CHAPTER 2

DISPATCH TABLES

In Chapter 1, we saw how to make functions more flexible by parametrizing their behaviors in terms of other functions. For example, instead of hardwiring the `hanoi()` function to print a certain message every time it wanted to move a disk, we had it call a secondary function that was passed in from outside. By supplying an appropriate secondary function, we could make `hanoi()` print out a list of instructions, or check its own moves, or generate a graphic display, without recoding the basic algorithm. Similarly, we were able to abstract the directory-walking behavior away from the file-size-computing behavior of our `total_size()` function to get a more useful and generally applicable `dir_walk()` function that could be used to do all sorts of different things.

To abstract behavior out of `hanoi()` and `dir_walk()`, we made use of code references. We passed `hanoi()` and `dir_walk()` additional functions as arguments, effectively treating the secondary functions as pieces of data. Code references make this possible.

Now we'll leave recursion for a while and go off in a different direction that shows another use of code references.

2.1 CONFIGURATION FILE HANDLING

Let's suppose that we have an application that reads in a configuration file in the following format:

```
VERBOSITY       8
CHDIR           /usr/local/app
```

```
        LOGFILE         log
        ...             ...
```

We would like to read in this configuration file and take an appropriate action for each directive. For example, for the VERBOSITY directive, we just want to set a global variable. But for the LOGFILE directive, we want to immediately redirect the program's diagnostic messages to the specified file. For CHDIR we might like the program to chdir to the specified directory so that subsequent file operations are relative to the new directory. This means that in the preceding example the LOGFILE is /usr/local/app/log, and not the log file in whatever directory the user happened to be in at the time the program was run.

Many programmers would see this problem and immediately envision a function with a giant if-else switch in it, perhaps something like this:

```
sub read_config {
  my ($filename) = @_;
  open my($CF), $filename or return; # Failure
  while (<$CF>) {
    chomp;
    my ($directive, $rest) = split /\s+/, $_, 2;
    if ($directive eq 'CHDIR') {
      chdir($rest) or die "Couldn't chdir to '$rest': $!; aborting";
    } elsif ($directive eq 'LOGFILE') {
      open STDERR, ">>", $rest
        or die "Couldn't open log file '$rest': $!; aborting";
    } elsif ($directive eq 'VERBOSITY') {
      $VERBOSITY = $rest;
    } elsif ($directive eq ...) {
      ...
    } ...
    } else {
      die "Unrecognized directive $directive on line $. of $filename; aborting";
    }
  }
  return 1; # Success
}
```

This function is in two parts. The first part opens the file and reads lines from it one at a time. It separates each line into a `$directive` part (the first word) and a `$rest` part (the rest). The `$rest` part contains the arguments to the directive, such as the name of the log file to open when supplied with the `LOGFILE` directive. The second part of the function is a big `if-else` tree that checks the `$directive` variable to see which directive it is, and aborts the program if the directive is unrecognized.

This sort of function can get very large, because of the many alternatives in the `if-else` tree. Each time someone wants to add another directive, they change the function by adding another `elsif` clause. The contents of the branches of the `if-else` tree don't have much to do with each other, except for the inessential fact that they're all configurable. Such a function violates an important law of programming: Related things should be kept together; unrelated things should be separated.

Following this law suggests a different structure for this function: The part that reads and parses the file should be separate from the actions that are performed when the configuration directives are recognized. Moreover, the code for implementing the various unrelated directives should not be lumped together into a single function.

2.1.1 Table-Driven Configuration

We can do better by separating the code for opening, reading, and parsing the configuration file from the unrelated segments that implement the various directives. Dividing the program into two halves like this will give us better flexibility to modify each of the halves, and to separate the code for the directives.

Here's a replacement for `read_config()`:

CODE LIBRARY
rdconfig-tabular

```
sub read_config {
  my ($filename, $actions) = @_;
  open my($CF), $filename or return; # Failure
  while (<$CF>) {
    chomp;
    my ($directive, $rest) = split /\s+/, $_, 2;
    if (exists $actions->{$directive}) {
      $actions->{$directive}->($rest);
    } else {
      die "Unrecognized directive $directive on line $. of $filename; aborting";
    }
  }
  return 1; # Success
}
```

We open, read, and parse the configuration file exactly as before. But we dispense with the giant `if-else` switch. Instead, this version of `read_config` receives an extra argument, `$actions`, which is a table of actions; each time `read_config()` reads a configuration directive, it will perform one of these actions. This table is called a *dispatch table*, because it contains the functions to which `read_config()` will dispatch control as it reads the file. The `$rest` variable has the same meaning as before, but now it is passed to the appropriate action function as an argument.

A typical dispatch table might look like this:

```
$dispatch_table =
{ CHDIR      => \&change_dir,
  LOGFILE    => \&open_log_file,
  VERBOSITY  => \&set_verbosity,
  ...        =>  ...,
};
```

The dispatch table is a hash, whose keys (generically called *tags*) are directive names, and whose values are *actions*, references to subroutines that are invoked when the appropriate directive name is recognized. Action functions expect to receive the `$rest` variable as an argument; typical actions look like these:

```
sub change_dir {
  my ($dir) = @_;
  chdir($dir)
    or die "Couldn't chdir to '$dir': $!; aborting";
}

sub open_log_file {
  open STDERR, ">>", $_[0]
    or die "Couldn't open log file '$_[0]': $!; aborting";
}

sub set_verbosity {
  $VERBOSITY = shift
}
```

If the actions are small, we can put them directly into the dispatch table:

```
$dispatch_table =
  { CHDIR      => sub { my ($dir) = @_;
                        chdir($dir) or
                          die "Couldn't chdir to '$dir': $!; aborting";
                      },

    LOGFILE    => sub { open STDERR, ">>", $_[0] or
                          die "Couldn't open log file '$_[0]': $!; aborting";
                      },

    VERBOSITY  => sub { $VERBOSITY = shift },
    ...        => ...,
};
```

By switching to a dispatch table, we've eliminated the huge `if-else` tree, but in return we've gotten a table that is only a little smaller. That might not seem like a big win. But the table provides several benefits.

2.1.2 Advantages of Dispatch Tables

The dispatch table is data, instead of code, so it can be modified at run time. You can insert new directives into the table whenever you want to. Suppose the table has:

```
'DEFINE' => \&define_config_directive,
```

where `define_config_directive()` is:

CODE LIBRARY
`def-conf-dir`

```
sub define_config_directive {
  my $rest = shift;
  $rest =~ s/^\s+//;
  my ($new_directive, $def_txt) = split /\s+/, $rest, 2;

  if (exists $CONFIG_DIRECTIVE_TABLE{$new_directive}) {
    warn "$new_directive already defined; skipping.\n";
    return;
  }

  my $def = eval "sub { $def_txt }";
  if (not defined $def) {
    warn "Could not compile definition for '$new_directive': $@; skipping.\n";
    return;
  }

  $CONFIG_DIRECTIVE_TABLE{$new_directive} = $def;
}
```

The configurator now accepts directives like this:

```
DEFINE HOME          chdir('/usr/local/app');
```

`define_config_directive()` puts `HOME` into `$new_directive` and `chdir('/usr/local/app');` into `$def_txt`. It uses `eval` to compile the definition text into a subroutine, and installs the new subroutine into a master configuration table, `%CONFIG_DIRECTIVE_TABLE`, using `HOME` as the key. If `%CONFIG_DIRECTIVE_TABLE` were in fact the dispatch table that was passed to `read_config()` in the first place, then `read_config()` will see the new definition, and will have an action associated with `HOME` if it sees the `HOME` directive on a later line of the input file. Now a config file can say:

```
DEFINE HOME       chdir('/usr/local/app');
CHDIR /some/directory
...
HOME
```

The directives in `...` are invoked in the directory `/some/directory`. When the processor reaches `HOME`, it returns to its home directory. We can also define a more robust version of the same thing:

```
DEFINE PUSHDIR   use Cwd; push @dirs, cwd(); chdir($_[0])
DEFINE POPDIR    chdir(pop @dirs)
```

`PUSHDIR` *dir* uses the `cwd()` function provided by the standard `Cwd` module to figure out the name of the current directory. It saves the name of the current directory in the variable `@dirs`, and then changes to *dir*. `POPDIR` undoes the effect of the last `PUSHDIR`:

```
PUSHDIR /tmp
A
PUSHDIR /usr/local/app
```

```
B
POPDIR
C
POPDIR
```

The program changes to `/tmp`, then executes directive A. Then it changes to `/usr/local/app` and executes directive B. The following `POPDIR` returns the program to `/tmp`, where it executes directive C; finally the second `POPDIR` returns it to wherever it started out.

In order for `DEFINE` to modify the configuration table, we had to store it in a global variable. It's probably better if we pass the table to `define_config_directive` explicitly. To do that we need to make a small change to `read_config`:

CODE LIBRARY
rdconfig-tablearg

```
sub read_config {
  my ($filename, $actions) = @_;
  open my($CF), $filename or return; # Failure
  while (<$CF>) {
    chomp;
    my ($directive, $rest) = split /\s+/, $_, 2;
    if (exists $actions->{$directive}) {
      $actions->{$directive}->($rest, $actions);
    } else {
      die "Unrecognized directive $directive on line $. of $filename; aborting";
    }
  }
  return 1; # Success
}
```

Now `define_config_directive` can look like this:

CODE LIBRARY
def-cdir-tablearg

```
sub define_config_directive {
  my ($rest, $dispatch_table) = @_;
  $rest =~ s/^\s+//;
  my ($new_directive, $def_txt) = split /\s+/, $rest, 2;

  if (exists $dispatch_table->{$new_directive}) {
    warn "$new_directive already defined; skipping.\n";
    return;
  }

  my $def = eval "sub { $def_txt }";
  if (not defined $def) {
    warn "Could not compile definition for '$new_directive': $@; skipping.\n";
    return;
  }

  $dispatch_table->{$new_directive} = $def;
}
```

With this change, we can add a really useful configuration directive:

```
DEFINE INCLUDE    read_config(@_);
```

This installs a new entry into the dispatch table that looks like this:

```
INCLUDE => sub { read_config(@_) }
```

Now, when we write this in the configuration file:

```
INCLUDE extra.conf
```

the main `read_config()` will invoke the action, passing it two arguments. The first argument will be the `$rest` from the configuration file; in this case the filename `extra.conf`. The second argument to the action will be the dispatch table again. These two arguments will be passed directly to a recursive call of `read_config`. `read_config` will read `extra.conf`, and when it's finished it will return control to the main invocation of `read_config`, which will continue with the main configuration file, picking up where it left off.

In order for the recursive call to work properly, `read_config()` must be reentrant. The easiest way to break reentrancy is to use a global variable, for example by using a global filehandle instead of the lexical filehandle we did use. If we had used a global filehandle, the recursive call to `read_config()` would open `extra.conf` with the same filehandle that was being used by the main invocation; this would close the main configuration file. When the recursive call returned, `read_config()` would be unable to read the rest of the main file, because its filehandle would have been closed.

The `INCLUDE` definition was very simple and very useful. But it was also ingenious, and it might not have occurred to us when we were writing `read_config`. It would have been easy to say "Oh, `read_config` doesn't need to be reentrant." But if we had written `read_config` in a nonreentrant way, the useful `INCLUDE` definition wouldn't have worked. There's an important lesson to learn here: make functions reentrant by default, because sometimes the usefulness of being able to call a function recursively will be a surprise.

Reentrant functions exhibit a simpler and more predictable behavior than nonreentrant functions. They are more flexible because they can be called recursively. Our `INCLUDE` example shows that we might not always anticipate all the reasons why someone might want to invoke a function recursively. It's better and safer to make everything reentrant if possible.

Another advantage of the dispatch table over hardwired code in `read_config()` is that we can use the same `read_config` function to process two unrelated files that have totally different directives, just by passing a different dispatch table to `read_config()` each time. We can put the program into "beginner mode" by passing a stripped-down dispatch table to `read_config()`. Or we can re-use `read_config()` to process a different file with the same basic syntax by passing it a table with a different set of directives; an example of this appears in Section 2.1.4.

2.1.3 Dispatch Table Strategies

In our implementation of `PUSHDIR` and `POPDIR`, the action functions used a global variable, `@dirs`, to maintain the stack of pushed directories. This is unfortunate. We can get around this, and make the system more flexible, by having `read_config()` support a *user parameter*. This is an argument, supplied by the caller of `read_config()`, which is passed verbatim to the actions:

CODE LIBRARY
rdconfig-uparam

```
sub read_config {
  my ($filename, $actions, $user_param) = @_;
  open my($CF), $filename or return; # Failure
  while (<$CF>) {
    my ($directive, $rest) = split /\s+/, $_, 2;
    if (exists $actions->{$directive}) {
      $actions->{$directive}->($rest, $user_param, $actions);
    } else {
      die "Unrecognized directive $directive on line $. of $filename; aborting";
    }
  }
  return 1; # Success
}
```

This eliminates the global variable, because we can now define `PUSHDIR` and `POPDIR` like this:

```
DEFINE PUSHDIR  use Cwd; push @{$_[1]}, cwd(); chdir($_[0])
DEFINE POPDIR   chdir(pop @{$_[1])
```

The `$_[1]` parameter refers to the user-parameter argument that is passed to `read_config()`. If `read_config()` is called with:

```
read_config($filename, $dispatch_table, \@dirs);
```

then `PUSHDIR` and `POPDIR` will use the array `@dirs` as their stack; if it is called with:

```
read_config($filename, $dispatch_table, []);
```

then they will use a fresh, anonymous array as the stack.

It's often useful to pass an action callback the name of the tag on whose behalf it was invoked. To do this, we change `read_config()` like this:

CODE LIBRARY
rdconfig-tagarg

```
sub read_config {
  my ($filename, $actions, $user_param) = @_;
  open my($CF), $filename or return; # Failure
  while (<$CF>) {
    my ($directive, $rest) = split /\s+/, $_, 2;
    if (exists $actions->{$directive}) {
      $actions->{$directive}->($directive, $rest, $actions, $user_param);
    } else {
      die "Unrecognized directive $directive on line $. of $filename; aborting";
    }
  }
  return 1; # Success
}
```

Why is this useful? Consider the action we defined for the `VERBOSITY` directive:

```
VERBOSITY => sub { $VERBOSITY = shift },
```

It's easy to imagine that there might be several configuration directives that all follow this general pattern:

```
VERBOSITY => sub { $VERBOSITY = shift },
TABLESIZE => sub { $TABLESIZE = shift },
PERLPATH  => sub { $PERLPATH = shift },
... etc ...
```

We would like to merge the three similar actions into a single function that does the work of all three. To do that, the function needs to know the name of the directive so that it can set the appropriate global variable:

```
VERBOSITY => \&set_var,
TABLESIZE => \&set_var,
PERLPATH  => \&set_var,
... etc ...

sub set_var {
  my ($var, $val) = @_;
  $$var = $val;
}
```

Or, if you don't like a bunch of global variables running around loose, you can store configuration information in a hash, and pass a reference to the hash as the user parameter:

```
sub set_var {
  my ($var, $val, undef, $config_hash) = @_;
  $config_hash->{$var} = $val;
}
```

In this example, not much is saved, because the action is so simple. But there might be several configuration directives that need to share a more complicated function. Here's a slightly more complicated example:

```
sub open_input_file {
  my ($handle, $filename) = @_;
  unless (open $handle, $filename) {
    warn "Couldn't open $handle file '$filename': $!; ignoring.\n";
  }
}
```

This `open_input_file()` function can be shared by many configuration directives. For example, suppose a program has three sources of input: a history file, a template file, and a pattern file. We would like the locations of all three files to be configurable in the configuration file; this requires three entries in the dispatch table. But the three entries can all share the same `open_input_file()` function:

```
...
HISTORY  => \&open_input_file,
TEMPLATE => \&open_input_file,
PATTERN  => \&open_input_file,
...
```

Now suppose the configuration file says:

```
HISTORY          /usr/local/app/history
TEMPLATE         /usr/local/app/templates/main.tmpl
PATTERN          /home/bill/app/patterns/default.pat
```

`read_config()` will see the first line and dispatch to the `open_input_file()` function, passing it the argument list `('HISTORY', '/usr/local/app/history')`. `open_input_file()` will take the `HISTORY` argument as a filehandle name, and open the `HISTORY` filehandle to come from the `/usr/local/app/history` file. On the second line, `read_config()` will dispatch to the `open_input_file()` again, this time passing it `('TEMPLATE', '/usr/local/app/templates/main.tmpl')`. This time, `open_input_file()` will open the `TEMPLATE` filehandle instead of the `HISTORY` filehandle.

2.1.4 Default Actions

Our example `read_config()` function dies when it encounters an unrecognized directive. This behavior is hardwired in. It would be better if the dispatch table itself carried around the information about what to do for an unrecognized directive. It's easy to add this feature:

CODE LIBRARY
rdconfig-default

```
sub read_config {
  my ($filename, $actions, $userparam) = @_;
  open my($CF), $filename or return; # Failure
  while (<$CF>) {
    chomp;
    my ($directive, $rest) = split /\s+/, $_, 2;
    my $action = $actions->{$directive} || $actions->{_DEFAULT_};
    if ($action) {
      $action->($directive, $rest, $actions, $userparam);
    } else {
      die "Unrecognized directive $directive on line $. of $filename; aborting";
    }
  }
  return 1; # Success
}
```

Here the function looks in the action table for the specified directive; if it isn't there, if looks for a _DEFAULT_ action, and dies only if there is no default specified in the dispatch table. Here's a typical _DEFAULT_ action:

```
sub no_such_directive {
  my ($directive) = @_;
  warn "Unrecognized directive $directive at line $.; ignoring.\n";
}
```

Since the directive name is passed as the first argument to the action function, the default action knows what unrecognized directive it was called on behalf of. Since the no_such_directive() function also gets passed the entire dispatch table, it can extract the real directive names and do some pattern matching to figure out what might have been meant. Here no_such_directive() uses a hypothetical score_match() function to decide which table entries are good matches for the unrecognized directive:

```
sub no_such_directive {
  my ($bad, $rest, $table) = @_;
  my ($best_match, $best_score);
  for my $good (keys %$table) {
    my $score = score_match($bad, $good);
    if ($score > $best_score) {
      $best_score = $score;
      $best_match = $good;
    }
  }
  warn "Unrecognized directive $bad at line $.;\n";
  warn "\t(perhaps you meant $best_match?)\n";
}
```

The system we have now has only a little code, but it's extremely flexible. Suppose our program is also going to read a list of user IDs and email addresses in the following format:

```
fred          fred@example.com
bill          bvoehno@plover.com
warez         warez-admin@plover.com
...           ...
```

We can re-use `read_config()` and have it read and parse this file, by supplying the appropriate dispatch table:

```
$address_actions =
  { _DEFAULT_ => sub { my ($id, $addr, $act, $aref) = @_;
                       push @$aref, [$id, $addr];
                     },
  };

read_config($ADDRESS_FILE, $address_actions, \@address_array);
```

Here we've given `read_config()` a very small dispatch table; all it has is a `_DEFAULT_` entry. `read_config()` will call this default entry once for each line in the address file, passing it the "directive name" (which is actually the user ID) and the address (which is the `$rest` value). The default action will take this information and add it to `@address_array`, which can be used later by the program.

2.2 CALCULATOR

Let's get away from the configuration file example for a while. Obviously, dispatch tables are going to make sense in many similar situations. For example, a conversational program that must process commands from a user can use a dispatch table to dispatch the user's commands. We'll look at a different example, a very simple calculator.

The input to this calculator is a string that contains an arithmetic expression in *reverse Polish notation* (RPN). Conventional arithmetic notation is ambiguous. If you write $2 + 3 \cdot 4$, it's not immediately clear whether we do the addition or the multiplication first. We have to have special conventions to say that multiplication always happens before addition, or we have to disambiguate the expression by inserting parentheses, for example, $(2 + 3) \cdot 4$.

Reverse Polish notation solves the problem in a different way. Instead of putting the operator symbols in between the arguments that they operate on, RPN puts the operators after their arguments. For example, instead of $2 + 3$ we write `2 3 +`. Instead of $(2 + 3) \cdot 4$, we write `2 3 + 4 *`. The `+` follows `2` and `3`, so the 2 and 3 are added; the `*` says to multiply the two preceding expressions, which are `2 3 +` and `4`. To express $2 + (3 \cdot 4)$ in RPN, we would write `2 3 4 * +`. The `+` applies to the two preceding arguments; the first of these is `2` and the second is `3 4 *`. Because the operator always follows its arguments, such expressions are said to be in *postfix form*; this is to contrast them with the usual form, where the operators are in between their arguments, which is called *infix form*.

It's easy to compute the value of an expression in RPN. To do this, we maintain a stack, and read the expression from left to right. When we see a number, we push it on the stack. When we see an operator, we pop the top two elements off the stack, operate on them, and push the result back on the stack. For example, to evaluate `2 3 + 4 *`, we first push 2 and then 3, and then when we see the + we pop them off and push back the sum, 5. Then we push 4 on top of the 5, and then the * tells us to pop the 4 and the 5 and push back the final answer, 20. To evaluate `2 3 4 * +` we push 2, then 3, then 4. The * tells us to pop back the 3 and the 4 and push the product 12; the + tells us to pop the 12 and the 2 and push the sum, 14, which is the final answer.

Here's a small calculator program that evaluates the RPN expression supplied in its command-line argument:

CODE LIBRARY
rpn-ifelse

```
my $result = evaluate($ARGV[0]);
print "Result: $result\n";

sub evaluate {
  my @stack;
  my ($expr) = @_;
  my @tokens = split /\s+/, $expr;
  for my $token (@tokens) {
    if ($token =~ /^\d+$/) { # It's a number
      push @stack, $token;
    } elsif ($token eq '+') {
       push @stack, pop(@stack) + pop(@stack);
    } elsif ($token eq '-') {
       my $s = pop(@stack);
       push @stack, pop(@stack) - $s
    } elsif ($token eq '*') {
       push @stack, pop(@stack) * pop(@stack);
    } elsif ($token eq '/') {
       my $s = pop(@stack);
       push @stack, pop(@stack) / $s
    } else {
      die "Unrecognized token '$token'; aborting";
    }
   }
  return pop(@stack);
}
```

The function splits the argument on whitespace into *tokens*, which are the smallest meaningful portions of the input. Then the function loops over the tokens one at a time, from left to right. If a token matches `/^\d+$/`, then it is a number, so the function pushes it onto the stack. Otherwise, it's an operator, so the function pops two values off the stack, operates on them, and pushes the result back onto the stack. The auxiliary `$s` variable in the code for subtraction is there because `5 3 -` should yield 2, not −2. If we had used:

```
push @stack, pop(@stack) - pop(@stack);
```

then for `5 3 -` the first `pop` would pop the 3, the second would pop the 5, and the result would have been −2. There is similar code in the division branch for the same reason. For multiplication and addition, the order of the operands doesn't matter.

When the function runs out of tokens, it pops the top value off the stack; this is the final result. This code ignores the possibility that the stack might finish with several values; this would mean that the argument contained more than one expression. `10 2 * 3 4 +` leaves 20 and 7 on the stack, in that order. It also ignores the possibility that the stack might become empty. For example, `2 *` and `2 3 + *` are invalid expressions, because in each, the `*` has only one argument instead of two. In evaluating these, the function finds itself doing an operation when the stack is empty. It should signal an error in that case, but I omitted the error handling to keep the example small.

We can make the example simpler and more flexible by replacing the large `if-else` switch with a dispatch table:

CODE LIBRARY
rpn-table

```
my @stack;
my $actions = {
  '+' => sub { push @stack, pop(@stack) + pop(@stack) },
  '*' => sub { push @stack, pop(@stack) * pop(@stack) },
  '-' => sub { my $s = pop(@stack); push @stack, pop(@stack) - $s },
  '/' => sub { my $s = pop(@stack); push @stack, pop(@stack) / $s },
  'NUMBER' => sub { push @stack, $_[0] },
  '_DEFAULT_' => sub { die "Unrecognized token '$_[0]'; aborting" }
};


my $result = evaluate($ARGV[0], $actions);
print "Result: $result\n";


sub evaluate {
  my ($expr, $actions) = @_;
  my @tokens = split /\s+/, $expr;
  for my $token (@tokens) {
    my $type;
    if ($token =~ /^\d+$/) { # It's a number
      $type = 'NUMBER';
    }

    my $action = $actions->{$type}
              || $actions->{$token}
              || $actions->{_DEFAULT_};
    $action->($token, $type, $actions);
  }
  return pop(@stack);
}
```

The main driver, `evaluate()`, is now much smaller and more general. It selects an action based on the token's "type," if it has one; otherwise, the action is based on the value of the token itself, and if there is no such action, a default action is used. The `evaluate()` function does a pattern match on the token to try to determine a token type, and if the token looks like a number, the selected type is `NUMBER`. We can add a new operator by adding an entry to the `%actions` dispatch table:

```
...
'sqrt' => sub { push @stack, sqrt(pop(@stack)) },
...
```

Again, because of the dispatch table construction, we can get a different behavior from the evaluator by supplying a different dispatch table. Instead of reducing the expression to a number, the evaluator will compile it into an *abstract syntax tree* (AST) if we supply this dispatch table:

```
my $actions = {
  'NUMBER'    => sub { push @stack,   $_[0] },
  '_DEFAULT_' => sub { my $s = pop(@stack);
                       push @stack,
                          [ $_[0], pop(@stack), $s ]
                       },
};
```

The result of compiling `2 3 + 4 *` is the abstract syntax tree `[ '*', [ '+', 2, 3 ], 4 ]`, which we can also represent as in Figure 2.1.

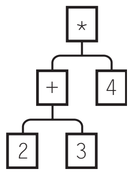

FIGURE 2.1 The AST for the expression `2 3 + 4 *`.

This is the most useful internal form for an expression because all the structure is represented directly. An expression is either a number, or it has an operator and two operands; the two operands are also expressions. An abstract syntax tree is either a number, or a list of an operator and two other ASTs. Once we have an AST, it's easy to write a function to process it. For example, here is a function to convert an AST to a string:

CODE LIBRARY
AST-to-string

```
sub AST_to_string {
  my ($tree) = @_;
  if (ref $tree) {
    my ($op, $a1, $a2) = @$tree;
    my ($s1, $s2) = (AST_to_string($a1),
                     AST_to_string($a2));
    "($s1 $op $s2)";
  } else {
    $tree;
  }
}
```

Given the tree of Figure 2.1, the `AST_to_string()` function produces the string `"((2 + 3) * 4)"`. The function first checks to see if the tree is trivial; if it is not a reference, then it must be a number, and the string version is just that number. Otherwise, the string has three parts: an operator symbol, which is stored in `$op`, and two arguments, which are ASTs. The function calls itself recursively to convert the two argument trees to strings `$s1` and `$s2`, and then produces a new string that has `$s1` and `$s2` with the appropriate operator symbol in between, surrounded by parentheses to avoid ambiguity. We have just written a system to convert postfix expressions to infix expressions, because we can feed the original postfix expression to `evaluate()` to generate an AST, and then give the AST to `AST_to_string()` to generate an infix expression.

The `AST_to_string()` function is recursive because the definition of an AST is recursive; the definition of an AST is recursive because the structure of an expression is recursive. The structure of `AST_to_string()` directly reflects the structure of an expression.

2.2.1 HTML Processing Revisited

In Chapter 1 we saw `walk_html()`, a recursive HTML processor. The HTML processor got two functional arguments: `$textfunc`, a function to call for a section of untagged text, and `$elementfunc`, a function to call for an HTML element. But "HTML element" is vague because there are many sorts of elements, and we might want our function to do something different for each kind of element.

We've seen several ways to accomplish this already. The most straightforward is for the user to simply put a giant `if-else` switch into `$elementfunc`. As we've already seen, that has some disadvantages. The user might like to supply a dispatch table to the `$elementfunc` instead. The structure of such a dispatch table is easy to see: the keys of the table will be tag names, and the values will be actions performed for each kind of element. Instead of supplying a single `$elementfunc` that knows how to deal with every possible element, the user will supply a dispatch table that provides one action for each kind of element, and also a generic `$elementfunc` that dispatches the appropriate action.

The `$elementfunc` might get access to the dispatch table in any of several ways. The dispatch table might be hardwired into the element function:

```
sub elementfunc {
  my $table = { h1        => sub { shift; my $text = join '', @_;
                                   print $text; return $text ;
                                 }
                _DEFAULT_ => sub { shift; my $text = join '', @_;
                                              return $text ;
              };
  my ($element) = @_;
  my $tag = $element->{_tag};
  my $action = $table->{$tag} || $table{_DEFAULT_};
  return $action->(@_);
}
```

Alternatively, we could build dispatch table support directly into `walk_html()`, so that instead of passing a single `$elementfunc`, the user passes the dispatch table directly to `walk_html()`. In that case, `walk_html()` would look something like this:

CODE LIBRARY
walk-html-disp

```
sub walk_html {
  my ($html, $textfunc, $elementfunc_table) = @_;
  return $textfunc->($html) unless ref $html; # It's a plain string
  my ($item, @results);
  for $item (@{$html->{_content}}) {
    push @results, walk_html($item, $textfunc, $elementfunc_table);
  }
  my $tag = $html->{_tag};
  my $elementfunc = $elementfunc_table->{$tag}
               || $elementfunc_table->{_DEFAULT_}
               || die "No function defined for tag '$tag'";
  return $elementfunc->($html, @results);
}
```

Yet another option is to change `walk_html()` to pass a user parameter to the `$textfunc` and `$elementfunc`. Then the user could have the dispatch table passed to the `$elementfunc` via the user parameter mechanism:

CODE LIBRARY
walk-html-uparam

```
sub walk_html {
  my ($html, $textfunc, $elementfunc, $userparam) = @_;
  return $textfunc->($html, $userparam) unless ref $html;
  my ($item, @results);
  for $item (@{$html->{_content}}) {
    push @results, walk_html($item, $textfunc, $elementfunc, $userparam);
  }
  return $elementfunc->($html, $userparam, @results);
}
```

Now it is up to the users to design their `$elementfuncs` to process the dispatch table appropriately.

One important and subtle point here: notice that we passed the user parameter to the `$textfunc` as well as to the `$elementfunc`. If the user parameter is a tag dispatch table, it is probably not useful to the `$textfunc`. Why did we pass it, then? Because it might not be a tag dispatch table; it might be something else. For example, the user might have called `walk_html()` like this:

```
walk_html($html_text,

          # $textfunc
          sub { my ($text, $aref) = @_;
                push @$aref, $text },

          # $elementfunc does nothing
          sub { },
                # user parameter
                \@text_array
               );
```

Now walk_html() will walk the HTML tree and push all the untagged plain text into the array @text_array. The user parameter is the reference to @text_array; it is passed to the $textfunc, which pushes the text onto the referred-to array. The $elementfunc doesn't use the user parameter at all. Since we, the authors of walk_html(), don't know in advance which sort of user parameter the user will require, we had better pass it to both the $textfunc and the $elementfunc; a function that doesn't need the user parameter is free to ignore it.