print shift, "\n" });  # still prints 120
```

Our new tail recursive CPS `factorial()` function takes an additional continuation argument and passes that to `factorial_helper()`. `factorial_helper()` either goes to the continuation with the result, or goes to itself with new values for `$n` and `$result`; but since it has no deferred computation, it does not need to construct a new continuation and just passes the existing continuation to the recursive call.

The take home message here is that this tail recursive definition of `factorial()` using `factorial_helper()` translates into a CPS where neither the stack nor the continuation grows. This is a general result: functions written to be tail-recursive consume no stack when TCO'd, and do not build new continuations when rewritten into CPS.

——— • ———

The "normal" way, or at least the easiest way to produce CPS code is to do what we did above: take non-CPS code and translate it into CPS. In the next section we're going to look at a few examples of simple, hypothetical function forms and how they translate into CPS.

13.3 Example CPS Transformations

To keep these examples simple, we'll ignore any issues of TCO. These examples should allow us to proceed with more confidence into the subsequent rewrite of our interpreter.

- The simplest kind of function is one that takes no arguments and returns a constant:

  ```
  sub A {
      return 'hello';
  }
  ```

  The CPS form of this takes a continuation as argument and calls the continuation on the constant:

  ```
  sub A {
      my ($ret) = @_;

      $ret->('hello');
  }
  ```

  I've called the continuation `$ret` instead of `$cont` to emphasize it's equivalence with a `return` statement. One of the guiding principles of converting to CPS is that calling the argument continuation in CPS is equivalent to doing a return in non-CPS. In fact you can mentally substitute `return(...)` for `$ret->(...)` in many of these examples without disturbing the sense of them.

- The next simplest form is a function that takes arguments and performs only primitive operations such as addition on them, returning the result:

```
sub A {
    my ($x, $y) = @_;

    return $x + $y;
}
```

In this case, since primitive operations can't take a continuation, and because they are "terminal" operations that won't run away off up the stack, we again can just call the continuation on the result:

```
sub A {
    my ($x, $y, $ret) = @_;

    $ret->($x + $y);
}
```

- Next come simple functions that just call another function without any deferred computation:

  ```
  sub A {
      my ($x) = @_;

      return B($x);
  }
  ```

  Here, since there is no deferred computation, there need be no new continuation, we just pass the existing continuation to the called function:

  ```
  sub A {
      my ($x, $ret) = @_;

      B($x, $ret);
  }
  ```

  This is just saying to `B()` "return your result here."

- Now we're starting to get into areas where there *is* deferred computation, and this is where it starts to get just a little bit tricky:

  ```
  sub A {
      return C(B());
  }
  ```

  `B()` gets called first, and the value it returns is passed as argument to `C()`. In CPS `B()` would never return so we must also call it first, passing it a *new* continuation that calls `C()` with `B()`'s result and the current continuation:

```
sub A {
    my ($ret) = @_;

    B(
        sub {
            my ($B_result) = @_;
            C($B_result, $ret);
         }
    );
}
```

The new continuation calls `C()` with two arguments: the result of the call to `B()`, and the original continuation `$ret` to which `C()` should return its result. Since the original call to `C()` was returned as the result of the call to `A()`, `C()` is being told "return your result here".

- Sequential function calls present a slightly different problem.

```
sub A {
    B();
    C();
    D();
}
```

Here we must construct a nest of continuations to ensure that `C()` and `D()` get called in the correct order after `B()`:

```
sub A {
    my ($ret) = @_;
    B(
        sub {
            C(
                sub{
                    D($ret);
                }
            );
        }
    );
}
```

We call `B()` with a continuation that will call `C()` with a continuation that will call `D()` with the original continuation `$ret` since the result of the call to `D()` was the result of the original call to `A()`. Again this is just saying to `D()` "return your result here."

- Next let's look at a conditional expression:

```
sub A {
    my ($x) = @_;

    if (B($x)) {
        C($x);
    } else {
        D($x);
    }
}
```

The call to B() in the condition will never return, so we must pass it a continuation that tests its result and decides which branch to take accordingly, passing the the original continuation to the chosen branch:

```
sub A {
    my ($x, $ret) = @_;

    B($x,
        sub {
            my ($B_result) = @_;
            if ($B_result) {
                C($x, $ret);
            } else {
                D($x, $ret);
            }
        }
    );
}
```

Both the true and the false branch used to make a single call in tail position to C() or D(), so now we simply pass the original continuation unchanged as an additional argument to C() or D().

- Finally, for now, lets look at a looping function.

```
sub A {
    my $i = 0;
    while ($i < 10) {
        B();
        ++$i;
    }
}
```

This needs a bit more thought. It turns out to be easiest to do a preliminary rewrite of this example into a recursive form as follows:

```
sub A {
    A_h(0);
}

sub A_h {
    my ($i) = @_;
    if ($i < 10) {
        B();
        A_h($i + 1);
    }
}
```

Turning that into CPS then becomes just a re-application of examples we've seen before:

```
sub A {
    my ($ret) = @_;
    A_h(0, $ret);
}

sub A_h {
    my ($i, $ret) = @_;
    if ($i < 10) {
        B(
            sub { A_h($i + 1, $ret); }
        );
    }
}
```

> `A()` calls `A_h()` with the continuation unchanged (`A_h()`, return your result here.) Since `B()` will not return, it is passed a continuation that carries on the recursion on `A_h()`, passing the original continuation `$ret`.

The examples above give a taste of the sorts of transformations that we shall be applying to our interpreter soon. There are other more difficult cases that might appear impossible at first sight (uses of `map` for example,) but again they can be resolved by first re-writing the expressions in a more tractable form before converting to CPS. We'll see examples of this sort of thing when we get to them.

It happens that there does exist a formal methodology for transforming statements in any language capable of supporting CPS into CPS. The above example transformations are samples from that ruleset. All such transformations can be automated. When I started this chapter I was hopeful that perhaps something in the **B** package, the Perl compiler, would be available that could perform the transform but that appears not to be the case. Anyhow we'll learn a lot more about CPS by performing the transform manually, so that is the best approach to take.

13.4 The Trampoline

I promised that there was an alternative to all the messy assignments to `@_` and the `goto`s that constitute TCO. Well that falls out of three closely related properties of a fully realised CPS:

1. No function call ever returns, therefore:
2. Every function call must be in tail position, and therefore:
3. If you *were* to return something it would be guaranteed to return all the way down the stack to the originating caller[5].

Now just suppose that at well chosen points we *do* return something, and not just anything. Suppose we return another continuation, this time taking no arguments, that when called just continues the calculation from where it left off!

That is one of the surprising things about continuations, that they are *completely* self-contained and require no external context to operate. You may need to convince yourself that this will work: Since we can TCO a CPS function, such that it uses no Perl stack at all, then even if the CPS code is not TCO'd there can be nothing on the Perl stack that it actually needs, just a long chain of return adresses that it pases through after the computation is finished. Returning a continuation like this merely interleaves this otherwise laborious chain of returns with the normal flow of control up the stack.

So how do we deal with this returned continuation? A handler routine, called a *trampoline*, starts off by being called with a continuation of no arguments. It loops, repeatedly calling the continuation and assigning the result (another continuation of no arguments) back to the continuation itself until the result is `undef`. The code is easier to write than to describe:

```
sub trampoline {
    my ($cont) = @_;
    $cont = $cont->() while defined $cont;
}
```

To give you a feel of how this might work, let's return once more to our CPS `factorial()` function and re-write it to make use of a trampoline instead of TCO. First to refresh your memory here's our first CPS attempt again (slightly modified) before we TCO'd it:

```
sub factorial {
    my ($n, $ret) = @_;
    if ($n == 0) {
        $ret->(1);
    } else {
        factorial($n - 1,
                  sub {
                      my ($a) = @_;
                      $ret->($n * $a)
                  });
    }
}

factorial(5, sub { print shift, "\n"; });  # still prints 120
```

and here it is rewritten to use a trampoline.

[5] Actually, we're also relying on the fact that perl implicitly returns the value of a tail call as the value of a function.

```
sub factorial {
    my ($n, $ret) = @_;
    if ($n == 0) {
        return sub { $ret->(1); };
    } else {
        return sub {
            factorial($n - 1,
                      sub {
                          my ($a) = @_;
                          return sub { $ret->($n * $a) }
                      });
        };
    }
}

sub trampoline {
    my ($cont) = @_;
    $cont = $cont->() while defined $cont;
}

trampoline(
    sub {
        factorial(5, sub { print shift, "\n"; return undef; });
    }
); # still prints 120
```

Changes from the original are in bold as usual. The key to understanding this is to notice that whenever a function call was done in the original, either to `factorial()` or to the continuation, a closure which will make that call is returned to the trampoline instead. Each time this happens the stack is completely cleared down and the trampoline resumes the computation by calling the returned closure. Finally, at the end of the computation, the original continuation passed to `factorial()` gets invoked, printing the result and returning `undef` to the trampoline causing it to stop.

Like TCO, the trampoline technique is not specific to CPS, but both techniques require that the modified calls be in tail position, making CPS a prime candidate for either kind of optimisation[6].

Well that's pretty scary stuff. Both TCO and the trampoline are simply alternative strategies to avoid unlimited use of the stack, and you may be wondering if the trampoline has any advantages over TCO at this point. I'd like to make a few arguments in favour of the trampoline here.

1. Our `factorial()` example is a very tight piece of code which somewhat overemphasizes the role of the trampoline by doing a lot with it in a small space. Particularily the explicit return of a closure to make the recursive call does not have to be done for *every* tail call, we just need to ensure it

[6] The trampoline further requires that all intermediate calls also be in tail position, and that the values of all tail calls are returned, so that a returned value would be guaranteed to reach the trampoline. Fortunately, CPS satisfies the first of these requirements, and Perl satisfies the second.

happens fairly regularly on our way up the stack. For example in a set of mutually tail recursive subroutines, `A()` calling `B()` calling `C()` calling `A()`..., only one of those subroutines need do that return. This is in contrast to TCO, where any unoptimised tail call constitutes a permanently unclaimed stack frame.

2. Some languages do not allow the possibility of doing TCO, so any CPS implementation using such a language would have to use a trampoline.

3. We can hide the trampoline from client CPS code by representing continuations as objects which contain the closures, and putting the **`return`** to the trampoline in the method that invokes the closure (provided that method is invoked in tail position).

It is the third argument that swings the case, and that's exactly what we'll be doing. If you don't get that argument yet, hold on and it will be made clear later.

13.5 Using CPS

Thinking back to our original exposition of CPS from Section 13.2 on page 164 where we suggested that normal procedural programming consisted primarily of calling functions and returning values, we said that CPS eliminates the second of these two operations. In fact that was a slight oversimplification. There is a certain amount of equivalence between the operations of "call" and "return", it is just the *direction* of the flow of data that differs, "upwards" to the called function via its arguments, versus "downwards" from the called function via its return value. Continuation passing style in fact *unifies* these two operations, returning a value is the *same* as calling a function. Since in CPS data flows in only one direction, in some sense CPS is actually a simplification!

Furthermore, an application written in CPS with complete TCO needs no stack at all: TCO allows us to eliminate the use of stack by tail calls, and in CPS all function calls are tail calls[7].

So now you understand continuations, but how do you use them? Well at each step of a computation you have a continuation representing the current function's **`return`** statement. But a continuation is a variable, a reference to a subroutine, and *you can do whatever you like with it!*. You don't *have* to call it (return through it) just when everyone is expecting you to, you might call (return through) a completely different continuation instead, or you might pass it to another function that can call it (return through it) if it likes. And when a continuation is called (returned through), control flow transfers to wherever the **`return`** statement equivalent to that continuation would have returned! Put another way, you have always had control over what value your function returns, and when it returns it, but not until now have you had control over where it returns it to!

And there's more. Although code written in CPS retains the notion of a stack since functions call functions and return values (via continuations); as we've already noted the stack is not really relevant, or even necessarily present. Any continuation is as valid as any other. It is perfectly permissable to call a continuation that resumes control in a function that has already returned, in effect jumping *across* the call graph that a stack based language is constrained by!

Let's give an explicit example to illustrate this last point. Consider the following simple perl script:

```
sub A {
    print "in A\n";
    B();
```

[7] This is what is meant by "Stackless Python": an implementation of that language in CPS with complete TCO.

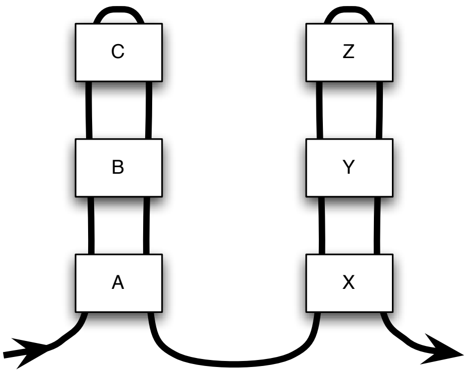
```
    print "back in A\n";
}

sub B {
    print "    in B\n";
    C();
    print "    back in B\n";
}

sub C {
    print "        in C\n";
}

sub X {
    print "in X\n";
    Y();
    print "back in X\n";
}

sub Y {
    print "    in Y\n";
    Z();
    print "    back in Y\n";
}

sub Z {
    print "        in Z\n";
}

A();
X();
```

`A()` calls `B()` which calls `C()`, and `X()` calls `Y()` which calls `Z()`. The top level calls `A()` then `X()`. You shouldn't take too long to convince yourself that it will produce the following output:

```
in A
    in B
        in C
    back in B
back in A
in X
    in Y
        in Z
    back in Y
back in X
```

Just to hammer home the simple point, Figure 13.4 on the facing page shows the thread of control flow passing through `A()`, `B()`, `C()`, `X()`, `Y()` and `Z()`.

Figure 13.4: Control flow for the simple script



Now let's re-write that program into CPS, without changing any of it's behaviour:

```
sub A {
    my ($ret) = @_;
    print "in A\n";
    B(sub { $ret->(print "back in A\n") });
}

sub B {
    my ($ret) = @_;
    print "    in B\n";
    C(sub { $ret->(print "    back in B\n") });
}

sub C {
    my ($ret) = @_;
    $ret->(print "        in C\n");
}

sub X {
    my ($ret) = @_;
    print "in X\n";
    Y(sub { $ret->(print "back in X\n") });
}

sub Y {
    my ($ret) = @_;
    print "    in Y\n";
    Z(sub { $ret->(print "    back in Y\n") });
}
```

```
sub Z {
    my ($ret) = @_;
    $ret->(print "        in Z\n");
}

A(sub { X(sub {})});
```

There are no new tricks that haven't already been described in Section 13.3 on page 172 above, the only difference is that since none of the original functions actually returned anything interesting (they returned the results of print statements), the equivalent continuations don't bother looking at their arguments.

This produces identical output to the original program, and exhibits exactly the same control flow. Now let's make just three tiny changes.

```
my $C_ret;

sub A {
    my ($ret) = @_;
    print "in A\n";
    B(sub { $ret->(print "back in A\n") });
}

sub B {
    my ($ret) = @_;
    print "    in B\n";
    C(sub { $ret->(print "    back in B\n") });
}

sub C {
    my ($ret) = @_;
    $C_ret = $ret;
    $ret->(print "        in C\n");
}

sub X {
    my ($ret) = @_;
    print "in X\n";
    Y(sub { $ret->(print "back in X\n") });
}

sub Y {
    my ($ret) = @_;
    print "    in Y\n";
    Z( sub { $ret->(print "    back in Y\n") });
}

sub Z {
    my ($ret) = @_;
    $C_ret->(print "        in Z\n");
}

A(sub { X(sub {})});
```

The first change is to declare a `$C_ret` variable to hold a continuation. Then `C()`, before it calls its continuation, stores it in this `$C_ret` variable. Finally `Z()`, instead of calling its own continuation `$ret`, calls the saved continuation `$C_ret` instead.

This produces the output below. Whether or not you find this surprising will depend on how closely you've been following the discussion:

```
in A
    in B
        in C
    back in B
back in A
in X
    in Y
        in Z
    back in B
back in A
in X
    in Y
        in Z
    back in B
back in A
in X
    in Y
        in Z
    back in B
back in A
...
```

All proceeds normally until we reach the first call to `Z()`. Since `Z()` calls the continuation that `C()` saved, `Z()` instead of returning to `X()`, returns to `B()` instead. Then normal service is resumed, starting from the return to `B()`, until the next return from `Z()`, which again returns to `B()` and so on, *ad infinitum.* what we have achieved is the control flow shown in Figure 13.5 on the following page.
(Cue the Mony Python music.)

If this still isn't clear, which I suspect may be the case, look at Figure 13.6 on the next page. In this figure I've "broken apart" the functions from their continuations. `A()` calls `B()` calls `C()` which calls the continuation of `B()` (e.g. `cB()`) which calls the continuation of `A()` etc. Now the continuation of `B()` is just "return to `A()`" (call `cA()`) and the continuation of `A()` is to call `X()` etc.
I'm deliberately down-playing the idea of "return" now, this really is just function calls, and in that case Figure 13.7 on the following page shows that there is really nothing special about `Z` calling `cB`, it's just a recursive loop, and TCO or a trampoline will take care of the stack for us.
This is what I meant by saying that CPS is a simplification. It *linearizes* control flow, so that it is just a straight line of function calls. Once you get that idea, a whole world of possibilities opens up. For

Figure 13.5: Control flow with continuations

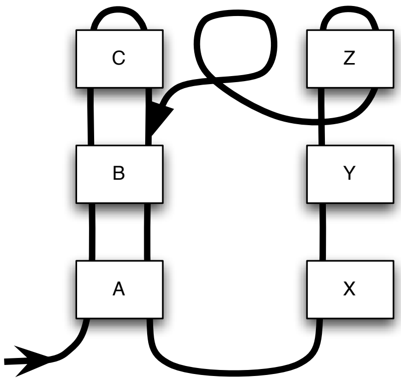


Figure 13.6: Continuations are just (anonymous) subroutines

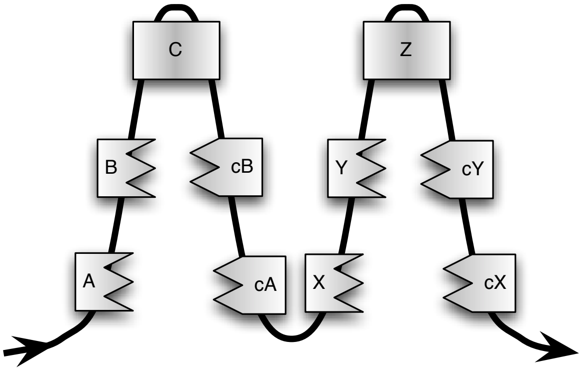
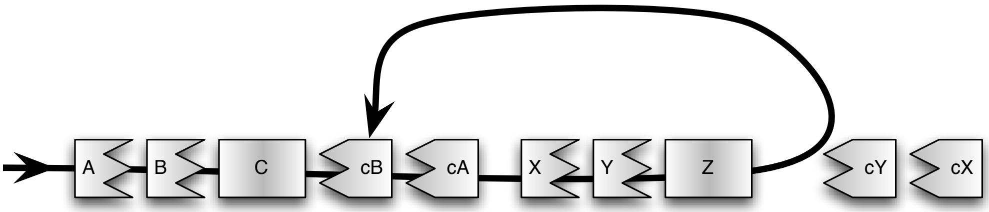


Figure 13.7: Continuations *really are* just subroutines



instance you can probably imagine at this stage that with a little more work, adding loops and passing continuations around, we could easily arrive at a coroutine implementation, where control does jump from the heart of one loop to the heart of another and back again without disturbing the state of either loop.

There is a big downside to writing in CPS however, and that is that it makes your head hurt. A far better approach is to use a language that has continuations built in “under the hood”. Then when you write “`return $val`” you are really calling a continuation on `$val`, but you don’t have to worry about it, and when you need to get hold of a continuation, you can ask for one. A language like that provides continuations as *first class objects* in that they can be passed around as variables, much in the same way as Perl provides anonymous subroutines (closures) as first class objects.

For example, if Perl had built-in continuations, and we could get at the current continuation by i.e. taking a reference to the `return` keyword[8], then we could rewrite all of this example without CPS, as follows:

```
my $cont;

sub A {
    print "in A\n";
    B();
    print "back in A\n";
}

sub B {
    print "    in B\n";
    C();
    print "    back in B\n";
}

sub C {
    $cont = \return;
    print "        in C\n";
}

sub X {
    print "in X\n";
    Y();
    print "back in X\n";
}

sub Y {
    print "    in Y\n";
    Z();
    print "    back in Y\n";
}

sub Z {
    print "        in Z\n";
    $cont->()
}
```

[8] Thanks to Tom Christiansen for this idea.

```
A();
X();
```

Bold text shows the differences from the original non-CPS version.

We are going to turn our PScheme interpreter into just such a language. The next few sections will describe the changes we need to make.

13.6 Implementation

Rather than attempting to rewrite the interpreter of Chapter 12 on page 135 from start to finish in CPS, We're going to backtrack to our first "interesting" interpreter, from Chapter 5 on page 59 which has only `let` and `lambda`, and re-implement that. This has the advantage that we get a real working interpreter with continuations which we can test early on, and we can demonstrate some of the power of continuations with it. Then I'll gloss the re-writing of the final interpreter in stages by working through the intermediate versions pausing only to study any previously unencountered constructs that require novel treatment. Finally we'll have a continuation-passing version of the interpreter from Chapter 12 on page 135 to play with.

13.6.1 Our Trampoline Implementation

Our implementation of the trampoline does not differ significantly from the example that we presented for `factorial()` above. But it is still best introduced gradually, so this section is still pseudocode, to a certain extent.

Firstly we need to rewrite the read-eval-print loop into CPS, so that we can call the whole thing from the trampoline. This isn't actually very difficult to do, the repl for version 0.0.2 conceptually is as simple as

```
sub repl {
    my ($reader, $outfh) = @_;
    while (my $expr = $reader->read()) {
        my $result = $expr->Eval(new env);
        $result->Print($outfh);
    }
}
```

We've already seen in Section 13.3 on page 172 that the easiest way to transform a `while` loop into CPS is first to rewrite it into a recursive form, and this is easy to do here:

```
sub repl {
    my ($reader, $outfh) = @_;
    if (my $expr = $reader->read()) {
        my $result = $expr->Eval(new env);
        $result->Print($outfh);
        repl($reader, $outfh);
    }
}
```

Now to recast that into CPS is fairly trivial, especially if we remember that the reader `PScm::Read::Read()` already returns `undef` on EOF, and it can continue to do so, telling the trampoline to stop, and only calling its continuation if there is something to evaluate.

```
sub repl {
    my ($reader, $outfh, $ret) = @_;
    $reader->read(
        sub {
            my ($expr) = @_;
            $expr->Eval(
                new env,
                sub {
                    my ($result) = @_;
                    $result->Print(
                        $outfh,
                        sub { repl($reader, $outfh, $ret) }
                    )
                }
            )
        }
    )
}
```

So apart from the return of `undef` by the reader, where would we put these `return` statements that return a continuation to the trampoline? Well as I've said we could place them throughout the code, but there's a better idea.

Instead of continuations being simple anonymous subroutines, we make them into objects that *contain* those anonymous subroutines, with a `Cont()` method to invoke the underlying closure. Then instead of just writing:

```
$ret->($arg);
```

to invoke a continuation, we say:

```
$ret->Cont($arg);
```

Then, in that `Cont()` method, instead of just saying

```
sub Cont {
    my ($self, $arg) = @_;
    $self->{cont}->($arg);
}
```

we instead write

```
sub Cont {
    my ($self, $arg) = @_;
    return sub { $self->{cont}->($arg) };
}
```

we have both effected the return of a continuation to the trampoline, and completely hidden the fact from the client code[9]!

In reality there are a few minor complications with this approach, but the above discussion is very close to our final implementation.

13.6.2 CPS **let and lambda**

In this section we re-implement the interpreter version 0.0.2 from Chapter 5 on page 59 in continuation passing style. Once that is done, we introduce a new construct, `call/cc`, which allows the language direct access to continuations.

First of all we need a new **PScm::Continuation** class. I don't want to show you all of that class just yet, but here's its `new()` method:

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

It takes an anonymous subroutine as argument and stores it in a `cont` field. We don't want to be writing `new PScm::Continuation( sub {...} )` all over the place, so we sweeten things with a little syntactic sugar:

```
sub cont(&) {
    my ($cont) = @_;
    return __PACKAGE__->new($cont);
}
```

This is put on **PScm::Continuation**'s `@EXPORT` list[10] so after importing it with "`use PScm::Continuation;`", instead of writing `new PScm::Continuation( sub {...} )` we just write `cont {...}` instead. If you're not familiar with this technique, see [13, pp225–231].

As discussed above, we add a `trampoline()` subroutine to **PScm** which repeatedly invokes the continuation returned from its previous invocation, until the invocation returns `undef`, signaling the end of the computation. Here's `trampoline()`:

```
sub trampoline {
    my ($cont) = @_;
    $cont = $cont->Bounce() while defined $cont;
}
```

It's functionally equivalent to the prototype `trampoline()` subroutine discussed above. The `Bounce()` method is defined in **PScm::Continuation** to immediately invoke the continuation with no arguments:

[9] Assuming of course that the `Cont()` method is invoked in tail position, but we've been here before.

[10] In general it is always considered better form to use `@EXPORT_OK` rather than `@EXPORT`. However it is justifiable here firstly because **PScm::Continuation** is part of PScheme, not a standalone library module, and secondly the only reason another class would `use PScm::Continuation` would be to gain access to the `cont` construct.

```
039 sub Bounce {
040     my ($self) = @_;
041     $self->{cont}->();
042 }
```

Only `trampoline()` calls `Bounce()`.

Now we need to look at the Read-Eval-Print loop from **PScm**, with the trampoline in place.

```
032 sub ReadEvalPrint {
033     my ($infh, $outfh) = @_;
034
035     $outfh ||= new FileHandle(">-");
036     my $reader = new PScm::Read($infh);
037     trampoline(cont { repl($reader, $outfh) });
038 }
```

It's the same as we've seen before up to Line 37 where instead of entering it's loop, it invokes the trampoline with a continuation. That continuation invokes a new helper routine `repl()` with the reader and the current output handle as arguments. Here's `repl()`.

```
040 sub repl {
041     my ($reader, $outfh) = @_;
042     $reader->Read(
043         cont {
044             my ($expr) = @_;
045             $expr->Eval(
046                 new PScm::Env(
047                     let    => new PScm::SpecialForm::Let(),
048                     '*'    => new PScm::Primitive::Multiply(),
049                     '-'    => new PScm::Primitive::Subtract(),
050                     if     => new PScm::SpecialForm::If(),
051                     lambda => new PScm::SpecialForm::Lambda(),
052                     'call/cc' =>
053                       new PScm::SpecialForm::CallCC(),
054                 ),
055                 cont {
056                     my ($result) = @_;
057                     $result->Print(
058                         $outfh,
059                         cont {
060                             repl($reader, $outfh);
061                         }
062                     )
063                 }
064             )
065         }
066     )
067 }
```

So the guts of the old `ReadEvalPrint()` have been moved to `repl()`. It's just an expansion of the CPS pseudocode for `repl()` in the previous section, and not nearly as bad as it might first appear, it's really just `Read()` calling `Eval()` calling `Print()` calling `repl()`, all through passed continuations.

There is also something new added to the environment. we'll see what that new binding `call/cc` on Line 53 is about later.

So the `Read()`, `Eval()` and `Print()` methods now all take continuations and must be modified accordingly. Thankfully the modifications to `Read()` and `Print()` are trivial.

First we need to look at the CPS `Print()` method.

```
075 sub Print {
076     my ($expr, $outfh, $cont) = @_;
077     print $outfh $expr->as_string, "\n";
078     $cont->Cont($expr);
079 }
```

It just does what it used to do, then calls its continuation with an arbitrary argument. That is the continuation that will restart the repl and it doesn't actually expect an argument, but `Cont()` does so we're just playing nice.

Notice that on Line 77 we *don't* pass a continuation to the `as_string()` method. This is just a normal non-CPS method call. The reasoning behind that is that although `as_string()` is potentially recursive, at no point will it cause evaluation of any PScheme expressions. Since we are only interested in continuations that might be exposed to user code, we can classify any method call that cannot result in a call to `Eval()` as a *simple expression* and deal with it as an atomic operation. Contrarywise, calls to `Eval()` or calls to methods that might result in a call to `Eval()` are classified as *significant expressions*, and must be rewritten into CPS. This distinction makes our rewrite much simpler[11].

As described above, that `Cont()` method actually *returns* a continuation of zero arguments which the trampoline will execute (by calling `Bounce()` on it). This is the trick I was enthusing about earlier: to return a continuation to the trampoline that will call the current continuation, rather than just directly calling the current continuation. The `return` will fall all the way back to the trampoline, effecting a complete cleardown of whatever stack might have accumulated up to this point, then the trampoline will kick things off again:

```
sub Cont {
    my ($self, $arg) = @_;
    return cont { $self->{cont}->($arg) };
}
```

The really neat thing about this is that the code that is written to use this method neither knows nor cares that the continuation is not simply being invoked directly at this point. The presence of the trampoline is completely invisible to the client CPS code.

Let's take a look at `Read()` in **PScm::Read** next. In fact what we have done is to rename `Read()` to `_read()`, leaving it otherwise unchanged:

[11] The reasoning is that each call to `Eval()` within the interpreter corresponds to a value being calculated and, most importantly, returned in the PScheme language. These points of return are exactly the points that require continuations to be used instead. If however in the later CPS rewrite of the object system from Chapter 12 on page 135 we wanted to allow PScheme objects to supply some sort of `to-string` method, and have that called in preference to the underlying Perl `as_string()` method, then we would have to rewrite `as_string()` into CPS.

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

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

    return $token unless $token->is_open_token;

    my @res = ();

    while (1) {
        $token = $self->_read;
        die "unexpected EOF"
          if !defined $token;
        last if $token->is_close_token;
        push @res, $token;
    }

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

Then we provide a new `Read()` that handles the continuation.

```
sub Read {
    my ($self, $cont) = @_;
    my $res = $self->_read();
    return undef unless defined $res;
    $cont->Cont($res);
}
```

`Read()` collects the result of the call to `_read()`, and if it is `undef` signifying EOF it returns `undef` to the trampoline telling it to stop. Otherwise it calls its continuation on the result.

Next we need to take a look at `Eval()`.

All `Eval()` methods now also take an additional continuation as argument. All the `Eval()` methods are in subclasses of **PScm::Expr**. Let's start by looking at the simplest of those expressions: literals and symbols.

The old `Eval()` method in **PScm::Expr** just returned `$self` (numbers, strings and anything else by default evaluate to themselves). The new version is little different, it calls its argument continuation on itself:

```
sub Eval {
    my ($self, $env, $cont) = @_;
    $cont->Cont($self);
}
```

Now for `PScm::Expr::Symbol::Eval()`. The old `Eval()` method in **PScm::Expr::Symbol** returned `$env->LookUp($self)`. Our CPS version calls its continuation on that result instead, because we can treat the call to `LookUp()` as a simple expression:

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

Evaluation of lists is a little more tricky, so to refresh our memories here's the original `PScm::Expr::List::Eval()` before CPS transformation:

```
062 sub Eval {
063     my ($self, $env) = @_;
064     my $op = $self->first()->Eval($env);
065     return $op->Apply($self->rest, $env);
066 }
```

On Line 64 It evaluates the first component of the list to get the operator `$op`, then on Line 65 it applies the operation `$op` to the rest of the unevaluated list.

Here's the CPS form:

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

There's rather a lot going on here, so best we approach it in two stages.

Firstly the `return cont { ... }` block wraps the entire method body in a continuation that we return to the trampoline. Apart from `PScm::Continuation::Cont()` this is the only other place where we explicitly fall back down to the trampoline. This is because all recursive calls to `Eval()` and `Apply()` must pass through this single function, and so we can stop all runaway stack consumption by `Eval()` and `Apply()` here[12]. You can just ignore the `return cont` wrapper and consider only the body of the

[12] Remember our CPS `factorial()` example before TCO, where the stack built up during calls to `factorial()` and built up further when the continuations actually triggered on `factorial(0)`. Well the `return` in the `PScm::Continuation::Cont()` method takes care of the second of these contingencies, and returning a continuation here takes care of the first. If you don't believe me, wait until we have rewritten the complete interpreter in CPS then enter the following definition in `t/interactive`:

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

Then try

continuation as if it were the body of the function. It would still work, but might run out of stack in the long run.

Secondly inside the method proper we assume that the first call to `Eval()`, in order to to get the `$op`, will not return, so we pass it a continuation which accepts the result `$op`, and applies it to the rest of the list, passing in the original continuation (Line 70). We must pass the original continuation `$cont` to `Apply()`, rather than just calling the continuation on the result of the `Apply()`, because the `Apply()` might make calls to `Eval()` to evaluate arguments to the `$op`, among other things, and must therefore be rewritten in CPS.

So that's it for the rewrite of all of the `Eval()` methods in **PScm::Expr**. Now we need to follow the chain of continuation passing into the various `Apply()` methods we have. Since this is an early version of the interpreter, there aren't too many, in fact they are in:

- **PScm::Primitive**;
- **PScm::SpecialForm::Let**;
- **PScm::SpecialForm::If**;
- **PScm::SpecialForm::Lambda** and
- **PScm::Closure::Function**.

Starting with `PScm::Primitive::Apply()`, you'll remember that all primitive operations share a common `Apply()` method. Now individual primitives do not have to accept continuations because they are terminal operations, so all that we have to do is to call the continuation that was passed to the shared primitive `Apply()` on the result of applying the individual primitive to its arguments.

Unfortunately this is complicated by the fact that the primitive `Apply()` must first evaluate its arguments. The original primitive `Apply()` did it with `map`:

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

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

This is a little tricky to rewrite in CPS, so we're going to attack it in stages. Stage one will be to write a recursive version of the builtin `map`, which instead of taking a sub and list, takes a listref and an environment, and for each element of the listref, calls that element's `Eval()` method with the environment as argument, accumulating the result in a new listref. But wait a minute, don't we already have such

```
> (factorial 170)
7257415615307998967396728211129263114716991681296451376543577798900561
8434017061578523507492426174595114909912378385207766660225654427530253
2890077320751090240043028005829560396661259965825710439855829425756896
6313439612262571094946806711205568880457193340212661452800000000000000
0000000000000000000000000000
```

Perl would have complained long before it hit that many levels of recursion.

a recursive map_eval() method? Yes, we wrote just such a method when we implemented true list processing for version 0.0.5 back in Section 8.5.1 on page 92.

Here's that method again.

```
sub map_eval {
    my ($self, $env) = @_;
    return $self->Cons($self->[FIRST]->Eval($env),
                       $self->[REST]->map_eval($env));
}
```

Now remember that that is code from 0.0.5, and here we're just rewriting version 0.0.2, so we don't have true list processing yet, lots of our methods are still expecting Perl array references, we don't have a Cons() method, and we don't have any **PScm::Expr::List::Null** class. Nonetheless we can cast this method back in to 0.0.2 terms quite easily.

This 0.0.2 map_eval() method is not yet in CPS form:

```
sub map_eval {
    my ($self, $env) = @_;
    if (@$self) {
        return [ $self->first->Eval($env),
                 @{ $self->rest->map_eval($env) } ];
    } else {
        return [];
    }
}
```

This is pretty straightforward.

The second stage of our attack is to re-write map_eval() in CPS. It will take an additional continuation argument, then, much as our factorial() example did, if the recursion has reached its limit (the argument list is empty) it calls its continuation on the empty list. Otherwise it has not finished, and it evaluates its first component, passing a continuation that arranges to evaluate the rest of the list by recursing:

```
sub map_eval {
    my ($self, $env, $cont) = @_;

    if (@$self) {
        $self->first->Eval(
            $env,
            cont {
                my ($evaluated_first) = @_;
                $self->rest->map_eval(
                    $env,
                    cont {
                        my ($evaluated_rest) = @_;
                        $cont->Cont([$evaluated_first,
                                     @$evaluated_rest]);
                    }
                );
            }
        );
    } else {
        $cont->Cont([]);
    }
}
```

This is the trickiest piece of code in the entire CPS re-write. Fortunately having done it, it is useful in a number of other scenarios. Now that we have `map_eval()` we can use it to re-write `PScm::SpecialForm::Primitive::Apply()`:

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

    $form->map_eval(
        $env,
        cont {
            my ($ra_evaluated_args) = @_;
            $cont->Cont($self->_apply(@$ra_evaluated_args));
        }
    );
}
```

Not too bad. The `map_eval()` is passed a continuation that applies the primitive operation to the evaluated arguments and calls the original argument continuation on the result.

It is worth noting again that there was no need to pass any continuation to the individual private `_apply()` methods for each primitive, so **PScm::Primitive::Multiply** etc. are unchanged.

The rest of the CPS transformations are much simpler, on the whole, and others that require the rewriting of `map` can make use of `map_eval()`.

Next up is **PScm::SpecialForm::Let**, here's the changes:

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

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

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

    $env->Extend(
        \@symbols,
027         \@values,
028         cont {
029             my ($newenv) = @_;
030             $body->Eval($newenv, $cont);
031         }
032     );
033 }
```

The changes are in bold on Lines 14 and 27–31

If you remember, the old version at the end simply said

```
return $body->Eval($env->Extend(\@symbols, \@values));
```

Since we know that the call to `$env->Extend()` will not return (those `@values` are still to be evaluated), we instead have to pass a continuation that will accept the resulting extended environment and evaluate the body in it. We have already dealt with all the `Eval()` methods (They're all in **PScm::Expr**) and they all take a continuation, so we pass the original continuation argument, since the `Eval()` is the expression that this `Apply()` was previously returning.

Remembering to add `PScm::Env::Extend()` to our list of methods that will need looking at, we proceed to `PScm::SpecialForm::If::Apply()`. We've already discussed how to transform a conditional expression into CPS form, but since this is our first encounter in the wild, let's refresh our memory by first looking at the original non-cps version:

```
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);
    }
}
```

It evaluates the condition in the current env, and calls `isTrue()` on the result, then uses that to decide whether to evaluate the true branch or the false branch, both in the current environment.

Our CPS version is not that different:

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

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

    $condition->Eval(
        $env,
        cont {
            my ($result) = @_;
            if ($result->isTrue) {
                $true_branch->Eval($env, $cont);
            } else {
                $false_branch->Eval($env, $cont);
            }
        }
    );
}
```

It evaluates the condition in the current environment and passes a continuation that will accept the result. That continuation calls isTrue() on the result and uses that to decide, in exactly the same way, whether to evaluate the true branch or the false branch. In either case the original continuation that was argument to PScm::SpecialForm::If::Apply() is passed to the chosen branch's Eval() method.

Staying with the program, our next task is the invocation of lambda handled by PScm::SpecialForm::Lambda::Apply():

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

    my ($args, $body) = $form->value;

    $cont->Cont(PScm::Closure::Function->new($args, $body, $env));
}
```

There's nothing very interesting here. lambda just creates a closure. There are no calls to Eval() that it must make during this creation, so we can treat the call to new as a simple expression and invoke our argument continuation on the result.

That just leaves PScm::Closure::Function::Apply() and PScm::Env::Extend(). Let's start with **PScm::Closure::Function**. The original just mapped Eval() over its arguments then called a private _apply() method on the results:

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

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

Another job for map_eval() then:

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

    $form->map_eval(
        $env,
        cont {
            my ($ra_evaluated_args) = @_;
            $self->_apply($ra_evaluated_args, $cont);
        }
    );
}
```

Note however that we need to pass the current continuation to that private _apply() method. That's because the closure will be calling Eval() on its body. Let's take a look at PScm::Closure::_apply().

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

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

It differs in that it takes a reference to an array of args and a continuation, rather than just an array of args, and it passes the continuation in to the call to Eval() on its body.

Finally, we need to rewrite PScm::Env::Extend().

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

    PScm::Expr::List->new(@$ra_values)->map_eval(
        $self,
        cont {
            my ($ra_evaluated_values) = @_;
            $cont->Cont(
                $self->ExtendUnevaluated(
                    $ra_symbols, $ra_evaluated_values
                )
            );
        }
    );
}
```

It uses map_eval() to evaluate its list of values, passing the result to a continuation that extends the environment with those values. It calls the argument continuation $cont on the result.

At this point in the discussion, we have a working CPS version of interpreter 0.0.2, and all the original tests that were written for that version still pass. However we seem to have done a lot of hard work for no benefit, since the interpreter is externally equivalent to the original 0.0.2 version. We can remedy that by giving the interpreter an additional construct that provides direct access to the underlying continuations.

There are many ways that this could be done, but one of the best-known and most powerful ways is with a form that goes by the unwieldy title call-with-current-continuation, usually abbreviated to call/cc. This form takes a function as argument and calls it, passing the current continuation as an explicit argument to the function, for example:

```
> (call/cc (lambda (cont) (cont 10)))
10
```

When the function invokes the continuation as a function, control returns to the `call/cc` and the argument to the continuation becomes the result of the call to `call/cc`.

If the previous example doesn't seem too exciting, how about this:

```
> (call/cc
>     (lambda (cont)
>         (if (cont 10)
>             20
>             30)))
10
```

Here the call to `(cont 10)` produced an immediate return of the value `10` through the `call/cc` even though it was executed in the conditional position of an `if` statement.

These two examples only show control passing down the "stack" when a continuation is invoked. However it is perfectly reasonable for control to return *up* the stack to a procedure that has already returned. It is simply not easy to demonstrate with this version of the interpreter. Once we have an interpreter with assignment and sequences, it becomes much easier.

`call/cc` is in fact a low-level, if not the lowest level continuation tool. It is possible to build higher level control constructs using it. Abandoning pscheme for a moment, consider this Fibonacci[13] sequence generator in some hypothetical Perl-like language that supports co-routines:

```
sub fib {
    my ($i, $j) = (0, 1);
    for (;;) {
        yield $i;
        ($i, $j) = ($j, $i + $j);
    }
}

while ((my $i = fib()) < 22) { # prints 0 1 2 3 5 8 13 21
    print "$i ";
}
```

That `yield` call not only behaves like a return statement, but also remembers the current state of the function so that the next time the function is called control resumes where it last left off. With built in continuations this sort of control flow is very easy to achieve.

Anyway I hope this has whet your appetite a little for what `call/cc` can do, so let's have a look at its implementation.

It is of course a special form, and as usual it has an `Apply()` method:

[13] The Fibonacci series starts with 0 and 1. the next number in the series is always the sum of the previous two, e.g. 0, 1, 1, 2, 3, 5, 8, 13, 21 ...

```
package PScm::SpecialForm::CallCC;

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

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

    $form->first->Eval(
        $env,
        cont {
            my ($closure) = @_;
            $closure->Apply(new PScm::Expr::List($cont),
                $env, $cont);
        }
    );
}

1;
```

It evaluates its first argument, which should result in a function of one argument, passing the `Eval()` a continuation which will `Apply()` the function to a form explicitly containing the current continuation. It also passes the current env and the current continuation a second time, this time as the normal implicit argument.

That's all there is to it. Of course the continuation itself will need an `Apply()` method so that it can be invoked as an operator.

We're now ready to see the whole of the **PScm::Continuation** package, in Listing 13.10.1 on page 225.
We've already seen most of this, only the `Apply()` method is new.

```
023 sub Apply {
024     my ($self, $form, $env, $cont) = @_;
025     $form->map_eval(
026         $env,
027         cont {
028             my ($ra_evaluated_args) = @_;
029             $self->Cont($ra_evaluated_args->[0]);
030         }
031     );
032 }
```

`Apply()` on Lines 23-32 is another method that makes use of `map_eval()` to evaluate its arguments. It passes it a continuation that calls *itself* on the first of its evaluated arguments, totally ignoring the passed-in, current continuation, and effecting transfer of control to whatever context this continuation represents.

And we're done.

A simple test of `call/cc` can be seen in Listing 13.10.2 on page 226.

Full source code for this version of the interpreter is available at
`http://billhails.net/Book/releases/PScm-0.1.2.tgz`

13.6.3 CPS `letrec`

The interpreter version 0.0.3 back in Chapter 6 on page 73 introduced `letrec` (`let recursive`) which allowed environments to be created in such a way that closures would extend the environment that they were themselves defined in, allowing them to make recursive calls.

This subsection takes the additions to version 0.0.3 and reimplements them in CPS. We're going to start to pick up the pace somewhat from hereon in, but I'll still present all of the changes, starting with the `letrec` special form itself. Here's the original v3:

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

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

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

The CPS version calls a modified `PScm::Env::ExtendRecursively()`, passing a continuation that takes the recursively extended environment and evaluates the body in it, passing the original continuation to that `Eval()`.

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

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

    $env->ExtendRecursively(
        $ra_symbols,
        $ra_values,
        cont {
            my ($extended_env) = @_;
            $body->Eval($extended_env, $cont);
        }
    );
}
```

`PScm::Env::ExtendRecursively()` calls `PScm::Env::ExtendUnevaluated()` as a simple expression then calls `_eval_values()` on the extended environment, passing the original continuation:

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

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

Here's the new CPS `_eval_values()`:

```
sub _eval_values {
    my ($self, $cont) = @_;
    $self->_map_bindings([keys %{ $self->{bindings} }], $cont);
}
```

It uses a new helper `_map_bindings()`, where the original `_eval_values()` just used `map`. This works in a similar way to `map_eval()`, evaluating each value in the environment but then assigning the result back to the original binding:

```
sub _map_bindings {
    my ($self, $ra_keys, $cont) = @_;
    my (@keys) = @$ra_keys;
    if (@keys) {
        my $firstkey = shift @keys;
        $self->{bindings}{$firstkey}->Eval(
            $self,
            cont {
                my ($value) = @_;
                $self->{bindings}{$firstkey} = $value;
                $self->_map_bindings([@keys], $cont);
            }
        );
    } else {
        $cont->Cont($self);
    }
}
```

Other methods are unchanged so we have completed the CPS rewrite of `letrec`, and all tests for 0.0.3 still pass in 0.1.3.

Full source code for this version of the interpreter is available at
`http://billhails.net/Book/releases/PScm-0.1.3.tgz`

13.6.4 CPS let*

The interpreter version 0.0.4 back in Chapter 7 on page 81 added `let*`, a shorthand way of creating nested environments providing the appearence of sequential assignment within the bindings of the `let*` expression. This was a simple addition, the rewrite will be equally simple.

Here's the new `PScm::SpecialForm::LetStar::Apply()`:

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

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

    $env->ExtendIteratively(
        $ra_symbols,
        $ra_values,
        cont {
            my ($extended_env) = @_;
            $body->Eval($extended_env, $cont);
        }
    );
}
```

Just like `letrec` (which called a modified `PScm::Env::ExtendRecursively()`,) this calls a modified `ExtendIteratively()`, passing a continuation that evaluates the body of the `let*` in the new environment with the original continuation.

Here's the modifications to `PScm::Env::ExtendIteratively()`:

```
sub ExtendIteratively {
    my ($self, $ra_symbols, $ra_values, $cont) = @_;
    my @symbols = @$ra_symbols;
    my @values  = @$ra_values;
    if (@symbols) {
        my $symbol = shift @symbols;
        my $value  = shift @values;
        $self->Extend(
            [$symbol],
            [$value],
            cont {
                my ($extended) = @_;
                $extended->ExtendIteratively([@symbols], [@values],
                    $cont);
            }
        );
    } else {
        $cont->Cont($self);
    }
}
```

The old version just iterated over the name/value pairs, creating an additional nested environment each time around the loop and returning the final result. CPS is easier with recursive definitions so this `ExtendIteratively()` has been recast as a recursive method. It still does the same job, but additionally arranges that the original continuation gets called on the final, extended environment.

Full source code for this version of the interpreter is available at
`http://billhails.net/Book/releases/PScm-0.1.4.tgz`

13.6.5 CPS List Processing

Our next iteration of the interpreter, version 0.0.5 back in Chapter 8 on page 87, added the list manipulation functions `quote`, `list`, `car`, `cdr` and `cons` to the language, and additionally changed the internal implementation of **PScm::Expr::List** from simple perl listrefs to linked lists, making **PScm::Expr::List** abstract, adding a **PScm::Expr::List::Pair** class to represent the cons cells, and adding a **PScm::Expr::List::Null** class to represent the empty list.

Surprisingly, The CPS rewrite of all of this is quite minimal. First, here's the new `quote` in `PScm::SpecialForm::Quote::Apply()`:

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

The original returned its first argument unevaluated, the CPS form calls its continuation on it. Remember that the `quote` system was re-written for a later version of the interpreter to support `unquote` back in Section 9.2.2 on page 112, so we'll be returning to `quote` later on, in Section 13.6.6 on page 208, where we rewrite that rewrite!

The other additional functions: `car`, `cdr`, `cons` and `list` are all primitives that share an `Apply()` method that has already been rewritten into CPS in Section 13.6.2 on page 188.

Among the **PScm::Expr** classes, the only thing that changes is the `map_eval()` method. That method was introduced in version 0.0.5 to work with pscheme lists, then re-introduced at an earlier stage of the CPS rewrite, in version 0.1.2, because we needed a recursive alternative to Perl's `map`. Finally, here, we combine the two implementations. Here's `PScm::Expr::List::Pair::map_eval()`:

```
sub map_eval {
    my ($self, $env, $cont) = @_;

    $self->[FIRST]->Eval(
        $env,
        cont {
            my ($evaluated_first) = @_;
            $self->[REST]->map_eval(
                $env,
                cont {
                    my ($evaluated_rest) = @_;
                    $cont->Cont($self->Cons($evaluated_first,
                                            $evaluated_rest));
                }
            );
        }
    );
}
```

And here's the new default `PScm::Expr::map_eval()` that terminates the recursion of `map_eval()` if `$self` is **PScm::Expr::Null**, does the right thing if the `cdr` of the list is not a list, and handles continuations, all in one tiny method:

```
sub map_eval {
    my ($self, $env, $cont) = @_;
    $self->Eval($env, $cont);
}
```

Full source code for this version of the interpreter is available at
`http://billhails.net/Book/releases/PScm-0.1.5.tgz`

13.6.6 CPS `macro` and `unquote`

Version 0.0.6 of our interpreter, from Chapter 9 on page 107, introduced the `macro` special form. This special form took arguments in an identical manner to `lambda` and created a variant type of closure **PScm::Closure::Macro**. Normal lambda closure evaluation proceeds by evaluating the arguments to the closure then evaluating the body of the closure with those arguments bound. In contrast macro closure evaluation proceeds by evaluating the body of the closure with its unevaluated arguments bound, then re-evaluating the result.

The rewrite into CPS is trivial, first here's the CPS form of `PScm::SpecialForm::Macro::Apply()`:

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

Just as with `PSCm::SpecialForm::Lambda::Apply()`, the creation of the closure can be treated as a simple expression (no calls to `Eval()`) and the continuation called on the result.

Now `PScm::Closure::Macro::Apply()`:

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

    $self->_apply(
        $form,
        cont {
            my ($new_form) = @_;
            $new_form->Eval($env, $cont);
        }
    );
}
```

Here we pass a continuation to the core `_apply()` method that takes the resulting new form and evaluates that in the current environment with the current continuation as an additional argument. The core `_apply()` extends the environment with unevaluated arguments, then calls the body of the macro with the new environment and the passed in continuation.

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

    my $extended_env =
      $self->{env}->ExtendUnevaluated($self->{args}, $args);
    $self->{body}->Eval($extended_env, $cont);
}
```

But you get the idea.

That's all there is to the rewrite of the `macro` extension into CPS. However Chapter 9 on page 107 also rewrote **PScm::SpecialForm::Quote** to support the `unquote` keyword which allows the interpolation of evaluated sub-expressions within a quoted expression. That proves more interesting to recast into CPS.

Let's start at the top by looking at the new CPS version of `PScm::SpecialForm::Quote::Apply()`:

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

So far so good, we just pass the current continuation along with the current environment to the `Quote()` method of whatever expression we're quoting.

Let's deal with the easy stuff first. `PScm::Expr::Quote()` used to just return `$self`, the CPS version calls the continuation on `$self` instead:

```
sub Quote {
    my ($self, $env, $cont) = @_;
    $cont->Cont($self);
}
```

That leaves **PScm::Expr::List::Pair**'s `Quote()`:

```
sub Quote {
    my ($self, $env, $cont) = @_;
    if ($self->[FIRST]->is_unquote) {
        $self->[REST]->first->Eval($env, $cont);
    } else {
        $self->quote_rest($env, $cont);
    }
}
```

Great! since the calls to both `Eval()` and `quote_rest()` are in tail position, it need only pass the continuation along to both. All the `Eval()` methods have already been dealt with of course, so that leaves `quote_rest()`. Let's first refresh our memories by looking at the non-CPS original:

```
sub quote_rest {
    my ($self, $env) = @_;
    return $self->Cons(
        $self->[FIRST]->Quote($env),
        $self->[REST]->quote_rest($env)
    );
}
```

This is *definately* not tail recursive. But if we think it through there are no problems. The first thing it does is call `Quote()` on its `first()` element, then it calls itself on the `rest()` of the list, then finally it calls `Cons()` on those two results. Both the call to `Quote()` and `quote_rest()` could potentially result in calls to `Eval()` so we need to pass continuations to both. We can rewrite it a little first to make the order of operations more explicit:

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

    my $quoted_first = $self->first->Quote($env);
    my $quoted_rest = $self->rest->quote_rest($env);

    return $self->Cons($quoted_first, $quoted_rest);
}
```

Now all we do is rewrite that so that the call to `Quote()` gets a continuation that performs the remaining two operations, including passing a second continuation to `quote_rest()` that performs the last `Cons()`. here's the CPS rewrite:

```
172 sub quote_rest {
173     my ($self, $env, $cont) = @_;
174     $self->[FIRST]->Quote(
175         $env,
176         cont {
177             my ($quoted_first) = @_;
178             $self->[REST]->quote_rest(
179                 $env,
180                 cont {
181                     my ($quoted_rest) = @_;
182                     $cont->Cont(
183                         $self->Cons($quoted_first, $quoted_rest));
184                 }
185             );
186         }
187     );
188 }
```

It calls `Quote()` on its first element, passing a continuation (Lines 176-186) that accepts the `$quoted_first` and then calls `quote_rest()` on the rest of the elements, passing *that* a continuation (Lines 180-184) that accepts the `$quoted_rest` and calls the original continuation on the result of `Cons()`-ing the `$quoted_first` and `$quoted_rest` together.

Finally, as before, where the default `PScm::Expr::quote_rest()` just returned `$self`, now it calls its aregument continuation on `$self`:

```
sub quote_rest {
    my ($self, $env, $cont) = @_;
    $cont->Cont($self);
}
```

That's it for our CPS rewrite of the `unquote` facility. All other methods are unchanged.

Full source code for this version of the interpreter is available at
`http://billhails.net/Book/releases/PScm-0.1.6.tgz`

13.6.7 CPS Sequences and Assignment

Next up are the sequences (`begin`) and assignment (`set!`) introduced by interpreter 0.0.7 in Chapter 10 on page 123. Let's start with sequences and the `begin` special form.

To recap `begin` takes a sequence of zero or more expressions and evaluates them in strict left to right order, returning the last result. If given no arguments, in this implementation it returns the empty list. Here's its `Apply()`:

```
sub Apply {
    my ($self, $form, $env, $cont) = @_;
    if ($form->is_pair) {
        $self->apply_next($form, $env, $cont);
    } else {
        $cont->Cont($form);
    }
}
```

The original was iterative. This version has been recast in a recursive mould to make the CPS transform easier and to take advantage of **PScm::Expr::List**. If the list is empty it calls the continuation on the empty list, otherwise it passes the form, environment and continuation to a helper method `apply_next()`:

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

    $form->first->Eval(
        $env,
        cont {
            my ($val) = @_;
            if ($form->rest->is_pair) {
                $self->apply_next($form->rest, $env, $cont);
            } else {
                $cont->Cont($val);
            }
        }
    );
}
```

It evaluates the first element of the list, passing a continuation that accepts the result. If there is more of the list to process, it calls itself recursively on the rest of the list, otherwise it calls the original continuation on the `$val` (the value of the last expression on the list). Note the similarity between this method and `map_eval()`. The diference is only that `apply_next()` does not need to construct a new list of all the evaluated results.

Next and last is `set!`. `set!` uses `PScm::Env::Assign()` that was developed for `letrec` to locate the nearest binding for a symbol and change its value. `set!` is a special form since it evaluates its second argument (the value) but not its first (the symbol).

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

The CPS rewrite is pretty straightforward. It evaluates the value passing a continuation that will perform the assignment (a simple expression) calling the original continuation on the result.

Full source code for this version of the interpreter is available at
`http://billhails.net/Book/releases/PScm-0.1.7.tgz`

13.6.8 CPS `define`

Version 0.0.8 in Chapter 11 on page 131 finally introduced `define`, delayed until then for reasons I won't reiterate here. Akin to `set!`, `define` takes a symbol and a value and binds the symbol to the value in the current environment. It is a special form since it does not evaluate the symbol. The CPS rewrite of `PScm::SpecialForm::Define::Apply()` proceeds much as the one for `set!` did:

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

    $expr->Eval(
        $env,
        cont {
            my ($value) = @_;
            $cont->Cont($env->Define($symbol, $value));
        }
    );
}
```

It evaluates the expression, passing a continuation that accepts the result and calls `PScm::Env::Define()` to bind the result to the symbol in the current environment. We can treat the call to `PScm::Env::Define()` as a simple expression and just call the original continuation on the result.

Full source code for this version of the interpreter is available at
`http://billhails.net/Book/releases/PScm-0.1.8.tgz`

13.6.9 CPS OOP

Version 0.0.9 in Chapter 12 on page 135 introduced an object-oriented extension to PScheme, by means of the `make-class` special form and the `root` parent class. Although there was a fair amount of code added to implement that extension, it turns out that very little of that code needs changing to produce a CPS version. As usual we just have to hunt down the calls to `Eval()` and ensure that there is a continuation available to pass in. Let's start with `PScm::SpecialForm::MakeClass::Apply()`.

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

    my $parent_expr = $form->first;
    my $fields = $form->rest->first;
    my $methods = $form->rest->rest;

    $parent_expr->Eval(
        $env,
        cont {
            my ($parent_class) = @_;
            $cont->Cont(
                PScm::Class->new(
                    $parent_class, $fields, $methods, $env
                )
            );
        }
    );
}
```

The first thing the old version did was to evaluate the parent expression (the value of the parent class) then use that along with the fields and methods (unevaluated) to create the new class. Our rewrite evaluates the parent expression passing a continuation that will create the class. We can treat the class creation as a simple expression (it makes no further calls to `Eval()`) and just call the original continuation on the result.

Nothing to follow up on there, so next we turn our attention to the application of a class to arguments, which creates a new object. This is in `PScm::Class::Apply()`:

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

    my $new_object = $self->make_instance();
    $new_object->call_method(
        $new_object,
        "init", $form, $env,
        cont {
            $cont->Cont($new_object);
        }
    );
}
```

As usual, `Apply()` now takes a continuation. The old version called `make_instance()` to create a new instance of the class, then called the `init` method of the new object with the arguments to the class, then returned the new object. The CPS version can still just call `make_instance()` as a simple expression, but needs to pass a continuation to `call_method()` because `call_method()` will be calling `Eval()` on both the arguments and the body of the method. The continuation just calls the original continuation on the new object (as if it were returning it).

Remembering that objects in this implementation are just environments, Here's the rewrite of `PScm::Env::call_method()`:

```
sub call_method {
    my ($self, $this, $method_name, $args, $env, $cont) = @_;

    if (my $method = $self->_lookup_method($method_name)) {
        $method->ApplyMethod($this, $args, $env, $cont);
    } else {
        $cont->Cont(undef);
    }
}
```

Hey, this isn't too bad at all. If `call_method()` finds the method, it calls `ApplyMethod()` on it, passing the original continuation as an extra argument. Wheras the old version implicitly returned `undef` if the method was not found, the CPS version must explicitly call the continuation on `undef`[14].

Next we need to look at `PScm::Closure::Method::ApplyMethod()`:

```
sub ApplyMethod {
    my ($self, $this, $form, $env, $cont) = @_;

    $form->map_eval(
        $env,
        cont {
            my ($evaluated_args) = @_;
            $self->_apply(
                PScm::Expr::List->Cons($this, $evaluated_args),
                $cont);
        }
    );
}
```

Again we use `map_eval()` on the `$form` (the arguments to the method) to evaluate them, this time passing a continuation that applies the method to its evaluated arguments, using `PScm::Expr::List::Cons()` to prepend the current object `$this` to the **PScm::Expr::List** of arguments to the core `PScm::Closure::_apply()` method, and passing in the original continuation. We've already discussed that core `_apply()` method in Section 13.6.2 on page 188 when we re-wrote `lambda`.

That just leaves the calling of methods on the objects themselves. Both **PScm::Env** and **PScm::Env::Super** have an `Apply()` method. The one from **PScm::Env::Super** arranges to pass the current value of `this` to the called method, otherwise they are very similar. Here's `PScm::Env::Apply()`:

[14] Note that calling a continuation on `undef` is not the same as *returning* `undef`, the trampoline will never see this `undef` and terminate the computation prematurely.

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

    my ($method, $args) = ($form->first, $form->rest);
    $self->CallMethodOrDie($self, $method, $args, $env, $cont);
}
```

And here's `PScm::Env::Super::Apply()`:

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

    my ($method, $args) = ($form->first, $form->rest);
    my $this = $env->LookUp(PScm::Expr::Symbol->new("this"));
    $self->CallMethodOrDie($this, $method, $args, $env, $cont);
}
```

They both now make use of a new support method `PScm::Env::CallMethodOrDie()` which just does what it says.

```
sub CallMethodOrDie {
    my ($self, $this, $method, $args, $env, $cont) = @_;
    $self->call_method(
        $this,
        $method->value,
        $args, $env,
        cont {
            my ($res) = @_;
            if (defined $res) {
                $cont->Cont($res);
            } else {
                die "method ", $method->value, " not found\n";
            }
        }
    );
}
```

It calls `call_method()` on `$self` and passes a continuation that checks if the result is defined. If the result is defined that means that `call_method()` sucessfully found the method and invoked it, in which case the original continuation is called on the result. If the result is `undef` then the method was not found and `CallMethodOrDie()` lives up to its name.

To make some of the subsequent tests easier I've added a `print` primitive. It takes a single evaluated argument and calls its `Print()` method. The only problem to solve is how to tell it about the current output filehandle. Ideally we would create a new **PScm::Expr::FileHandle** type and install one containing the default output filehandle in the environment for `print` to find, however it is easier for our limited purposes to just pass the output filehandle to the **PScm::SpecialForm::Print** constructor. You can see this in the changes to `ReadEvalPrint()` between 0.0.9 and 0.1.9 here:

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

    $outfh ||= new FileHandle(">-");
    my $reader      = new PScm::Read($infh);
    my $initial_env;
    $initial_env = new PScm::Env(
        let          => new PScm::SpecialForm::Let(),
        '*'          => new PScm::Primitive::Multiply(),
        '-'          => new PScm::Primitive::Subtract(),
        '+'          => new PScm::Primitive::Add(),
        if           => new PScm::SpecialForm::If(),
        lambda       => new PScm::SpecialForm::Lambda(),
        list         => new PScm::Primitive::List(),
        car          => new PScm::Primitive::Car(),
        cdr          => new PScm::Primitive::Cdr(),
        cons         => new PScm::Primitive::Cons(),
        letrec       => new PScm::SpecialForm::LetRec(),
        'let*'       => new PScm::SpecialForm::LetStar(),
        eval         => new PScm::SpecialForm::Eval(),
        macro        => new PScm::SpecialForm::Macro(),
        quote        => new PScm::SpecialForm::Quote(),
        'set!'       => new PScm::SpecialForm::Set(),
        begin        => new PScm::SpecialForm::Begin(),
        define       => new PScm::SpecialForm::Define(),
        'make-class' => new PScm::SpecialForm::MakeClass(),
        'call/cc'    => new PScm::SpecialForm::CallCC(),
        print        => new PScm::SpecialForm::Print($outfh),
    );

    $initial_env->Define(
        PScm::Expr::Symbol->new("root"),
        PScm::Class::Root->new($initial_env)
    );
    trampoline(cont { repl($initial_env, $reader, $outfh) });
}
```

The **PScm::SpecialForm::Print** package is unusual, therefore, in that it has a `new()` method and creates its own instance, because it needs to save the argument filehandle. Other than that it is a standard CPS special form:

```
package PScm::SpecialForm::Print;

use base qw(PScm::SpecialForm);

use PScm::Continuation;

```

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

sub Apply {
    my ($self, $form, $env, $cont) = @_;
    $form->first->Eval($env, cont {
            my ($thing) = @_;
            $thing->Print(
                $self->{outfh},
                cont {
                    $cont->Cont($thing)
                }
            );
    });
}

1;
```

`PScm::SpecialForm::Print::Apply()` invokes its argument `$form`'s `Eval()` method with the current environment and a continuation that will print the result, then call the original continuation on the evaluated result. So `print` returns the expression that it printed.

Full source code for this version of the interpreter is available at
`http://billhails.net/Book/releases/PScm-0.1.9.tgz`

13.7 CPS Without Closures

Perl has made it relatively easy for us to implement CPS by passing closures as continuations, albeit with an object wrapper. We should be glad of that and continue to use this feature. However the question must be asked: how would we go about implementing CPS in a language that does not support closures?

The answer, or one answer in any case is to roll our own closures as separate objects. This would mean a separate class for every occurrence of the `cont{}` construct in the PScheme source. The object would be initialized with all of the values that the closure referred to, and a method in the class would perform the same duty as the individual closure, referring to those saved values as attributes of `$self` rather than as lexical variables. Consider the `PScm::Expr::List::Eval()` method we discussed early on, in its CPS form:

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

You can see three continuations in this one method: the continuation `$cont` passed in as argument, the outer `cont{}` returned to the trampoline, and the inner `cont{}` that applies the evaluated `$op` to the as-yet unevaluated arguments. There is a lot of dependancy on the lexical scope of various variables in this code. If we were going to do this without closures we would have to make all that explicit. Here's an attempt:

```
sub Eval {
    my ($self, $env, $cont) = @_;
    return new Bouncer(
        new ListEvalFirstCont($self, $env, $cont)
    );
}
```

The **Bouncer** class would cope with any continuations returned to the trampoline, it would have a `new()` method to capture the argument continuation and a `Bounce()` method that invoked the captured continuation:

```
package Bouncer;

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

```
sub Bounce {
    my ($self) = @_;
    $self->{cont}->Cont();
}
```

Then **ListFirstEvalCont** would need a `new()` method, and a `Cont()` method of no arguments, since that is what the **Bouncer** would call on it it:

```
package ListEvalFirstCont;

sub new {
    my ($class, $list, $env, $cont) = @_;
    bless {
        list => $list,
        env  => $env,
        cont => $cont,
    }, $class;
}

sub Cont {
    my ($self) = @_;
    $self->{list}->first()->Eval(
        $self->{env},
        new ListEvalRestCont($self->{list},
                             $self->{env},
                             $self->{cont});
    );
}
```

You can see that the `Cont()` method here is doing the same thing that the closure was doing in the original code, but it in turn must create a new **ListEvalRestCont** object rather than a closure. That **ListEvalRestCont** would in turn need a `new()`, and a `Cont()` method, since that is what the operations `Eval()` method would call:

```
package ListEvalRestCont;

sub new {
    my ($class, $list, $env, $cont) = @_;
    bless {
        list => $list,
        env  => $env,
        cont => $cont,
    }, $class;
}

sub Cont {
    my ($self, $op) = @_;
    $op->Apply($self->{list}->rest,
               $self->{env},
               $self->{cont});
}
```

And that's not the end of it, since really that last `Cont()` method should be returning a continuation to the trampoline rather than just calling `Apply()` directly...

Admittedly this is just first pass untested code to give you an idea, but I'm not writing this just to scare you off. The point to note is that there are three basic things that have to be kept track of when implementing CPS without continuations. One is the current position in the control flow (in this case the list being evaluated.) The second is the current environment: the values of variables that the continuations need to execute; the third is the containing continuation. In fact what the closure implementation makes implicit and effectively hides from us is that there is a *chain of continuations*, from closure to closure, back to the originating continuation in the repl. This is an important observation. It will greatly simplify the writing of a PScheme compiler later.

There are even some advantages to implementing continuations in this way. Primarily because the chain of continuations is explicit, it can be traversed and searched making all sorts of additional control flow constructs easier to implement. For example a `try` / `throw` / `catch` / `resume` mechanism need merely traverse back up the continuation chain looking for a catching continuation, invoke it with the originating continuation, and if the `catch` block could fix the problem it would `resume` from the instruction after the `throw`. While this is possible with our existing implementation, it is more tricky.

13.8 CPS Fun

We're done! Let's play around with what we have.

13.8.1 An `error` Handler

There are many uses of `call/cc`, the simplest is probably the definition of *escape procedures*, procedures that when called, *escape* the current context and return control to a containing outer context. The simplest type of escape procedure is `error`. If at any point in your code you invoke `error`, the argument error message is printed, the current context is abandoned and control is returned to the top level. For example (pretending we have a divide "`/`" operator):

```
> (define div
>   (lambda (numerator denominator)
>     (if denominator
>         (/ numerator denominator)
>         (error "division by zero"))))
div
> (+ (div 2 0) 1)
Error: division by zero
()
>
```

The addition never happened. Control returned directly to the top level. Using call/cc, we can define an error escape procedure in the PScheme language itself, without needing to make further changes to the interpreter.

All error has to do is to print its error message and call a continuation that returns control to the top level. So, assuming that top level continuation is already installed as ^escape (I'm just using a caret, "^", to prefix any continuation names so they stand out,) the error procedure itself is straightforward:

```
(define error
  (lambda (msg)
    (begin
      (print msg)
      (^escape ()))))
```

Note that the ^escape continuation expects an argument, so error passes the empty list (), and that is what the repl prints as the result of whatever expression error is called from.

Next we need to create that top level continuation. Here's a first attempt:

```
(define ^escape (call/cc (lambda (c) c)))
```

This looks promising. The call/cc calls the anonymous lambda expression passing in the current continuation. The lambda expression just returns its argument, which is what call/cc returns, and that is what gets bound to the global ^escape. This is good, as far as it goes, the current continuation certainly is bound to ^escape.

The problem is that the operation of defining ^escape is part of the continuation saved in ^escape. Put another way, the first time error calls ^escape, control resumes at the point that call/cc is returning its value and so the define is re-executed, binding the empty list to ^escape and forgetting the previously bound continuation. So the ^escape continuation is only useable once.

There are two ways to fix this.

The first way would be to change error so that it passes ^escape as argument to itself: (^escape ^escape), The downside of that is that you will get a **PScm::Continuation** printed out as the result of any call to error.

The other way to fix this is to change the way we set up ^escape in the first place. First of all we create a global ^escape variable with an arbitrary initial binding:

```
(define ^escape 0)
```

Then we use call/cc as before but call an expression that directly assigns to ^escape:

```
(call/cc
  (lambda (cont)
    (begin
      (set! ^escape cont))))
```

This way the call/cc has no enclosing context, no deferred operations to perform, and when ^escape is invoked control returns directly to the top level.

A test of the error handler, which just duplicates the above code, is in Listing 13.10.3 on page 227. Next let's try something a bit more challenging.

13.8.2 yield

Remember that strange Fibonacci generator that was described towards the end of Section 13.6.2 on page 188, the one that used a hypothetical `yield` command to return a value while remembering its current position so that a subsequent call to the function would resume where it left off? Well here's how we'd like to write it in PScheme:

```
(define fib
  (yielder
    (letrec ((fib-loop
               (lambda (i j)
                 (begin
                   (yield i)
                   (fib-loop j (+ i j))))))
      (fib-loop 0 1))))
```

and here's how we'd like to call it:

```
(let ((n 10))
  (while n
    (begin
      (set! n (- n 1))
      (print (fib)))))
```

Note that there is nothing at all special about the code that uses `fib`. All of the interesting details are in the definition of `fib` itself. The function `fib` is defined as a `yielder` (my term). It creates a recursive helper routine `fib-loop` and calls it with arguments `0` and `1`. That helper routine `yield`s the current value of its first argument `i`, then calls itself with argument `i` replaced by `j` and `j` replaced with `i + j`.

The second part of the example loops 10 times printing the next value from `fib` each time around the loop.

Of course this presupposes a few features that PScheme doesn't appear to have. The `while` macro was already introduced in Section 9.2.2 on page 112, but here it is again just for completeness:

```
(define while
  (macro (test body)
    '(letrec
      ((loop
         (lambda ()
           (if ,test
             (begin
               ,body
               (loop))
             ()))))
      (loop))))
```

You can see that it's a recursive definition, but by now that shouldn't worry you: as each call to `loop` is in tail position we won't grow any context or eat any stack with this definition.

The other two features we don't have yet are `yield` and `yielder`. `yield` is actually quite easy to write. When we say `(yield value)`, what we really want to do is return not only the value, but also the current continuation so that the next time we are called that continuation can be called instead, returning us to where we left off. We can return more than one value by wrapping the values in a list, so notionally `(yield value)` means:

```
(^return (list current-continuation value))
```

where `^return` is another continuation set up by the caller of the function.

Now we need a way of getting the current continuation and the only way to do that is with `call/cc` so, still notionally, but getting closer, `(yield value)` means:

```
(call/cc
  (lambda (current-continuation)
    (^return (list current-continuation value))))
```

In fact that's it, all we need to do is wrap that up in a macro, and we have `yield`:

```
(define yield
  (macro (value)
    '(call/cc
      (lambda (^here)
        (^return (list ^here ,value))))))
```

Next we need to look at `yielder`. You probably won't be surprised to find out it's another macro. It returns a function that, when called for the first time, invokes the body of the `yielder` expression, saving the current continuation in a `^return` variable. When `yield` is invoked control returns through the `^return` continuation. The `^here` continuation part of the returned result is saved and the `value` part is returned by the function as a whole.

On subsequent calls, rather than invoking the body of the `yielder` again, the saved `^here` continuation is invoked, returning control to where the `yield` left off. Here's `yielder`:

```
(define yielder
  (macro (body)
   '(let ((firsttime 1)
          (^resume 0)
          (^return 0))
     (lambda ()
       (if firsttime
         (let ((res (call/cc
                      (lambda (^cont)
                        (begin
                          (set! ^return ^cont)
                          ,body)))))
           (begin
               (set! firsttime 0)
               (set! ^resume (car res))
               (car (cdr res))))
```

```
        (let ((res (call/cc
                     (lambda (^cont)
                       (begin
                         (set! ^return ^cont)
                         (^resume))))))
          (begin
              (set! ^resume (car res))
              (car (cdr res))))))))
```

Again, continuations are flagged with a caret.

That really is all there is to it. If this seems like a solution looking for a problem, then let me point out a very real problem for which this would be the perfect solution. Consider the venerable **File::Find** package from the core Perl distribution. This package allows you to specify criteria for recursively locating files in a filesystem, along with a callback subroutine which is called on each file so located. Sometimes this is exactly what you need, but sometimes you would prefer **File::Find** to behave as an *iterator*, returning the next file found on each call. With proper support for co-routines in perl, the callback function that you provide to **File::Find** need only `yield` the file to achieve exactly that, without changing a single line of the **File::Find** package itself!

Tests for this `yield` feature, which just duplicate the above code, are in Listing 13.10.4 on page 228.

13.9 Summary

This has been a long chapter and a difficult one, particularily if you were unfamiliar with the subject of continuations. It has however provided numerous real-world examples of CPS during the rewrite of the interpreter and hopefully the basic principles of CPS have been well covered.

To reiterate the basic idea, continuations can be thought of as the “rest of the computation”, or perhaps more graphically as a “reference to a return statement” that can be called as a function.

Anyway, having achieved a CPS interpreter in Section 13.6.2 on page 188 we then introduced the `call/cc` form, which passes the current continuation to its argument function. If Perl functions could take a reference to their `return` statement with a syntax like `\return`, then we could write `call/cc` in perl like this:

```
sub call_cc {
    my ($sub) = @_;
    $sub->(\return);
}
```

Unfortunately Perl 5 does not support that syntax yet, but lest you think this is all irrelevant to Perl, you should be aware that the Parrot virtual machine which will run Perl6 has continuations built in from the ground up!

Finally, having completely re-worked the interpreter in CPS, in Section 13.8 on page 220 we showed how we could use `call/cc` in conjunction with `macro` to create two high-level control constructs: `error` and `yield` from within the PScheme language itself.

——— • ———

The next few chapters will take continuations a little further, to show the sorts of things that can be done by varying the internal details of the implementation of continuations.

13.10 Listings

13.10.1 PScm/Continuation.pm

```perl
package PScm::Continuation;

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

require Exporter;

our @ISA = qw(PScm Exporter);

our @EXPORT = qw(cont);

sub new {
    my ($class, $cont) = @_;
    bless { cont => $cont }, $class;
}

sub cont(&) {
    my ($cont) = @_;
    return __PACKAGE__->new($cont);
}

sub Apply {
    my ($self, $form, $env, $cont) = @_;
    $form->map_eval(
        $env,
        cont {
            my ($ra_evaluated_args) = @_;
            $self->Cont($ra_evaluated_args->[0]);
        }
    );
}

sub Cont {
    my ($self, $arg) = @_;
    return cont { $self->{cont}->($arg) };
}

sub Bounce {
    my ($self) = @_;
    $self->{cont}->();
}

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

1;
```

13.10.2 t/PScm_CallCC.t

```
use strict;
use warnings;
use Test::More;
use lib './t/lib';
use PScm::Test tests => 2;

BEGIN { use_ok('PScm') }

eval_ok(<<EOF, '10', 'call/cc');
(let ((a (lambda (return)
           (if (return (* 2 5))
             20
             30))))
  (call/cc a))
EOF

# vim: ft=perl
```

13.10.3 t/CPS_Error.t

```
use strict;
use warnings;
use Test::More;
use lib ’t/lib’;
use PScm::Test tests => 2;

BEGIN { use_ok(’PScm’) }

eval_ok(<<EOF, <<EOR, ’error’);
(define ^error 0)
(call/cc
  (lambda (cont)
    (begin
      (set! ^error cont))))

(define error
  (lambda (msg)
    (begin
      (print msg)
      (^error ()))))

(define div
  (lambda (numerator denominator)
    (if denominator
        (/ numerator denominator)
        (error "division by zero"))))

(+ (div 2 0) 1)

EOF
^error
PScm::Continuation
error
div
"division by zero"
()
EOR

# vim: ft=perl
```

13.10.4 t/CPS_Yield.t

```
use strict;
use warnings;
use Test::More;
use lib 't/lib';
use PScm::Test tests => 2;

BEGIN { use_ok('PScm') }

eval_ok(<<EOF, <<EOR, 'yield');
(define yield
  (macro (value)
    '(call/cc
      (lambda (^here)
        (^return (list ^here ,value))))))

(define yielder
  (macro (body)
   '(let ((firsttime 1)
         (^resume 0)
         (^return 0))
     (lambda ()
       (if firsttime
         (let ((res (call/cc
                      (lambda (^cont)
                        (begin
                          (set! ^return ^cont)
                          ,body)))))
           (begin
               (set! firsttime 0)
               (set! ^resume (car res))
               (car (cdr res))))
         (let ((res (call/cc
                      (lambda (^cont)
                        (begin
                          (set! ^return ^cont)
                          (^resume))))))
           (begin
               (set! ^resume (car res))
               (car (cdr res)))))))))

(define while
  (macro (test body)
    '(letrec
      ((loop
         (lambda ()
           (if ,test
             (begin
               ,body
               (loop))
             ()))))
      (loop))))
```

```

(define fib
  (yielder
    (letrec ((fib-loop
               (lambda (i j)
                 (begin
                   (yield i)
                   (fib-loop j (+ i j))))))
      (fib-loop 0 1))))

(let ((n 10))
  (while n
    (begin
      (set! n (- n 1))
      (print (fib)))))
EOF
yield
yielder
while
fib
0
1
1
2
3
5
8
13
21
34
()
EOR

# vim: ft=perl
```

Chapter 14

Threads

Continuations make the implementation of threads almost trivial. The trick is in the trampoline. Our old trampoline method repeatedly called `Bounce()` on the current continuation to get the next continuation, until a continuation returned `undef`:

```
sub trampoline {
    my ($cont) = @_;
    $cont = $cont->Bounce() while defined $cont;
}
```

If you think about it, a continuation already represents a single thread of computation. The trampoline is just managing that single thread, ensuring that it does not consume too much stack. Suppose that `trampoline()`, instead of just repeatedly invoking the current continuation, kept a queue of continuations, and after bouncing the one at the front of the queue put the result onto the back of the queue (if the result was not `undef`,) looping until the queue was empty. This version does exactly that:

```
sub trampoline {
    while (@thread_queue) {
        my $cont = shift @thread_queue;
        $cont = $cont->Bounce();
        push @thread_queue, $cont if defined $cont;
    }
}
```

Note that it no longer takes an argument continuation, instead it gets the next continuation from the front of the queue. `@thread_queue` is a new lexical "`my`" variable in the **PScm** package.

We place new threads on that queue with a `new_thread()` method, also in the **PScm** package:

```
sub new_thread {
    my ($self, $cont) = @_;
    push @thread_queue, $cont;
}
```

Very simple. it takes a continuation and pushes it onto the queue.

Next we need a way of creating threads from the PScheme language. This is done using a new special form `spawn`. `spawn` takes no arguments and returns `0` to one thread and `1` to the other. This means you can write code that does different things in different threads by testing the result, much like the UNIX `fork` system call does:

```
> (if (spawn)
>   (begin
>     (print "hello")
>     (print "hello")
>     1)
>   (begin
>     (print "goodbye")
>     (print "goodbye")
>     (exit)))
"hello"
"goodbye"
"hello"
"goodbye"
1
```

Notice that although both threads run in parallel, one thread does an `(exit)` so only the result `1` from the other thread gets printed.

`spawn` is a new special form in **PScm::SpecialForm::Spawn**, and it's surprisingly easy to implement:

```
286 package PScm::SpecialForm::Spawn;
287
288 use base qw(PScm::SpecialForm);
289 use PScm::Continuation;
290
291 sub Apply {
292     my ($self, $form, $env, $cont) = @_;
293
294     PScm->new_thread(cont {
295         $cont->Cont(new PScm::Expr::Number(0));
296     });
297
298     $cont->Cont(new PScm::Expr::Number(1));
299 }
```

On Line 294 it calls `new_thread()` with a new continuation that will call the current continuation with argument `0`, and on Line 298 it directly calls the current continuation with an argument of `1`. This is so easy I feel like I have cheated!, but really that's all there is to it. The new continuation will get executed in turn when the `Cont()` on Line 298 returns control to the trampoline, and the trampoline will continue executing any threads on its queue until all threads have finished and the queue is empty.

`exit` is even more trivial. It has to be a special form because individual primitives do not get called in tail position, but all that it has to do is to `return undef` to the trampoline:

```
327 package PScm::SpecialForm::Exit;
328
```

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

sub Apply {
    my ($self, $form, $env, $cont) = @_;
    return undef;
}

1;
```

Incidentally, `exit` provides a useful way of terminating the interactive interpreter. Typing `(exit)` at the prompt while only one thread is running will result in an empty `$thread_queue` so the trampoline will finish.

All that remains is to wire this in to the repl:

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

    $outfh ||= new FileHandle(">-");
    my $reader      = new PScm::Read($infh);
    my $initial_env;
    $initial_env = new PScm::Env(
        let          => new PScm::SpecialForm::Let(),
        ’*’          => new PScm::Primitive::Multiply(),
        ’-’          => new PScm::Primitive::Subtract(),
        ’+’          => new PScm::Primitive::Add(),
        if           => new PScm::SpecialForm::If(),
        lambda       => new PScm::SpecialForm::Lambda(),
        list         => new PScm::Primitive::List(),
        car          => new PScm::Primitive::Car(),
        cdr          => new PScm::Primitive::Cdr(),
        cons         => new PScm::Primitive::Cons(),
        letrec       => new PScm::SpecialForm::LetRec(),
        ’let*’       => new PScm::SpecialForm::LetStar(),
        eval         => new PScm::SpecialForm::Eval(),
        macro        => new PScm::SpecialForm::Macro(),
        quote        => new PScm::SpecialForm::Quote(),
        ’set!’       => new PScm::SpecialForm::Set(),
        begin        => new PScm::SpecialForm::Begin(),
        define       => new PScm::SpecialForm::Define(),
        ’make-class’ => new PScm::SpecialForm::MakeClass(),
        ’call/cc’    => new PScm::SpecialForm::CallCC(),
        print        => new PScm::SpecialForm::Print($outfh),
        spawn        => new PScm::SpecialForm::Spawn(),
        exit         => new PScm::SpecialForm::Exit(),
    );

    $initial_env->Define(
        PScm::Expr::Symbol->new("root"),
        PScm::Class::Root->new($initial_env)
    );
    __PACKAGE__->new_thread(cont { repl($initial_env, $reader, $outfh) });
    trampoline();
}
```

Apart from the addition of `spawn` and `exit` to the initial environment, there is only one change. The repl uses `new_thread()` to add the initial thread (continuation) to the `@thread_queue` then calls `trampoline()` with no arguments, rather than passing the continuation directly to `trampoline()`.

14.1 Variations

A more complete thread implementation would also provide mechanisms for collecting the result of one thread in another with a `wait` command—not so easy, you'd need to put the waiting thread on a separate queue and have the `exit` command take an argument and put it somewhere that the `wait` command could find.

You would also need to be able to prevent concurrent access to sections of code, best done with some sort of atomic semaphore operation. But atomicity is easy to guarantee at the level of the interpreter internals, as long as no continuations are called during the claiming of a semaphore.

These variations are exercises you can try at home.

14.2 Tests

A simple test for `spawn` is in Listing 14.3.1 on the facing page.

14.3 Listings

14.3.1 t/CPS_Spawn.t

```
use strict;
use warnings;
use Test::More;
use lib 't/lib';
use PScm::Test tests => 3;

BEGIN { use_ok('PScm') }

eval_ok(<<EOF, <<EOR, 'spawn');
(if (spawn)
    (begin
        (print "hello")
        (print "hello")
        1)
    (begin
      (print "goodbye")
      (print "goodbye")
      (exit)))
EOF
"hello"
"goodbye"
"hello"
"goodbye"
1
EOR

eval_ok('(exit)', '', 'exit');

# vim: ft=perl
```

Full source code for this version of the interpreter is available at
`http://billhails.net/Book/releases/PScm-0.1.10.tgz`

Chapter 15

Better Error Handling

Section 13.8.1 on page 220 described how we could write our own `error` routine in the PScheme language, using an escape procedure to return control to the top level and resuming the read-eval-print loop. That implementation had a couple of drawbacks however.

- Apart from printing an error message, the error handler returned a value (the empty list) to the repl which printed it.
- It would be useful if we could gain access to the error continuation from within the PScheme interpreter, so that recoverable errors would no longer have to be fatal.

In this short chapter we remedy these deficiencies by providing a built-in `error` primitive, and show how our interpreter can interface with it.

15.1 The Built in `error` Primitive

All the `error` builtin has to do is to print its argument message and restart the repl. In order to restart the repl it must have a continuation to do that, therefore much like the `print` special form, our `error` will have to be initialised with arguments, this time both the output filehandle on which to print the error, and the continuation to invoke afterwards. Here's how we wire it in to the repl.

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

    $outfh ||= new FileHandle(">-");
    my $reader      = new PScm::Read($infh);
    my $initial_env;
    $initial_env = new PScm::Env(
        let          => new PScm::SpecialForm::Let(),
        ’*’          => new PScm::Primitive::Multiply(),
        ’-’          => new PScm::Primitive::Subtract(),
        ’+’          => new PScm::Primitive::Add(),
        if           => new PScm::SpecialForm::If(),
        lambda       => new PScm::SpecialForm::Lambda(),
        list         => new PScm::Primitive::List(),
        car          => new PScm::Primitive::Car(),
        cdr          => new PScm::Primitive::Cdr(),
        cons         => new PScm::Primitive::Cons(),
        letrec       => new PScm::SpecialForm::LetRec(),
        'let*'       => new PScm::SpecialForm::LetStar(),
        eval         => new PScm::SpecialForm::Eval(),
        macro        => new PScm::SpecialForm::Macro(),
        quote        => new PScm::SpecialForm::Quote(),
        'set!'       => new PScm::SpecialForm::Set(),
        begin        => new PScm::SpecialForm::Begin(),
        define       => new PScm::SpecialForm::Define(),
        'make-class' => new PScm::SpecialForm::MakeClass(),
        'call/cc'    => new PScm::SpecialForm::CallCC(),
        print        => new PScm::SpecialForm::Print($outfh),
        spawn        => new PScm::SpecialForm::Spawn(),
        exit         => new PScm::SpecialForm::Exit(),
        error        => new PScm::SpecialForm::Error(
                        $outfh,
                        cont { repl($initial_env, $reader, $outfh) }
                     ),
    );

    $initial_env->Define(
        PScm::Expr::Symbol->new("root"),
        PScm::Class::Root->new($initial_env)
    );
    __PACKAGE__->new_thread(cont { repl($initial_env, $reader, $outfh) });
    trampoline();
}
```

You can see the token "`error`" being bound to a new **PScm::SpecialForm::Error** object, and the constructor for that object is passed both the current `$outfh` and a continuation which just calls `repl()` with appropriate arguments.

The constructor for **PScm::SpecialForm::Error** just stashes its arguments:

```
sub new {
    my ($class, $outfh, $cont) = @_;
    bless {
        outfh => $outfh,
        cont => $cont,
    }, $class;
}
```

When we invoke `error` with for example `(error "my error message")` its `Apply()` method is invoked. Here it is:

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

    $form->first->Eval($env, cont {
        my ($msg) = @_;
        $self->do_error($msg->display_string());
    });
}
```

It has to use CPS because the error message itself might be computed, we can't just assume that it is already a string. So it `Eval()`'s the message, passing in a continuation that will first of all convert the resulting message to a string suitable for display, and then call a secondary method `do_error()` on that string. the `display_string()` method is defined in **PScm::Expr** to just call `as_string()`:

```
sub display_string { $_[0]->as_string() }
```

but **PScm::Expr::String** overrides this to call `$self->value()` instead:

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

The upshot of this is that the error message, if it's a **PScm::Expr::String**, won't be wrapped in quotes when printed which is what the `PScm::Expr::String::as_string()` method would have done.

Returning to **PScm::SpecialForm::Error**, `do_error()` is also quite simple:

```
sub do_error {
    my ($self, $errstr) = @_;
    $errstr =~ s/\n$//;
    $self->{outfh}->print("Error: ", $errstr, "\n");
    return $self->{cont};
}
```

It expects only a simple perl string. it strips any trailing newline from the error message, prints it to the stored output file handle, then returns the stored continuation to the trampoline. That continuation will restart the repl, skipping the print stage of the current loop.

Apart from making the code a little easier on the eye, there is another reason for having a separate `do_error()` method, and that brings us to the second part of this chapter.

15.2 Using the `error` Builtin for Internal Errors

It would be very useful if we could avail ourselves of this `error` builtin to report and recover from internal errors such as type check and variable lookup failures. This is actually easy to do. All we have to do is look up the error handler in the current environment and invoke its `do_error()` method. A new method `Error()` in the **PScm** base class does exactly this, and so is available everywhere:

```
sub Error {
    my ($self, $msg, $env) = @_;
    my $error = $env->LookUp(new PScm::Expr::Symbol('error'));
    $error->do_error($msg);
}
```

The only thing it has to be careful of is that it calls `do_error()` in tail position, so that the continuation gets returned to the trampoline.

Let's look at a few places where we can make use of this new method. If you remember, way back in Section 3.6 on page 27 we saw how the various primitive operations made use of a `check_type()` method, which would `die` if the argument object was not of the desired type. Now we can cheat a little, and rather than rewriting those primitives in CPS, we just catch the error with a (Perl) `eval` in the shared `PScm::Primitive::Apply()` method, and call `Error()` with argument `$@` if an error was detected. Here's the previous version of that `PScm::Primitive::Apply()`:

```
008 sub Apply {
009     my ($self, $form, $env, $cont) = @_;
010
011     $form->map_eval(
012         $env,
013         cont {
014             my ($evaluated_args) = @_;
015             $cont->Cont($self->_apply($evaluated_args->value));
016         }
017     );
018 }
```

and here's the changes:

```
008 sub Apply {
009     my ($self, $form, $env, $cont) = @_;
010
011     $form->map_eval(
012         $env,
013         cont {
014             my ($evaluated_args) = @_;
015             my $result = eval {
016                 $self->_apply($evaluated_args->value);
017             };
018             if ($@) {
019                 $self->Error($@, $env);
020             } else {
021                 $cont->Cont($result);
022             }
023         }
024     );
025 }
```

It is safe for `Apply()`, on Line 16 to evaluate the individual primitive separately, since it is not in CPS form. Then all it has to do is either call the current continuation on the result, or invoke `Error()` with `$@`, both calls being in tail position.

Apart from primitive expressions, another place where we throw an exception on a recoverable error is in the LookUp() method of **PScm::Env**, when we don't find a binding for a variable. Unfortunately LookUp() was treated as a simple expression in our CPS rewrite, so we need to backtrack to find the CPS code that invokes LookUp() in order to install the error handling. Fortunately there is only one place where that happens, when a symbol is evaluated. Here's the previous PScm::Expr::Symbol::Eval().

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

and the changes:

```
sub Eval {
    my ($self, $env, $cont) = @_;
    my $result = eval { $env->LookUp($self) };
    if ($@) {
        $self->Error($@, $env);
    } else {
        $cont->Cont($result);
    }
}
```

Again, as with the primitive Apply() above, it is safe for it to execute the LookUp() first, since LookUp() is not in CPS. Then, depending on $@, it either invokes Error() or calls the continuation on the result of the lookup.

We next make an identical change to PScm::SpecialForm::Set::Apply(). If you remember the set! special form uses PScm::Env::Assign() to replace an existing binding with a new value, and it is an error if Assign() can't find a binding to change. It is Assign() that dies if the binding is not found, and since Assign() is a simple expression that has not been rewritten into CPS, PScm::SpecialForm::Set::Apply() must trap the exception and throw the PScheme error. Firstly here's PScm::SpecialForm::Set::Apply() before the changes:

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

and here's the changes:

```
sub Apply {
    my ($self, $form, $env, $cont) = @_;
    my ($symbol, $expr) = $form->value;
    $expr->Eval(
        $env,
        cont {
            my ($val) = @_;
            my $result = eval { $env->Assign($symbol, $val) };
            if ($@) {
                $self->Error($@, $env);
            } else {
                $cont->Cont($result);
            }
        }
    );
}
```

Another place where we died was in `PScm::Env::_populate_bindings()` where we handle the possibility of dot notation and single values in the formal arguments to a `lambda` expression. This routine is only called by `ExtendUnevaluated()`, but unfortunately `ExtendUnevaluated()` is not yet in CPS form. In this case, because `ExtendUnevaluated()` is called from a number of places and all those places would have to be aware that `ExtendUnevaluated()` could throw a Perl exception, it seems better to rewrite `ExtendUnevaluated()` into CPS, and change its callers to use the CPS form. Here's the CPS version of `ExtendUnevaluated()`.

```
sub ExtendUnevaluated {
    my ($self, $symbols, $values, $cont) = @_;

    my %bindings;
    eval {
        $self->_populate_bindings(\%bindings, $symbols, $values);
    };
    if ($@) {
        $self->Error($@, $self);
    } else {
        my $newenv = $self->new(%bindings);
        $newenv->{parent} = $self;
        $cont->Cont($newenv);
    }
}
```

Most of the methods that call `ExtendUnevaluated()` are already in CPS so we don't really need to see the changes to them. One method, `make_instance()` in **PScm::Class** is not in CPS, so we need to rewrite that too:

```
sub make_instance {
    my ($self, $cont) = @_;

    $self->{parent}->make_instance(cont {
        my ($parent_instance) = @_;

        $self->{env}->ExtendUnevaluated(
            new PScm::Expr::List(
                PScm::Expr::Symbol->new("class"),    # $self
                PScm::Expr::Symbol->new("super"),    # $parent_instance
                @{ $self->{fields} },                # 0...
            ),
            new PScm::Expr::List(
                $self,                               # "class"
                PScm::Env::Super->new(super => $parent_instance)
                ,                                    # "super"
                ((PScm::Expr::Number->new(0)) x @{ $self->{fields} })
                ,                                    # field...
            ),
            $cont
        );
    });
}
```

And the equivalent method in **PScm::Class::Root**:

```
sub make_instance {
    my ($self, $cont) = @_;

    $self->{env}->ExtendUnevaluated(
        new PScm::Expr::Symbol("class"),
        $self,
        $cont
    );
}
```

The caller of `make_instance()`, `PScm::Class::Apply()`, was already in CPS so transforming that to call the CPS form of `make_instance()` is trivial:

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

    $self->make_instance( cont {
        my ($new_object) = @_;
        $new_object->call_method(
            $new_object,
            "init", $form, $env,
            cont {
                $cont->Cont($new_object);
            }
        );
    });
}
```

The last place where we die unnecessarily is in PScm::Env::CallMethodOrDie() where it is an error if a method can not be found. Fortunately CallMethodOrDie() is already in CPS so it is even easier to change. Here's the original:

```
sub CallMethodOrDie {
    my ($self, $this, $method, $args, $env, $cont) = @_;
    $self->call_method(
        $this,
        $method->value,
        $args, $env,
        cont {
            my ($res) = @_;
            if (defined $res) {
                $cont->Cont($res);
            } else {
                die "method ", $method->value, " not found\n";
            }
        }
    );
}
```

and here are the changes:

```
sub CallMethodOrDie {
    my ($self, $this, $method, $args, $env, $cont) = @_;
    $self->call_method(
        $this,
        $method->value,
        $args, $env,
        cont {
            my ($res) = @_;
            if (defined $res) {
                $cont->Cont($res);
            } else {
                $self->Error(
                    "method " . $method->value . " not found\n",
                    $env
                );
            }
        }
    );
}
```

Very simple: the `die` was already in tail position, so where it used to `die`, it invokes `Error()` instead.

15.3 Tests

A few simple tests for `error` are in Listing 15.4.1 on the next page. Primarily, besides demonstrating that the `error` builtin works, they show that the repl is still up and running afterwards.

15.4 Listings

15.4.1 t/CPS_BuiltInError.t

```
use strict;
use warnings;
use Test::More;
use lib ’t/lib’;
use PScm::Test tests => 8;

BEGIN { use_ok(’PScm’) }

eval_ok(<<EOF, <<EOR, ’built in error’);
(define div
  (lambda (numerator denominator)
    (if denominator
        (/ numerator denominator)
        (error "division by zero"))))
(+ (div 2 0) 1)
EOF
div
Error: division by zero
EOR

eval_ok(<<EOF, <<EOR, ’argument to error need not be a string’);
(error ’(an error "message"))
EOF
Error: (an error "message")
EOR

eval_ok(<<EOF, <<EOR, ’internal type error and recovery’);
(* 2 "2")
(* 2 2)
EOF
Error: wrong type argument(PScm::Expr::String) to PScm::Primitive::Multiply
4
EOR

eval_ok(<<EOF, <<EOR, ’internal lookup error and recovery’);
x
2
EOF
Error: no binding for x in PScm::Env
2
EOR

eval_ok(<<EOF, <<EOR, ’method lookup error and recovery’);
(define testclass
  (make-class
    root
    ()
    (say-hello () ’hello)))
(define testobj (testclass))
```

```
(testobj say-goodbye)
(testobj say-hello)
EOF
testclass
testobj
Error: method say-goodbye not found
hello
EOR

eval_ok(<<EOF, <<EOR, ’set! error and recovery’);
(set! x 1)
2
EOF
Error: no binding for x in PScm::Env
2
EOR

eval_ok(<<EOF, <<EOR, ’lambda error and recovery’);
(define test
  (lambda (a b c)
    (list a b c)))
(test 1 2)
2
EOF
test
Error: not enough arguments
2
EOR

# vim: ft=perl
```

Full source code for this version of the interpreter is available at
`http://billhails.net/Book/releases/PScm-0.1.11.tgz`

Chapter 16

Chronological Backtracking

Wheras, with the exception of Chapter 12 on page 135, we have been extending this interpreter to be more like a complete scheme implementation, this chapter makes a deliberate departure from the R^{6}RS [12] specification to add a feature not normally found in functional or procedural languages. This feature is best introduced by example, but suffice to say that it is one step on the way to implementing a logic programming language.

Understanding this chapter relies heavily on previous chapters. If you have skipped ahead to here, you should at least make sure that you understand the implementation details of CPS from Chapter 13 on page 159 before diving in to the details that follow. However you can read the next few sections on their own if you want to get a taste of what this chapter has to offer.

16.1 Introducing amb

The feature we shall be adding is called `amb` [1, pp412–437]. `amb` is short for "ambivalent"—in the sense of "having more than one value". As I've said it is best introduced by example, and the simplest example is this:

```
> (amb 1 2 3)
1
> ?
2
> ?
3
> ?
Error: no more solutions
> ?
Error: no current problem
```

What's going on here? Well `amb` is given a list of values, and returns all of them. But it returns them one at a time. When a "`?`" is typed at the PScheme prompt control *backtracks* to `amb` and it returns its next result. So the execution of expressions involving `amb` is somehow threaded into the read-eval-print loop itself. I should probably point out that this new behaviour is not specific to `amb`, but rather a general property of the interpreter:

```
> ?
Error: no current problem
> (+ 2 2)
4
> ?
Error: no more solutions
> ?
Error: no current problem
```

The **Error: no current problem** message means just that: there is no current problem so no backtracking is possible, wheras the **Error: no more solutions** message means that the current "problem" has just exhahsted all of its posibilities. With no occurence of `amb` in the "problem" there is only one possible outcome (4 in the `(+ 2 2)` example above) so the repl continues to behave as normal for "normal" input.

`amb` will only return a subsequent value if it is told that the previous value is not acceptable. One way of doing that, as we have seen, is by typing "`?`" at the scheme prompt. We can do the same thing within our code however, as I'll demonstrate next:

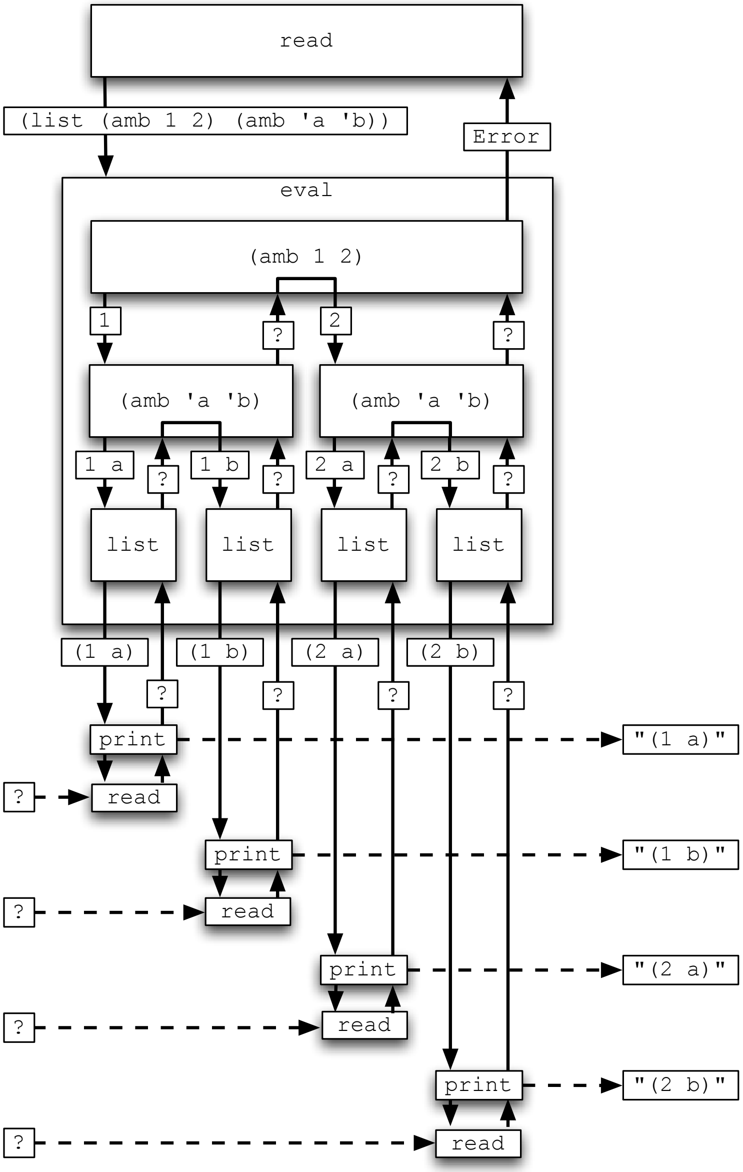
```
> (list (amb 1 2) (amb 'a 'b))
(1 a)
> ?
(1 b)
> ?
(2 a)
> ?
(2 b)
> ?
Error: no more solutions
```

Now that's interesting. There are two calls to `amb`, and `list` collects the results. Best we go through this one step at a time.

1. The expression first returns a list of the first arguments to each call to `amb`, namely `1` and `a`.
2. When we tell the interpreter that we'd like to see more results by typing `?` at the prompt, the second `amb` call intercepts the request and returns its second argument, so the whole expression returns `(1 b)`.
3. When we ask for a third result, the second `amb` again intercepts the request, but this time it has run out of arguments, so it fails to satisfy the request and control propogates back to the *first* call to `amb`. The first `amb` now returns its second result, `2`, and control passes forwards again to the second `amb`. This second `amb` is now being called afresh, as it were, and is back in its initial state where it returns its first argument, so the whole third result is `(2 a)`.
4. The request for a fourth result proceeds as the request for the second result did, with the second `amb` producing `b`, resulting in `(2 b)`.
5. With the fifth and final request, the second `amb` again fails, so propogates the failure back to the first `amb`, but this time the first `amb` has also exhausted its results, so propogates the failure back to the command loop and we get the "error".

The diagram in Figure 16.1 attempts to show this control flow in action[1].

Figure 16.1: Control flow during `(list (amb 1 2) (amb 'a 'b))`



So in what way does this demonstrate that we can control the backtracking behaviour of `amb`? Simple. When `amb` itself fails it propogates control back to the chronologically previous call to `amb`, just as typing a "?" at the prompt does. When the second `amb` call ran out of options in the example, control propogated back to the first `amb` call. Now a call to `amb` with no arguments must immediately fail, because it has no arguments to choose from:

[1] Of course none of this would be possible without continuations. Only with CPS do we have a situation where `Read()` calls `Eval()` and so forth, but that's best left for Section 16.3 on page 266, which discusses implementation.

```
> (amb)
Error: no more solutions
```

So calling amb with no arguments forces any previous amb to deliver up its next value[2]. We can wrap that behaviour in a function that tests some condition, and forces another choice if the condition is false. That function is called require:

```
(define require
  (lambda (x)
    (if x x (amb))))
```

The return value of x if the test succeeds is merely utilitarian, it is the call to amb with no arguments if the test fails that is important. So how can we use requre? Well for example let's assume we have a predicate even? that returns true if its argument is even. We can use that to filter the results of an earlier amb:

```
> (let ((x (amb 1 2 3 4 5 6)))
>      (begin
>        (require (even? x))
>        x))
2
> ?
4
> ?
6
> ?
Error: no more solutions
```

The expression (require (even? x)) filtered out the odd values of x, so only the even values were propogated to the result(s) of the expression.

You should be starting to see how amb and CPS are deeply interlinked, and how backtracking can therefore return to *any* chronologically previous point in the computation, not just "down the stack" to a caller of the code that initiates the backtracking.

16.2 Examples of amb in Action

Now we know what amb does, what can we use it for? That example with (require even? x) above should give you some idea, but in a word: search[3].

16.2.1 The "Liars" Puzzle

Consider the following logic problem, one of a classic and simple type[4].

[2] However you can't type (amb) at the prompt instead of "?" to prompt backtracking on the previous "problem" because the interpreter reguards anything other than "?" as the start of a new "problem."

[3] I'm not talking about searching the web, I mean searching for solutions to problems.

[4] This puzzle appears as an exercise in [1, p420] and they in turn accredit Hubert Philips, 1934, The Sphinx Problem Book.

Liars

> Five schoolgirls sat for an examination. Their parents—so they thought—showed an undue degree of interest in the result. They therefore agreed that, in writing home about the examination, each girl should make one true statement and one untrue one. The following are the relevant passages from their letters:
>
> **Betty:** “Kitty was second in the examination, I was only third.”
>
> **Ethel:** “You’ll be glad to hear that I was on top. Joan was second.”
>
> **Joan:** “I was third, and poor old Ethel was bottom.”
>
> **Kitty:** “I came out second. Mary was only fourth.”
>
> **Mary:** “I was fourth. Top place was taken by Betty.”
>
> What in fact was the order in which the five girls were placed?

amb makes it easy to solve this type of problem by merely enumerating all the possibilities then eliminating those possibilities that are wrong in some way:

```
> (define liars
>   (lambda ()
>     (let ((betty (amb 1 2 3 4 5))
>           (ethel (amb 1 2 3 4 5))
>           (joan  (amb 1 2 3 4 5))
>           (kitty (amb 1 2 3 4 5))
>           (mary  (amb 1 2 3 4 5)))
>         (begin
>           (require (distinct? (list betty ethel joan kitty mary)))
>           (require (xor (eq? kitty 2) (eq? betty 3)))
>           (require (xor (eq? ethel 1) (eq? joan  2)))
>           (require (xor (eq? joan  3) (eq? ethel 5)))
>           (require (xor (eq? kitty 2) (eq? mary  4)))
>           (require (xor (eq? mary  4) (eq? betty 1)))
>           ’((betty ,betty)
>             (ethel ,ethel)
>             (joan  ,joan)
>             (kitty ,kitty)
>             (mary  ,mary))))))
liars
> (liars)
((betty 3) (ethel 5) (joan 2) (kitty 1) (mary 4))
```

The bindings in the `let` supply all possible grades to each of the girls, then the first `require` in the body of the let makes sure that all the girls have different grades: the `distinct?` function only returns true if there are no duplicates in its argument list. I’ll show you the implementation, and that of other functions here, later. The remaining requirements simply list the two parts of each girl’s statement, requiring that one is true and one is false: the `xor` (exclusive or) function returns true only if one of its arguments is true and the other is false. The `eq?` function tests if two expressions are equal.

So we start out by requiring that all five girls have distinct positions in the exam results. The we go on to require that exactly one of each of the girls two statements is true. Finally we build and return a list of pairs of the girl's names, and the associated positions that satisfy all the requirements, using `quote` and `unquote`.

Of course this is horribly inefficient. There are $5^5 = 3125$ permutations of `betty`, `ethel`, `joan`, `kitty` and `mary`, and that first `distinct` requirement forces a re-evaluation of all but $5! = 120$ of them[5], so about 96% of the initial possibilities are pruned at the first step, and backtracking is provoked. In fact, when writing tests for this `amb` example, this single function took so long to run (about 14 seconds on my laptop) that I was forced to find ways to optimize it. The optimizations demonstrate some additional behaviour of `amb`, so here's the optimized version:

```
(define liars
  (lambda ()
    (let* ((betty (amb 1 2 3 4 5))
           (ethel (one-of (exclude (list betty)
                                   (list 1 2 3 4 5))))
           (joan  (one-of (exclude (list betty ethel)
                                   (list 1 2 3 4 5))))
           (kitty (one-of (exclude (list betty ethel joan)
                                   (list 1 2 3 4 5))))
           (mary  (car    (exclude (list betty ethel joan kitty)
                                   (list 1 2 3 4 5)))))
         (begin
           (require (xor (eq? kitty 2) (eq? betty 3)))
           (require (xor (eq? ethel 1) (eq? joan  2)))
           (require (xor (eq? joan  3) (eq? ethel 5)))
           (require (xor (eq? kitty 2) (eq? mary  4)))
           (require (xor (eq? mary  4) (eq? betty 1)))
           '((betty ,betty)
             (ethel ,ethel)
             (joan  ,joan)
             (kitty ,kitty)
             (mary  ,mary))))))
```

It starts out as before, setting `betty` to an `amb` choice from the available positions, but then calls a couple of new functions to calculate the value for `ethel` and the rest of the girls. `exclude` returns a list of all the elements in its second list that aren't in its first list. So for example if `betty` is `1`, then `ethel` only gets the choice of values `2` through `5`. I'll show you `exclude` later. `one-of` is more interesting, since it makes use of `require` and `amb`. It does the same thing as `amb`, but takes a single list of values as argument rather than individual arguments:

```
(define one-of
  (lambda (lst)
    (begin
        (require lst)
        (amb (car lst) (one-of (cdr lst))))))
```

[5] Yes, a practical application of the factorial function!

Firstly it requires that the list is not empty, then it uses amb to choose either the car of the list, or one-of the cdr of the list. This in fact demonstrates that amb must be a special form: this function would not work if amb had its arguments evaluated for it; if both arguments to that second amb were evaluated before amb saw them then one-of would get recursively executed until the list was empty, then the first amb to be actually invoked would be the one that terminates recursion when (require lst) fails, so this function would always fail if amb were a primitive.

Back to our optimized liars example. The use of let* instead of let makes the values of the previous bindings available to subsequent ones. By the time we get to assigning to mary, there is only one choice left, so we just take it with car rather than using one-of. Since our values are now guaranteed to be distinct, we can remove that explicit requirement from the code, and the optimized version runs in a little under a second on the same machine.

——— • ———

As promised, here are the rest of the scheme functions needed to implement the solution to the "Liars" puzzle and other examples seen earlier. You can skim these if you're not interested in the details, they don't really do anything new.

To make these functions easier to write (and read) I've introduced the boolean short circuiting special forms and and or to this version of the interpreter: (and a b) will return a without evaluating b if a is false, and (or a b) will return a without evaluating b if a is true.

Some of these functions also make use of not. and and or have been added as special forms to the interpreter and so interact with the amb rewrite, so you'll have to wait to see those, but not is just:

```
(define not
  (lambda (x)
    (if x 0 1)))
```

And so to our support routines. Firstly even?:

```
(define divisible-by
  (lambda (n)
    (lambda (v)
      (begin
        (define loop
          (lambda (o)
            (if (eq? o v)
                1
                (if (> o v)
                    0
                    (loop (+ o n))))))
        (loop 0)))))

(define even?
  (lambda (a)
    ((divisible-by 2) a)))
```

This is really just demonstrating the functional programming style that Scheme promotes[6]. The function divisible-by takes an argument number n and returns another function that will return true if its

[6] and that we can still get away without a "/" primitive.

argument is divisible by `n`. It creates an inner `loop` method which loops over $0, n, 2n, 3n \ldots$ until either equal to or greater than the number being tested. `even?` uses this to create a function that tests for divisibility by 2, and calls it on its argument `a`. It is total overkill to do it this way, but fun.

Next `distinct?`:

```
(define distinct?
  (lambda (lst)
    (if lst
        (and (not (member? (car lst) (cdr lst)))
             (distinct? (cdr lst)))
        1)))
```

`distinct?` says if the list is not empty, then it is distinct if its first element (its `car`) is not a member of the rest of the list and the rest of the list is distinct. If the list *is* empty, then it is distinct. `distinct` makes use of another function, `member?`, shown next.

```
(define member?
  (lambda (item lst)
    (if lst
        (or (eq? item (car lst))
            (member? item (cdr lst)))
        0)))
```

`member?` determines if its argument `item` is a member of its argument `lst`. It says if the list is not empty, then the item is a member of the list if it is equal to the `car` of the list or a member of the `cdr` of the list. The item is not a member of an empty list. `member?` uses another function `eq?` to test equality, but that's been added to the interpreter as a primitive, so we'll leave that for later.

Next up for consideration is `xor`. `xor` takes two arguments and returns true only if precisely one of those arguments is false.

```
(define xor
  (lambda (x y)
    (or (and x (not y))
        (and y (not x)))))
```

Lastly for the support routines, our optimized example made use of `exclude` which returns its second argument list after removing any items on its first argument list. It's easy to do now that we have `member?`:

```
(define exclude
  (lambda (items lst)
    (if lst
        (if (member? (car lst) items)
            (exclude items (cdr lst))
            (cons (car lst)
                  (exclude items (cdr lst))))
        ())))
```

For a non-empty list: if the first element is to be excluded then just return the result of calling `exclude` on the rest of the list. If it is not to be excluded, then prepend it to the result of calling `exclude` on the rest of the list. For an empty list the only result can be the empty list.

16.2.2 Barrels of Fun

Our next example is another logic puzzle, from [2]. It is somewhat different, but requires much the same approach.

> **Barrels of Fun**
>
> A wine merchant has six barrels of wine and beer containing:
>
> - 30 gallons
> - 32 gallons
> - 36 gallons
> - 38 gallons
> - 40 gallons
> - 62 gallons
>
> Five barrels are filled with wine and one with beer. The first customer purchases two barrels of wine. The second customer purchases twice as much wine as the first customer. Which barrel contains beer?

Here's a solution:

```
(define barrels-of-fun
  (lambda ()
    (let* ((barrels (list 30 32 36 38 40 62))
           (beer (one-of barrels))
           (wine (exclude (list beer) barrels))
           (barrel-1 (one-of wine))
           (barrel-2 (one-of (exclude (list barrel-1) wine)))
           (purchase (some-of (exclude (list barrel-1 barrel-2) wine))))
      (begin
        (require (eq? (* 2 (+ barrel-1 barrel-2))
                      (sum purchase)))
        beer))))
```

Again, it is more or less just a statement of the problem. We start off by picking the beer barrel at random. Then we say that the wine barrels are the remaining barrels. Next we randomly pick the first barrel of wine bought from the wine barrels, and the second from the remaining wine barrels. We don't know how many barrels the second customer bought, so we merely assign some-of the remaining barrels to that purchase. Finally in the body of the let we require that the second customer buys twice as much wine as the first, then return the beer barrel (the answer is 40 by the way).

We haven't seen some-of before. It is very similar to one-of described above, and makes direct use of amb.

```
(define some-of
  (lambda (lst)
    (begin
      (require lst)
      (amb (list (car lst))
           (some-of (cdr lst))
           (cons (car lst)
                 (some-of (cdr lst)))))))
```

It requires the list to be non-empty, then chooses between just the first element of the list (as a list), some of the rest of the list, or the first element prepended to some of the rest of the list. This will eventually produce all non-empty subsets of the list.

The only other function we haven't seen before is sum. It adds all the values in its argument list and is quite trivial:

```
(define sum
  (lambda (lst)
    (if lst
        (+ (car lst)
           (sum (cdr lst)))
        0)))
```

The sum of a list is the car of the list plus the sum of the cdr of the list. The sum of an empty list is zero.

16.2.3 Pythagorean Triples

As another example of amb, consider generating so-called *pythagorean triples*, triples of integers x, y and z such that $x^2 + y^2 = z^2$. This should be pretty easy.

```
> (define square
>   (lambda (x)
>     (* x x)))
square
> (define pythagorean-triples
>   (lambda ()
>      (let ((x (amb 1 2 3 4 5 6 7 8))
>            (y (amb 1 2 3 4 5 6 7 8))
>            (z (amb 1 2 3 4 5 6 7 8 9 10 11 12)))
>        (begin
>          (require (eq? (+ (square x)
>                           (square y))
>                        (square z)))
>          '((x ,x) (y ,y) (z ,z))))))
pythagorean-triples
> (pythagorean-triples)
((x 3) (y 4) (z 5))
> ?
((x 4) (y 3) (z 5))
> ?
```

```
((x 6) (y 8) (z 10))
> ?
((x 8) (y 6) (z 10))
> ?
Error: no more solutions
```

And so it was. After defining `square`, we pick some ranges of numbers `x`, `y` and `z`, then `require` that the sum of the squares of `x` and `y` equals the square of `z`.

Although it is simple and easy to understand, that's a terribly naiive implementation. We just guessed the range 1…8 for `x` and `y` based on a fixed range 1…12 for `z`. Plus the result includes duplicates: `((x 3) (y 4) (z 5))` is the same as `((x 4) (y 3) (z 5))`. Plus, the number of results is constrained by the highest value of `z`, altogether not very satisfactory.

With the addition of a couple more functions, we can remedy all of these deficiencies. Firstly, here's a function `integers-between` that will ambivalently return every number between its lower bound and its upper bound, in ascending order:

```
(define integers-between
  (lambda (lower upper)
    (begin
      (require (<= lower upper))
      (amb lower
           (integers-between (+ lower 1) upper)))))
```

It begins by requiring that its lower bound is less than or equal to its upper bound, then ambivalently returns first the lower bound, then the result of calling itself with its lower bound incremented by one.

Thinking about it, if we were to remove the bounds check from `integers-between` it would continue to produce new integers, one at a time, *ad-infinitum*, and without the bounds check it would have no need for the upper bound argument. That realisation gives us our second function, `integers-from`:

```
(define integers-from
  (lambda (x)
    (amb x
         (integers-from (+ x 1)))))
```

This function will just carry on returning one integer after another as long as it is backtracked to.

Given these two simple functions we can write a much more satisfactory version of `pythagorean-triples`:

```
(define pythagorean-triples
  (lambda ()
    (let* ((z (integers-from 1))
           (x (integers-between 1 z))
           (y (integers-between x z)))
      (begin
        (require (eq? (+ (square x)
                         (square y))
                      (square z)))
        '((x ,x) (y ,y) (z ,z))))))
```

It uses `let*` to make the value of `z` available to the definition of `x`, and likewise the value of `x` available to the definition of `y`, much as in the `liars` puzzle above. It lets `z` equal each of the positive integers in turn, then it lets `x` range over the values 1 to `z`. Then, to avoid duplication, it only allows `y` to range over the values `x` to `z`. The rest of the implementation is unchanged. It's a little slow, but it will continue to generate unique pythagorean triples as long as you keep asking it for more:

```
> (pythagorean-triples)
((x 3) (y 4) (z 5))
> ?
((x 6) (y 8) (z 10))
> ?
((x 5) (y 12) (z 13))
> ?
((x 9) (y 12) (z 15))
> ?
((x 8) (y 15) (z 17))
> ?
((x 12) (y 16) (z 20))
> ?
((x 7) (y 24) (z 25))
> ?
...
```

To wrap up this section, although it should be obvious, it's probably worth pointing out that there is a pitfall to using `amb` to generate infinite sequences like this. The function `integers-from` can *never fail*, so unless it is the first call to `amb` in your program, any previous calls to `amb` will never get backtracked to. This works out pretty well for `pythagorean-triples`: since we need the current value of `z` to constrain the values of `x`, the call to `integers-from` *had* to happen first, but even if we hadn't needed the value of `z` first, we would still have to have calculated it first, otherwise any previous calls to `amb` would never get a chance to yield more than their first result. For example the following just won't work:

```
...
    (let ((x (integers-from 1))
          (y (integers-from 1))
          (z (integers-from 1)))
...
```

The last call to `integers-from` to provide the value of `z`, when backtracked to (by hitting "?" or by some downstream call to `amb`), would just keep on producing values, so the declarations of `x` and `y` would never get backtracked to and never produce alternative values.

16.2.4 Parsing Natural Language

Our last example of `amb` is a little different. It turns out that `amb` is extremely useful for parsing. Because `amb` can backtrack and is capable of trying many alternative strategies, it is much more powerful than any simple bottom-up parser like the one used to parse PScheme itself. In fact it is quite capable of parsing some restricted subsets of natural language.

To understand what follows, it is essential to realise that even `set!`, when backtracked through, will have its effect undone. This is what is meant by "chronological backtracking": chronological backtracking really does restore the state of the machine to a *previous time*, as if nothing since the `amb` being backtracked to ever happened. I think that is quite amazing.

To start the discussion on parsing, consider the following two English sentences:

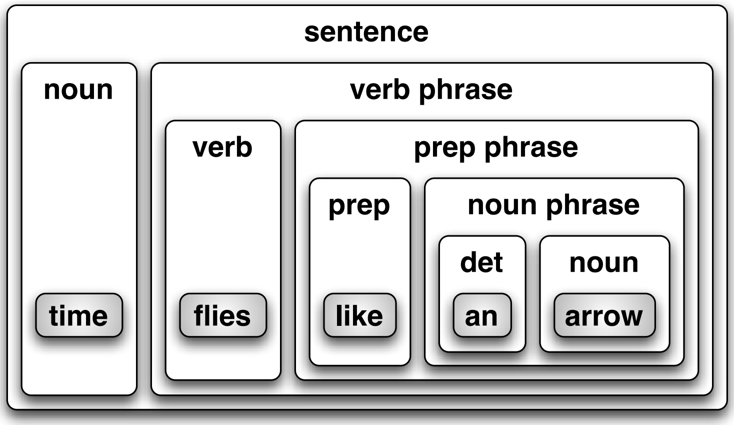
- "Time flies like an arrow."
- "Fruit flies like a bannanna."

Although superficially very similar, the two sentences have radically different structures and semantics: Time, "the indefinite continued progress of existence" is noted to always fly forward in the manner of an arrow, wheras fruit flies of the genus *Melanogaster* are known to be quite partial to bannanas.

This demonstrates quite vividly that it is in fact impossible to correctly parse natural language without involving semantics, and of course it is impossible to extract the semantics without parsing; a chicken and egg problem that I hope to show `amb` can neatly circumvent.

Drawing on old school grammar lessons, Figure 16.2 shows a reasonable parse tree for the first sentence. It consists of the noun "time" and a verb phrase. The verb phrase consists of the verb "flies" and a prepositional phrase. The prepositional phrase consists of the preposition "like" and a noun phrase. The noun phrase consists of the determinant "an" and the noun "arrow".

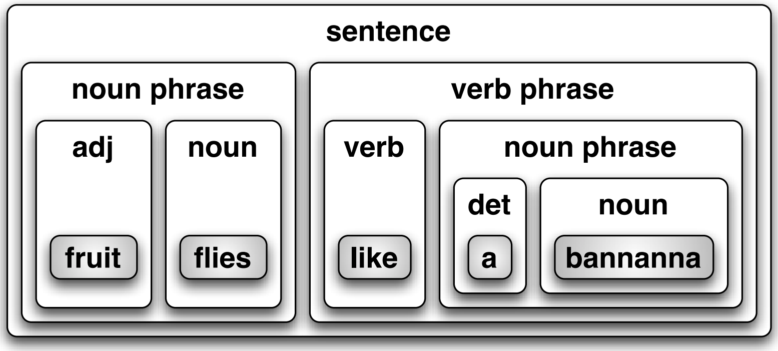
Figure 16.2: One possible parse of "Time flies like an arrow"



Similarily, Figure 16.3 on the next page shows a parse tree for the second sentence. This time the sentence breaks down into the classic noun phrase plus verb phrase structure (as did the first, but the noun phrase just contained a noun). The noun phrase contains the adjective "fruit" and the noun "flies". The verb phrase contains the verb "like" and another noun phrase. This second noun phrase consists of the determinant "a" and the noun "bannanna".

In order to parse these sentences, we can start off by categorizing the individual words:

Figure 16.3: A parse tree for "Fruit flies like a bannanna"



```
(define verbs        '(verb   flies like))
(define nouns        '(noun   time fruit flies bannanna arrow))
(define determinants '(det    a an))
(define adjectives   '(adj    time fruit))
(define prepositions '(prep   like))
```

The first symbol on each list identifies the type of the rest of the words on the list. Note that a number of the words occur on more than one of the lists: "like" acts as a preposition in the first sentence, while it is a verb in the second. Similarily "flies" is the verb in the first sentence, but a noun in the second. Additionally, I've added a couple of categorisations that aren't needed to parse those sentences correctly, but would nonetheless be present in a sufficiently general lexicon: "fruit" is certainly a noun, and "time" is a perfectly acceptable adjective ("time travel" for example). These additional classifications are exactly what cause us to do that double take when we first encounter these two sentences, and will make the parsing more realistic.

Next we create a global variable `*unparsed*` to hold the words remaining to be parsed. this is initially defined to be empty:

```
(define *unparsed* ())
```

Then we define a top level `parse` routine:

```
(define parse
  (lambda (input)
    (begin
      (set! *unparsed* input)
      (let ((sentence (parse-sentence)))
        (begin
          (require (not *unparsed*))
          sentence)))))
```

`parse` starts by setting the global `*unparsed*` to its argument. Then it calls `parse-sentence`, collecting the result. Finally it `requires` that there is nothing left in `*unparsed*` and returns the result of `parse-sentence`.

Readers who appreciate the dangers of global state and mutation might be wondering what on earth is going on here. A function that accepts an argument then just assigns it to a global variable? Worse, it then proceeds to mutate that global as the parse proceeds? Surely that is the antithesis of good programming? There is a very sound reason that it is done this way, and that is to demonstrate what amb is capable of. Please bear with me.

parse will be called like (parse '(fruit flies like a bannanna)) and should return a parse tree with the nodes of the tree labelled, like:

```
(sentence (noun-phrase (adj fruit)
                       (noun flies))
          (verb-phrase (verb like)
                       (noun-phrase (det a)
                                    (noun bannanna))))
```

We have seen that parse calls parse-sentence, and we shall see shortly that parse-sentence calls out to other parse-noun-phrase etc. routines to futher break down the sentence. The various parse-* routines all indirectly consume tokens from the global *unparsed* variable, but the only function that directly removes tokens from *unparsed* is the function parse-word:

```
(define parse-word
  (lambda (words)
    (begin
      (require *unparsed*)
      (require (member? (car *unparsed*) (cdr words)))
      (let ((found-word (car *unparsed*)))
        (begin
          (set! *unparsed* (cdr *unparsed*))
          (list (car words) found-word))))))
```

The argument words will be one of the lists of words defined above, where the car is the type of the words and the cdr is the actual words to be recognized. Hence the use of car and cdr to get the appropriate components.

So parse-word is called like (parse-word nouns) and will succeed and return a list of a type and a word if the first word of *unparsed* is one of its argument words. For example if *unparsed* is '(flies like an arrow) and we call (parse-word nouns) it should return the list (noun flies) and as a side effect set *unparsed* to '(like an arrow).

parse-word requires that there are tokens left to parse, then requires that the first word of *unparsed* is a member of its list of candidate words. If so then it removes the first word from *unparsed* and returns it, appended to the category of words that matched. If there are no words left to parse, or if the next word in *unparsed* is not one of the argument words, then parse-word fails and control backtracks to the previous decision point where the next alternative is tried. It is important to remember here that the effect of set! on *unparsed* can be undone by the backtracking of amb.

Back to parse. parse calls parse-sentence:

```
(define parse-sentence
  (lambda ()
    (amb (list 'sentence
               (parse-word nouns)
               (parse-verb-phrase))
         (list 'sentence
               (parse-noun-phrase)
               (parse-verb-phrase)))))
```

`parse-sentence` ambivalently chooses to parse either the structure of the first sentence or the structure of the second. It prepends the result with the appropriate grammatical label just as `parse-word` did.

Since the second part of both sentences is the same (a verb phrase) we could equivalently have said:

```
(define parse-sentence
  (lambda ()
    (list 'sentence
          (amb (parse-word nouns)
               (parse-noun-phrase))
          (parse-verb-phrase))))
```

In fact this second formulation is likely to be more efficient since it doesn't have to backtrack through `parse-verb-phrase` unneccessarily.

Next let's look at `parse-verb-phrase`. Our two example verb phrases are different. The first consists of a verb and a prepositional phrase, the second consists of a verb and a noun phrase. We can combine the two, eliminating the duplication on verbs for a slightly more efficient parse. Here's `parse-verb-phrase`:

```
(define parse-verb-phrase
  (lambda ()
    (list 'verb-phrase
          (parse-word verbs)
          (amb (parse-prep-phrase)
               (parse-noun-phrase)))))
```

Going bredth-first from `parse-sentence`, next up is `parse-noun-phrase`:

```
(define parse-noun-phrase
  (lambda ()
    (list 'noun-phrase
          (amb (parse-word adjectives)
               (parse-word determinants))
          (parse-word nouns))))
```

We have two example noun phrases: an adjective followed by a noun and a determinant followed by a noun. Again we've removed the duplication, this time on the noun.

Lastly, we have to parse prepositional phrases, of which we have only one example: a preposition followed by a noun phrase:

```
(define parse-prep-phrase
  (lambda ()
     (list 'prep-phrase
           (parse-word prepositions)
           (parse-noun-phrase))))
```

With these definitions in place, we can attempt to parse our two sentences (output reformatted manually to aid readability):

```
> (parse '(time flies like an arrow))
(sentence (noun time)
          (verb-phrase (verb flies)
                       (prep-phrase (prep like)
                                    (noun-phrase (det an)
                                                 (noun arrow)))))
> ?
(sentence (noun-phrase (adj time)
                       (noun flies))
          (verb-phrase (verb like)
                       (noun-phrase (det an)
                                    (noun arrow))))
> ?
Error: no more solutions
```

and

```
>(parse '(fruit flies like a bannanna))
(sentence (noun fruit)
          (verb-phrase (verb flies)
                       (prep-phrase (prep like)
                                    (noun-phrase (det a)
                                                 (noun bannanna)))))
> ?
(sentence (noun-phrase (adj fruit)
                       (noun flies))
          (verb-phrase (verb like)
                       (noun-phrase (det a)
                                    (noun bannanna))))
> ?
Error: no more solutions
```

So while time does indeed fly like an arrow, and fruit flies are fond of bannannas, other valid parses of the sentences imply that some strange creatures called “time flies” are attracted to arrows, and that fruit does fly much like a bannanna does.

What makes this really exciting is that we are still in backtracking mode when the parse is complete. If we don’t like a particular result we can request further results by hitting “?” at the prompt, but *this option is also available to client code of the parser*: Subsequent downstream analysis of the result may reject it on semantic grounds (fruit can’t fly, no such thing as “time flies”), and request an alternative parse.

So you’ve seen what amb can do. The rest of this chapter discusses its implementation.

16.3 Implementing amb

The core idea behind amb is to use an additional continuation to let us do *backtracking*. Instead of just passing around one continuation that specifies the point of return for the called function, we pass around two continuations. The first continuation is the same as before, and that takes care of normal control flow. The second continuation is a "failure" continuation of no arguments that gets called by amb when it runs out of options, and by the repl when you type in a "?". That failure continuation resumes execution at the previous decision point.

Here's a useful analogy to help keep track of what is going on. If you consider normal control flow, both calls to methods and calls (returns) to normal continuations to be always "downstream" towards the successful production of a result, then the invocation of the failure continuation causes control to pass back "upstream" to a previous point in the computation and resume from there.

The initial failure continuation is passed to Read() by the repl to produce the "no current problem" error and restart the repl (in fact Error() already restarts the repl for us). Then when the repl invokes Eval() on an expression it has read, it passes an alternative "no more solutions" failure continuation which again calls Error(). These initial failure continuations are as far "upstream" as you can get because they exist before the computation is even attempted.

Now if amb is invoked, it replaces the current failure continuation with a new one that, if called, will cause amb either to pass its next value back "downstream" again (to the success continuation) or, if there are no more choices, to retreat even further "upstream" to the previous failure continuation. That's all we have to achieve really, the rest of this chapter is just the details.

There are however another couple of places where the failure continuation needs to be treated specially. Remember my little rant about purely functional languages at the start of Section 10.1 on page 123? Well in a purely functional language there would be no such extra places, because it is only side effects that need to be undone as the failure continuation backtracks upstream through them. Both define and set! need to install their own failure continuations that will undo whatever change they made, then call the previous failure continuation to continue back upstream.

As you might have guessed, amb requires another rewrite of our interpreter. However this time the rewrite is, on the whole, a purely mechanical one. Apart from the places mentioned above, the failure continuation is always simply passed through untouched. It is an extra argument to all the methods that take a continuation argument, and the success continuations themselves now all take an extra failure continuation as argument too, since the failure continuation must not be lost track of.

16.3.1 Changes to Continuations

Notice that we now have three kinds of continuation: a success continuation for normal control flow, a failure continuation for backtracking, and let's not forget the continuation of no arguments returned to the trampoline to clear the stack. It was becoming obvious that if I just stuck with cont{} to create all continuations, I would have to start to litter the code with comments to the effect of "this is the success continuation", "this is the failure continuation" etc. It makes much more sense to make the original **PScm::Continuation** class abstract, and to have concrete subclasses for each of these types. Then, instead of the generic cont{} construct to create a continuation, we now have three separate prototyped subroutines to create continuations of the appropriate type. The cont{} construct still creates the "normal" continuations, but new constructs fail{} and bounce{} create the other types of continuation.

So without further ado, here's the new abstract **PScm::Continuation** class:

```
package PScm::Continuation;

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

require Exporter;

push our @ISA, qw(Exporter);

our @EXPORT = qw(bounce cont fail);

sub new {
    my ($class, $cont) = @_;
    bless { cont => $cont }, $class;
}

sub cont(&) {
    my ($cont) = @_;
    return PScm::Continuation::Cont->new($cont);
}

sub fail(&) {
    my ($fail) = @_;
    return PScm::Continuation::Fail->new($fail);
}

sub bounce(&) {
    my ($bounce) = @_;
    return PScm::Continuation::Bounce->new($bounce);
}
```

The `cont` sub now creates a **PScm::Continuation::Cont** object instead of a **PScm::Continuation** object. The new `fail` and `bounce` subs are completely analogous.

Additionally, instead of the three new continuation classes sharing a common `Cont()` method to invoke the continuation, `Cont()` has been moved to the **PScm::Continuation::Cont** class, and the **PScm::Continuation::Fail** class has a `Fail()` method. These are both almost identical to the earlier generic `Cont()` method, but expect the correct number of arguments etc. The old `Bounce()` method, which invoked the continuation directly, has just been moved into the **PScm::Continuation::Bounce** class.

16.3.2 Mechanical Transformations of the Interpreter

I don't want to show you those other **PScm::Continuation** derived classes yet, because that would jump the gun on the passing of the new failure continuation around in `Apply()` etc. Instead, now we know what `fail{}` and `bounce{}` do, lets take a look at some examples of how this mechanical rewrite of the interpreter will proceed.

The default `Eval()` method in **PScm::Expr** demonstrates the simplest kind of transformation. The previous version simply called its continuation on `$self`, so by default expressions evaluate to themselves. The new `amb` version takes an extra `$fail` continuation as argument, and passes it along to the original continuation as an extra argument:

```
sub Eval {
    my ($self, $env, $cont, $fail) = @_;
    $cont->Cont($self, $fail);
}
```

Next up, let's take a look at transforming an example method that creates a new continuation. The `PScm::SpecialForm::Let::Apply()` method does that. It extends the current environment with the new bindings for the `let` expression, passing a continuation that will evaluate the body of the `let` in that new environment. The new version for `amb` is not that different. As you can see all the method calls that used to take a single continuation as argument now take an extra `$fail` continuation, and the original continuations themselves now take an extra `$fail` continuation, passing it to any method that now expects it. Otherwise, it's unchanged:

```
013 sub Apply {
014     my ($self, $form, $env, $cont, $fail) = @_;
015
016     my ($symbols, $values, $body) = $self->UnPack($form);
017
018     $env->Extend(
019         $symbols, $values,
020         cont {
021             my ($newenv, $fail) = @_;
022             $body->Eval($newenv, $cont, $fail);
023         },
024         $fail
025     );
026 }
```

Please note however that there are two `$fail` variables here. The first one is passed to `Apply()` as argument on Line 14 and gets passed on as an additional argument to `Extend()` on Line 24. The second `$fail` is argument to the new continuation on Line 21 and is passed on as an additional argument to `Eval()` on Line 22. It is very important that these two `$fail` variables are kept distinct.

Before we finally get around to some code that actually does more than just pass the failure continuation around, let's take a look at a fairly involved use of continuations, and the (still mechanical) transformation that `amb` requires of it. In Section 8.5.1 on page 92 we introduced `PScm::Expr::List::Pair::map_eval()`, which evaluates each component of its list and returns an arrayref of those evaluated components. That method was introduced even earlier in our CPS rewrite in Section 13.6.2 on page 188 and was finally reunited with its original list implementation in Section 13.6.5 on page 206 where it deals with both continuations and true PScheme lists. Here's `PScm::Expr::List::Pair::map_eval()` so far:

```
sub map_eval {
    my ($self, $env, $cont) = @_;

    $self->[FIRST]->Eval(
        $env,
        cont {
            my ($evaluated_first) = @_;
            $self->[REST]->map_eval(
                $env,
                cont {
                    my ($evaluated_rest) = @_;
                    $cont->Cont($self->Cons($evaluated_first,
                                            $evaluated_rest));
                }
            );
        }
    );
}
```

And here it is after the `amb` changes:

```
sub map_eval {
    my ($self, $env, $cont, $fail) = @_;

    $self->[FIRST]->Eval(
        $env,
        cont {
            my ($evaluated_first, $fail) = @_;
            $self->[REST]->map_eval(
                $env,
                cont {
                    my ($evaluated_rest, $fail) = @_;
                    $cont->Cont(
                        $self->Cons($evaluated_first,
                                    $evaluated_rest),
                        $fail
                    );
                },
                $fail
            );
        },
        $fail
    );
}
```

It's a bit longer, but I hope you can see that the only change is that extra `$fail` argument alongside each passed continuation, and as an extra argument to any continuation which is actually called. Note again that it's very important that each continuation actually declares its extra argument. Although the same `$fail` variable name is used throughout, the actual scope of each variable is different, and could easily have a different value. Having said that, this is the main reason that this rewrite is so mechanical.

16.3.3 Remaining Continuation Changes

Now that we've seen examples of how the failure continuation gets passed around, it's safe to return to our **PScm::Continuation** classes and show the details of the various methods therein. Firstly, here's the **PScm::Continuation::Cont** class.

```
034 package PScm::Continuation::Cont;
035 
036 use strict;
037 use warnings;
038 use base qw(PScm::Continuation);
039 
040 BEGIN {
041     *cont = \&PScm::Continuation::cont;
042     *bounce = \&PScm::Continuation::bounce;
043 }
044 
045 sub Apply {
046     my ($self, $form, $env, $cont, $fail) = @_;
047     $form->map_eval(
048         $env,
049         cont {
050             my ($evaluated_args, $fail) = @_;
051             $self->Cont($evaluated_args->first, $fail);
052         },
053         $fail
054     );
055 }
056 
057 sub Cont {
058     my ($self, $arg, $fail) = @_;
059 
060     bounce { $self->{cont}->($arg, $fail) }
061 }
```

We have to manually import the `cont` and `bounce` subroutines from **PScm::Continuation** because they're in the same file (a failure of `use base`.) Then on Lines 45-55 we see the `Apply()` method for continuations (remember `call/cc` presents continuations as functions so they need an `Apply()` method.) `Apply()` is unchanged except for the passing of the extra `$fail` continuation. This means that the failure continuation is kept track of even through the use of `call/cc`.

Lastly the `Cont()` method is similarily unchanged except that it uses `bounce{}` instead of `cont{}` to create a **PScm::Continuation::Bounce** for the trampoline, and of course it has the extra `$fail` continuation to pass on.

PScm::Continuation::Fail is somewhat shorter:

```
064 package PScm::Continuation::Fail;
065 
```

```
use strict;
use warnings;
use base qw(PScm::Continuation);

BEGIN { *bounce = \&PScm::Continuation::bounce; }

sub Fail {
    my ($self) = @_;
    bounce { $self->{cont}->() }
}
```

Again we must manually import the `bounce{}` construct that we need, but then the `Fail()`, method, which takes no arguments, merely returns a `bounce{}` continuation to the trampoline that will invoke the failure continuation with no arguments.

PScm::Continuation::Bounce is even shorter. It's single `Bounce()` method, again with no arguments, directly invokes its stored continuation as it always did.

```
package PScm::Continuation::Bounce;

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

sub Bounce {
    my ($self) = @_;
    $self->{cont}->();
}

1;
```

So back to the rewrite. How about actually doing something with the failure continuation? As I've said, there are only a few places in the interpreter where a new failure continuation is constructed, namely

- in the `Apply()` method for `amb` iteslf;
- in the `Apply()` method for `define`;
- in the `Apply()` method for `set!`;
- in the repl.

These are the only places in the interpreter where the `amb` rewrite is not purely mechanical. We'll go through these cases in the same order, starting with `PScm::SpecialForm::Amb::Apply()`.

16.3.4 `amb` Itself

`amb` is the whole point of this chapter, and so deserves some attention. This special form must evaluate and return its first argument "downstream" when it is called, but if control backtracks to it then it must return its next argument, and so on, until the argument list is exhausted at which point it should invoke the failure continuation that it was originally called with and backtrack further upstream:

```
477 package PScm::SpecialForm::Amb;
478
479 use base qw(PScm::SpecialForm);
480
481 use PScm::Continuation;
482
483 sub Apply {
484     my ($self, $choices, $env, $cont, $fail) = @_;
485     if ($choices->is_pair) {
486         $choices->first->Eval(
487             $env,
488             $cont,
489             fail {
490                 $self->Apply(
491                     $choices->rest,
492                     $env,
493                     $cont,
494                     $fail
495                 )
496             }
497         )
498     } else {
499         $fail->Fail();
500     }
501 }
502
503 1;
```

It's really not that bad. It takes the same arguments as any normal `Apply()` method, including the extra failure continuation. On Line 485 it tests to see if the argument `$choices` (the actual arguments to the `amb` function) is the empty list. If `$choices` is not empty, then on Line 486 it evaluates the first choice, passing the original success continuation `$cont` which will return the result downstream to the caller. But instead of just passing in its argument `$fail` continuation, on Lines 489-496 it passes a new `fail{}` continuation that will, if backtracked to, call `Amb::Apply()` again on the *rest* of the arguments. Note that on Line 494 the new failure continuation passes `amb`'s original failure continuation to `Amb::Apply()`. So if `amb` itself decides to backtrack by calling that, control will pass immediately back to whatever failure continuation was in place before `amb` installed this new one. If on the other hand the new failure continuation is ever invoked downstream of this, it will cause control to proceed back upstream to this occurence of `amb` which then returns its next value back downstream via `Apply()` to the current success continuation.

If the list of arguments is empty, then on Line 499 `Apply()` invokes its original argument `$fail` continuation causing execution to immediately backtrack further upstream.

16.3.5 Changes to `define`

Next, let's take a look at `define`. `define` installs its symbol/value pair in the current environment frame, reguardless of the presence or absence of any previous binding. The `amb` version of define must undo whatever it did if it is backtracked through, so it needs to remember the previous value, if any. Here's the `amb` version of `PScm::SpecialForm::Define::Apply()`:

```
331 sub Apply {
332     my ($self, $form, $env, $cont, $fail) = @_;
333     my ($symbol, $expr) = $form->value;
334     my $old_value = $env->LookUpHere($symbol);
335 
336     $expr->Eval(
337         $env,
338         cont {
339             my ($value, $fail) = @_;
340             $cont->Cont(
341                 $env->Define($symbol, $value),
342                 fail {
343                     if (defined $old_value) {
344                         $env->Assign($symbol, $old_value);
345                     } else {
346                         $env->UnSet($symbol);
347                     }
348                     $fail->Fail();
349                 }
350             );
351         },
352         $fail
353     );
354 }
```

`define` in the previous version evaluated its value part in the current environment, passing a continuation that would call the top environment frame's `Define()` method on the symbol and the result. This new version must additionally keep track of the previous value of the symbol, if any, and arrange that its failure continuation restores that value before backtracking further. Apart from the extra `$fail` argument, the first thing that is new is that on Line 334 it calls a new method `PScm::Env::LookUpHere()` which only looks in the top frame and returns either the value of the argument symbol or `undef`. Then things proceed as normal apart from the extra `$fail` continuation until Lines 342-349 where a replacement `fail{}` continuation is passed to `define`'s original success continuation.

That new `fail{}` continuation checks to see if the `$old_value` is defined. If it is, then it calls `Assign()` on the environment to restore the old value. If it is not defined (there was no previous value) then it must call a new method of **PScm::Env**, `UnSet()`, to remove the binding from the top frame. In either case, it finally returns through the original `$fail` continuation to backtrack further upstream.

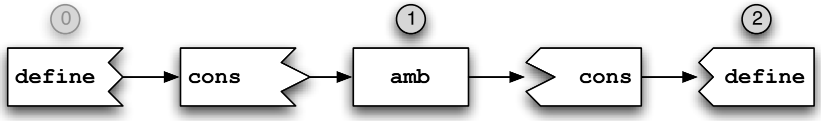
The location of the `fail{}` is quite subtle, in fact an earlier version of this code had a bug that went unnoticed for a considerable time. Consider the following PScheme fragment (assume `x` is already defined):

```
(define x (cons (amb 1 2) x))
```

Obviously, when backtracking, we want the previous value of `x` to be restored before we `cons` the next value from `amb` on to it, otherwise we would be breaking the semantics of chronological backtracking. i.e. if `x` starts out as `(5)`, then after the first time throught it will obviously be `(1 5)`, and the second time through it should be `(2 5)`.

Now, referring to Figure 16.4, think about the order that things happen here. Passing continuations is much like tearing a function into two or more pieces: the first piece is the "head" of the function, before it makes any calls of its own. The remaining pieces are the continuations that it passes to the functions that it calls. This figure omits many details, but you can see that `define` calls `cons` which calls `amb`, then `amb` calls the continuation of `cons` which in turn calls the waiting continuation of `define`. By the way this is another example of CPS being a simplification in that it linearizes control flow.

Figure 16.4: `define` installs a failure continuation *last*



In this figure, "downstream" is left to right and "upstream" is right to left. Additionally the circles represent new failure continuations being created and passed downstream. If control backtracks upstream into this piece of code, it will first encounter the most recently installed, e.g. the *rightmost* failure continuation. You can see that `amb` installs a new failure continuation at 1, and that in order for `define` to have its failure continuation supplant the one set up by `amb` it must be created downstream of `amb`'s. Therefore it is `define`'s *success continuation* that must install the failure continuation, at 2 in the figure.

If instead the initial code on entry to `define` had installed the failure continuation at 0 (by passing it as the last argument to the outermost `Eval`), then backtracking would find `amb`'s failure continuation first, and `define` would not get a chance to undo its effect before `amb` sent its next value downstream again.

That was the bug of course, setting up the failure continuation at 0 instead of 2—it works almost all of the time, unless evaluation of the second argument to `define` or `set!` results in a call to `amb`.

16.3.6 Changes to set!

Next we're going to look at `set!`. `set!` searches the environment for a binding and replaces it, throwing an error if no binding can be found. First of all, here's the new definition for `PScm::SpecialForm::Set::Apply()`:

```
sub Apply {
    my ($self, $form, $env, $cont, $fail) = @_;
    my ($symbol, $expr) = $form->value;
    my $old_value = $env->LookUpNoError($symbol);
    $expr->Eval(
        $env,
        cont {
            my ($val, $fail) = @_;
            my $result = eval { $env->Assign($symbol, $val) };
            if ($@) {
                $self->Error($@, $env);
            } else {
                $cont->Cont(
                    $result,
                    fail {
                        $env->Assign($symbol, $old_value);
                        $fail->Fail();
                    }
                );
            }
        },
        $fail
    );
}
```

This is similar to `PScm::SpecialForm::Define::Apply()` above, but it uses another new method of **PScm::Env**, `LookUpNoError()` to see if the variable being set had a previous value in any frame. Then it proceeds as it did, Passing a continuation to the `Eval()` of the expression that will call `Assign()` on the environment, trapping any error. But then it installs a new failure continuation that will, if invoked, restore the previous value and backtrack further. This new failure continuation just assigns the previous value and backtracks. It need not worry that there was no previous value, since in that case the code in the success continuation would have invoked `Error()`, thus restarting the repl, and the failure continuation would never be backtracked through.

The same subtleties described in `define` apply here: `set!` must pass its new failure continuation to the original success continuation passed to `set!`, rather than to the `eval` of the value to be assigned, in case that evaluation contains an `amb`. The `set!` failure continuation that undoes the mutation must be backtracked through before any `amb` failure continuation.

16.3.7 Changes to `repl()`

It is not without reason that I've kept the changes to the repl until last. This is the most complex part of the `amb` rewrite. If you remember from Section 13.6.2 on page 188 `ReadEvalPrint()` now calls a helper routine `repl()` to do the heavy lifting, and it is `repl()` that we see here undergoing significant change. However there is nothing here that is really new, now that you've seen the mechanics of the rest of the rewrite. It is mostly complicated because the original `repl()` method was already complicated. Here's `repl()` from the previous version of the interpreter:

```
sub repl {
    my ($env, $reader, $outfh) = @_;
    $reader->Read(
        cont {
            my ($expr) = @_;
            $expr->Eval(
                $env,
                cont {
                    my ($result) = @_;
                    $result->Print(
                        $outfh,
                        cont {
                            repl($env, $reader, $outfh);
                        }
                    )
                }
            )
        }
    )
}
```

As I've said before, it's really just `Read()` called with a continuation that calls `Eval()` with a continuation that calls `Print()` with a continuation that calls `repl()` again. As before there are going to be extra failure continuations passed around, but that part of the rewrite is purely mechanical. The additional complications are because `repl()` must additionally install the final upstream failure continuations, and additionally must check if the expression just read is a "?" request to backtrack. Bearing all that in mind it's really not too bad:

```
sub repl {
    my ($env, $reader, $outfh, $fail1) = @_;
    $fail1 ||= fail { __PACKAGE__->Error("no current problem", $env) };
    $reader->Read(
        cont {
            my ($expr, $fail2) = @_;
            $expr->Eval(
                $env,
                cont {
                    my ($result, $fail3) = @_;
                    $result->Print(
                        $outfh,
                        cont {
                            my ($dummy, $fail4) = @_;
                            repl($env, $reader, $outfh, $fail4);
                        },
                        $fail3
                    )
                },
                fail {
                    __PACKAGE__->Error("no more solutions", $env);
                }
109             )
110         },
111         $fail1
112     )
113 }
```

To aid readability somewhat, I've named the various occurences of the failure coninuation separately: `$fail1`, `$fail2` etc. They could all just be called `$fail` without breaking anything, but it would be more confusing.

The `$fail1` argument to `repl()` is optional. Neither `Error()` nor the `new_thread()` call that initially installs the repl on the trampoline bother to pass one. If no failure continuation is passed, then on Line 89 `repl()` defaults it to a call to `Error()` with a "`no current problem`" message.

Then, as before `repl()` calls `Read()` with a continuation that will call `Eval()` etc. `Read()` itself changes slightly however: if it reads a "`?`" it will invoke the current failure continuation.

If the expression read is not a retry request, then everything proceeds as normal, bar the extra failure continuations: `Eval()` is called with a continuation that calls `Print()` with a continuation that calls `repl()` again. Note that on Line 101 the continuation passed to `Print()` calls `repl()` with *its* argument failure continuation `$fail4`, which is how backtracking works when a "`?`" is read subsequently.

One last thing to notice. On Lines 106-108 The failure continuation passed to `Eval()` produces the "`no more solutions`" error, which will be printed if required before `repl()` reinstates the default "`no current problem`" failure.

As mentioned above, `Read()` has changed a little. Here's the new definition:

```
094 sub Read {
095     my ($self, $cont, $fail) = @_;
096     my $res = $self->_read();
097     return undef unless defined $res;
098     if ($res->is_retry) {
099         $fail->Fail();
100     } else {
101         $cont->Cont($res, $fail);
102     }
103 }
```

Before returning its value, `Read()` must first check that the expression it is about to pass to its success continuation is not "?". On Line 98 `Read()` checks to see if the expression returned by `_read()` is a retry request. `is_retry()` is defined to return false in **PScm::Expr**, but **PScm::Expr::Symbol** redefines this to return true if the symbol's `value()` is "?". If it is a retry request, `Read()` invokes its argument failure continuation. At the very start this will be the `"no current problem"` error, so typing "`?`" at a fresh PScheme prompt will produce this error.

That's really all there is to `amb`. The rest of this section joins the dots by showing the support routines that I've glossed over.

16.3.8 Additional Changes

There's not really many of those to describe. If you remember there were a couple of extra methods added to **PScm::Env** to aid `define` and `set!` in undoing their changes. The first of these was `LookUpHere()`:

```
sub LookUpHere {
    my ($self, $symbol) = @_;
    if (exists($self->{bindings}{ $symbol->value })) {
        return $self->{bindings}{ $symbol->value };
    } else {
        return undef;
    }
}
```

`LookUpHere()` just checks the current frame to see if the binding exists. It is called by our new `define` to save any previous value before `define` replaces it.

Next is `LookUpNoError()`:

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

    if (defined(my $ref = $self->_lookup_ref($symbol))) {
        return $$ref;
    } else {
        return undef;
    }
}
```

It uses the existing `_lookup_ref()` method to locate the symbol, either dereferencing and returning the value if it was found, or returning `undef`. `LookUpNoError()` is called by `set!` before assigning a new value to the found variable.

The other addition to **PScm::Env** was an `UnSet()` method which would remove a binding from the environment.

```
sub UnSet {
    my ($self, $symbol) = @_;
    delete $self->{bindings}{ $symbol->value };
}
```

This method just deletes a binding from the current frame. It is only called by `define` when backtracking to remove the setting that `define` added to the current environment frame, so it need not, and should not recurse.

Finally, and most trivially, there is an `is_retry()` method of **PScm::Expr**, so that the continuation passed to `Read()` can ask politely if the expression just read is a request to backtrack ("?"). The base **PScm::Expr** class defines this to be false as a default:

```
sub is_retry { 0 }
```

But **PScm::Expr::Symbol** redefines this to return true if the symbol's `value()` is `"?"`.

```
sub is_retry {
    my ($self) = @_;
    return $self->value eq "?";
}
```

16.4 Support for Testing amb

The examples at the start of this chapter in Section 16.1 on page 249 made use of quite a few support functions of various sorts. Most of those functions could be defined directly in the PScheme language, but a few remaining functions were left to be implemented in the interpreter itself. Specifically, those functions were:

- `and` and `or`. Both are special forms so that arguments do not get evaluated unnecessarily and short-circuit evaluation is possible.
- Numeric inequality tests `>`, `<`, `>=` and `<=`.
- A general equality test `eq?`. This function will work for numbers, symbols, strings and lists (two lists are considered `eq?` if their `car`s are `eq?` and their `cdr`s are `eq?`)[7].

16.4.1 and and or

It is best to make these both special forms, so that they do not evaluate their arguments unnecessarily. In fact they share quite a bit in common with the existing `begin` special form. `begin` evaluates all its arguments in turn, wheras `and` and `or` evaluate each of their arguments until some condition is met. If you remember from Section 13.6.7 on page 211, `PScm::SpecialForm::Begin::Apply()` called out to a helper function `apply_next()` if its argument list was non-empty. What I've done is to create a common abstract base class **PScm::SpecialForm::Sequence** for all of `begin`, `and` and `or`, because they can all share a common `Apply()` method:

```
package PScm::SpecialForm::Sequence;

use base qw(PScm::SpecialForm);

sub Apply {
    my ($self, $form, $env, $cont, $fail) = @_;
    if ($form->is_pair) {
        $self->apply_next($form, $env, $cont, $fail);
    } else {
        $cont->Cont($form, $fail);
    }
}
```

PScm::SpecialForm::Begin inherits from that to get its `Apply()` method, its `apply_next()` is unchanged other than having the extra failure continuation:

[7] This differs from scheme which has separate equality tests for symbols and lists, for efficiency reasons.

```perl
package PScm::SpecialForm::Begin;

use base qw(PScm::SpecialForm::Sequence);
use PScm::Continuation;

sub apply_next {
    my ($self, $form, $env, $cont, $fail) = @_;

    $form->first->Eval(
        $env,
        cont {
            my ($val, $fail) = @_;
            if ($form->rest->is_pair) {
                $self->apply_next($form->rest, $env, $cont, $fail);
            } else {
                $cont->Cont($val, $fail);
            }
        },
        $fail
    );
}
```

PScm::SpecialForm::And reimplements `apply_next()` to return false as soon as an evaluated value is false. If all of the arguments to `and` are true, `and` returns the value of the last argument.

```perl
package PScm::SpecialForm::And;

use base qw(PScm::SpecialForm::Sequence);
use PScm::Continuation;

sub apply_next {
    my ($self, $form, $env, $cont, $fail) = @_;

    $form->first->Eval(
        $env,
        cont {
            my ($val, $fail) = @_;
            if ($form->rest->is_pair) {
                if ($val->isTrue) {
                    $self->apply_next($form->rest, $env, $cont, $fail);
                } else {
                    $cont->Cont($val, $fail);
                }
            } else {
                $cont->Cont($val, $fail);
            }
        },
        $fail
    );
}
```

PScm::SpecialForm::Or behaves similarily. it evaluates each of its arguments until one of them is true, in which case it returns that result. If all of its arguments are false, it returns false.

```
package PScm::SpecialForm::Or;

use base qw(PScm::SpecialForm::Sequence);
use PScm::Continuation;

sub apply_next {
    my ($self, $form, $env, $cont, $fail) = @_;

    $form->first->Eval(
        $env,
        cont {
            my ($val, $fail) = @_;
            if ($form->rest->is_pair) {
                if ($val->isTrue) {
                    $cont->Cont($val, $fail);
                } else {
                    $self->apply_next($form->rest, $env, $cont, $fail);
                }
            } else {
                $cont->Cont($val, $fail);
            }
        },
        $fail
    );
}
```

16.4.2 Numeric Inequality Tests

The next thing we'll need is a numeric inequality test ">". The full standard set of numeric inequality tests "<", ">", "<=", and ">=" now exist as primitives in the interpreter. They are all under **PScm::Primitive**, in fact they all descend from a subclass of that called **PScm::Primitive::Compare** which provides a common _apply() method:

```
package PScm::Primitive::Compare;

use base qw(PScm::Primitive);

sub _apply {
    my ($self, @numbers) = @_;
    $self->_check_type($numbers[0], 'PScm::Expr::Number') if @numbers;
166     while (@numbers > 1) {
167         my $number = shift @numbers;
168         $self->_check_type($numbers[0], 'PScm::Expr::Number');
169         return PScm::Expr::Number->new(0)
170             unless $self->_compare($number, $numbers[0]);
171     }
172     return PScm::Expr::Number->new(1);
173 }
```

So they all take an arbitrary number of arguments like the arithmetic primitives. For example `(<= 2 3 3 4)` is true because each argument is "`<=`" the next argument. `_apply()` iterates over its arguments, checking each one is a number and calling a separate `_compare()` method on each pair. The `_compare()` method called on Line 170 is implemented separately by each of **PScm::Primitive::Lt**, **PScm::Primitive::Gt**, **PScm::Primitive::Le** and **PScm::Primitive::Ge**. They all go exactly the same way, so for example here's **PScm::Primitive::Lt**:

```
package PScm::Primitive::Lt;

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

sub _compare {
    my ($self, $first, $second) = @_;
    return $first->value < $second->value;
}
```

Only the actual comparison operator differs between the implementations.

16.4.3 `eq?`

Finally `eq?`. The `eq?` implementation is a bit more interesting. It can be used to compare any PScheme data types that inherit from **PScm::Expr**. Equality is a relative term however. For instance, unlike Perl, a string and a number will never be considered equal, however two lists with the same content *are* considered equal. Here's the new **PScm::Primitive::Eq** class:

```
package PScm::Primitive::Eq;

use base qw(PScm::Primitive);

sub _apply {
    my ($self, @things) = @_;
    while (@things > 1) {
        my $thing = shift @things;
        return PScm::Expr::Number->new(0) unless $thing->Eq($things[0]);
    }
    return PScm::Expr::Number->new(1);
}
```

As you can see, like the inequality tests above, it will take an arbitrary number of arguments. `Apply()` keeps comparing adjacent arguments by calling their `Eq()` method until a test fails, or all tests pass. The `Eq()` method is defined differently for various types of **PScm::Expr**. A default method in the base **PScm::Expr** just compares object identity:

```
sub Eq {
    my ($self, $other) = @_;
    return $self == $other;
}
```

This means that, for example, two functions with the same arguments, env and body would still not be considered equal. This could be fixed, but I'm not sure it's worth it.

Anyway **PScm::Expr::Atom** overrides this `Eq()` method to do a string comparison on the (scalar) values of the two objects, first checking that the two objects are of the same type. This is good enough for strings, numbers and symbols:

```
sub Eq {
    my ($self, $other) = @_;
    return 0 unless $other->isa(ref($self));
    return $self->value eq $other->value;
}
```

`PScm::Expr::List::Pair::Eq()` is more interesting. Firstly it does a quick check for object identity, that will save unnecessary recursion if the two objects are actually the same object. Then it checks that the object is a list, and finally it recursively calls itself on both `first()` and `rest()` to complete the test:

```
sub Eq {
    my ($self, $other) = @_;
    return 1 if $self == $other;
    return 0 unless $other->is_pair;
    return $self->[FIRST]->Eq($other->[FIRST]) &&
           $self->[REST]->Eq($other->[REST]);
}
```

Last of the `Eq()` methods is in **PScm::Expr::List::Null**. This method returns true only if the other object is also a **PScm::Expr::List::Null**, since null is only equal to null:

```
sub Eq {
    my ($self, $other) = @_;
    return $other->is_null;
}
```

16.4.4 Wiring it up

Finally, here's the additional methods wired in to `ReadEvalPrint()`. You can also see that on Line 67 the `new_thread()` routine installs a `bounce{}` continuation that starts the repl. That continuation doesn't pass a failure continuation to `repl()`, so `repl()` will default that to the **`Error: no current problem`** error.

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

    $outfh ||= new FileHandle(">-");
    my $reader      = new PScm::Read($infh);
    my $initial_env;
    $initial_env = new PScm::Env(
        let          => new PScm::SpecialForm::Let(),
        ’*’          => new PScm::Primitive::Multiply(),
        ’-’          => new PScm::Primitive::Subtract(),
        ’+’          => new PScm::Primitive::Add(),
        if           => new PScm::SpecialForm::If(),
        lambda       => new PScm::SpecialForm::Lambda(),
        list         => new PScm::Primitive::List(),
        car          => new PScm::Primitive::Car(),
        cdr          => new PScm::Primitive::Cdr(),
        cons         => new PScm::Primitive::Cons(),
        letrec       => new PScm::SpecialForm::LetRec(),
        ’let*’       => new PScm::SpecialForm::LetStar(),
        eval         => new PScm::SpecialForm::Eval(),
        macro        => new PScm::SpecialForm::Macro(),
        quote        => new PScm::SpecialForm::Quote(),
        ’set!’       => new PScm::SpecialForm::Set(),
        begin        => new PScm::SpecialForm::Begin(),
        define       => new PScm::SpecialForm::Define(),
        ’make-class’ => new PScm::SpecialForm::MakeClass(),
        ’call/cc’    => new PScm::SpecialForm::CallCC(),
        print        => new PScm::SpecialForm::Print($outfh),
        spawn        => new PScm::SpecialForm::Spawn(),
        exit         => new PScm::SpecialForm::Exit(),
        error        => new PScm::SpecialForm::Error(
                           $outfh,
                           bounce { repl($initial_env, $reader, $outfh) }
                        ),
        amb          => new PScm::SpecialForm::Amb(),
        ’eq?’        => new PScm::Primitive::Eq(),
        ’>’          => new PScm::Primitive::Gt(),
        ’<’          => new PScm::Primitive::Lt(),
        ’>=’         => new PScm::Primitive::Ge(),
        ’<=’         => new PScm::Primitive::Le(),
        and          => new PScm::SpecialForm::And(),
        or           => new PScm::SpecialForm::Or(),
    );

    $initial_env->Define(
        PScm::Expr::Symbol->new("root"),
        PScm::Class::Root->new($initial_env)
    );
    __PACKAGE__->new_thread(bounce { repl($initial_env, $reader, $outfh) });
    trampoline();
}
```

16.5 Summary and Directions

`amb` demonstrates a very simple but unfortunately inefficient mechanism for embedding so-called "non-deterministic" programming into an otherwise procedural language. The term "non-deterministic" means that the control flow through the interpreter is not, on the surface, determined solely by a single set of conditions at a particular point: there are choices available.

The main reason that `amb` is so inefficient is that it uses what is called "chronological backtracking". It really is as though the interpreter "winds back the clock" when a condition fails, going back to a prior moment in time and retrying a different choice at that point. Of course this is an illusion, but a useful and simple analogy to use. However chronological backtracking is a brute-force approach to search, since all possible solutions are attempted and in a typical search the vast majority of these possible solutions are discarded (remember the `distinct` function in the first solution to the "Liars" puzzle.)

There are alternatives to chronological backtracking. They are beyond the scope of this chapter, but to give you some idea, the most popular and successful of these alternatives is known as *dependancy-directed backtracking*. This technique gains more control over the next choice made by noting the reason for the previous failure and avoiding choices that would again produce the same failure. For example, in the "Liars" puzzle from Section 16.1 on page 249, if one of the conditions fails, for instance `(xor (eq? kitty 2) (eq?  betty 3))`, it is only the choices of `kitty` and `betty` that cause the failure, but naiive chronological backtracking would try many other alternatives before actually arriving at any choices that affect that failure. Dependancy-directed backtracking on the other hand, would backtrack directly to those choice points.

While simple enough to describe, dependancy-directed backtracking is by no means easy to implement in practice.

I mentioned in the introduction to this chapter that `amb` was one step towards a logic programming language. As I've just described some sort of dependancy-directed backtracking is a necessary second step for any production-quality language. The third component to logic programming is a very special and interesting technique called *unification*. Ignoring any efficiency concerns that `amb` might have, we will nonetheless be looking at unification, and how it facilitates logic programming, in the next chapter.

16.5.1 An Alternative Implementation

Before finally moving on from `amb` it is worth considering a slightly different, and potentially more powerful approach to its implementation. There is a design pattern called *Parameter Object* which goes something like this: "If you are always passing the same set of parameters around from method to method, then wrap those parameters in a single object and pass it as a single parameter." Now the `$cont` and `$fail` parameters of `amb` are perfect candidates for the application of this pattern. The main reasons I haven't done it are 1. I wanted to keep the code explicit, and 2. the `cont{}` and `fail{}` constructs would have to be made a lot cleverer in order to manipulate an existing composite continuation parameter.

However if we had gone for the Parameter Object pattern, there would have been a very interesting payoff: if you could have two continuations, why not three? four? etc. Why might you want such a thing? Well, imagine a language where all the control flow (`for` and `while` loops, `break`, `continue`, `return` etc.) were implemented by continuations. Then a `for` loop would install `break` and `continue` continuations (and uninstall them again), a subroutine would install a `return` continuation, etc.

Even more exciting, consider an *environment* of continuations as a parameter object. Then for example nested `for` loops would push and pop their `break` and `continue` continuations. It would then be relatively easy to `break` or `continue` or `return` to an arbitrary containing point.

These alternatives, while exciting, are left as an open exercise should you wish to pursue them.

16.6 Tests

The first set of tests in Listing 16.7.1 on the next page tries out or equality and inequality operators. It's nice to know they all work as expected.

The next set of tests in Listing 16.7.2 on page 289 exercizes the new repl itself ensuring that the appropriate error messages are produced after various requests for backtracking, and that the repl recovers gracefully in all situations.

The last set of tests, in Listing 16.7.3 on page 290 gives amb a thorough workout. It tests most of the examples that we have seen in this chapter, plus a few more for good measure. Additionally, it tests that `set!` and `define` do in fact undo their assignments in the face of backtracking

16.7 Listings

16.7.1 t/PScm_Compare.t

```
use strict;
use warnings;
use Test::More;
use lib ’./t/lib’;
use PScm::Test tests => 38;

BEGIN { use_ok(’PScm’) }

eval_ok(’(eq? 1 1)’, ’1’, ’eq numbers’);
eval_ok(’(eq? 1 2)’, ’0’, ’neq numbers’);
eval_ok(’(eq? 1 "1")’, ’0’, ’neq numbers and strings’);
eval_ok("(eq? 1 ’a)", ’0’, ’neq numbers and symbols’);
eval_ok("(eq? 1 (list 1))", ’0’, ’neq numbers and lists’);

eval_ok(’(eq? "a" "a")’, ’1’, ’eq strings’);
eval_ok(’(eq? "a" "b")’, ’0’, ’neq strings’);
eval_ok(’(eq? "1" 1)’, ’0’, ’neq strings and numbers’);
eval_ok(’(eq? "a" \’a)’, ’0’, ’neq strings and symbols’);
eval_ok(’(eq? "a" (list "a"))’, ’0’, ’neq strings and lists’);

eval_ok("(eq? ’a ’a)", ’1’, ’eq symbols’);
eval_ok("(eq? ’a ’b)", ’0’, ’neq symbols’);
eval_ok("(eq? ’a 1)", ’0’, ’neq symbols and numbers’);
eval_ok(’(eq? \’a "a")’, ’0’, ’neq symbols and strings’);
eval_ok("(eq? ’a (list ’a))", ’0’, ’neq symbols and lists’);

eval_ok("(eq? (list 1 2) (list 1 2))", ’1’, ’eq lists’);
eval_ok("(eq? (list 1 2) (list 1 2 3))", ’0’, ’neq lists’);
eval_ok("(eq? (list 1) 1)", ’0’, ’neq lists and numbers’);
eval_ok(’(eq? (list "a") "a")’, ’0’, ’neq lists and strings’);
eval_ok("(eq? (list ’a) ’a)", ’0’, ’neq lists and symbols’);

eval_ok("(eq? () ())", ’1’, ’eq empty lists’);
eval_ok("(eq? () (list 1))", ’0’, ’neq empty lists’);
eval_ok("(eq? 1 1 1 1)", ’1’, ’eq multiple arguments’);
eval_ok("(eq? 1 1 1 2)", ’0’, ’neq multiple arguments’);

eval_ok("(< 1 2 3 4)", ’1’, ’< multiple arguments’);
eval_ok("(< 1 2 3 3)", ’0’, ’!< multiple arguments’);

eval_ok("(<= 1 2 3 3)", ’1’, ’<= multiple arguments’);
eval_ok("(<= 1 2 3 2)", ’0’, ’!<= multiple arguments’);

eval_ok("(> 4 3 2 1)", ’1’, ’> multiple arguments’);
eval_ok("(> 4 3 2 2)", ’0’, ’!> multiple arguments’);

eval_ok("(>= 4 3 2 2)", ’1’, ’>= multiple arguments’);
eval_ok("(>= 4 3 2 3)", ’0’, ’!>= multiple arguments’);

eval_ok("(and 1 2 3)", "3", ’and success’);
eval_ok("(and 1 2 () 3)", "()", ’and failure’);

eval_ok("(or 1 2 3)", "1", ’or success’);
eval_ok("(or 0 0 3 0)", "3", ’or success [2]’);
eval_ok("(or 0 0 0 0)", "0", ’or failure’);

# vim: ft=perl
```

16.7.2 t/AMB_repl.t

```
use strict;
use warnings;
use Test::More;
use lib ’./t/lib’;
use PScm::Test tests => 2;

BEGIN { use_ok(’PScm’) }

eval_ok(<<EOT, <<EOR, ’sequential repl and amb’);
1
? ?
+1
(amb ’x ’y ’z)
? ? ? ?
(list (amb 1 2) (amb 5 6))
? ? ?
(list (amb 1 2) (amb 5 6))
(list (amb 1 2) (amb 5 6))
EOT
1
Error: no more solutions
Error: no current problem
1
x
y
z
Error: no more solutions
Error: no current problem
(1 5)
(1 6)
(2 5)
(2 6)
(1 5)
(1 5)
EOR

# vim: ft=perl
```

16.7.3 t/AMB_amb.t

```
use strict;
use warnings;
use Test::More;
use lib ’./t/lib’;
use PScm::Test tests => 8;

BEGIN { use_ok(’PScm’) }

my $prereqs = <<EOT;
(define require
  (lambda (x)
    (if x x (amb))))

(define member?
  (lambda (item lst)
    (if lst
        (if (eq? item (car lst))
            1
            (member? item (cdr lst)))
        0)))

(define distinct?
  (lambda (lst)
    (if lst
        (if (member? (car lst)
                     (cdr lst))
            0
            (distinct? (cdr lst)))
        1)))

(define one-of
  (lambda (lst)
    (begin
        (require lst)
        (amb (car lst) (one-of (cdr lst))))))

(define exclude
  (lambda (items lst)
    (if lst
        (if (member? (car lst) items)
            (exclude items (cdr lst))
            (cons (car lst)
                  (exclude items (cdr lst))))
        ())))

(define abs
  (lambda (x)
    (if (< x 0)
        (- x)
        x)))

```

```
(define not
  (lambda (x)
    (if x 0 1)))

(define xor
  (lambda (x y)
    (or (and x (not y))
        (and y (not x)))))

(define difference
  (lambda (a b)
    (abs (- a b))))

(define divisible-by
  (lambda (n)
    (lambda (v)
      (begin
        (define test
          (lambda (o)
            (if (eq? o v)
                1
                (if (> o v)
                    0
                    (test (+ o n))))))
        (test 0)))))

(define even?
  (lambda (a)
    ((divisible-by 2) a)))
EOT

my $prereqs_output = <<EOT;
require
member?
distinct?
one-of
exclude
abs
not
xor
difference
divisible-by
even?
EOT

$prereqs_output =~ s/\n$//s;


eval_ok(<<EOT, <<EOR, ’even?’);
$prereqs
(define test
  (lambda ()
    (let ((x (amb 1 2 3 4 5)))
      (begin
        (require (even? x))
        x))))
(test)
?
?
EOT
$prereqs_output
test
2
4
Error: no more solutions
EOR

eval_ok(<<EOT, <<EOR, 'Barrels of Fun');
$prereqs
(define some-of
  (lambda (lst)
    (begin
      (require lst)
      (amb (list (car lst))
           (some-of (cdr lst))
           (cons (car lst)
                 (some-of (cdr lst)))))))

(define sum
  (lambda (lst)
    (if lst
      (+ (car lst)
         (sum (cdr lst)))
      0)))

(define barrels-of-fun
  (lambda ()
    (let* ((barrels (list 30 32 36 38 40 62))
           (beer (one-of barrels))
           (wine (exclude (list beer) barrels))
           (barrel1 (one-of wine))
           (barrel2 (one-of (exclude (list barrel1) wine)))
           (barrels (some-of (exclude (list barrel1 barrel2) wine))))
      (begin
        (require (eq? (* 2 (+ barrel1 barrel2))
                      (sum barrels)))
        beer))))
(barrels-of-fun)
EOT
$prereqs_output
some-of
sum
barrels-of-fun
40
```

```
EOR

# Baker, Cooper, Fletcher, Miller and Smith live on different
# floors of a five-storey building. Baker does not live on the
# top floor. Cooper does not live on the bottom floor. Fletcher
# does not live on the top or the bottom floor. Miller lives
# on a higher floor than Cooper. Smith does not live on a
# floor adjacent to Fletcher's. Fletcher does not live on a floor
# adjacent to Cooper's. Where does everyone live?

eval_ok(<<EOT, <<EOR, 'amb');
$prereqs

(define multiple-dwelling
  (lambda ()
    (let* ((baker (one-of (list 1 2 3 4)))
           (cooper (one-of (exclude (list baker) (list 2 3 4 5))))
           (fletcher (one-of (exclude (list baker cooper) (list 2 3 4))))
           (miller (one-of (exclude (list baker cooper fletcher)
                                     (list 1 2 3 4 5))))
           (smith (car (exclude (list baker cooper fletcher miller)
                                 (list 1 2 3 4 5)))))
       (begin
         (require (> miller cooper))
         (require (not (eq? (difference smith fletcher) 1)))
         (require (not (eq? (difference cooper fletcher) 1)))
         (list (list 'baker baker)
               (list 'cooper cooper)
               (list 'fletcher fletcher)
               (list 'miller miller)
               (list 'smith smith))))))

(multiple-dwelling)
EOT
$prereqs_output
multiple-dwelling
((baker 3) (cooper 2) (fletcher 4) (miller 5) (smith 1))
EOR

eval_ok(<<EOF, <<EOR, 'Liars (optimized)');
$prereqs
(define liars
  (lambda ()
    (let* ((betty (amb 1 2 3 4 5))
           (ethel (one-of (exclude (list betty)
                                    (list 1 2 3 4 5))))
           (joan (one-of (exclude (list betty ethel)
                                   (list 1 2 3 4 5))))
           (kitty (one-of (exclude (list betty ethel joan)
                                    (list 1 2 3 4 5))))
           (mary (car (exclude (list betty ethel joan kitty)
                                (list 1 2 3 4 5)))))
         (begin
           (require (xor (eq? kitty 2) (eq? betty 3)))
           (require (xor (eq? ethel 1) (eq? joan 2)))
           (require (xor (eq? joan 3) (eq? ethel 5)))
           (require (xor (eq? kitty 2) (eq? mary 4)))
           (require (xor (eq? mary 4) (eq? betty 1)))
           '((betty ,betty)
             (ethel ,ethel)
             (joan ,joan)
             (kitty ,kitty)
             (mary ,mary))))))
(liars)
?
EOF
$prereqs_output
liars
((betty 3) (ethel 5) (joan 2) (kitty 1) (mary 4))
Error: no more solutions
EOR

eval_ok(<<EOF, <<EOR, 'set! backtracking');
(let ((x 1))
     (let ((y (amb 'a 'b)))
          (begin
            (print (list 'x x))
            (set! x 2)
            (print (list 'x x))
            y)))
?
EOF
(x 1)
(x 2)
a
(x 1)
(x 2)
b
EOR

eval_ok(<<EOF, <<EOR, 'define backtracking');
(define x 1)
(let ((y (amb 'a 'b)))
     (begin
       (print (list 'x x))
       (define x 2)
       (print (list 'x x))
       y))
?
EOF
x
(x 1)
(x 2)
a
```

```
(x 1)
(x 2)
b
EOR

eval_ok(<<EOT, <<EOR, ’parsing’);
$prereqs

(define proper-nouns ’(john paul))
(define nouns ’(car garage))
(define auxilliaries ’(will has))
(define verbs ’(put))
(define articles ’(the a his))
(define prepositions ’(in to with))
(define degrees ’(very quite))
(define adjectives ’(red green old new))

(define parse-sentance
  (lambda ()
    (amb (list (parse-noun-phrase)
               (parse-word auxilliaries)
               (parse-verb-phrase))
         (list (parse-noun-phrase)
               (parse-verb-phrase)))))

(define parse-noun-phrase
  (lambda ()
    (amb (parse-word proper-nouns)
         (list (parse-word articles)
               (parse-adj-phrase)))))

(define parse-adj-phrase
  (lambda ()
    (amb (list (parse-deg-phrase)
               (parse-adj-phrase))
         (parse-word nouns))))

(define parse-deg-phrase
  (lambda ()
    (amb (list (parse-word degrees)
               (parse-deg-phrase))
         (parse-word adjectives))))

(define parse-verb-phrase
  (lambda ()
    (list (parse-word verbs)
          (parse-noun-phrase)
          (parse-prep-phrase))))

(define parse-prep-phrase
  (lambda ()
    (list (parse-word prepositions)
          (parse-noun-phrase))))

(define parse-word
  (lambda (words)
    (begin
      (require *unparsed*)
      (require (member? (car *unparsed*) words))
      (let ((found-word (car *unparsed*)))
        (begin
          (set! *unparsed* (cdr *unparsed*))
          found-word)))))

(define *unparsed* ())

(define parse
  (lambda (input)
    (begin
      (set! *unparsed* input)
      (let ((sentance (parse-sentance)))
        (begin
          (require (not *unparsed*))
          sentance)))))

(parse '(john will put his car in the garage))
(parse '(paul put a car in his garage))
(parse '(paul has put a very very old car in his quite new red garage))
EOT
$prereqs_output
proper-nouns
nouns
auxilliaries
verbs
articles
prepositions
degrees
adjectives
parse-sentance
parse-noun-phrase
parse-adj-phrase
parse-deg-phrase
parse-verb-phrase
parse-prep-phrase
parse-word
*unparsed*
parse
(john will (put (his car) (in (the garage))))
(paul (put (a car) (in (his garage))))
(paul has (put (a ((very (very old)) car)) (in (his ((quite new) (red garage))))))
EOR

# vim: ft=perl
```

Full source code for this version of the interpreter is available at
`http://billhails.net/Book/releases/PScm-0.1.12.tgz`

Chapter 17

Unification and Logic Programming

This chapter first gives an example of the sort of things that logic programming is capable of, then gets on with the implementation by introducing the concept of *pattern matching*. Then it explores a generalization of pattern matching called *unification*. Having implemented unification and other more specialized support routines in the interpreter core, we then proceed, with the help of `amb` from Chapter 16 on page 249, to build a logic programming system in the PScheme language itself. The advantages of doing this in a language which has continuation-passing and backtracking built in should become very apparent by the end of the chapter. This implementation is based on the one given in [7, pp295–300], but made a little easier by using `amb`.

17.1 Logic Programming Examples

Logic programs consist of a database of known facts about a problem, and then typically a query or queries that interrogate this database.

17.1.1 Mary and John

Here is a very simple database of facts about a problem domain.

```
(define the-rules
  '(((mary likes wine))
    ((mary likes cheese))
    ((john likes beer))
    ((john likes wine))
    ((john likes chips))
    ((person mary))
    ((person john))))
```

Our database is called `the-rules` and it is a list of statements of various sorts. This database lists facts about what `mary` and `john` like, and also the facts that both `mary` and `john` are people. There is nothing special about the structure of these individual facts, just as long as we remain consistent in their use, and write queries that interrogate the database appropriately. In other words it is we, the programmers, who decide the "meaning" of `((mary likes wine))`: the system attaches no particular significance to that structure in itself; in particular `likes` is not a keyword or an operator of the logic programming system, it is just a symbol that I have chosen to use.

In our logic programming implementation, we will use symbols with initial capital letters as *pattern variables* which can match parts of the database. So given the database above, the system can respond to a query such as `(mary likes X)` with the facts `(mary likes wine)` and `(mary likes cheese)`, and can respond to the question `(person X)` with the facts `(person mary)` and `(person john)`. Of course there is nothing here that a simple SQL query could not do, so let's make things more interesting by adding a *rule* to the facts.

This rule states "mary likes anyone who likes chips." This is written as:

```
((mary likes X) (person X) (X likes chips))
```

You can read this as "`mary likes X` *if* `person X` *and* `X likes chips`."

Rules all have this general form. The first statement in the rule is true if all of the other statements in the rule are true or can be proved to be true. The first statement is called the *head* of the rule, and the remaining statements are called the *body* of the rule.

Looked at in this way, bare facts are just rules with no body: they are true in themselves because there is nothing left to prove. This explains the apparently redundant extra parentheses around each fact in our example database.

Given our extended database, which now looks like this:

```
(define the-rules
  '(((mary likes wine))
    ((mary likes cheese))
    ((john likes beer))
    ((john likes wine))
    ((john likes chips))
    ((person mary))
    ((person john))
    ((mary likes X) (person X) (X likes chips)))
```

the system can answer the question `(mary likes john)` in the affirmative, and furthermore, when prompted to list all the things that mary likes with `(mary likes X)`, john will be among the results:

```
> (prove '((mary likes X)))
> ?
((mary likes wine))
> ?
((mary likes cheese))
> ?
((mary likes john))
```

Also note that the statement to be proved has those apparently redundant extra braces too. This is because the system can be asked to prove any number of things at once, for example:

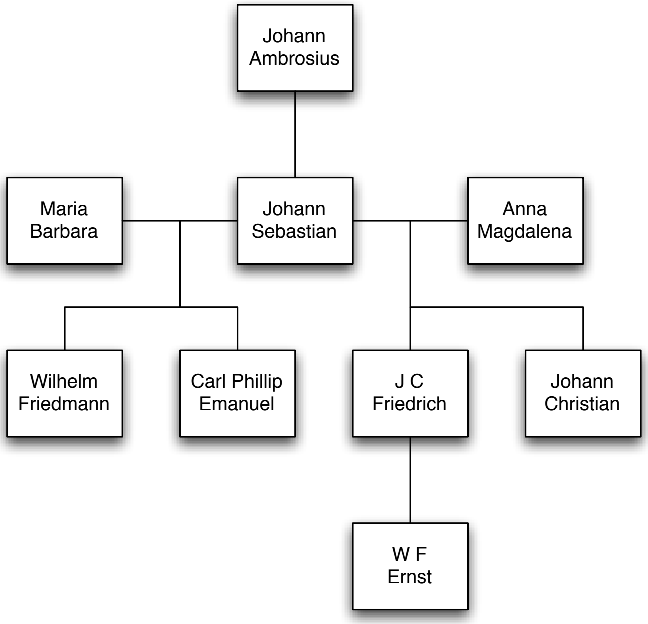
```
> (prove '((mary likes X) (X likes beer)))
((mary likes john) (john likes beer))
> ?
Error: no more solutions
```

This can be read as "prove `mary likes X` *and* `X likes beer`." The query only succeeds if all of the components succeed, so it is just like the body of a rule in this respect.

17.1.2 J. S. Bach and Family

This type of programming can also be used for recursive search. Figure 17.1 shows a small snippet of the Bach family tree, with old J. S. at the centre.

Figure 17.1: Bach's Family Tree



We can translate that into the following facts:

```
(((johann ambrosius) father-of (j s)))
(((j c friedrich) father-of (w f ernst)))
(((j s) (anna magdalena) parents-of (j c friedrich)))
(((j s) (anna magdalena) parents-of (johann christian)))
(((j s) (maria barbara) parents-of (wilhelm friedmann)))
(((j s) (maria barbara) parents-of (c p e)))
```

Now, we have a mixture of facts about fathers alone, and parents where the mother is known. But we can rectify that with a few extra rules as follows:

```
((X father-of Y) (X _ parents-of Y))
((X mother-of Y) (_ X parents-of Y))
((X parent-of Y) (X father-of Y))
((X parent-of Y) (X mother-of Y))
```

That underscore is a special pattern variable that always matches *anything*. The first rule then says "`X` is the father of `Y` if `X` and *anybody* are parents of `Y`" (the `parents-of` facts always list the father first.) Likewise the second rule says "`X` is the mother of `Y` if *anybody* and `X` are parents of `Y`." The last pair of rules (actually really one rule in two parts) says that "X is a parent of Y if X is the father of Y or X is the mother of Y."

So we have a general way of expressing both and and or in our rules: and is expressed by adjacent statements in the body of a single rule, while or is expressed as alternative versions of the same rule.

Now that we have a general parent-of relation, we can add a recursive ancestor-of relationship as follows:

```
((X ancestor-of Y) (X parent-of Y))
((X ancestor-of Y) (X parent-of Z) (Z ancestor-of Y))
```

This says "X is an ancestor of Y if X is a parent of Y or X is a parent of some Z and Z is an ancestor of Y." Given that definition, and the rest of the facts and rules, we can start to ask interesting questions of the system:

```
> (prove '((X ancestor-of (w f ernst))))
(((j c friedrich) ancestor-of (w f ernst)))
> ?
(((johann ambrosius) ancestor-of (w f ernst)))
> ?
(((j s) ancestor-of (w f ernst)))
> ?
(((anna magdalena) ancestor-of (w f ernst)))
> ?
Error: no more solutions
```

We can add yet more rules to this database. For example X and Y are siblings if X and Y have the same parent:

```
((X sibling-of Y) (Z parent-of X)
                  (Z parent-of Y)
                  (require (not (eq? 'X 'Y))))
```

If we don't require that X and Y are not equal then this rule would think that old J. S. was his own brother. Given that, we can ask:

```
> (prove '(((c p e) sibling-of X)))
(((c p e) sibling-of (johann christian)))
```

etc.

17.1.3 Appending Lists

In case you're starting to think that logic programming is just some sort of glorified database lookup, here's a more meaty example. Consider the problem of appending two lists. You can write an append function in PScheme very easily[1] as:

[1] Of course you can define it even more easily in Perl as (@a, @b) but that's only because Perl already *has* an append operation, and so does any complete scheme implementation.

```
(define append
  (lambda (a b)
    (if a
        (cons (car a)
              (append (cdr a)
                      b))
        b)))
```

It walks to the end of the list a, then at that point returns the list b, and as the recursion unwinds it builds a copy of a prepended to b. So for example:

```
> (append '(a b) '(c d))
(a b c d)
```

This definition is useful enough in itself, but logic programming allows a much more flexible definition of append as follows:

```
(define the-rules
  (list '((append () Y Y))
        '((append (A . X) Y (A . Z)) (append X Y Z))))
```

Note that this uses the PScheme dotted pair notation introduced in Section 8.4.1 on page 89. So the expression (A . X) refers to a list who's car is A and who's cdr is X.

The first rule says that the result of appending something to the empty list is just that something. The second rule says that you can join (A . X) and Y to make (A . Z) if you can join X and Y to make Z.

Why is that more powerful than the PScheme append? Because we can ask lots of questions of it. Not only can we ask "what do we get if we append (a b) and (c d)?":

```
> (prove '((append (a b) (c d) X)))
((append (a b) (c d) (a b c d)))
```

We can also ask "what do we need to append to (a) to get (a b c d e)?":

```
> (prove '((append (a) X (a b c d e))))
((append (a) (b c d e) (a b c d e)))
```

And even "what can we append together to make (a b c d)?":

```
> (prove '((append X Y (a b c d))))
((append () (a b c d) (a b c d)))
> ?
((append (a) (b c d) (a b c d)))
> ?
((append (a b) (c d) (a b c d)))
> ?
((append (a b c) (d) (a b c d)))
> ?
((append (a b c d) () (a b c d)))
> ?
Error: no more solutions
```

This idea is absolutely core to the concept of logic programming. A rule that states that "A and B make C" is equally capable of describing "what do I need to make C?" provided C has a value when the question is asked. It is as if the relationship described by the rule can be inspected from many different angles when seeking a solution to a problem.

Back to `append`. The rules for `append` can of course be made available to other rules, for example (peeking ahead a bit)

```
((sentence S) (append NP VP S) (noun-phrase NP) (verb-phrase VP))
```

Says `S` is a sentence if you can append `NP` and `VP` to make `S`, and `NP` is a noun phrase, and `VP` is a verb phrase. Rules for `noun-phrase` and `verb-phrase` would be very similar, and rules for individual words would just be of the form `((noun (garage)))` etc.

17.1.4 Factorial Again

The above examples have already demonstrated that our database of rules can be recursive, so how about the king of recursive functions, our old friend the factorial function? Here it is recast into a pair of rules:

```
(define the-rules
  (list
     '((factorial 0 1))
     '((factorial N X) (T is (- N 1)) (factorial T U) (X is (* N U)))))
```

This isn't as bad as it might first look. The first rule is the bare fact that the factorial of `0` is `1`. The second rule says that the factorial of `N` is `X` *if* `T` is `N - 1` *and* the factorial of `T` is `U` *and* `X` is `N * U`. The special infix `is` operator forces arithmetic evaluation of its right hand side, then requires that its left hand side is the same as the result of that evaluation.

given the above we can calculate factorials:

```
> (prove '((factorial 10 X)))
((factorial 10 3628800))
```

However there is a limitation here. Because of the unidirectional nature of that `is` operator, we cannot ask "what number can we apply `factorial` to to get `x`":

```
>(prove '((factorial X 3628800)))
Error: no more solutions
```

So it goes.

17.1.5 Symbolic Differentiation

If you're still not impressed, how about a more difficult problem? I hope you don't mind a little maths. This next example states some of the rules of symbolic differentiation then asks the system to work out the differential of an equation. The rules we'll be using are:

- The derivative of x in x is 1.
- The derivative of any constant number in x is 0.

- The derivative of x^{n} in x is $n \times x^{\mathrm{n\text{-}1}}$.
- The derivative of $f + g$ in x is $df + dg$ if the derivative of f in x is df and the derivative of g in x is dg.
- The derivative of $f - g$ in x is $df - dg$ if the derivative of f in x is df and the derivative of g in x is dg.
- The derivative of $f \times g$ in x is $f \times dg + g \times df$ if the derivative of f in x is df and the derivative of g in x is dg.
- The derivative of $1/f$ in x is $-df/f^2$ if the derivative of f in x is df.
- The derivative of f/g in x is $(g \times df - f \times dg)/f^2$ if the derivative of f in x is df and the derivative of g in x is dg.

If you don't remember the maths from school, just sit back and enjoy the ride. These rules can translate directly into our logic system as:

```
(define the-rules
  (list '((derivative X X 1))
        '((derivative N X 0) (require (number? 'N)))
        '((derivative (X ^ N) X (N * (X ^ P))) (P is (- N 1)))
        '((derivative (F + G) X (DF + DG))
            (derivative F X DF)
            (derivative G X DG))
        '((derivative (F - G) X (DF - DG))
            (derivative F X DF)
            (derivative G X DG))
        '((derivative (F * G) X ((F * DG) + (G * DF)))
            (derivative F X DF)
            (derivative G X DG))
        '((derivative (1 / F) X ((- DF) / (F * F)))
            (derivative F X DF))
        '((derivative (F / G) X (((G * DF) - (F * DG)) / (G * G)))
            (derivative F X DF)
            (derivative G X DG))))
```

This is obviously way more complex than Mary and John, but there isn't actually much that you haven't seen before. The first and most important thing to realise is that, with one exception, the arithmetic operators "+", "-" etc. mean nothing special to the logic program: they are just symbols in patterns to be matched. Having said that we do need ways to perform numeric tests and to do arithmetic. The body of second rule `requires` that `N` is a number, and the body of third rule evaluates `(- N 1)` before "assigning" it to `P`. Rules of the form `(require` ⟨*expr*⟩`)` and `(`⟨*var*⟩ `is` ⟨*expr*⟩`)` *are* recognized and treated specially by the system.

Anyway, having entered these rules we can ask the system:

```
(prove '((derivative
           (((x ^ 2) + x) + 1)
           x
           X)))
```

That is to say "prove the differential of $x^2 + x + 1$ in x is `X`". The system replies:

```
((derivative
  (((x ^ 2) + x) + 1)
  x
  (((2 * (x ^ 1)) + 1) + 0)))
```

The last line is the computed value for `X`, the pattern variable in the query. For now we will have to simplify that manually:

```
(((2 * (x ^ 1)) + 1) + 0)
  (2 * x      ) + 1
```

You can see that the result is indeed the differential of $x^2 + x + 1$, namely $2x + 1$.

—— • ——

That should be enough to whet your appetite for what the rest of this chapter has to offer. We next turn our attention to pattern matching, which is the basis of unification, which along with `amb` from the previous chapter is the basis of our logic programming implementation.

17.2 Pattern Matching

The kind of pattern matching we will be discussing here has very little if anything to do with regular expressions. This sort of pattern matching is about matching a pattern against a *structure*, not a string, and furthermore the pattern itself is a structure.

A *pattern* or a *structure*, for the purposes of our discussion is a PScheme expression: a string, number, symbol or list. However a pattern may also contain *pattern variables*. These are not the normal variables of PScheme programs: any symbol is a variable in that sense; these are a special type of symbol, recognised as a variable only by the pattern matching system. As far as the rest of PScheme is concerned, they are just ordinary symbols. For our purposes a pattern variable will be any symbol that starts with a capital letter[2]. Although I'm calling these symbols "pattern variables" here, I'll drop that convention in future and just call them "variables" when the context makes it clear what I mean.

First let's look at a few examples of pattern matching.

- The pattern `(a b c)` will *only* match the structure `(a b c)` because the pattern contains no variables.

- The pattern `(a b c)` will *not* match the structure `(a b foo)` because `c` does not equal `foo`.

[2] A real Scheme implementation is case-insensitive, so does not have this luxury. PScheme is case-sensitive which allows us to follow the convention of the logic programming language Prolog, where capital letters introduce pattern variables.

- The pattern `(a X c)` will match the structure `(a b c)` because the variable `X` can stand for the symbol `b`.
- The pattern `X` will match the structure `(a b c)` because `X` can stand for the entire structure.
- The pattern `(a X c)` will match the structure `(a (1 2 3) c)` because `X` can stand for `(1 2 3)`.
- The pattern `(a (1 X 3) c)` will match the structure `(a (1 2 3) c)` because `X` can stand for `2`.
- The pattern `(a X Y)` will match the structure `(a b c)` because `X` can stand for `b` and `Y` can stand for `c`.
- The pattern `(a X X)` will *not* match the structure `(a b c)` because the variable `X` must stand for the same thing throughout the matching process: it cannot be both `b` and `c`.

I'm sure you get the idea by now.

The result of a pattern match is a set of bindings, one for each variable that matched. For example after matching the pattern `(a X Y)` against `(a (1 2 3) b)` the result is the set of bindings `X => (1 2 3)` and `Y => b`. This result is much the same as an environment, and we'll take advantage of this equivalence later.

17.2.1 A Perl Pattern Matcher

As I've said, pattern matching is merely a precursor to unification, which is our goal. Implementing a pattern matching system in Pscheme is fairly trivial, either in the PScheme language itself or in the underlying Perl, but we're not actually going to do that, because it isn't powerful enough for our purposes. However we need to start somewhere, so this section discusses a standalone pattern-matching implementation in Perl. It's easy enough to understand, and the subsequent section builds on that to produce a standalone *unification* implementation. Having arrived at a standalone unification implementation, we can wire that in to our interpreter.

But first we need to look at pattern matching. This standalone Perl pattern matcher can take patterns of the form:

```
['a', { b => 'X'}, 'Y']
```

where capitalized strings represent the pattern variables. It matches them against structures such as

```
['a', { b => [2, 3] }, 'c']
```

returning a hashref that will look like

```
{ X => [2, 3], Y => 'c' }
```

Here's a birds-eye view of how our first pattern matcher will work. It walks both the pattern and the structure in parallel, also passing an additional, initially empty environment around. If it encounters a variable in the pattern then it checks to see if the variable is already set in the environment. If it is, then it checks that the current structure component is the same as the value of the variable, failing if it is not. If the variable is not set in the environment, it extends the environment, binding the variable to the current structure component, and continues. Of course matching will also fail if the pattern and the structure are not otherwise identical. On success, it returns the environment, and on failure, it `dies`.

Both for the fun of it, and for completeness' sake, the pattern matcher described here can handle perl hashrefs as well as listrefs. This means it will accept patterns and structures containing mixtures of both types. However only the values of a hash in a pattern will be recognised as pattern variables, the keys will not.

Here's the top-level `match()` routine:

```
sub match {
    my ($pattern, $struct, $env) = @_;
    $env ||= {};
    if (var($pattern)) {
        match_var($pattern, $struct, $env);
    } elsif (hashes($pattern, $struct)) {
        match_hashes($pattern, $struct, $env);
    } elsif (arrays($pattern, $struct)) {
        match_arrays($pattern, $struct, $env);
    } elsif (strings($pattern, $struct)) {
        match_strings($pattern, $struct, $env);
    } else {
        fail();
    }
    return $env;
}
```

Firstly, if `$env` is not passed, then `match()` initializes it to an empty hashref. The pattern matching algorithm is never required to undo any variable bindings that it creates, so we can just pass around a hash by reference and allow the bindings to accumulate in it. Then we see various tests for the types of `$pattern` and `$struct`. The `var()` check is just:

```
sub var {
    my ($thing) = @_;
    !ref($thing) && $thing =~ /^[A-Z]/;
}
```

So a var is any string that starts with a capital letter.

The `hashes()` check returns true if both arguments are hashrefs:

```
sub hashes {
    my ($a, $b) = @_;
    hash($a) && hash($b);
}
```

Where `hash()` is just:

```
sub hash {
    my ($thing) = @_;
    ref($thing) eq 'HASH';
}
```

The other two checks, `arrays()` and `strings()` are defined equivalently:

```
sub arrays {
    my ($a, $b) = @_;
    array($a) && array($b);
}

sub array {
    my ($thing) = @_;
    ref($thing) eq 'ARRAY';
}

sub strings {
    my ($a, $b) = @_;
    string($a) && string($b);
}

sub string {
    my ($thing) = @_;
    !var($thing) && !ref($thing);
}
```

Ok, that's the administrative support out of the way.

So the first thing that `match()` does, if its `$pattern` is a `var()`, is to call `match_var()` on the `$pattern` and the `$struct`, passing the current environment. Here's `match_var()`:

```
sub match_var {
    my ($var, $struct, $env) = @_;
    if (exists($env->{$var})) {
        match($env->{$var}, $struct, $env);
    } else {
        $env->{$var} = $struct;
    }
}
```

It checks to see if the `$var` is in the environment. If it is then it attempts to `match()` the value of the variable against the `$struct`[3]. If the `$var` is not already in the environment then it puts it there with a value equal to the current `$struct` (an unassigned variable will always match the current structure component, and will be instantiated to it.)

Returning to `match()`, if both the `$pattern` and the `$struct` are `arrays()`, `match()` calls `match_arrays()` on them.

`match_arrays()` walks both arrays in tandem, calling `match()` on each pair of elements. If the arrays are not the same length then they cannot possibly match, so this sanity check is performed first:

```
sub match_arrays {
    my ($pattern, $struct, $env) = @_;
    my @patterns = @$pattern;
```

[3] We use `match()` here only because it is convenient: neither the value of the variable nor the structure will contain variables, so `match()` is being used as a recursive equality test (like `eq?`.)

```
    my @structs = @$struct;
    if (@patterns != @structs) { fail(); }
    while (@patterns) {
        match(shift @patterns, shift @structs, $env);
    }
}
```

The `fail()` sub `dies` with a `"match failed"` message:

```
sub fail { die "match failed\n"; }
```

Back to `match()` again. If the `$pattern` and the `$struct` are both `hashes()`, then `match()` calls `match_hashes()` on them:

```
sub match_hashes {
    my ($pattern, $struct, $env) = @_;
    check_keys_eq($pattern, $struct);
    foreach my $key (sort keys %$pattern) {
        match($pattern->{$key}, $struct->{$key}, $env);
    }
}
```

Much as `match_arrays()` checks that the two arrays are the same length, `match_hashes()` checks that the two hashes have the same keys using `check_keys_eq()`:

```
sub check_keys_eq {
    my ($as, $bs) = @_;
    my $astr = join(’.’, sort keys %$as);
    my $bstr = join(’.’, sort keys %$bs);
    fail unless $astr eq $bstr;
}
```

This is a cheap trick and could easily be fooled, but it’s good enough for our demonstration purposes.

Assuming that the hashes have equal keys (this pattern matcher does not allow—or at least expect—hash keys to be variables), `match_hashes()` walks the keys matching the individual components in much the same way as `match_arrays()` did. It sorts the keys before traversing them to ensure the order of any variable assignment is at least deterministic.

Back to `match()` yet again. If both the `$pattern` and the `$struct` are `strings()`, `match()` calls `match_strings()` on them. This is the most trivial of the matching subroutines: it just compares the strings and fails if they are not equal:

```
sub match_strings {
    my ($pattern, $struct, $env) = @_;
    fail if $pattern ne $struct;
}
```

This completes our prototype pattern matching implementation.

While very simple, this pattern matcher can be made to do useful work. Consider a “database” of facts in a Perl list:

```
my @facts = (
    {
        composer => 'beethoven',
        initials => 'lv',
        lived => [1770, 1829]
    },
    {
        composer => 'mozart',
        initials => 'wa',
        lived => [1756, 1791]
    },
    {
        composer => 'bach',
        initials => 'js',
        lived => [1685, 1750]
    }
);
```

We can use the matcher to extract information from this list:

```
foreach my $fact (@facts) {
    eval {
        my $result = match({
            composer => 'COMPOSER',
            initials => 'js',
            lived => 'LIVED'
        }, $fact);
        print "The composer with initials 'js'",
              " is $result->{COMPOSER}",
              " who lived from $result->{LIVED}[0]",
              " to $result->{LIVED}[1]\n";
    };
}
```

That's it for pattern matching. We next turn our attention to Unification, which as I've said is an extension to Pattern Matching and is much more interesting.

17.3 Unification

Unification, in a nutshell, is the matching of two patterns. It solves the problem "given two patterns, that might both contain variables, find values for those variables that will make the two patterns equal."

Our pattern matcher from the previous section is a good jumping off point for implementing true unification. In fact it has most of the things we'll need already in place. The next section discusses the modifications we will need to make to it.

17.3.1 A Perl Unifier

This unifier can solve a broader class of problems than a simple pattern matcher can. For example given the two patterns:

```
['f', ['g', 'A'], 'A']
```

and

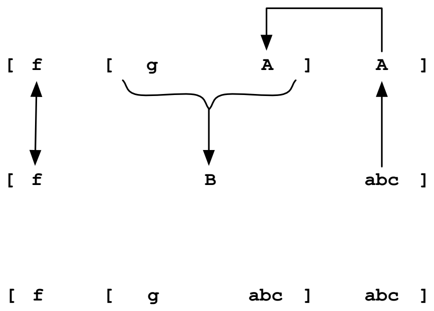
```
['f', 'B', 'abc']
```

It can correctly deduce:

```
A => 'abc',
B => ['g', 'abc']
```

You can see the process graphically in Figure 17.2. The variable `'A'` unifies with the term `'abc'` while the variable `'B'` unifies with the compound term `['g', 'A']`, where `'A'` is provided with a value `'abc'` from the previous unification.

Figure 17.2: Unification of `['f', ['g', 'A'], 'A']` with `['f', 'B', 'abc']`



Unification is capable of even more complex resolutions, for example it can unify (ommitting quotes for brevity this time)

```
[F, [A, 1, B], [A, B], 2]
```

with

```
[C, [D, D, E], C, E]
```

To show that

- `A = 1`
- `B = 2`
- `C = [1, 2]`

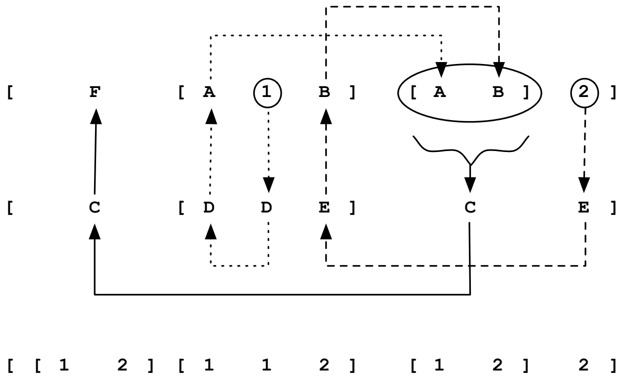
- `D = 1`

- `E = 2`

- `F = [1, 2]`

You can see this in action in Figure 17.3 if you just follow the differently styled arrows starting from the three ringed nodes in the figure as they propogate information around.

Figure 17.3: A more complex unification example



———•———

This unifier has additional feature: the *anonymous variable* “_” (underscore) behaves like a normal variable but will always match anything, since it is never instantiated. This allows you to specify a “don’t care” condition. For example, going back to our database of composers, the pattern:

```
{
    composer => 'COMPOSER',
    initials => '_',
    lived => '_'
}
```

will just retrieve all of the composers names from the database, without testing or instantiating any other variables. Not also that since “_” always matches, and is never instantiated, it can be reused throughout a pattern.

This unifier is a direct modification of the pattern matcher from the previous section, so let’s just concentrate on the differences. Firstly `match()` has been renamed to `unify()`, and it has an extra clause, in case the old structure, which is now also a pattern, contains variables. The various `match_*` subs have also been renamed `unify_*`, and the variables `$pattern` and `$struct`, now both patterns, have been renamed to just `$a` and `$b`:

```
sub unify {
    my ($a, $b, $env) = @_;
    $env ||= {};
    if (var($a)) {
        unify_var($a, $b, $env);
    } elsif (var($b)) {
        unify_var($b, $a, $env);
    } elsif (hashes($a, $b)) {
        unify_hashes($a, $b, $env);
    } elsif (arrays($a, $b)) {
        unify_arrays($a, $b, $env);
    } elsif (strings($a, $b)) {
        unify_strings($a, $b, $env);
    } else {
        fail();
    }
    return $env;
}
```

The extra clause, if `$b` is a var, simply reverses the order of the arguments to `unify_var()`. Note that the single environment means that variables will share *across* the two patterns. If you don't want this, simply make sure that the two patterns don't use the same variable names.

`unify_hashes()`, `unify_arrays()` and `unify_strings()` are identical to their `match_*` equivalents, except that `unify_arrays()` and `unify_hashes()` call `unify()` instead of `match()` on their components.

The `var()` check is slightly different, to allow for the anonymous variable:

```
sub var {
    my ($thing) = @_;
    !ref($thing) && ($thing eq '_' || $thing =~ /^[A-Z]/);
}
```

That leaves `unify_var()`, where the action is. `unify_var()` is still quite similar to `match_var()`, it just has more things to watch out for:

```
sub unify_var {
    my ($var, $other, $env) = @_;
    if (exists($env->{$var})) {
        unify($env->{$var}, $other, $env);
    } elsif (var($other) && exists($env->{$other})) {
        unify($var, $env->{$other}, $env);
    } elsif ($var eq '_') {
        return;
    } else {
        $env->{$var} = $other;
    }
}
```

So `$struct` was renamed to `$other`, and `unify_var()` calls `unify()` instead of `match()`. If `$var` is not set in the environment, instead of immediately assuming it can match `$other`, `unify_var()` looks to see if `$other` is a var and already has a value. If so it calls `unify()` on `$var` and the value of `$other`. If `$other` is not a var, or has no binding, `unify_var()` next checks to see if `$var` is the anonymous variable. If it is, then because the anonymous variable always matches and is never instantiated, it just returns. Lastly, only when all other options have been tried, it adds a binding from the `$var` to `$other` and returns.

Let's walk through the actions of `unify()` as it attempts to unify the two patterns `['f', ['g', 'A'], 'A']` and `['f', 'B', 'abc']`.

- `unify(['f', ['g', 'A'], 'A'], ['f', 'B', 'abc'], {})` is called with the two complete patterns and an empty environment, and determines that both patterns are arrays, so calls `unify_arrays()`.
    - `unify_arrays(['f', ['g', 'A'], 'A'], ['f', 'B', 'abc'], {})` simply calls `unify()` on each component.
        - `unify('f', 'f', ())` determines that both its arguments are strings, and calls `unify_strings()`.
            - `unify_strings('f', 'f', {}) = {}` succeeds but the environment is unchanged.
        - `unify(['g', 'A'], 'B', {})` determines that it's second argument is a variable and so calls `unify_var()` with the arguments reversed.
            - `unify_var('B', ['g', 'A'], {}) = {B => ['g', 'A']}` succeeds, and `unify_var()` extends the environment with `'B'` bound to `['g', 'A']`.
        - `unify('A', 'abc', {B => ['g', 'A']})` determines that it's first argument is a variable and so calls `unify_var()` again, passing the environment that was extended by the previous call to `unify_var()`.
            - `unify_var('A', 'abc', {B => ['g', 'A']}) = {B => ['g', 'A'], A => 'abc'}` also succeeds, extending the environment with a new binding of `'A'` to `'abc'`. This environment is the final result of the entire unification.

So the final result `{B => ['g', 'A'], A => 'abc'}` falls a little short of our expectations, because the value for `'B'` still contains a reference to the variable `'A'`[4]. However this is not a problem as such. we can patch up the result with a separate routine called `resolve()`.

```
sub resolve {
    my ($pattern, $env) = @_;
    while (var($pattern)) {
        if (exists $env->{$pattern}) {
            $pattern = $env->{$pattern};
        } else {
            return $pattern;
        }
    }
    if (hash($pattern)) {
```

[4] In fact it is quite possible to retrieve bindings of one variable directly to another, like `{A => 'B'}` in other circumstances, for example if `'B'` did not have a value when it was unified with `'A'`.

```
        my $ret = {};
        foreach my $key (keys %$pattern) {
            $ret->{$key} = resolve($pattern->{$key}, $env);
        }
        return $ret;
    } elsif (array($pattern)) {
        my $ret = [];
        foreach my $item (@$pattern) {
            push @$ret, resolve($item, $env);
        }
        return $ret;
    } else {
        return $pattern;
    }
}
```

`resolve()` takes a pattern, and the environment that was returned by `unify()`. If and while the pattern is a variable, it repeatedly replaces it with its value from the environment, returning the variable if it cannot further resolve it. Then if the result is a hash or an array reference, `resolve()` recursively calls itself on each component of the result, also passing the environment. The final result of `resolve()` is a structure where any variables in the pattern that have values in the environment have been replaced by those values. Note that `resolve()` does not change the environment in any way.

This completes our stand-alone implementation of `unify()`. Hopefully seeing it out in the open like this will make the subsequent implementation within PScheme easier to digest. It is a little difficult to demonstrate the utility of `unify()` at this point, since it's purpose is mostly part of the requirements of logic programming, however to give you some idea, consider that the result of one unification, an environment or hash, can be passed as argument to a second unification, thus constraining the values that the pattern variables in the second unification can potentially take.

There is one extremely interesting and useful application of unification outside of logic programming which makes use of this idea. Consider that we might want to check that the types of the variables in a PScheme expression are correct before we eval the expression. Assume that we know the types of the arguments and return values from all primitives in the language. Furthermore we can also detect the types of any variables which are assigned constants directly. It is therefore possible to detect if a variable's assigned value does not match it's eventual use, even if that eventual use is remote (through layers of function calls) from the original assignment. Such a language, which does not declare types but is nonetheless capable of detecting type mismatches, is called an *implicitly typed language*. We can use unification to do this type checking, by treating PScheme variables as pattern variables and unifying them with their types and with each other across function calls, accumulating types of arguments and return values for `lambda` expressions and functions in the process.

That however, is for another chapter. Next we're going to look at the implementation of `unify` in PScheme.

17.3.2 Implementing `unify` in PScheme

To get the ball rolling with the implementation of `unify`, notice that the previous implementation of `unify()` frequently tests the types of its arguments. Obviously in an object-oriented implementation like PScheme we can distribute a `Unify()` method around the various data types and avoid this explicit type checking for the most part.

A second point worth noting is that where the above `unify()` did a `die` on failure, our new `Unify()` can quite reasonably invoke backtracking instead, to try another option, which fits in quite neatly with our existing `amb` implementation.

A third and final point. `unify()` above made use of a flat hash to keep track of variable bindings, but PScheme already has a serviceable environment implementation and we should make use of it. This will mean exposing the environment as a PScheme data type since that is what is explicitly passed to and returned by `Unify()`, but this is not a concern since we have done this once before in our classes and objects extension from Chapter 12 on page 135.

We'd better start by looking at the `unify` command in action in the interpreter. The result of a call to `unify` is a **PScm::Env** which isn't much direct use. However we can add another PScheme command that will help us out there. `substitute` takes a form and an environment, and replaces any pattern variables in the form with their values from the argument environment. It also performs the `resolve()` function on each value before substitution. So for example:

```
> (substitute
>     '(f B A)
>     (unify '(f (g A) A)
>            '(f B abc)))
(f (g abc) abc)
```

The call to `unify` provides the second argument, an environment, to `substitute`, which then performs the appropriate replacements on the expression `(f B A)` to produce the result `(f (g abc) abc)`. Note that in all cases we have to quote the expressions to prevent them from being evaluated. We do need the interpreter to evaluate the arguments to `substitute` and `unify` in most cases however, because the actual forms being substituted and unified may be passed in as (normal PScheme) variables or otherwise calculated.

I should probably also demonstrate that `unify` does proper backtracking if it fails:

```
> (unify '(a b c) '(a b d))
Error: no more solutions
```

It's still somewhat difficult to demonstrate the usefulness of `unify` combined with `amb` at this stage however. That will have to wait until the next section where we finally get to see logic programming in action.

The first thing we need to do then, is to create a new special form **PScm::SpecialForm::Unify** and give it an `Apply()` method. This special form will be bound to the symbol `unify` in the top-level environment. `unify` will take two or three arguments. The first two arguments are the patterns to be unified. The third, optional argument is an environment to extend. If `unify` is not passed an environment, it will create a new, empty one. We have to make `unify` a special form because it needs access to the failure continuation. Here's **PScm::SpecialForm::Unify**:

```
package PScm::SpecialForm::Unify;

use base qw(PScm::SpecialForm);

use PScm::Continuation;

sub Apply {
    my ($self, $form, $env, $cont, $fail) = @_;
    $form->map_eval(
        $env,
        cont {
            my ($evaluated_args, $fail) = @_;
            my ($a, $b, $qenv) = $evaluated_args->value;
            $qenv ||= new PScm::Env();
            $a->Unify($b, $qenv, $cont, $fail);
        },
        $fail
    );
}

1;
```

You can see that it uses `map_eval()` from Section 13.6.5 on page 206 to evaluate its argument `$form`, passing it a continuation that breaks out the patterns `$a` and `$b`, and the optional environment `$qenv` from the evaluated arguments. Then it defaults `$qenv` to a new, empty environment, and calls `Unify()` on `$a` passing it the other pattern, the query environment and the success and failure continuations.

Referring back to our test implementation of `unify()` in Section 17.3.1 on page 312 we can see that the first thing that implementation does is to check if its first argument is a var, and if so call `unify_var()` on it. We can replace this explicit conditional with polymorphism by putting a `Unify()` method at an appropriate place in the **PScm::Expr** hierarchy. But the **PScm::Expr::Symbol** class is not the best place: not all symbols are pattern variables, only those starting with capital letters or underscores. So here's the trick. We create a new subclass of **PScm::Expr::Symbol** called **PScm::Expr::Var** and put the method there. `Read()` can detect pattern variables on input and create instances of this new class instead of **PScm::Expr::Symbol**. Since the new class inherits from **PScm::Expr::Symbol**, and we do not override any of that class's existing methods, these new **PScm::Expr::Var** objects behave exactly like ordinary symbols to the rest of the PScheme implementation. Here's the change to `_next_token()` from **PScm::Read** to make this happen.

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

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

    for ($self->{Line}) {
        s/^\(\s*// && return PScm::Token::Open->new();
        s/^\)\s*// && return PScm::Token::Close->new();
        s/^\'\s*// && return PScm::Token::Quote->new();
079         s/^\,\s*// && return PScm::Token::Unquote->new();
080         s/^\.\s*// && return PScm::Token::Dot->new();
081         s/^([-+]?\d+)\s*//
082           && return PScm::Expr::Number->new($1);
083         s/^"((?:(?:\\.)|([^"]))*)"\s*// && do {
084             my $string = $1;
085             $string =~ s/\\//g;
086             return PScm::Expr::String->new($string);
087         };
088         s/^([A-Z_][^\s\(\)]*)\s*//
089           && return PScm::Expr::Var->new($1);
090         s/^([^\s\(\)]+)\s*//
091           && return PScm::Expr::Symbol->new($1);
092     }
093     die "can't parse: $self->{Line}";
094 }
```

The only change is on Lines 88–89 where if the token matched starts with a capital letter or underscore then `_next_token()` returns a new **PScm::Expr::Var** where otherwise it would have returned a **PScm::Expr::Symbol**.

Now we have somewhere to hang the functionality equivalent to `unify_var()` from our test implementation, we can put it in a method called `Unify()` in **PScm::Expr::Var**:

```
sub Unify {
    my ($self, $other, $qenv, $cont, $fail) = @_;

    if (defined(my $value = $qenv->LookUpNoError($self))) {
        $value->Unify($other, $qenv, $cont, $fail);
    } elsif ($other->is_var &&
             defined (my $other_value = $qenv->LookUpNoError($other))) {
        $other_value->Unify($self, $qenv, $cont, $fail);
    } elsif ($self->is_anon) {
        $cont->Cont($qenv, $fail);
    } else {
        $qenv->ExtendUnevaluated(
            new PScm::Expr::List($self),
            new PScm::Expr::List($other),
            $cont,
            $fail
        );
    }
}
```

It's identical to the earlier `unify_var()` except that it makes use of method calls and is written in CPS. The `is_var()` method is defined to be false at the root of the expression hierarchy in **PScm::Expr**, but is overridden to be true in **PScm::Expr::Var** alone. Equivalently `is_anon()` is defined false in **PScm::**

Expr but defined to be true if the value of the var is "_" in **PScm::Expr::Var**[5]:

```
sub is_anon {
    my ($self) = @_;
    $self->value eq '_';
}
```

The only other occurrence of `Unify()` is at the root of the hierarchy in **PScm::Expr**:

```
sub Unify {
    my ($self, $other, $qenv, $cont, $fail) = @_;
    if ($other->is_var) {
        $other->Unify($self, $qenv, $cont, $fail);
    } else {
        $self->UnifyType($other, $qenv, $cont, $fail);
    }
}
```

This really just takes care of the case where the first pattern is not a var but the second pattern is. If the second pattern is a var it calls `Unify()` on it, passing `$self` as the argument, reversing the order in the same way as our prototype `unify()` did. If the second pattern is not a var, then it calls a new method `UnifyType()` on `$self`. `UnifyType()` is just another name for `Unify()` and allows a second crack at polymorphism since it is implemented separately in a couple of places in the **PScm::Expr** hierarchy.

The first such place is in **PScm::Expr** itself.

```
sub UnifyType {
    my ($self, $other, $qenv, $cont, $fail) = @_;
    if ($self->Eq($other)) {
        $cont->Cont($qenv, $fail);
    } else {
        $fail->Fail();
    }
}
```

This works for all atomic data types. If the two patterns are `Eq()` then succeed, otherwise fail. This is the first place we've actually seen the failure continuation invoked. Note that the equality test `Eq()` implicitly deals with type equivalence for us, so we don't need the `arrays()` routines etc. from the prototype. Now the only other place we need to put `UnifyType()` is in **PScm::Expr::List::Pair**

```
sub UnifyType {
    my ($self, $other, $qenv, $cont, $fail) = @_;
    if ($other->is_pair) {
        $self->[FIRST]->Unify(
            $other->[FIRST],
            $qenv,
```

[5] We could have further subclassed **PScm::Expr::Var** and had the reader recognize underscores as this type, but `is_anon()` is the only method that would then need to be specialized to this type.

```
            cont {
                my ($qenv, $fail) = @_;
                $self->[REST]->Unify(
                    $other->[REST],
                    $qenv,
                    $cont,
                    $fail
                );
            },
            $fail
        );
    } else {
        $fail->Fail();
    }
}
```

This `PScm::Expr::List::Pair::UnifyType()` is in fact simpler in one sense than the `unify_arrays()` from our test implementation. First it performs a simple check that the `$other` is a list. If not it calls the failure continuation. Then, rather than walking both lists, it only needs to call `Unify()` on its `first()` and `rest()`, passing the `$other`'s `first()` or `rest()` appropriately. Of course this is a little complicated because it's in CPS, but nonetheless that is all it has to do.

That's all for `unify` itself. If you remember from the start of this section, we will also need a `substitute` builtin to replace pattern variables with values in the environment. It is called like `(substitute` ⟨*pattern*⟩ ⟨*env*⟩`)` and returns the ⟨*pattern*⟩ suitably instantiated. We can make this a primitive rather than a special form because it has no need of an environment (other than the one that is explicitly passed) and no need to access the failure continuation (it always succeeds). Here's **PScm::Primitive::Substitute**:

```
package PScm::Primitive::Substitute;

use base qw(PScm::Primitive);

sub _apply {
    my ($self, $body, $qenv) = @_;
    $body->Substitute($qenv->ResolveAll());
}
```

It does nothing much by itself, merely calling a `ResolveAll()` method on the argument environment then passing the result to the `$body`'s `Substitute()` method. We'll take a look at that new `PScm::Env::ResolveAll()` method first.

```
sub ResolveAll {
    my ($self) = @_;
    my %bindings;
    foreach my $var ($self->Keys) {
        $bindings{$var} = $self->Resolve(new PScm::Expr::Var($var));
    }
    return $self->new(%bindings);
}
```

This `ResolveAll()` loops over each key in the environment, calling a subsidary `Resolve()` method on each and saving the result in a temporary `%bindings` hash. Then it creates and returns a new **PScm::Env** with those bindings.

If you refer back to our `resolve()` function in the prototype, you can see that in the first stage, if the `$pattern` is a variable, it repeatedly attempts to replace it with a value from the environment until either it is not a variable anymore or it cannot find a value. This `ResolveAll()` is effectively pre-processing the environment so that any subsequent lookup for a pattern variable will not need to perform that iteration.

`Keys()` just collects all the keys from the environment:

```
sub Keys {
    my ($self, $seen) = @_;
    $seen ||= {};
    foreach my $key (keys %{$self->{bindings}}) {
        $seen->{$key} = 1;
    }
    if ($self->{parent}) {
        $self->{parent}->Keys($seen);
    }
    return (keys %$seen);
}
```

and `Resolve()` does exactly what `resolve()` did in our test implementation: it repeatedly replaces the variable with its value from the environment until the variable is not a variable any more, or cannot be found. If it finds a non-variable value it calls its `ResolveTerm()` method on it, passing the env `$self` as argument, and returning the result.

```
sub Resolve {
    my ($self, $term) = @_;
    while ($term->is_var) {
        if (my $val = $self->LookUpNoError($term)) {
            $term = $val;
        } else {
            return $term;
        }
    }
    return $term->ResolveTerm($self);
}
```

`ResolveTerm()` gives any compound term a chance to resolve any pattern variables it may contain. There are two definitions of `ResolveTerm()`. The only compound terms in PScheme are lists, and pattern variables themselves have already been resolved, so the default `ResolveTerm()` in **PScm::Expr** just returns `$self`:

```
sub ResolveTerm {
    my ($self, $qenv) = @_;
    return $self;
}
```

The second definition of ResolveTerm() is, not surprisingly, in **PScm::Expr::List::Pair**:

```
sub ResolveTerm {
    my ($self, $qenv) = @_;
    return $self->Cons($qenv->Resolve($self->[FIRST]),
                       $qenv->Resolve($self->[REST]));
}
```

It walks itself, calling the argument $qenv's Resolve() method on each component, and returning a new **PScm::Expr::List** of the results.

So we're talking about how substitute works, and we saw that primitive's _apply() method called the argument $qenv's ResolveAll() method to return a new environment with any pattern variables in the values replaced, where possible. Then _apply() passed that new environment to its argument $body's Substitute() method. We've just seen how ResolveAll() works, now we can look at Substitute() itself.

Only pattern variables can be substituted, but lists need to examine themselves to see if they contain any pattern variables. So a default Substitute() method in **PScm::Expr** takes care of all the things that can't be substituted, it just returns $self:

```
sub Substitute {
    my ($self, $qenv) = @_;
    return $self;
}
```

The Substitute() in **PScm::Expr::Var** returns either its value from the environment or itself if it is not in the environment:

```
sub Substitute {
    my ($self, $qenv) = @_;
    return $qenv->LookUpNoError($self) || $self;
}
```

Finally, the Substitute() in **PScm::Expr::List::Pair** recursively calls Substitute() on each of its components, constructing a new list of the results:

```
sub Substitute {
    my ($self, $qenv) = @_;
    return $self->Cons($self->[FIRST]->Substitute($qenv),
                       $self->[REST]->Substitute($qenv));
}
```

And that's substitute. To sum up, it tries as hard as it can to replace all pattern variables in the form with values from the environment, recursing not only into the form it is substituting, but also into the values of the variables themselves.

There are a few more things we need to add to the interpreter before we can show off its new prowess. Firstly we will have occasion to pass an empty environment into unify (indirectly), and for that we'll need a `new-env` primitive. This is as simple as it gets:

```
package PScm::Primitive::NewEnv;

use base qw(PScm::Primitive);

sub _apply {
    new PScm::Env();
}
```

Another thing we'll need goes back to a passing comment I made a while back. It can be a problem if you try to unify two patterns that inadvertantly use the same pattern variable names. Sometimes you want the variables to share a value, and sometimes you don't. For this reason we need something that will take a pattern and replace its variable names with new variable names that are guaranteed to be unique. The same variable occurring more than once in the pattern should correspond to the same new variable occurring more than once in the result, but we should be reasonably confident that the new variable name won't appear anywhere else in the program by accident.

This new PScheme function is called `instantiate`. It could be written in the PScheme language, but that would require adding other less germane primitives for creating symbols etc. so all in all it is probably better to build it in to the core. It can be a primitive, but it will need a bit more than just an `_apply()` method:

```
package PScm::Primitive::Instantiate;

use base qw(PScm::Primitive);

sub new {
    my ($class) = @_;
    bless {
        seen => {},
        counter => 0,
    }, $class;
}

sub _apply {
    my ($self, $body) = @_;
    $self->{seen} = {};
    $body->Instantiate($self);
}

sub Replace {
```

```
286     my ($self, $var) = @_;
287     return $var if $var->is_anon;
288     unless (exists($self->{seen}{$var->value})) {
289         $self->{seen}{$var->value} =
290             new PScm::Expr::Var($self->{counter}++);
291     }
292     return $self->{seen}{$var->value};
293 }
294 
295 1;
```

Remember that there is only one instance of any given primitive or special form in the PScheme environment, and that persists for the duration of the repl. So by giving this primitive its own `new()` method we provide a convenient place to store a singleton counter that we can use to generate new symbols. The `seen` field of the object however, which keeps track of which variables the `instantiate` function has already encountered, must be re-initialized to an empty hash on each separate application of the primitive. After (re-)initializing `seen` on Line 281, `_apply()` calls its argument `$body`'s `Instantiate()` method passing `$self` as argument.

Obviously the only **PScm::Expr** type that will avail itself of the instantiate object is **PScm::Expr::Var**, and that will make use of the callback method `Replace()` to find or generate a replacement variable. `Replace()` then, first checks to see if the variable is the anonymous variable (Line 287). If so then it just returns the variable, since the anonymous variable never shares and should never be replaced by a variable that will share. Then unless it has already seen the variable it creates a new alias for it by using the incrementing counter (Lines 288-291). This works well because the reader would never create a **PScm::Expr::Var** from a number.

Just as with `Unify()` and `Substitute()`, a default method `Instantiate()` in **PScm::Expr** handles most cases and just returns `$self`:

```
079 sub Instantiate {
080     my ($self, $instantiator) = @_;
081     return $self;
082 }
```

The **PScm::Expr::Var** version of instantiate calls the `$instantiator`'s `Replace()` callback to get a new variable, passing `$self` as argument because `Replace()` needs to keep track of the variables it has seen already:

```
412 sub Instantiate {
413     my ($self, $instantiator) = @_;
414     return $instantiator->Replace($self);
415 }
```

Finally `PScm::Expr::List::Pair::Instantiate()` recurses on both its `first()` and `rest()` components, calling `Instantiate()` on both and constructing a new list on the way back out:

```
316 sub Instantiate {
317     my ($self, $instantiator) = @_;
```

```
    return $self->Cons($self->[FIRST]->Instantiate($instantiator),
                       $self->[REST]->Instantiate($instantiator));
}
```

The last thing we're going to need, for pragmatic reasons, is a way to check the type of various expressions from within the PScheme language. A proper scheme implementation has a full set of such type checking functions, but we're only going to need `pair?`, `number?` and `var?` (note the question marks.) They are all primitives, and in fact have so much in common that we will create an abstract parent class called **PScm::Primitive::TypeCheck** and put a shared `_apply()` method in there:

```
package PScm::Primitive::TypeCheck;

use base qw(PScm::Primitive);

sub _apply {
    my ($self, $body) = @_;
    if ($self->test($body)) {
        return new PScm::Expr::Number(1);
    } else {
        return new PScm::Expr::Number(0);
    }
}
```

You can see that it calls a `test()` method which we must subclass for each test, then it returns an appropriate true or false value depending on the test. Here's the test for `pair?` in **PScm::Primitive::TypeCheck::Pair**:

```
package PScm::Primitive::TypeCheck::Pair;
use base qw(PScm::Primitive::TypeCheck);

sub test { $_[1]->is_pair }
```

We've already seen that `is_pair()` is defined false in **PScm::Expr** and overridden to be true in **PScm::Expr::List::Pair** alone. The equivalent `number?` and `var?` PScheme functions are bound to **PScm::Primitive::TypeCheck::Number** and **PScm::Primitive::TypeCheck::Var**, and make use of equivalent `is_number()` and `is_var()` methods in **PScm::Expr**.

We have now implemented the four components we need to get on with defining a logic programming language: `unify`, `substitute`, `new-env` and `instantiate`. Along with those we have also added the three type tests `pair?`, `var?` and `number?` which just check if their argument is of that type. They are all wired in to the repl in the normal way, here's the additions:

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

    $outfh ||= new FileHandle(">-");
    my $reader      = new PScm::Read($infh);
    my $initial_env;
    $initial_env = new PScm::Env(
        let          => new PScm::SpecialForm::Let(),
        ’*’          => new PScm::Primitive::Multiply(),
        ’-’          => new PScm::Primitive::Subtract(),
        ’+’          => new PScm::Primitive::Add(),
        if           => new PScm::SpecialForm::If(),
        lambda       => new PScm::SpecialForm::Lambda(),
        list         => new PScm::Primitive::List(),
        car          => new PScm::Primitive::Car(),
        cdr          => new PScm::Primitive::Cdr(),
        cons         => new PScm::Primitive::Cons(),
        letrec       => new PScm::SpecialForm::LetRec(),
        ’let*’       => new PScm::SpecialForm::LetStar(),
        eval         => new PScm::SpecialForm::Eval(),
        macro        => new PScm::SpecialForm::Macro(),
        quote        => new PScm::SpecialForm::Quote(),
        ’set!’       => new PScm::SpecialForm::Set(),
        begin        => new PScm::SpecialForm::Begin(),
        define       => new PScm::SpecialForm::Define(),
        ’make-class’ => new PScm::SpecialForm::MakeClass(),
        ’call/cc’    => new PScm::SpecialForm::CallCC(),
        print        => new PScm::SpecialForm::Print($outfh),
        spawn        => new PScm::SpecialForm::Spawn(),
        exit         => new PScm::SpecialForm::Exit(),
        error        => new PScm::SpecialForm::Error(
                           $outfh,
                           bounce { repl($initial_env, $reader, $outfh) }
                        ),
        amb          => new PScm::SpecialForm::Amb(),
        ’eq?’        => new PScm::Primitive::Eq(),
        ’>’          => new PScm::Primitive::Gt(),
        ’<’          => new PScm::Primitive::Lt(),
        ’>=’         => new PScm::Primitive::Ge(),
        ’<=’         => new PScm::Primitive::Le(),
        and          => new PScm::SpecialForm::And(),
        or           => new PScm::SpecialForm::Or(),
        unify        => new PScm::SpecialForm::Unify(),
        substitute   => new PScm::Primitive::Substitute(),
        ’new-env’    => new PScm::Primitive::NewEnv(),
        instantiate  => new PScm::Primitive::Instantiate(),
        'pair?'      => new PScm::Primitive::TypeCheck::Pair(),
        'var?'       => new PScm::Primitive::TypeCheck::Var(),
        'number?'    => new PScm::Primitive::TypeCheck::Number(),
    );

    $initial_env->Define(
        PScm::Expr::Symbol->new("root"),
        PScm::Class::Root->new($initial_env)
    );
    __PACKAGE__->new_thread(bounce { repl($initial_env, $reader, $outfh) });
    trampoline();
}
```

17.4 Logic Programming

You may want to refer back to Section 17.1 on page 299 at the start of this chapter where we saw some examples of logic propgramming in action. However in order to explain how this all works, let's move away from our first examples and consider instead a very simple logic problem:

Socrates

- All men are mortal.
- Socrates is a man.
- Is socrates mortal?

While a mortal man should have no difficulty answering "yes" to the above puzzle, SQL queries might have some difficulty.

Here's a formulation of the rules in our system:

```
(define the-rules
  '(
    ((mortal X) (man X))
    ((man socrates))
   ))
```

The first rule on the list should be read as `(mortal X)` *if* `(man X)` that is, "`X` is mortal if `X` is a man," or colloquially "all men are mortal".

The second rule is just the bare fact "Socrates is a man."

In response to a query `(mortal socrates)` the system will respond in the affirmative. In response to a query like `(mortal aristotle)` you will just get **`Error: no more solutions`**.

You already know what unification does, so you should be able to start to see what is happening here. The system is given the query `((mortal socrates))` so it scans through `the-rules` looking only at the head of each rule, trying to unify it with the first term in the query, `(mortal socrates)`. It succeeds in unifying it with `(mortal X)`. The result of that unification is an environment where `X` is bound to `socrates`. In the context of that environment, it descends into the body of the rule, attempting to prove each component of the body just as if it had been entered as a direct query, but with the variable parts substituted for their values. In this case that means trying to find a rule that matches `(man X)` where `X=socrates`. This succeeds matching the fact `(man socrates)` and so the entire query succeeds.

What about asking “Who is mortal?”

In response to the query `(mortal X)` the head of each rule is again scanned. But I haven’t told you the full story at this point.

If you remember from Section 17.3.2 on page 316 we said there might be problems trying to unify two patterns which happened to contain the same variable names, and for that reason we implemented `instantiate` to replace the variables in a pattern with others that were equivalent, but guaranteed to be unique. In fact when the database of rules is scanned, `instantiate` is called on each rule before the unification with the head is attempted. This has no effect on our first example query `(mortal socrates)` but it does make a difference for `(mortal X)`, because the `X` appears in both the query and the rule. Thanks to `instantiate`, `(mortal X)` unifies with the head of a rule that looks like `((mortal ⟨0⟩) (man ⟨0⟩))`.

So the variable `X` unifies with the variable ⟨0⟩ rather than itself, and it is in this environment that the body of the rule `(man ⟨0⟩)` is investigated.

The statement `(man ⟨0⟩)` succeeds in unifying with `(man socrates)`, and in the process ⟨0⟩ is bound to `socrates`. Now the entire rule has succeeded and the query succeeds, resulting in an environment where `X=⟨0⟩` and `⟨0⟩=socrates`. `substitute` is given the form `(mortal X)` and that environment, and produces the result `(mortal socrates)`.

So let’s see how to build this system from our “toolkit” of `unify`, `substitute`, `instantiate`, and `new-env`. Before diving into the code, here is a slightly more formal definition of what we shall be doing.

- Start with an empty environment.
- For each term in a query of the form (⟨$term_1$⟩ ... ⟨$term_n$⟩)
  - For each rule
    - Instantiate a copy of the rule
    - If the term unifies with the head of the copied rule
      - Recurse on the body of the instantiated rule as another query, with the environment augmented by variables that were instantiated by unifying the term with the head.
      - Augment the environment with variables that were instantiated by solving the term.
    - Else
      - Fail (backtrack.)
  - Fail (backtrack.)

So you can see that the body of a rule is treated exactly the same as a top-level query, except that the current environment may contain variables that were instantiated by prior unifications. Therefore our implementation can recurse on the body of a rule, re-using the code that we shall write to process a top-level query. You should also be aware that the environment always accumulates on its way downstream to a solution, only backtracking causes bindings in the accumulating environment to be discarded on the way back upstream.

This is the first time we have really used the PScheme language to implement any serious piece of functionality. I’ve chosen to do it this way for two reasons. Firstly it emphasises a nice abstraction barrier between our "toolkit" of primitive Perl operations and the PScheme "glue" that binds them into a logic programming system; secondly and perhaps more importantly, it's actually easier to do in PScheme than it would have been in Perl, which I'm honestly quite pleased about. Having said all that, the code may take a little more study if you're not used to reading Scheme programs yet, but it is blissfully short and sweet.

The top-level function, as you have seen from examples above, is called `prove`, and here's its definition:

```
(define prove
  (lambda (goals)
    (substitute goals
                (match-goals goals
                             (new-env)))))
```

`prove` is given a list of `goals` (statements to be proved.) It calls another function `match-goals` passing both the `goals` and a new empty environment. If `match-goals` succeeds, it will return an environment with appropriate variables bound, and `prove` passes that environment along with the original `goals` to `substitute`, which replaces variables in the `goals` with their values then returns the result. If `match-goals` fails, control will backtrack through `prove` and we will see **`Error: no more solutions`**.

Here's `match-goals`:

```
(define match-goals
  (lambda (goals env)
    (if goals
        (match-goals (cdr goals)
                     (match-goal (car goals)
                                 env))
        env)))
```

`match-goals` walks its list of `goals`, calling another function `match-goal` on each, and collecting the resulting extended environment. If there are no `goals` left to prove, then `match-goal` succeeds and returns the environment it was passed. Incidentally this means that `prove` with an empty list of goals will always succeed and return its argument environment unchanged.

Here's `match-goal`:

```
(define match-goal
  (lambda (goal env)
    (match-goal-to-rule goal
                        (one-of the-rules)
                        env)))
```

Here is where we start to see `amb` coming in to play. `match-goal` uses the `one-of` function that we defined in Section 16.1 on page 249 to pick one of the list of rules to try to match against the `goal`. It passes the `goal`, the chosen rule, and the environment to `match-goal-to-rule`, which does the actual unification and recursion.

Here's `match-goal-to-rule`:

```
(define match-goal-to-rule
  (lambda (goal rule env)
    (let* ((instantiated-rule (instantiate rule))
           (head (car instantiated-rule))
           (body (cdr instantiated-rule))
           (extended-env (unify (substitute goal env)
                                head
                                env)))
      (match-goals body extended-env))))
```

It uses `let*` to first create an instantiated copy of the rule, and then extract the head and the body from the `instantiated-rule`. Then it calls `unify` on the (substituted) goal and the head of the instantiated rule. If `unify` succeeds, then the result is an extended environment that `match-goal-to-rule` uses to recursively call `match-goals` on the body of the rule. This is the point of recursion discussed above, where the body of a rule is treated as a new query.

That's all there is to it! Of course what is implicit in the above code is the backtracking that both `unify` and `amb` provoke if a unification fails or the list of rules to try is exhausted. This is most apparent in `match-goal-to-rule` above: if `unify` fails, then control simply backtracks out of the function at that point. Likewise in `match-goal`, when `one-of` runs out of options, control backtracks and `match-goal-to-rule` is never called.

There are a couple of refinements we can make howver. We would like to support `require` and `is` from our examples at the start of the chapter[6].

Additionally, it would be useful if we could check that the result returned by `prove` does not contain any unresolved variables. If it does, this should be considered a failure. For that reason we make use of those two type checking functions `pair?` and `var?` to write a little recursive test called `no-vars?`:

```
(define no-vars?
  (lambda (expr)
    (if (pair? expr)
        (and (no-vars? (car expr))
             (no-vars? (cdr expr)))
        (not (var? expr)))))
```

We can use that to write a variant of `substitite` called `substitute-all` that first of all performs the substitution then requires that the result contains no vars:

```
(define substitute-all
  (lambda (expr env)
    (let ((subst-expr (substitute expr env)))
         (begin
           (require (no-vars? subst-expr))
           subst-expr))))
```

[6] `(requre` ⟨*expression*⟩`)` fails if ⟨*expression*⟩ is false, and `(`$\langle \mathit{expr}_1 \rangle$ `is` $\langle \mathit{expr}_2 \rangle$`)` fails if $\langle \mathit{expr}_1 \rangle$ does not equal $\langle \mathit{expr}_2 \rangle$, in both cases after variable substitution and evaluation.

If the `require` succeeds, then `substitute-all` returns the substituted expression just as `substitute` does. Otherwise `substitute-all` backtracks. We can use this instead of `substitute` in the top-level `prove` function:

```
(define prove
  (lambda (goals)
    (substitute-all goals
                    (match-goals goals
                                 (new-env)))))
```

We will find other uses for `substitute-all`.

Next up, remember I said that the system treated `(require` ⟨*expr*⟩`)` and `(`⟨*var*⟩ `is` ⟨*expr*⟩`)` specially. This is not difficult to achieve. The existing `match-goal` just calls `match-goal-to-rule` with the current goal, `one-of` the rules, and the current env. All we need to do is to extend this to first look out for `require` and `is`.

```
(define match-goal
  (lambda (goal env)
    (if (eq? (car (cdr goal)) 'is)
        (match-is goal env)
        (if (eq? (car goal) 'require)
            (match-require goal env)
            (match-goal-to-rule goal
                                (one-of the-rules)
                                env)))))
```

I've added two new functions `match-is` and `match-require` to actually deal with those special rules. neither of them are particularily complex. `match-is` extracts the var and the expression from the rule, then it calls `substitute-all` on the expression. It will therefore fail and backtrack if the expression contains variables that have not been bound. This makes sense because the next thing it does is to pass the expression to `eval` (defined in Section 9.2.3 on page 117) and it makes no sense for `eval` to operate on an expression which contains instantiated pattern variables (⟨*0*⟩, ⟨*1*⟩ etc.) that cannot possibly have values in the normal pscheme environment. If it gets that far, then `match-is` finally unifies the var term with the substituted and evaluated expression, returning the new environment:

```
(define match-is
  (lambda (goal env)
    (let* ((var (car goal))
           (value (car (cdr (cdr goal))))
           (svalue (substitute-all value env)))
          (unify var (eval svalue) env))))
```

Note particularily that "`is`" is not necessarilty a test, it is a unification that will fail if the left hand expression cannot be unified with the right, so it can be considered both an assertion and potentially an assignment. To be clear `match-is` allows us to deal with statements like:

```
(N is (+ X Y))
```

provided X and Y are bound. In this example either N must already have a numeric value equal to the sum of X and Y, or it must be unbound, in which case it will recieve that value.

match-require is quite similar to match-is.

```
(define match-require
(lambda (goal env)
(let ((sgoal (substitute-all goal env)))
  (begin
    (eval sgoal)
    env))))
```

It too calls substitute-all, this time on the entire expression, for the same reasons match-is did. Then it passes the whole expression to eval. If the require in the substituted goal fails, control will backtrack from that point as usual.

That concludes our implementation. Let's try it out!

17.5 More Logic Programming Examples

In this section we look at applying logic programming to some "real world" problems, beginning with parsing.

17.5.1 Parsing (again)

We saw in Section 16.2.4 on page 260 that amb by itself was very useful for parsing because of its built in backtracking capability. However combining amb with unification as we have done here makes parsing an extraordinarily simple task, at least if we are prepared to accept the accompanying inefficiencies.

Consider the following rules for parsing a set of sentences worked through in the tests from Chapter 16 on page 249, Listing 16.7.3 on page 290. This is a different set of sentences to those worked through in Section 16.2.4 on page 260, the sentences in question are:

- "John will put his car in the garage."
- "Paul put a car in his garage."
- "Paul has put a very very old car in his quite new red garage."

These are obviously more complex than the sentences we worked through with amb, but we can deal with them easily enough here:

```
(define the-rules
  (list
        '((proper-noun (john)))
        '((proper-noun (paul)))
        '((noun (car)))
        '((noun (garage)))
        '((auxilliary (will)))
```

```
'((auxilliary (has)))
'((verb (put)))
'((article (the)))
'((article (a)))
'((article (his)))
'((preposition (in)))
'((preposition (to)))
'((preposition (with)))
'((degree (very)))
'((degree (quite)))
'((adjective (red)))
'((adjective (green)))
'((adjective (old)))
'((adjective (new)))

'((append () Y Y))
'((append (A . X) Y (A . Z)) (append X Y Z))

'((sentence S)
    (append NP VP S)
    (noun-phrase NP) (verb-phrase VP))
'((sentence S)
    (append _X VP S) (append NP AUX _X)
    (noun-phrase NP) (auxilliary AUX) (verb-phrase VP))

'((noun-phrase NP)
    (append ART ADJP NP)
    (article ART) (adj-phrase ADJP))
'((noun-phrase NP) (proper-noun NP))

'((adj-phrase ADJP) (noun ADJP))
'((adj-phrase ADJP)
    (append DGP ADJP2 ADJP)
    (degree-phrase DGP) (adj-phrase ADJP2))

'((degree-phrase DGP) (adjective DGP))
'((deg-phrase DGP)
    (append DEG DGP2 DGP)
    (degree DEG) (deg-phrase DGP2))

'((verb-phrase VP)
    (append _X PP VP) (append VB NP _X)
    (verb VB) (noun-phrase NP) (prep-phrase PP))

'((prep-phrase PP)
    (append PR NP PP)
```

```
          (preposition PR) (noun-phrase NP))))
```

This is little more than a declaration of the rules of the grammar. Let's look at a few of those rules a little more closely.

```
'((adj-phrase ADJP) (noun ADJP))
'((adj-phrase ADJP)
    (append DGP ADJP2 ADJP)
    (degree-phrase DGP) (adj-phrase ADJP2))
```

This rule about adjectival phrases is in two parts. The first part says that ADJP is an adj-phrase if ADJP is a noun. The second part says ADJP is an adj-phrase if some DGP and ADJP2 append to form ADJP, and DGP is a degree-phrase, and ADJP2 is an adj-phrase.

```
'((noun (car)))
'((noun (garage)))
```

This pair of rules defines the nouns we know about. They say (car) is a noun and (garage) is a noun. The nouns themselves are in lists because the system deals with lists: consider the adj-phrase rules above, where ADJP must be a list for append to work on it.

Given the above, we can ask the obvious question:

```
> (prove '((sentence (john put his car in the garage))))
((sentence (john put his car in the garage)))
```

You get the desired response, eventually[7]. Interestingly, you can also ask it "what is a sentence":

```
> (prove '((sentence S)))
((sentence (john put john in the car)))
> ?
((sentence (john put john in the garage)))
> ?
((sentence (john put john in the red car)))
> ?
((sentence (john put john in the red garage)))
> ?
((sentence (john put john in the red red car)))
> ?
((sentence (john put john in the red red garage)))
> ?
((sentence (john put john in the red red red car)))
```

It gets stuck in a recursive rut after a while, but it's exciting to see that it has the potential to generate all sentences of a grammar if it could avoid those traps[8]. However that's not the real problem, the real

[7] it took over ten seconds on my computer.

[8] One way to avoid recursive traps would be to randomize the order in which one-of returns its values. Unfortunately that will cause other problems in general.

problem is that it is horribly inefficient. All of those calls to `append`, most of which produce useless results, consume huge amounts of resources.

However we can take a hint from the way that we implemented parsing with `amb` in Section 16.2.4 on page 260. In that implementation, the routine `parse-word` removed tokens from the front of the `*unparsed*` global, effectively "directing" the progress of the parse, since it exposed the next token and only certain words would match that newly exposed token. We can't use "global variables" in a logic programming system—the concept has no meaning—but we can nonetheless keep track of what has been parsed so far.

We can do this by making all of our grammar rules take a second argument. For example:

```
(noun-phrase S R)
```

Will succeed if there is a noun phrase at the start of `S`, and if `R` is the remainder of `S` after removing that noun phrase. Perhaps this approach is most easily demonstrated for individual words. Here's the new definition of `noun`:

```
'((noun (car . S) S))
'((noun (garage . S) S))
```

It will match if its first argument is a list starting with `car` or `garage`, an in the process instantiate its second argument to the remainder of the list. For example:

```
> (prove '((noun (garage is red) X)))
((noun (garage is red) (is red)))
```

If we write the other rules for single words similarily, we can start to build up more complex rules on top:

```
'((noun-phrase S X)
    (article S S1) (adj-phrase S1 X))
```

This says that there is a noun phrase at the start of `S`, leaving `X` remaining if there is an article at the start of `S` leaving `S1` remaining, and an adjectival phrase at the start of `S1`, leaving `X` remaining. Note that we no longer need to use `append` to "generate and test". We can further build on these intermediate rules just as with the previous grammar:

```
'((sentence S X)
    (noun-phrase S S1) (verb-phrase S1 X))
```

In order to use this new parser, we must remember to pass an empty list as the second argument to `sentence`, to ensure that all of the tokens in the first argument are consumed:

```
> (prove '((sentence (john put his car in the garage) ())))
((sentence (john put his car in the garage) ()))
```

Putting all of this together, we can see the considerably faster (but uglier) version in the test at the end of Listing 17.8.2 on page 342.

Ugliness is not just an aesthetic thing, it gets in the way of clear and readable code. For that reason any full Prolog implementation provides rewriting rules that will accept a simple grammar with statements like[9]:

[9] When I say "like", I'm not suggesting that Prolog looks exactly like this, it's actual syntax is somewhat different. I'm only saying these are conceptually alike.

```
(sentence --> noun-phrase verb-phrase)
```

It will transform them into the internal form:

```
((sentence X0 X1) (noun-phrase X0 X2) (verb-phrase X2 X1))
```

as it reads them in. I'm not going to do that, since it would require generating symbols etc. but it should be obvious that this sort of transformation is not difficult.

17.5.2 Simplifying Algebraic Expressions

If you remember back in Section 17.1.5 on page 304 The result of differentiating the expression `(((x ^ 2) + x) + 1)` was `(((2 * (x ^ 1)) + 1) + 0)`. While correct, this is somewhat unwieldy as it contains redundant operations such as the addition of zero. It is relatively easy to write a set of rules that can automatically simplify such expressions, however we have to be careful of the *order* in which we specify the rules. This is not only a feature of our implementation, the order of the rules, which is the order in which they will be searched, is significant in Prolog too.

We start off by saying that we cannot simplify anything unless it has some internal structure:

```
'((simplify E E) (require (not (pair? 'E))))
```

So if `E` is not a pair, then it will simplify to itself.

We next get to the main driver rule for simplification:

```
'((simplify (X OP Y) E)
    (simplify X X1)
    (simplify Y Y1)
    (s (X1 OP Y1) E))
```

This says that we can simplify an expression of the form `(X OP Y)` to `E` if

1. we can simplify `X` to `X1`;
2. we can simplify `Y` to `Y1`;
3. we can simplify `(X1 OP Y1)` to `E` using the auxilliary simplification rules `(s (`⟨*E1*⟩ ⟨*OP*⟩ ⟨*E2*⟩`)` ⟨*E3*⟩`)`.

These auxilliary simplification rules occupy the rest of the definition of simplification. Firstly here are the rules for addition:

```
'((s (X + 0) X))
'((s (0 + X) X))
'((s (X + Y) Z)
    (require (and (number? 'X) (number? 'Y)))
    (Z is (+ X Y)))
'((s (X + Y) (X + Y)))
```

The first rule says that `X` plus zero is just `X`. The second rule is a necessary repetition of the first, reversing `0` and `X`. The third rule says that we can simplify `(X + Y)` to `Z` if `X` and `Y` are both numbers, by making `Z` equal to their sum. The last rule is there in case the term can not be simplified, in which case the result is just the original.

The rules for simplifying multiplication are very similar, except that we can simplify both multiplication by zero and multiplication by one:

```
'((s (X * 0) 0))
'((s (0 * X) 0))
'((s (X * 1) X))
'((s (1 * X) X))
'((s (X * Y) Z)
    (require (and (number? 'X) (number? 'Y)))
    (Z is (* X Y)))
'((s (X * Y) (X * Y)))
```

Likewise for exponentiation, except that we don't have a "^" operator in PScheme so we can't perform any actual numeric computation in this case:

```
'((s (X ^ 0) 1))
'((s (X ^ 1) X))
'((s (X ^ Y) (X ^ Y)))))
```

With these rules in place we can ask the system to simplify that unwieldy expression we saw before:

```
> (prove '((simplify (((2 * (x ^ 1)) + 1) + 0) X)))
((simplify (((2 * (x ^ 1)) + 1) + 0) ((2 * x) + 1)))
```

Note that these rules for simplification are very incomplete, most noticeably they do not deal with subtraction or division. However they are sufficient to demonstrate that they work and you can add the extra rules yourself if you want to extend or experiment (this example is part of the tests in Listing 17.8.2 on page 342.)

17.6 Summary, Shortcomings and Short-cuts

While we've spent some time exploring the potential of logic programming, it would be wrong of me to suggest that PScheme offers anything like the power of a real Prolog system. In fact it is missing some fairly fundamental features, and differs from real Prolog quite significantly. If you're excited by what you've seen here I'd suggest you get acquainted with a real Prolog system: I've barely scratched the surface. You may be surprised to know that Prolog itself is commonly used to build *compilers* for even higher-level languages and, much like Scheme, it is relatively easy to define a meta-circular evaluator for Prolog in Prolog. I'm not going to go anywhere near that, this is getting silly enough as it is. Rather, In this section I'd like to discuss the shortcomings of this implementation by comparing it with a real Prolog.

17.6.1 Functors and Arity

In Prolog, facts and rules are not defined as simple lists. Prolog distinguishes between the *functor* of a rule and its arguments. For example the PScheme "fact":

```
((mary likes cheese))
```

would be written in Prolog as:

```
likes(mary, cheese).
```

and the rule:

```
((likes mary X) (person X) (X likes chips))
```

would be written as:

```
likes(mary, X) :- person(X), likes(X, chips).
```

In this example `likes` is called the *functor* of the rule. Because it takes two arguments, it is said to have an *arity* of 2, and is classified as `likes/2`. This is distinct from any `likes` functor with a different number of arguments, Just as in our implementation lists of different length cannot match.

The big advantage in distinguishing functor/arity like this is that Prolog can index its database on this basis. Unification is expensive, so when searching for a rule to match i.e. `likes(mary, X)` Prolog need only inspect rules that are `likes/2`.

Of course this means that the functor cannot be matched by a pattern variable. The expression:

```
X(mary, wine)
```

is not valid Prolog. However Prolog has mechanisms to extract the functor from a term as a variable, and to call an expression constructed with a variable functor, so this apparent limitation can be worked around.

Prolog also recognises *operators* and *operator precedence*, such that the normal mathematical operators are infix and are parsed with the correct precedence and associativity. Prolog operators are functors, so for example the form:

```
a + b * c
```

is equivalent to:

```
+(a, *(b, c))
```

Prolog also allows you to define new operators as any sequence of non-alphanumeric characters, and assign them a precedence and associativity. These user defined operators don't actually *do* anything: they just make it easier to write Prolog terms as they translate internally to normal functors just as the built-in operators do.

17.6.2 The Cut

Another reason that Prolog distinguishes its functors is to allow short circuiting of the search space. Prolog provides a special rule called *the cut* to accomplish this [3, pp69–92]. The cut is written with a single exclaimation mark "`!`". As part of the body of a rule, the cut always succeeds. But if it is backtracked through, it backtracks all the way past the point where the head of the rule was unified, and past the point where rules of that functor/arity were considered, back to the decision point before that. This is extremely useful behaviour.

For example, returning to our `simplify` example of algebraic simplification from Section 17.5.2 on page 337, our very first rule was:

```
'((simplify E E) (require (not (pair? 'E))))
```

This was saying that you can simplify `E` to `E` if `E` is not a pair (is atomic.) We might translate this into Prolog as:

```
simplify(E, E) :- atomic(E).
```

However this is better expressed in Prolog as:

```
simplify(E, E) :- atomic(E), !.
```

This says that if this rule succeeds, then no other `simplify/2` rules should be considered if backtracking occurs through the cut. If the expression `E` is atomic, then it cannot be simplified, end of story.

There are other uses of the cut, but they are best described in a book on Prolog.

In order to properly implement the cut, we would have to pass a third `cut{}` continuation around, which makes the Parameter Object pattern discussed in Section 16.5.1 on page 285 even more attractive.

17.6.3 Backtracking Efficiency

As I've already mentioned in Section 16.5 on page 285, the chronological backtracking exhibited by `amb` is hopelessly inefficient, and an alternative *dependancy-directed backtracking* is preferable. This kind of backtracking examines the cause of any failure, and backtracks immediately to the point of execution that most recently affected the failure. Dependancy directed backtracking is often implemented by a *constraint network*, which allows the backtracking to proceed directly to the last place that the value in question was altered. Again this is beyond the scope of this book[10].

17.7 Tests

The first set of tests in Listing 17.8.1 on the next page exercise the individual additions to this version of the interpreter.

The first test proves that `unify` elicits backtracking on failure.

The second test proves that `unify` returns a **PScm::Env** on success.

the third test demonstrates that `unify` plus `substitute` can resolve the variable terms in an example that we've seen before.

The last test in this file shows `instantiate` in action. Although the digits in the result look like numbers, they are actually **PScm::Expr::Var**s
The second set of tests in Listing 17.8.2 on page 342 tries out our logic programming system. It works through pretty much the examples we've already covered in this chapter.

[10] i.e. I don't know how to make it work yet.

17.8 Listings

17.8.1 t/PScm_Unify.t

```
use strict;
use warnings;
use Test::More;
use lib ’./t/lib’;
use PScm::Test tests => 5;

BEGIN { use_ok(’PScm’) }

eval_ok(<<’EOT’, <<’EOR’, ’simple unify’);
(unify ’(a b c) ’(a b d))
EOT
Error: no more solutions
EOR

eval_ok(<<’EOT’, <<’EOR’, ’simple unify 2’);
(unify ’(a b c) ’(a b c))
EOT
PScm::Env
EOR

eval_ok(<<’EOT’, <<’EOR’, ’unify and substitute’);
(substitute
    ’((a A) (b B))
    (unify ’(f (g A) A)
           ’(f B abc)))
EOT
((a abc) (b (g abc)))
EOR

eval_ok(<<’EOT’, <<’EOR’, ’instantiate’);
(instantiate ’((f (g A) A) (f B abc)))
EOT
((f (g 0) 0) (f 1 abc))
EOR

# vim: ft=perl
```

17.8.2 t/AMB_Unify.t

```
use strict;
use warnings;
use Test::More;
use lib './t/lib';
use PScm::Test tests => 13;

BEGIN { use_ok('PScm') }

my $prereqs = <<EOT;

(define not
  (lambda (x)
     (if x 0 1)))

(define require
  (lambda (x)
    (if x x (amb))))

(define one-of
  (lambda (lst)
    (begin
        (require lst)
        (amb (car lst) (one-of (cdr lst))))))

(define no-vars?
  (lambda (expr)
    (if (pair? expr)
        (and (no-vars? (car expr))
             (no-vars? (cdr expr)))
        (not (var? expr)))))

(define substitute-all
  (lambda (expr env)
    (let ((subst-expr (substitute expr env)))
         (begin
           (require (no-vars? subst-expr))
           subst-expr))))

(define prove
  (lambda (goals)
    (substitute-all goals
                    (match-goals goals
                                 (new-env)))))

(define match-goals
  (lambda (goals env)
    (if goals
        (match-goals (cdr goals)
                     (match-goal (car goals)
                                 env))
        env)))
```

```

(define match-goal
  (lambda (goal env)
    (if (eq? (car (cdr goal)) 'is)
        (match-is goal env)
        (if (eq? (car goal) 'require)
            (match-require goal env)
            (match-goal-to-rule goal
                                (one-of the-rules)
                                env)))))

(define match-is
  (lambda (goal env)
    (let* ((var (car goal))
           (value (car (cdr (cdr goal))))
           (svalue (substitute-all value env)))
          (unify var (eval svalue) env))))

(define match-require
(lambda (goal env)
(let ((sgoal (substitute-all goal env)))
  (begin
    (eval sgoal)
    env))))

(define match-goal-to-rule
  (lambda (goal rule env)
    (let* ((instantiated-rule (instantiate rule))
           (head (car instantiated-rule))
           (body (cdr instantiated-rule))
           (extended-env (unify (substitute goal env)
                                head
                                env)))
      (match-goals body extended-env))))
EOT

my $prereqs_output = <<EOT;
not
require
one-of
no-vars?
substitute-all
prove
match-goals
match-goal
match-is
match-require
match-goal-to-rule
EOT

$prereqs_output =~ s/\n$//s;

```

```

eval_ok(<<EOT, <<EOR, 'socrates');
$prereqs
(define the-rules
  (list '((man socrates))
        '((mortal X) (man X))))
(prove '((mortal socrates)))
EOT
$prereqs_output
the-rules
((mortal socrates))
EOR

my $rules = <<'EOT';
(define the-rules
  (list '((mary likes cheese))
        '((mary likes wine))
        '((john likes beer))
        '((john likes wine))
        '((john likes chips))
        '((person mary))
        '((person john))
        '((mary likes X) (person X)
                         (require (not (eq? 'X 'mary)))
                         (X likes Y)
                         (mary likes Y))))
EOT

eval_ok(<<EOT, <<EOR, 'mary and john [1]');
$prereqs
$rules
(prove '((mary likes john)))
EOT
$prereqs_output
the-rules
((mary likes john))
EOR

eval_ok(<<EOT, <<EOR, 'mary and john [2]');
$prereqs
$rules
(prove '((mary likes X)))
?
?
?
EOT
$prereqs_output
the-rules
((mary likes cheese))
((mary likes wine))
((mary likes john))
Error: no more solutions
```

```
EOR

$rules = <<’EOT’;
(define the-rules
  (list ’((append () Y Y))
        ’((append (A . X) Y (A . Z)) (append X Y Z))))
EOT

eval_ok(<<EOT, <<EOR, ’append [1]’);
$prereqs
$rules
(prove ’((append (a b) (c d) X)))
EOT
$prereqs_output
the-rules
((append (a b) (c d) (a b c d)))
EOR

eval_ok(<<EOT, <<EOR, ’append [2]’);
$prereqs
$rules
(prove ’((append X Y (a b c d))))
?
?
?
?
?
EOT
$prereqs_output
the-rules
((append () (a b c d) (a b c d)))
((append (a) (b c d) (a b c d)))
((append (a b) (c d) (a b c d)))
((append (a b c) (d) (a b c d)))
((append (a b c d) () (a b c d)))
Error: no more solutions
EOR

eval_ok(<<EOT, <<EOR, ’symbolic differentiation’);
$prereqs
(define the-rules
  (list ’((derivative X X 1))
        ’((derivative N X 0) (require (number? ’N)))
        ’((derivative (X ^ N) X (N * (X ^ P))) (P is (- N 1)))
        ’((derivative (log X) X (1 / X)))
        ’((derivative (F + G) X (DF + DG))
            (derivative F X DF)
            (derivative G X DG))
        ’((derivative (F - G) X (DF - DG))
            (derivative F X DF)
            (derivative G X DG))
        ’((derivative (F * G) X ((F * DG) + (G * DF)))
            (derivative F X DF)
            (derivative G X DG))
        '((derivative (1 / F) X ((- DF) / (F * F)))
            (derivative F X DF))
        '((derivative (F / G) X (((G * DF) - (F * DG)) / (G * G)))
            (derivative F X DF)
            (derivative G X DG))))

(prove '((derivative (((x ^ 2) + x) + 1) x X)))
EOT
$prereqs_output
the-rules
((derivative (((x ^ 2) + x) + 1) x (((2 * (x ^ 1)) + 1) + 0)))
EOR

$rules = <<EOF;
(define the-rules
  (list '((add X Y Z) (Z is (+ X Y)))
        '((add X Y Z) (X is (- Z Y)))
        '((add X Y Z) (Y is (- Z X)))))
EOF

eval_ok(<<EOF, <<EOR, 'builtin arithmetic');
$prereqs
$rules
(prove '((add 2 3 X)))
(prove '((add 2 X 5)))
(prove '((add X 3 5)))
EOF
$prereqs_output
the-rules
((add 2 3 5))
((add 2 3 5))
((add 2 3 5))
EOR

$rules = <<EOF;
(define the-rules
  (list '(((johann ambrosius) father-of (j s)))
        '(((j c friedrich) father-of (w f ernst)))
        '(((j s) (anna magdalena) parents-of (j c friedrich)))
        '(((j s) (anna magdalena) parents-of (johann christian)))
        '(((j s) (maria barbara) parents-of (wilhelm friedmann)))
        '(((j s) (maria barbara) parents-of (c p e)))
        '((X father-of Y) (X _ parents-of Y))
        '((X mother-of Y) (_ X parents-of Y))
        '((X parent-of Y) (X father-of Y))
        '((X parent-of Y) (X mother-of Y))
        '((X ancestor-of Y) (X parent-of Y))
        '((X ancestor-of Z) (X parent-of Y) (Y ancestor-of Z))))
EOF

```

```
# had to truncate this test - takes too long to run
eval_ok(<<EOF, <<EOR, 'genealogy');
$prereqs
$rules
(prove '((X ancestor-of (w f ernst))))
EOF
$prereqs_output
the-rules
(((j c friedrich) ancestor-of (w f ernst)))
EOR

$rules = <<EOF;
(define the-rules
  (list
        '((proper-noun (john . S) S))
        '((proper-noun (paul . S) S))
        '((noun (car . S) S))
        '((noun (garage . S) S))
        '((auxilliary (will . S) S))
        '((auxilliary (has . S) S))
        '((verb (put . S) S))
        '((article (the . S) S))
        '((article (a . S) S))
        '((article (his . S) S))
        '((preposition (in . S) S))
        '((preposition (to . S) S))
        '((preposition (with . S) S))
        '((degree (very . S) S))
        '((degree (quite . S) S))
        '((adjective (red . S) S))
        '((adjective (green . S) S))
        '((adjective (old . S) S))
        '((adjective (new . S) S))
        '((sentance S X)
            (noun-phrase S S1) (verb-phrase S1 X))
        '((sentance S X)
            (noun-phrase S S1) (auxilliary S1 S2) (verb-phrase S2 X))
        '((noun-phrase S X)
            (article S S1) (adj-phrase S1 X))
        '((noun-phrase S X) (proper-noun S X))
        '((adj-phrase S X) (noun S X))
        '((adj-phrase S X)
            (degree-phrase S S1) (adj-phrase S1 X))
        '((degree-phrase S X) (adjective S X))
        '((deg-phrase S X)
            (degree S S1) (deg-phrase S1 X))
        '((verb-phrase S X)
            (verb S S1) (noun-phrase S1 S2) (prep-phrase S2 X))
        '((prep-phrase S X)
            (preposition S S1) (noun-phrase S1 X))

    ))
```

```
EOF

eval_ok(<<EOF, <<EOT, ’parsing’);
$prereqs
$rules
(prove ’((sentance (john will put his car in the garage) ())))
EOF
$prereqs_output
the-rules
((sentance (john will put his car in the garage) ()))
EOT

$rules = <<EOF;
(define the-rules
  (list
    ’((simplify E E) (require (not (pair? ’E))))
    ’((simplify (X OP Y) E)
        (simplify X X1)
        (simplify Y Y1)
        (s (X1 OP Y1) E))
    ’((s (X + 0) X))
    ’((s (0 + X) X))
    ’((s (X + Y) Z)
        (require (and (number? ’X) (number? ’Y)))
        (Z is (+ X Y)))
    ’((s (X + Y) (X + Y)))
    ’((s (X * 0) 0))
    ’((s (0 * X) 0))
    ’((s (X * 1) X))
    ’((s (1 * X) X))
    ’((s (X * Y) Z)
        (require (and (number? ’X) (number? ’Y)))
        (Z is (* X Y)))
    ’((s (X * Y) (X * Y)))
    ’((s (X ^ 0) 1))
    ’((s (X ^ 1) X))
    ’((s (X ^ Y) (X ^ Y)))))
EOF

eval_ok(<<EOF, <<EOT, ’simplification’);
$prereqs
$rules
(prove ’((simplify (((2 * (x ^ 1)) + 1) + 0) X)))
EOF
$prereqs_output
the-rules
((simplify (((2 * (x ^ 1)) + 1) + 0) ((2 * x) + 1)))
EOT

$rules = <<EOR;
(define the-rules
  (list
     '((factorial 0 1))
     '((factorial N X) (T is (- N 1)) (factorial T U) (X is (* N U)))))
EOR

eval_ok(<<EOF, <<EOT, 'recursion');
$prereqs
$rules
(prove '((factorial 10 X)))
EOF
$prereqs_output
the-rules
((factorial 10 3628800))
EOT

eval_ok(<<EOF, <<EOT, 'factorial not bidirectional');
$prereqs
$rules
(prove '((factorial X 3628800)))
EOF
$prereqs_output
the-rules
Error: no more solutions
EOT

# vim: ft=perl
```

Full source code for this version of the interpreter is available at
`http://billhails.net/Book/releases/PScm-0.1.13.tgz`

Chapter 18

Summary

In this book we've watched the evolution of a programming language from humble beginnings to a powerful if somewhat incomplete implementation.

Starting from a global environment model in Chapter 3 on page 15 with basic arithmetic and conditional evaluation, in Chapter 4 on page 49 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 Chapter 5 on page 59 where we introduced function definition and closure.

We then went on to introduce recursive functions in Chapter 6 on page 73, and showed that a different kind of binding is necessary to get recursive functions to work. Moving on, in Chapter 7 on page 81, we introduced another type of binding which is performed sequentially. In the next chapter 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, in Chapter 9 on page 107 we added a macro facility that allowed the program to generate 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 moved on to add side effects, (both definition and assignment,) sequences and global definition.

Chapter 12 on page 135 described a simple object-oriented extension to the language, using our existing environment implementation to model objects.

In Chapter 13 on page 159 we re-wrote the entire interpreter in Continuation Passing Style, giving the language direct access to those continuations via `call/cc` (`call-with-current-continuation`) and showed how powerful a control tool continuations are.

In the short but sweet Chapter 14 on page 231 we showed how trivial a threaded interpreter is once continuations are available, and in the equally short Chapter 15 on page 237 we added built-in error handling and error recovery.

Chapter 16 on page 249 took continuations even further, making a radical departure[1] from a standard Scheme implementation to add the `amb` operator and backtracking. By showing that it is possible for an interpreter to pass both a normal (success) continuation and a failure continuation, backtracking was easily included into the PScheme core.

Chapter 17 on page 299 discussed pattern matching and unification, added unification and other support routines as extensions to the interpreter. Then it used `amb` alongside those extentions to implement a simple but complete logic programming application in the PScheme language, demonstrating the

[1] Why are departures always radical?

essence of logic programming languages.

That's all for now.

Bibliography

[1] Harold Abelson, Gerald Jay Sussman, and Julie Sussman. *Structure and Interpretation of Computer Programs, 2nd Edition.* The MIT Press, Cambridge, Massachusetts, 1996.

[2] Philip Carter and Ken Russel. *Logic Brainteasers.* Carlton, London, 2006.

[3] W. F. Clocksin and C. S. Mellish. *Programming in Prolog.* Springer-Verlag, Berlin, Heidelberg, New York, London, Paris, Tokyo, 1987.

[4] Mark Jason Dominus. *Higher Order Perl.* Morgan Kaufmann Publishers, Amsterdam, 2005.

[5] Martin Fowler. *UML Distilled.* Addison Wesley, Boston, 1999.

[6] Daniel P. Friedman and Matthias Felleisen. *The Little Schemer.* The MIT Press, Cambridge, Massachusetts, 1996.

[7] Daniel P. Friedman, Mitchell Wand, and Christopher T. Haynes. *Essentials of Programming Languages, 2nd Edition.* The MIT Press, Cambridge, Massachusetts, 2001.

[8] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns.* Addison Wesley, Boston, 1995.

[9] Adele Goldberg and David Robson. *Smalltalk-80 the Language.* Addison Wesley, Boston, 1989.

[10] John C. Mitchell. *Concepts in Programming Languages.* Cambridge University Press, Cambridge, England, 2003.

[11] Benjamin C. Pierce. *Types and Programming Languages.* The MIT Press, Cambridge, Massachusetts, 2002.

[12] Michael Sperber, R. Kent Dybvig, Matthew Flatt, Anton van Straaten, Richard Kelsey, William Clinger, Jonathan Rees, Robert Bruce Findler, and Jacob Matthews. Revised^6 report on the algorithmic language scheme. `http://www.r6rs.org/`.

[13] Larry Wall, Tom Christiansen, and Jon Orwant. *Programming Perl, 3rd Edition.* O'Reilly, Sebastopol, 2000.

Index