CHAPTER 9

DECLARATIVE PROGRAMMING

Beginning programmers often wish for a way to simply tell the computer what they want, and have the computer figure out how to do it. Declarative programming is an attempt to do that. The idea is that the programmer will put in the specifications for the value to be computed, and the computer will use the appropriate algorithm.

Nobody knows how to do this in general, and it may turn out to be impossible. But there are some interesting results we can get in specific problem domains. Regular expressions are a highly successful example of declarative programming. You write a pattern that represents the form of the text you are looking for, and then sit back and let the regex engine figure out the best way of locating the matching text.

Searching in general lends itself to declarative methods: the programmer specifies what they are searching for, and then lets a generic heuristic searching algorithm look for it. Database query languages are highly visible examples of this; consider SQL, or the query language of Chapter 8. The programming language Prolog is an extension of this idea to allow general computations.

We've seen searching in some detail already, so in this chapter we'll look at some other techniques and applications of declarative programming.

9.1 CONSTRAINT SYSTEMS

Suppose you wrote a program to translate Fahrenheit temperatures into Celsius:

```
sub f2c {
  my $f = shift;
  return ($f - 32) * 5/9;
}
```

Now you'd like to have a program to perform the opposite conversion, from Celsius to Fahrenheit. Although this calculation is in some sense the same, you'd have to write completely new code, from scratch:

```
sub c2f {
  my $c = shift;
  return 9/5 * $c + 32;
}
```

The idea of constraint systems is to permit the computer to be able to run this sort of calculation in either direction.

9.2 LOCAL PROPAGATION NETWORKS

One approach that seems promising is to distribute the logic for the calculation among several objects in a *constraint network* as shown in Figure 9.1.

There is a node in the network for each constant, variable, and operator. Lines between the nodes communicate numeric values between nodes, and they are called *wires*. A node can set the value on one of its wires; this sends a notification to the node at the other end of the wire that the value has changed. Because values are propagated only from nodes to their adjacent wires to the nodes attached at the other end of the wire, the network is called a *local propagation network*.

A constant node has one incident wire, and when the network is started up, the constant node immediately tries to set its wire to the appropriate constant

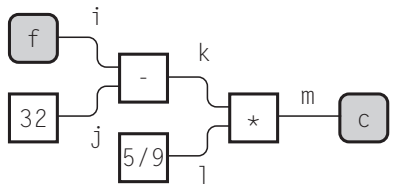


FIGURE 9.1 A constraint network for turning Fahrenheit temperatures to Celsius and back.

value. In the network shown in Figure 9.1, wire *j* initially has the value 32, and wire *l* initially has the value 5/9.

The nodes marked with variable names, `c` and `f` in this example, are input-output nodes. Initially, they do nothing, but the user of the program has the option to tell them to set their incident wires to certain values; that is how the user sends input into the network. If an input-output node notices that its incident wire has changed value, it announces that fact to the user; that's how output is emitted from the network.

We'll use the network of Figure 9.1 to calculate the Celsius equivalent for 212 Fahrenheit. We start by informing the `f` node that we want *f* to have the value 212. The `f` node obliges by setting the value of wire *i* to 212.

The change on wire *i* wakes up the attached - node, which notices that both of its input wires now have values: Wire *i* has the value 212 and wire *j* has the value 32. The - node performs subtraction; it subtracts 32 from 212 and sets its output wire *k* to the difference, 180.

The change on wire *k* wakes up the attached * node, which notices that both of *its* input wires now have values: Wire *k* has the value 180 and wire *l* has the value 5/9. The * node performs multiplication; it multiplies 180 by 5/9 and sets its output wire *m* to the product, 100.

The change on wire *m* wakes up the attached input-output node `c`, which notices that its input wire now has the value 100. It announces this fact to the user, saying something like:

```
c = 100
```

which is in fact the Celsius equivalent of 212 Fahrenheit.

What makes this interesting is that the components are so simple and so easily reversible. There's nothing about this process that requires that the calculation proceed from left to right. Let's suppose that instead of calculating the Celsius equivalent of 212 Fahrenheit, we wanted the Fahrenheit equivalent of 37 Celsius. We begin by informing the `c` input-output node that we want the

value of *c* to be 37. The `c` node will set wire *m* to value 37. This will wake up the * node, which will notice that two of its three incoming wires have values: *l* has value 5/9 and *m* has value 37. It will then conclude that wire *k* must have the value 37/(5/9) = 66.6, and set wire *k* accordingly.

The change in the value of wire *k* will wake up the attached - node, which will notice that the subtrahend *j* is 32 and the difference *k* is 66.6, and conclude that the minuend, *i*, must have the value 98.6. It will then set *i* to 98.6. This will wake up the attached `f` node, which will announce something like:

```
f = 98.6
```

which is indeed the Fahrenheit equivalent of 37 Celsius.

It's no trouble to attach more input-output nodes to the network to have it calculate several things at once. For example, we might extend the network as shown in Figure 9.2.

Now setting `c` to 37 causes values to propagate in two directions. The 37 will propagate left along wire *m* as before, eventually causing node `f` to announce the value 98.6. But wire *m* now has three ends, and the 37 will also propagate rightward, causing the - node to set wire *p* to 310.15, which is the value announced by the `k` node. The output looks something like:

```
f = 98.6
k = 310.15
```

which are the Fahrenheit and kelvin equivalents of 37 Celsius. Alternatively, we could have set node `k` to 0, which would have resulted in wire *m* being set to −273.15. Node `c` would announce that fact, and the * node would also

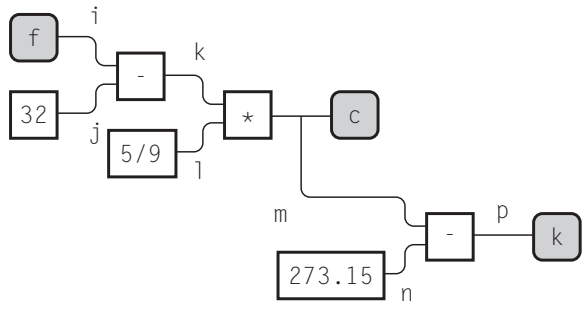


FIGURE 9.2 A constraint network for interconverting Fahrenheit, Celsius, and absolute temperatures.

take note; eventually wire *i* would be set to −459.67, and the output from the entire network would be:

```
c = -273.15
f = -459.67
```

which are the Celsius and Fahrenheit temperatures of absolute zero.

9.2.1 Implementing a Local Propagation Network

Clearly we will have two kinds of objects: wires and nodes. Wires store values. When a wire's value is set by a node, the wire remembers the value and also which node was responsible for setting it. This is so that the node can change or retract the value later. If the wire didn't remember the original source of its information, it wouldn't be able to distinguish the situation where the source changed its mind from the situation in which it was being given conflicting information. We'd like it to diagnose the latter but not the former:

CODE LIBRARY
Wire.pm

```
package Wire;
my $N = 0;
sub new {
  my ($class, $name) = @_;
  $name ||= "wire" . ++$N;
  bless { N => $name, S => undef, V => undef, A => [] } => $class;
}
```

The `$name` here is used for debugging purposes; we can supply a name to the constructor, or else the constructor will auto-generate one. `V` will be the stored value, initially undefined. `S` will be the identity of the node that supplied the stored value ("settor"), also initially undefined. `A` is a list of attached nodes. When the wire's value changes, it will notify the attached nodes.

It's common to need to manufacture several wires at once, so here's a utility function that does that:

```
sub make {
  my $class = shift;
  my $N = shift;
  my @wires;
```

```
    push @wires, $class->new while $N--;
    @wires;
  }
```

`Wire->make(5)` returns a list of five new wires.

The principal `Wire` method is `set`, which assigns a value to a wire:

```
sub set {
  my ($self, $settor, $value) = @_;
  if (! $self->has_settor || $self->settor_is($settor)) {
    $self->{V} = $value;
    $self->{S} = $settor;
    $self->notify_all_but($settor, $value);
  } elsif ($self->has_settor) {
    unless ($value == $self->value) {
      my $v = $self->value;
      my $N = $self->name;
      warn "Wire $N inconsistent value ($value != $v)\n";
    }
  }
}
```

The normal case is if the wire had no value before (`! $self->has_settor`) or if the old settor is changing the value, in which case the wire remembers the new value and the settor, and then calls `notify_all_but()` to notify the other attached nodes that the value has changed.

The other case of interest occurs when some other node, not the original settor, tries to notify the wire of a new value. In this case, if the old and new values are the same, all is well, and nothing need be done. But if the values differ, the wire should issue a diagnostic message. This might occur, for example, if we set the Fahrenheit input of a network to 212, and then tried to set the Celsius input to something other than 100.

The `notify_all_but()` function takes care of notifying the attached nodes of a change in value:

```
sub notify_all_but {
  my ($self, $exception, $value) = @_;
  for my $node ($self->attachments) {
    next if $node == $exception;
    $node->notify;
  }
}
```

When a wire is set to a certain value, it notifies all its attached nodes of the change *except* the one that set the value in the first place; this avoids infinite loops.

The accessors for attachments are trivial:

```
sub attach {
  my ($self, @nodes) = @_;
  push @{$self->{A}}, @nodes;
}

sub attachments { @{$_[0]->{A}} }
```

The other `Wire` accessor methods are similarly trivial:

```
sub name {
  $_[0]{N} || "$_[0]";
}

sub settor { $_[0]{S} }
sub has_settor { defined $_[0]{S} }
sub settor_is { $_[0]{S} == $_[1] }
```

The only unusual method here is `settor_is()`. `$wire->settor_is($node)` asks if the wire's settor is `$node`, and returns true if so. Note that objects can be compared for identity with the == operator; this actually compares the underlying machine addresses at which the objects are stored.

The opposite of `set()` is `revoke()`, which allows the settor node to revoke a previously set value:

```
sub revoke {
  my ($self, $revoker) = @_;
  return unless $self->has_value;
  return unless $self->settor_is($revoker);
  undef $self->{V};
  $self->notify_all_but($revoker, undef);
  undef $self->{S};
}
```

As far as the attached nodes are concerned, a revocation of a value is the same as setting the value to `undef`.

The final methods in the `Wire` class are the ones that query a wire for its current value. The code is short, but a little tricky. They're *almost* straightforward accessors, simply returning the value of `$self->{V}` or its definedness:

```
sub value { my ($self, $querent) = @_;
            return if $self->settor_is($querent);
            $self->{V};
          }
sub has_value { my ($self, $querent) = @_;
                return if $self->settor_is($querent);
                defined $_[0]{V};
              }
```

The exception is if the wire's settor is asking about the value. In this case, the wire returns `undef`, indicating that it doesn't know. This is necessary to support revocation of values. To see the reason for this, consider an adder node with addend wires *A* and *B*, and sum wire *C*. Suppose *A* and *B* have been set to 1 and 2 by some other components; the adder node itself then sets the sum *C* to 3. Now suppose the value of *B* is revoked. The adder node receives a notification and inspects the values of the wires. If not for the special case in `value()`, it would learn that *A* had value 1 and *C* had value 3, and conclude that *B* must have value 2, a conclusion that is no longer warranted. To avoid this, wire *C* will report a value of 3 to any *other* node that asks, but if the adder itself asks, the wire will say `undef`, meaning "If you're not sure what my value is supposed to be, then I'm not sure either."

We'll use an abstract class to represent nodes. We could subclass this to make the various node types, but since most of the node behavior is the same in all node types, we won't bother; the variable part of the behavior can be specified by supplying an anonymous function that will be stored in the node object.

Here's the generic constructor:

CODE LIBRARY
Node.pm

```
package Node;
my %NAMES;
sub new {
  my ($class, $base_name, $behavior, $wiring) = @_;
  my $self = {N => $base_name . ++$NAMES{$base_name},
              B => $behavior,
              W => $wiring,
             };
  for my $wire (values %$wiring) {
    $wire->attach($self);
  }
```

```
  bless $self => $class;
}
```

The constructor's first argument is a node type name, such as `adder`, which is used to construct a name for debugging. The important arguments are the other two. `$behavior` is a function that is invoked when one of the attached wires changes values; it is the responsibility of `$behavior` to calculate new values and to propagate them through the network. `$wiring` is a hash whose values are the wires themselves; each wire is associated with a name, through which `$behavior` will access it.

The primary method is `notify()`. When a node is notified that a wire has changed, it builds a hash of its current wire values, and passes the hash to its behavior function:

```
sub notify {
  my $self = shift;
  my %vals;
  while (my ($name, $wire) = each %{$self->{W}}) {
    $vals{$name} = $wire->value($self);
  }
  $self->{B}->($self, %vals);
}
```

The rest of the `Node` methods are simple utilities, intended to be used by the behavior function:

```
sub name {
  my $self = shift;
  $self->{N} || "$self";
}
```

`wire()` takes a name and returns the associated wire object:

```
sub wire { $_[0]{W}{$_[1]} }

sub set_wire {
  my ($self, $wire_name, $value) = @_;
  my $wire = $self->wire($wire_name);
  $wire->set($self, $value);
}

sub revoke_wire {
  my ($self, $wire_name) = @_;
```

```
  my $wire = $self->wire($wire_name);
  $wire->revoke($self);
}
```

We're finally at the meat of the program; we're ready to see the components themselves. Here's the behavior function for an adder:

```
{
  my $adder = sub {
    my ($self, %v) = @_;
    if (defined $v{A1} && defined $v{A2}) {
      $self->set_wire('S', $v{A1} + $v{A2});
    } else {
      $self->revoke_wire('S');
    }
    if (defined $v{A1} && defined $v{S}) {
      $self->set_wire('A2', $v{S} - $v{A1});
    } else {
      $self->revoke_wire('A2');
    }
    if (defined $v{A2} && defined $v{S}) {
      $self->set_wire('A1', $v{S} - $v{A2});
    } else {
      $self->revoke_wire('A1');
    }
  };

  # continues...
```

An adder has three wires: two addends, named `A1` and `A2`, and a sum, named `S`. When it receives a notification, it checks to see if `A1` and `A2` both have values; if so, it sets `S` to be the sum. If not, it revokes any value that it might have given to `S`; note that if `S` has no value, or if some other component was responsible for setting `S`, the revocation is harmless, because of the way we defined the `Wire::revoke()` method. There are two analogous blocks of code for inferring the two addends from the sum.

The function to build an adder node gets three wires as arguments and invokes `Node::new()` to build a node with those three wires and the adder behavior function:

```
  # continued...

  sub new_adder {
    my ($a1, $a2, $s) = @_;
```

```
  Node->new('adder',
            $adder,
            { A1 => $a1, A2 => $a2, S => $s });
  }
}
```

The behavior function for a multiplier node is a little more complicated. Not only does it need to infer a product from the two factors, and vice versa, but when a factor is 0, it can infer the product even without the other factor:

```
{
  my $multiplier = sub {
    my ($self, %v) = @_;
    if (defined $v{F1} && defined $v{F2}) {
      $self->set_wire('P', $v{F1} * $v{F2});
    } elsif (defined $v{F1} && $v{F1} == 0) {
      $self->set_wire('P', 0);
    } elsif (defined $v{F2} && $v{F2} == 0) {
      $self->set_wire('P', 0);
    } else {
      $self->revoke_wire('P');
    }

# continues...
```

The price of this free inference, however, is that the wires can be in an inconsistent state, which corresponds to a division by zero. If one factor is zero while the product is nonzero, the node won't be able to reason backwards, and will become upset:

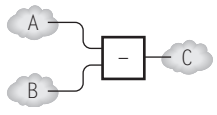
```
# continued...


  if (defined $v{F1} && defined $v{P}) {
    if ($v{F1} != 0) {
      $self->set_wire('F2', $v{P} / $v{F1});
    } elsif ($v{P} != 0) {
      warn "Division by zero\n";
    }
  } else {
    $self->revoke_wire('F2');
  }
```

```
    if (defined $v{F2} && defined $v{P}) {
      if ($v{F2} != 0) {
        $self->set_wire('F1', $v{P} / $v{F2});
      } elsif ($v{P} != 0) {
        warn "Division by zero\n";
      }
    } else {
      $self->revoke_wire('F1');
    }

};

# continues...
```

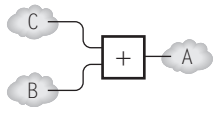
The function for building a multiplier node, `new_multiplier()`, is similar to `new_adder()`:

```
# continued...

sub new_multiplier {
  my ($f1, $f2, $p) = @_;
  Node->new('multiplier', $multiplier,
            { F1 => $f1, F2 => $f2, P => $p });
}
}
```

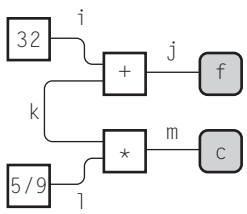
We could go on and build a subtraction node, but there's no need.



This network fragment expresses the constraint $A - B = C$. But that's the same as $C + B = A$, so the network shown below expresses the same thing.



With the transformation shown above, our Fahrenheit-to-Celsius network becomes the network shown here.



But for convenience, we could define:

```
# S - M = D
sub new_subtractor {
  my ($s, $m, $d) = @_;
  new_adder($d, $m, $s);
}

# V / S = Q
sub new_divider {
  my ($v, $s, $q) = @_;
  new_multiplier($q, $s, $v);
}
```

if we wanted.

Now all we need are constant nodes and input-output (IO) nodes. Constants, as you would expect, are very simple:

```
sub new_constant {
  my ($val, $w) = @_;
  my $node = Node->new('constant',
                       sub {},
                       {'W' => $w},
                      );
  $w->set($node, $val);
  $node;
}
```

The two arguments here are `$val`, the constant value, and `$w`, the outgoing wire. The behavior function is trivial and does nothing. The only fine point is that

the constructor needs to notify the attached wire of the outgoing constant value immediately after constructing the node, before anything else happens in the network.

Most of the code for IO nodes is for announcing changes in values on the one attached wire. `$announce` is a curried function. Its argument is the name of the IO node, and it returns a behavior function for that node:

```
{
  my $announce = sub {
    my $name = shift;
    sub {
      my ($self, %val) = @_;
      my $v = $val{W};
      if (defined $v) {
        print "$name : $v\n";
      } else {
        print "$name : no longer defined\n";
      }
    };
  };
  # continues...
```

The IO node itself is an ordinary node with this announcing behavior:

```
  # continued...

  sub new_io {
    my ($name, $w) = @_;
    Node->new('io',
              $announce->($name),
              { W => $w });
  }
}
```

There are two utility functions exposed to the main program for setting and revoking the values of IO nodes:

```
sub input {
  my ($self, $value) = @_;
  $self->wire('W')->set($self, $value);
}
```

```
sub revoke {
  my $self = shift;
  $self->wire('W')->revoke($self);
}
```

We can now build local propagation networks:

```
my ($F, $C);
{ my ($i, $j, $k, $l, $m) = Wire->make(5);
  $F = new_io('Fahrenheit', $i);
  $C = new_io('Celsius', $m);
  new_constant(32, $j);
  new_constant(5/9, $l);
  new_adder($i,$k,$j);
  new_multiplier($k,$l,$m);
}
```

And now we can use the network to calculate values:

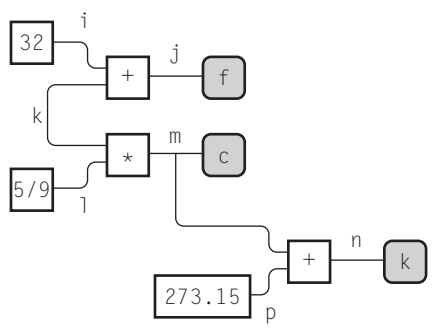
```
input($F, 212);
       Celsius : 100
input($F, 32);
       Celsius : 0
revoke($F);
       Celsius : no longer defined
input($C, 37);
       Fahrenheit : 98.6
input($F, 100);
       Wire wire3 inconsistent value (100 != 98.6)
revoke($C);
       Fahrenheit : no longer defined
input($F, 100);
       Celsius : 37.7777777777778
```

We can extend the network to handle kelvins by adding:

```
my ($F, $C);
my $K;
{ my ($i, $j, $k, $l, $m) = Wire->make(5);
  $F = new_io('Fahrenheit', $i);
  $C = new_io('Celsius', $m]);
```

```
    new_constant(32, $j);
    new_constant(5/9, $l);
    new_adder($i,$k,$j);
    new_multiplier($k,$l,$m);
    my ($n, $p) = Wire->make(2);
    $K = new_io('kelvin', $n);
    new_constant(273.15, $p);
    new_adder($m, $p, $n);
}
```

The final adder node expresses the constraint that $C + 273.15 = K$. Note that the wire `$m` has been attached to three nodes, as shown here.



These definitions of local propagation networks are quite verbose, but it's easy to imagine attaching a front-end that would allow the programmer to enter the desired constraints in ordinary algebraic notation:

```
C = (F+32)*5/9 ;
K = C + 273.15 ;
```

The front-end would have a parser for expressions like the ones we've already seen. The output from the parser would be a constraint network corresponding to the input expressions. Central to the parser would be productions like these, that would build up the appropriate constraint network as the input expression was analyzed:

```
$expression = operator($Term,
                       [lookfor(['OP', '+']),
```

```
  sub { my $sum = Wire->new;
        new_adder($_[0], $_[1], $sum);
        return $sum;
      }
 [lookfor(['OP', '-']),
  sub { my $difference = Wire->new;
        new_adder($difference, $_[1], $_[0]);
        return $difference;
      }
);
```

9.2.2 Problems with Local Propagation

If you've ever seen a discussion of local propagation networks before, you've probably seen the Fahrenheit-Celsius converter example. There's a good reason for this: It's one of the few examples for which local propagation actually works.

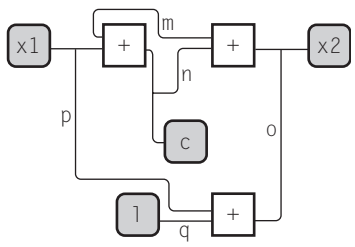
Let's consider a different problem, almost as simple. Suppose we're building a drawing system. A horizontal line has two endpoints at (x_1, y) and (x_2, y). Its center point is at (c, y), and its length is l. y is independent of the other parameters, but any two of x_1, x_2, c, and l determine the other two. We might reason that the center point is the one that is the same distance from each endpoint, and define the center point with the equation:

$$c - x_1 = x_2 - c$$

The length, of course, is the distance between the endpoints:

$$x_2 - x_1 = l$$

These two constraints yield the network shown here.



If we set x_1 to 3 and c to 5, everything works out. The 5 is propagated along wire n to the leftmost + node, which sets wire m to 2. This value, plus the 5 reaching the upper-right + node along wire n, causes wire o to be set to 7, which is reported as the value of x_2. Wire o, carrying 7, and wire p, carrying x_1's value of 3, arrive at the lower + node, allowing the network to deduce that the value of l is 4.

But suppose instead we set x_1 to 3 and x_2 to 7. The two values arrive at the lower + node, allowing the calculation of l as before. But there's a problem in the upper part of the diagram. Each of the two upper + nodes has only one defined input. Neither m nor n is defined, and each is needed for the deduction of the other one. Since wire n defines the value of c, the network has failed. Similarly, although the network above can compute x_2 from x_1 and l, it fails to compute c.

This kind of problem often arises when local constraint networks contain loops. In general, we can't avoid constraints that result in loops, so we need another technique.

One technique that's commonly used in such cases is called *relaxation*. We tell the network to *guess* a value for c, and to compute the consequences of the guess. In general, this will result in an inconsistent network. In the preceding example, we might guess that c is 0. This means that n is 0, and then the two upper addition nodes can compute values for wire m. The leftmost one computes that m is -3, and the rightmost one that m is -7. These are inconsistent, so the network averages them, getting -4, and tries that out as a value for m. If m is -4, then the two addition nodes want to set wire n to -1 and to 11, respectively. So the network once again tries the average, 5, for n. This time, the two addition nodes agree that m should be 2 — so the relaxation is complete, and has solved the constraint equations.

As with nearly all numerical techniques, relaxation is fraught with peril. Sometimes the relaxation process will diverge: Instead of reaching the correct value, the successive steps produce more and more grossly incorrect values. Sometimes the relaxation process converges slowly to the correct values, getting closer and closer but never quite making it.

Getting local propagation networks to work well is an active research area. I introduced the technique because it's an interesting exercise and a good introduction to the idea of constraint systems. But for the rest of the chapter, we're going to go a different way.

9.3 LINEAR EQUATIONS

As a large example, we'll develop a system, called `linogram`, for drawing diagrams.

Diagrams are usually drawn with a WYSIWYG structured drawing system. The big drawback of this kind of system is that if you want to change the diagram globally, you essentially have to start over. For example, suppose you were drawing a family tree, and you decided to represent each person with a rectangle 0.75 inches wide by 0.5 inches tall. You get the diagram done, but then you learn that the diagram will need to be printed in landscape mode, rather than portrait mode. You want to make the boxes shorter, to fit on the shorter page, but wider, to fit the text in — say 1 inch wide and only 0.4 inches tall. Also, you want to see how the diagram looks if the corners of the boxes are rounded off.

In the typical structured drawing system, you'd have to manually adjust each box and the text inside it. In a declarative drawing system, however, this kind of change is easy. A diagram is like a program, and there is a definition in the program that describes the kind of box you want to use for a person. By changing the definition, you change every box of that type in the entire diagram.

In the declarative drawing system, you can tell the computer to calculate the positions and sizes of drawing elements based on the positions and sizes of other elements. So, for example, you can easily tell the system that you want all the squares to be made into rectangles, or all the straight arrows made into curved arrows, or all the parts of the diagram that represent widgets to be drawn with three round knobs instead of two square knobs, just by changing a small part of the description of the diagram.

Since the input to the declarative system is a plain text file, it's also easy to get another program to generate diagrams as output.

If we're going to describe objects by giving constraints, we need some way of solving the constraints to figure out where the objects actually are. As we saw, local propagation won't do it. In general, the problem is very difficult, because constraints are equations, so solving constraints means solving equations. If solving equations were easy, we wouldn't have to suffer through four years of high school algebra learning how to do it, and we wouldn't need mathematicians to figure it out.

For general geometric problems, we have to solve general sorts of equations; these may involve higher algebra, or even trigonometry. There is one kind of equation, however, that's easy to solve. Linear equations are easy. The solution of:

$$ax + b = c$$

is:

$$x = \frac{c - b}{a}$$

Because diagrams usually involve many straight lines, linear equations usually do most of what we want. The kind of curves that appear in diagrams are unusually simple and highly constrained. It may require advanced mathematics to find the intersection of a lemniscate and a cardioid, but how often do you draw a diagram with a lemniscate and a cardioid? Diagrams do involve circles (which potentially opens up a can of trigonometric worms), but typically the circle is used as just another kind of box. If we allow drawing elements to be attached to circles only at the "corners" (the northmost, northwestmost, etc. points) then the circle is essentially an octagon as far as the equations are concerned; then once we figure out where the corners are located, we can join them with curves instead of straight sides.

9.4 `linogram`: A DRAWING SYSTEM

The entities with which our program deals are called *features*. A feature represents something like a box or a line. It might contain sub-features; for example, a box feature contains four sub-features that represent its four sides. A feature also contains a list of constraint equations that define the relationships between its sub-features; for example, in a box feature, the top and left sides are constrained to start at the same point.

The input to the drawing system will be a specification for a large compound feature, called the *root feature*, which represents the entire drawing. Here's an example specification for a root feature:

```
box F, plus, con32, times, C, con59;
line i, j, k, l, m;
number hspc, vspc, boxht, boxwd;

constraints {
  boxht = 1; boxwd = 1;
  hspc = 1 + boxwd; vspc = 1 + boxht;

  F.ht = boxht; F.wd = boxwd;

  plus = F + (hspc, 0);
  con32 = plus + (hspc, 0);
  times = plus + (0, vspc);
  C = times + (hspc, 0);

  con59 + (hspc, 0) = times;
```

```
  i.start = F.e;     i.end = plus.nw;
  j.start = plus.e;  j.end = con32.w;
  k.start = plus.sw; k.end = times.nw;
  l.start = con59.e; l.end = times.sw;
  m.start = times.e; m.end = C.w;

  F.nw = (0,0);
}
```

The first three lines declare the sub-features of the root feature; it contains six boxes, five lines, and four numbers. Numbers are primitive, and don't contain sub-features. The numbers `hspc` and `vspc` will be used to determine the amount of space between the boxes. If we want to move all the boxes closer together in the horizontal direction, we will need only to change the definition of `hspc` to a smaller value. Similarly, `boxht` and `boxwd` will be the height and width of each of the six boxes.

The `constraints` section is the really interesting part. It's a list of linear equations that specify the sizes and relative locations of the boxes and lines. The first four equations define the four numeric parameters `boxht`, `boxwd`, `hspc`, and `vspc`. `hspc` represents the minimum center-to-center horizontal separation of two nearby boxes, so it's defined in terms of `boxwd`: The distance between the two centers is the width of one box, plus one unit of space.

The next two equations define the height and the width of the `F` box by establishing constraints on its subfeatures `F.ht` and `F.wd`. The definition of the `box` type (which we'll see later) contains a declaration like:

```
number ht, wd;
```

to say that every box has these two properties, and other declarations that relate these numbers to the positions of the four sides.

The next equation,

```
plus = F + (hspc, 0);
```

constrains the size, shape, and position of the `plus` box. The `(hspc, 0)` is called a *tuple expression* and represents a displacement. The constraint says that the box `plus` is exactly like `F`, only displaced eastward by `hspc` units and southward by 0 units. Internally, this will translate into a series of constraints that force each of `plus`'s four corners and four sides to be `hspc` units east and 0 units south of the corresponding corners and sides of `F`.

Although this equation looks like an assignment, it isn't; it's a declaration. If `linogram` knows about `F`, it can deduce the corresponding information about `plus` — or vice versa. It can also deduce complete information about both from partial information. For example, if only the left side of `F` and the top side of `plus` are known, then the other sides of the two boxes can all be deduced: The left side of `plus` is like the left side of `F`, and the top side of `F` is like the top side of `plus`.

We could also have written this equation in any of these mathematically equivalent forms:

```
F + (hspc, 0) = plus;
plus + (-hspc, 0) = F;
plus - F = (-hspc, 0);
plus - (hspc, 0) - F = 0;
```

Sometimes it's convenient to write equations like this. For example, suppose we have four features, `A`, `B`, `C`, and `D`. We're not sure where `A`, `B`, and `C` are, but we know that we want `D`'s position relative to `C` to be the same as `B`'s position relative to `A` — if `B` is one furlong due north of `A`, we want `D` one furlong due north of `C`, or whatever. It's quite straightforward and intuitive to express it like this:

```
D - C = B - A;
```

Or suppose we wanted point `Z` to be one-third of the way from `X` to `Y` along the straight line between them:

```
Z - X = 1/3 * (Y - X);
```

In addition to a height and a width, every box has thirteen more subfeatures: four lines and nine points. The lines represent the four sides, and are named `top`, `bottom`, `left`, and `right`. The points aren't strictly necessary, but they're convenient. They are the four corner points, called `nw`, `ne`, `sw`, and `se`, the midpoints of the four sides, called `n`, `s`, `e`, and `w`, and the center point, called `c`.

Similarly, a line contains two sub-features, called `start` and `end`, that denote its two endpoints.

The next few declarations in our specification define the endpoints of the five lines `i` through `m`.

The declarations:

```
i.start = F.e;     i.end = plus.nw;
```

constrain line `i` to start at the midpoint of `F`'s east side, and to end at the northwest corner of box `plus`.

Finally, we have to tell the program the absolute location of at least one of the features, or it won't be able to figure out where anything is located. We force the issue by attaching the northwest corner of box `F` arbitrarily to `(0,0)`, although it doesn't really matter; we could as easily have attached any other point of any of the boxes.

In addition to these manifest constraints, there are a large number of hidden constraints that we don't see, inherent in the definitions of the `box` and `line` types. For example, the definition of `box` has, among others,

```
top.start = left.start;
nw = top.start;
top.start + wd = top.end;
n = top.center;
...
```

and the definition of `line` has:

```
center = (start + end)/2;
```

Again, although this looks like an assignment, it isn't; it's symmetric. If the start and end points of the line are known, the center will be calculated from them; if the start and center are known instead, the position of the end point will be calculated instead. Any two of the points imply the third.

The program's strategy for drawing a diagram is as follows. First it will read in the definition of the root feature, including the implied definitions of common sub-features such as `box`. It will accumulate a large set of linear constraint equations. These will include the explicit constraints, as well as many automatically generated implicit constraints. If the root feature contains a `box` named `F`, then it will also include `F`'s constraints implicitly, in the form of equations like these:

```
F.top.start = F.left.start;
F.nw = F.top.start;
F.top.start + F.wd = F.top.end;
F.n = F.top.center;
...
```

In fact, since `F` itself contains several sub-features, it will inherit constraints from these. `F`'s top side is a line, so `F` will inherit the constraint:

```
top.center = (top.start + top.end)/2;
```

from the definition of `line`; this will in turn be inherited by the root feature as:

```
F.top.center = (F.top.start + F.top.end)/2;
```

After accumulating all the constraint equations, the program will solve the equations. The result will be a complete description of where every part of each feature is located.

Associated with each feature will be one or more drawing functions. The program will invoke the drawing functions for each feature, passing them a hash containing the relevant variables. It's up to the drawing functions to generate the appropriate output. The output might be instructions in PostScript to be sent to a printer, or perhaps a "canvas" object containing a bitmap of the finished diagram.

Before we go any further with the main program, let's look at the definitions of the simpler sub-features such as boxes, which will be instructive. The simplest features that the program deals with are numbers, which are atomic. These are the only features whose definitions are built into the program. All other features are defined by a library file that specifies the feature's sub-features, constraints, and drawing methods.

After a number, the simplest feature is a `point`, which has `x` and `y` coordinates, but no constraints on them:

```
define point {
  number x, y;
}
```

When `linogram` wants to draw a feature, its default behavior is to recursively draw all the feature's sub-features. Thus it draws a `point` by trying to "draw" the two numbers `x` and `y`. Numbers are considered to be invisible, so the aggregate behavior for drawing a point is also to do nothing. The simplest visible feature is a line, which has start and end points:

```
define line {
  point start, end, center;
  constraints { center = (start + end)/2; }
  draw { &draw_line; }
}
```

As mentioned before, a line also has a `center` point, for convenience; it's constrained to be halfway between the start and end points (see Figure 9.3).

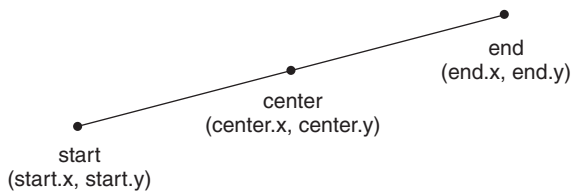


FIGURE 9.3 A line and its subfeatures.

The `draw` section is new. The declaration shown here is the name of a Perl subroutine responsible for drawing the feature. The `&` is a lexical marker that indicates that this is the name of a subroutine. When invoked, the subroutine will be passed a hash that indicates the positions of the sub-features of the line:

```
("start.x" => 5, "start.y" => 3,
 "end.x" => 3, "end.y" => 7,
 "center.x" => 4, "center.y" => 5,
)
```

If any of the sub-features are unknown, they'll be omitted from the hash; in that case, the function should complain. Since this chapter is about declarative programming, and not about graphics, we'll weasel out of doing any actual drawing, and use the following drawing function, which claims to draw lines even though it doesn't really draw anything. It does, however, give us a clear description of the line it *would have* drawn, which is enough to see whether the program is doing what it should be doing:

```
sub draw_line {
  my $env = shift;
  my $GOOD = 1;
  for my $k (qw(start.x start.y end.x end.y)) {
    unless (defined $env->{$k}) {
      warn "Can't draw line because '$k' is missing\n";
      $GOOD = 0;
    }
  }
  if ($GOOD) {
    print "Drawing line from ($env->{'start.x'}, $env->{'start.y'})
           to ($env->{'end.x'}, $env->{'end.y'})\n";
  }
}
```

Given the preceding hash, this will produce the output:

```
Drawing line from (5, 3) to (3, 7)
```

Even though we weaseled out of the drawing, creating a diagram in PostScript is barely more difficult. We would need to generate output something like this:

```
50 30 moveto 30 70 lineto stroke
```

This is almost the same, but there are a (very) few additional complications that I didn't want to have to consider, so we'll stick with the weasel drawing technique.

The other possible inhabitants of a `draw` section are the names of some of the sub-features that make up the feature. Only these sub-features will be drawn. If there is no `draw` section at all, the default is to draw all the sub-features.

We have enough machinery now to define boxes directly, but `linogram`'s standard library goes through a set of intermediate definitions first. The top and bottom sides of a box are constrained to be horizontal, and it's convenient to define a new feature type to represent a horizontal line:

```
define hline extends line {
  number y, length;
  constraints {
    start.y = end.y;
    start.y = y;
    start.x + length = end.x;
  }
}
```

This defines a new type, called `hline`, which has all of the sub-features and constraints that an ordinary `line` has, and some additional ones. The start and end points must have the same y-coordinate, and an `hline` also has an additional sub-feature, called `y`, which is defined to be equal to this y-coordinate. If we were trying to specify the location of a box `F`, this would allow us to abbreviate `F.top.start.y` as simply `F.top.y`, which is more natural. An `hline` also has a length, which is the distance between the endpoints. In general, the length of a line is not a linear function of the positions of the endpoints (because $\mathit{length} = \mathrm{sqrt}((\mathit{end}.x - \mathit{start}.x)^2 + (\mathit{end}.y - \mathit{start}.y)^2)$) and computing one point given the length and the other endpoint requires trigonometry, which `linogram` won't do. But for horizontal lines, the calculation is trivial.

The constraints in this definition are adjoined to those inherited from `line`, which imply the position of the `center` point of an `hline`, even though we never

mentioned it explicitly. The draw section is also inherited from line, so that the Perl draw_line function will be used for hline as well.

Vertical lines are almost exactly the same:

```
define vline  extends line {
  number x, height;
  constraints {
    start.x = end.x;
    start.x = x;
    start.y + height = end.y;
  }
}
```

Now we're ready to define box. It has a lot of machinery, but none of it is new:

```
define box {
  vline left, right;
  hline top, bottom;
  point nw, n, ne, e, se, s, sw, w, c;
  number ht, wd;

  constraints {
    left.start  = top.start;
    right.start = top.end;
    left.end    = bottom.start;
    right.end   = bottom.end;

    nw = left.start;
    ne = right.start;
    sw = left.end;
    se = right.end;
    n = top.center;
    s = bottom.center;
    w = left.center;
    e = right.center;

    c = (n + s)/2;

    ht = left.height;
    wd = top.length;
  }
}
```

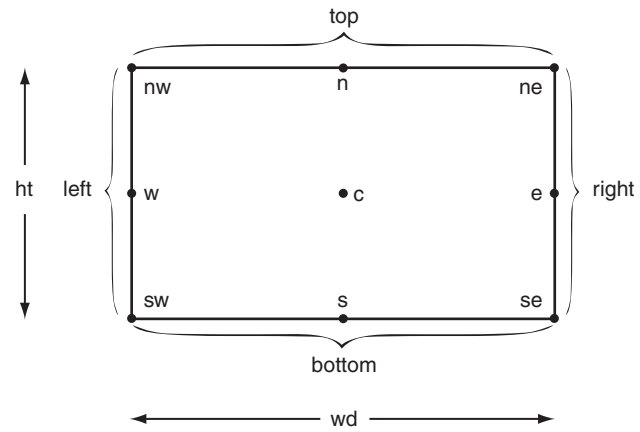


FIGURE 9.4 A box and its subfeatures.

A box has a left and a right side, which are `vlines`, and a top and a bottom side, which are `hlines`. It has nine named points, which are identical to various parts of the four sides, except for `c`, the center, which is halfway between the north and south points. It also has a height and a width, which are the same as the lengths of the left and top sides, respectively (see Figure 9.4). We didn't need to require that `ht = right.height`; this is already implicit in the other equations, although it wouldn't have hurt to put it in.

The `box` definition doesn't contain a `draw` section. The default behavior is for `linogram` to draw a box by drawing each of its fifteen sub-features. For the nine points and the two numbers, this does nothing at all; the other four sub-features are the four sides, which `linogram` draws by calling `draw_line`. Each box will therefore result in four calls to `draw_line`, which is just what we want.

To define a square, we need only write:

```
define square extends box {
  constraints { ht = wd; }
}
```

which defines a `square` to be the same as a `box` but with the height and width constrained to be equal. Another common constituent of diagrams is an arrow. From `linogram`'s point of view, this is nothing more than an oddly-drawn line:

```
define arrow extends line {
  draw { &draw_arrow; }
}
```

An arrow has a start and end point, just like a line; these are the start and end points of the arrow's shaft. The `draw_arrow` function is responsible for drawing the shaft (which it can do by calling `draw_line`) and then filling in the two whiskers at the endpoint.

If we're feeling creative, we might go on:

```
define golden_rectangle extends box {
  constraints { ht * 1.618 = wd; }
}

define circle {
  number r, d;
  point c, nw, n, ne, e, se, s, sw, w;
  constraints {
    d = 2*r;

    n = c - (0, r);
    s = c + (0, r);
    e = c + (r, 0);
    w = c - (r, 0);
    se = c + ( r, r)/1.4142;
    sw = c + (-r, r)/1.4142;
    ne = c + ( r,-r)/1.4142;
    nw = c + (-r,-r)/1.4142;
  }
  draw { &draw_circle; }
}

define diamond extends box {
  line nw_side(start=n, end=w),
       sw_side(start=s, end=w),
       ne_side(start=n, end=e),
       se_side(start=s, end=e);
  draw { nw_side;
         sw_side;
         ne_side;
         se_side;
  }
}
```

The `nw_side(start=n, end=w)` declaration in the last definition is a shorthand for:

```
line nw_side;
constraints { nw_side.start = n;
              nw_side.end = w;
            }
```

`linogram` has a few other features, but we'll see them in the course of seeing the program code. The program code comprises three major classes and several less-important classes. The three major classes are `Constraint`, which represents constraints; `Type`, which represents feature types such as `box` and `line`; and `Value`, which represents the value of an expression as it is being converted to a set of constraints. We'll see constraints and equations first.

9.4.1 Equations

The heart of `linogram` will be the module that solves systems of linear equations. The usual way to do this is to represent the system as a matrix, and then perform sequences of matrix transformations on it until the matrix is in a canonical form; this is called *Gaussian elimination*. Methods for doing this are well studied, and also available on CPAN. But for various reasons, the CPAN modules I found for solving linear equations didn't seem to be what I wanted, so I'll develop one here.

An `Equation` object is a hash. The equation:

$$14x + 9y - 3.5z = 28$$

is represented by the hash:

```
{ "x" => 14,
  "y" => 9,
  "z" => -3.5,
  ""  => -28,
}
```

The values 14, 9, and -3.5, are called the *coefficients* of x, y, and z, respectively. The -28 is the *constant part*. It's negative because the equation is actually:

$$14x + 9y - 3.5z - 28 = 0$$

The "" key in the hash is mandatory because every linear equation has a constant part, even if the constant part is 0. The equation:

$$x = 0$$

corresponds to the hash:

```
{ "x" => 1,
  "" => 0,
}
```

and the trivial equation $0 = 0$ is represented by the hash `{ "" => 0 }`.

Manipulating equations through these hashes is straightforward and easy to debug, although slow. If speed is an issue, the `Equation` module of the program should be replaced with one that uses a more abbreviated representation of equations, perhaps one implemented in C.

The constructor function takes an argument hash and puts it into a canonical form:

CODE LIBRARY
Equation.pm

```
sub new {
  my ($base, %self) = @_;
  $class = ref($base) || $base;
  $self{""} = 0 unless exists $self{""};
  for my $k (keys %self) {
    if ($self{$k} == 0 && $k ne "") { delete $self{$k} }
  }
  bless \%self => $class;
}
```

If the constant part is missing, the constructor sets it to 0; if the coefficients of any of the variables are 0, they are deleted. For example, `->new("x" => 0, "y" => 1)`, which represents $0x + 1y = 0$, is turned into `{ "y" => 1, "" => 0 }`.

`ref($base) || $base`

One idiom used here and elsewhere that you may not have seen is the `ref($base) || $base` trick. The goal is to write a function that can be called as either an object or a class method, either as:

```
Equation->new(...)
```

or as;

```
$some_equation->new(...)
```

In the former case, `$base` is the string `Equation`, and `ref $base` is false, since `$base` is a string rather than a reference. `$class` is therefore set equal to `$base`. In the latter case, `$base` is the object `$some_equation`, and `ref($base)` is the class into which `$some_equation` was blessed. `$class` is therefore set equal to `$some_equation`'s class. This is convenient when we'll be writing several other constructor methods that might get an `Equation` object as an argument and will want to create another object similar to it. For example, here's a method that makes a copy of an `Equation` object:

```
sub duplicate {
  my $self = shift;
  $self->new(%$self);
}
```

Note that:

```
# WRONG!
sub duplicate {
  my $self = shift;
  Equation->new(%$self);
}
```

doesn't work properly if its argument is an object of a class derived from `Equation`. The correct code creates a new object from the same derived subclass; the incorrect code creates a new `Equation` object regardless.

SOLVING EQUATIONS

For convenience, we set up a constant for the important trivial equation $0 = 0$:

```
BEGIN { $Zero = Equation->new() }
```

Equations have three important accessors. One retrieves the coefficient of a given variable:

```
sub coefficient {
  my ($self, $name) = @_;
```

```
  $self->{$name} || 0;
}
```

The second recovers the constant part:

```
# Constant part of an equation
sub constant {
  $_[0]->coefficient("");
}
```

The other returns the names of all the variables that the equation mentions:

```
sub varlist {
  my $self = shift;
  grep $_ ne "", keys %$self;
}
```

All equations can be scaled and added. If an equation is known to be true, you can multiply its constant and its coefficients by any number n, and the resulting equation is also true. For example, if:

$$14x + 9y - 3.5z = 28$$

then we can scale all the numbers by 2 and get:

$$28x + 18y - 7z = 56$$

which is equivalent.

If we have two equations that are true, we can add them together and get another true equation. For example, suppose we have:

$$x = 13$$
$$2y = 7$$

we can add these, getting:

$$x + 2y = 20$$

These two operations are fundamental to all methods of solving linear equations. For example, suppose we have:

$$x + y = 12$$
$$x - y = 2$$

If we add these two equations together, the $+y$ in the first and the $-y$ in the second cancel, yielding:

$$2x = 14$$

which we can then scale (by 1/2) to yield:

$$x = 7$$

We can then scale this by -1, yielding:

$$-x = -7$$

When we add this last equation to the very first equation, the x's cancel, and we're left with:

$$y = 5$$

And in fact $x = 7, y = 5$ is the solution of the equations.

The most important function in the `Equation` module is `arithmetic()`, which scales and adds equations:

```
sub arithmetic {
  my ($a, $ac, $b, $bc) = @_;
  my %new;
  for my $k (keys(%$a), keys %$b) {
    my ($av) = $a->coefficient($k);
    my ($bv) = $b->coefficient($k);
    $new{$k} = $ac * $av + $bc * $bv;
  }
  $a->new(%new);
}
```

Given two equations, `$a` and `$b`, and two numbers, `$ac` and `$bc`, `arithmetic()` scales `$a` by `$ac`, scales `$b` by `$bc`, and adds the two scaled equations together. Built atop this base are several simpler utility functions. For example, to add two

equations together, we use `arithmetic()`, with both scale factors set to 1:

```
sub add_equations {
  my ($a, $b) = @_;

  arithmetic($a, 1, $b, 1);
}
```

Similarly, to subtract one equation from another is the same as adding them, but with the second one negated:

```
sub subtract_equations {
  my ($a, $b) = @_;

  arithmetic($a, 1, $b, -1);
}
```

Scaling a single equation is yet another special case, where the second equation is zero:

```
sub scale_equation {
  my ($a, $c) = @_;
  arithmetic($a, $c, $Zero, 0);
}
```

Now suppose we have two equations:

$$ax + \text{some other stuff} = c$$
$$bx + \text{more stuff} = d$$

Here we can eliminate x from the first equation by scaling the second by $-a/b$ and adding the result to the first equation. The function `substitute_for()` is for eliminating a variable from an equation. The call:

```
$first->substitute_for("x", $second);
```

eliminates variable `"x"` from equation `$first` in this way, by combining it with an appropriately scaled version of `$second`:

```
# Destructive
sub substitute_for {
```

```
    my ($self, $var, $value) = @_;
    my $a = $self->coefficient($var);
    return if $a == 0;
    my $b = $value->coefficient($var);
    die "Oh NO" if $b == 0;  # Should never happen

    my $result = arithmetic($self, 1, $value, -$a/$b);
    %$self = %$result;
}
```

If `$a` is zero, then the first equation didn't contain the variable we were trying to eliminate, so nothing needs to be done. The `"Oh NO"` case occurs when the second equation doesn't contain the variable we're trying to eliminate; in this case there's no way to use it to eliminate the variable from the first equation. Note that the function is destructive: It modifies `$self` in place.

The cost of eliminating a variable like x is that the resulting equation might be more complicated than what we started with, depending on what else is in the equation we're using to reduce it. If we're not careful, we might even get stuck in an infinite loop. Suppose we had:

$$x + y = 3$$

$$y + z = 5$$

and we scale the second equation by -1 and add it to the first, to eliminate y:

$$x - z = -2$$

If we then add *this* equation to the second one to eliminate z, we're back where we started.

We'll adopt a simple strategy that prevents infinite loops. We'll take the first equation and use it to completely eliminate one of its variables from all the other equations. The variable will be present in that first equation only, so as long as we don't use the first equation again, we can't possibly reintroduce that variable. We'll then move to the second equation and use *it* to eliminate one of *its* variables from all the other equations. We'll repeat this for each equation.

To that end, here's a method that returns an arbitrarily chosen variable from an equation:

```
sub a_var {
  my $self = shift;
```

```
    my ($var) = $self->varlist;
    $var;
}
```

Let's see a small example of how this works. Consider the equations:

$$\begin{aligned} A: &\quad x + 2y = 8 \\ B: &\quad 2y + z = 10 \\ C: &\quad x + y + 2z = 13 \end{aligned}$$

First we use A to eliminate x from the other two equations. For equation B there is nothing to do; eliminating x from C leaves:

$$\begin{aligned} A: &\quad x + 2y = 8 \\ B: &\quad 2y + z = 10 \\ C: &\quad -y + 2z = 5 \end{aligned}$$

Now we use B to eliminate y from the other two equations. Eliminating y from A leaves:

$$\begin{aligned} A: &\quad x - z = -2 \\ B: &\quad 2y + z = 10 \\ C: &\quad -y + 2z = 5 \end{aligned}$$

Eliminating y from C leaves:

$$\begin{aligned} A: &\quad x - z = -2 \\ B: &\quad 2y + z = 10 \\ C: &\quad 2.5z = 10 \end{aligned}$$

Finally, we use C to eliminate z from the other two equations:

$$\begin{aligned} A: &\quad x = 2 \\ B: &\quad 2y = 6 \\ C: &\quad 2.5z = 10 \end{aligned}$$

At this point we have finished one complete pass through all the equations, so we are done. There's a final step that needs to be done to put the equations in

standard form: We must adjust the coefficients to 1:

$$\begin{aligned} A: &\quad x = 2 \\ B: &\quad y = 3 \\ C: &\quad z = 4 \end{aligned}$$

but this is a simple scaling operation.

Solving entire systems of equations is the job of the `Equation::System` module, whose objects represent whole systems of equations:

```
package Equation::System;

sub new {
  my ($base, @eqns) = @_;
  my $class = ref $base || $base;
  bless \@eqns => $class;
}
```

In the course of solving a system of equations, we often find that some of them are redundant. The way this appears in the mathematics is that we reduce an equation and find that we have nothing left. (That is, nothing but $0 = 0$, which adds no useful information.) We can detect such a ghostly equation with `Equation::is_tautology`:

```
package Equation;

sub is_tautology {
  my $self = shift;
  return $self->constant == 0 && $self->varlist == 0;
}
```

In such a case, we'll replace the ghostly equation with `undef`.

The important accessor for an `Equation::System` recovers the current list of equations, ignoring the ones we have nulled out:

```
package Equation::System;

sub equations {
  my $self = shift;
  grep defined, @$self;
}
```

A typical operation on a system of equations will be to transform each equation in some way:

```
sub apply {
  my ($self, $func) = @_;
  for my $eq ($self->equations) {
    $func->($eq);
  }
}
```

Now we're ready to see `Equation::System::solve`, the end product of all this machinery.

```
sub solve {
  my $self = shift;
  my $N = my @E = $self->equations;
  for my $i (0 .. $N-1) {
    next unless defined $E[$i];
    my $var = $E[$i]->a_var;
    for my $j (0 .. $N-1) {
      next if $i == $j;
      next unless defined $E[$j];
      next unless $E[$j]->coefficient($var);
      $E[$j]->substitute_for($var, $E[$i]);
      if ($E[$j]->is_tautology) {
        undef $E[$j];
      } elsif ($E[$j]->is_inconsistent) {
        return ;
      }
    }
  }
  $self->normalize;
  return 1;
}
```

The main loop selects an equation number *i*, selects one if its variables, `$var`, and then scans over all the other equations *j* reducing each one to remove `$var`. If the result is the trivial equation $0 = 0$, equation *j* is nulled out.

After each reduction, we test the resulting equation to make sure it makes sense. If we get an equation like $1 = 0$, we know something has gone wrong.

This will occur if the original equations were inconsistent. For example:

$$start.y = 1;$$
$$y = 2;$$
$$start.y - y = 0;$$

Eliminating *start.y* from the others yields:

$$start.y = 1;$$
$$y = 2;$$
$$-y = -1;$$

Then using the second equation to eliminate y from the others yields:

$$start.y = 1;$$
$$y = 2;$$
$$0 = 1;$$

which is no good, because it says that $0 = 1$. The `Equation::is_inconsistent` method detects bad equations like $0 = 1$ that have no variables, but whose constant part is nonzero:

```
package Equation;

sub is_inconsistent {
  my $self = shift;
  return $self->constant != 0 && $self->varlist == 0;
}
```

When the main loop is finished, we hope that the equations in the system have been reduced to the point where they contain only one variable each. As we saw, the equations might need one final adjustment. An equation like this:

$$2y = 6$$

should be adjusted to this:

$$y = 3$$

The Equation::System::normalize method adjusts the equations in this way:

```
package Equation::System;

sub normalize {
  my $self = shift;
  $self->apply(sub { $_[0]->normalize });
}
```

To normalize a single equation, we scale it appropriately:

```
package Equation;

sub normalize {
  my $self = shift;
  my $var = $self->a_var;
  return unless defined $var;
  %$self = %{$self->scale_equation(1/$self->coefficient($var))};
}
```

An equation like $y = 3$ is so simple that even the computer understands what it means. We say that this equation *defines* the variable y. The defines_var() method reports on whether an equation defines a variable:

```
sub defines_var {
  my $self = shift;
  my @keys = keys %$self;
  return unless @keys == 2;
  my $var = $keys[0] || $keys[1];
  return $self->{$var} == 1 ? $var : () ;
}
```

To define a variable, an equation must have the form *var* = *val*, and so must contain exactly two keys. One is the name of the variable; the other is the empty string. Moreover, the coefficient of the one variable must be 1. If all this is true, defines_var() returns the name of the variable so defined. The value of the variable can be recovered with - $equation->constant. (The minus sign is because $y = 7$ is represented as $y - 7 = 0$, which is { y => 1, "" => -7 }.)

The main entry to the equation-solving subsystem for outside functions is the values() method. This takes a system of equations, solves the equations,

and returns a hash that maps the names of known variables to their values:

```
package Equation::System;

sub values {
  my $self = shift;
  my %values;
  $self->solve;
  for my $eqn ($self->equations) {
    if (my $name = $eqn->defines_var) {
      $values{$name} = -$eqn->constant;
    }
  }
  %values;
}

1;
```

CONSTRAINTS

`linogram` will have another class, called `Constraint`, which represents constraints. Since constraints are essentially equations, `Constraint` will be a derived class of `Equation`:

CODE LIBRARY
Constraint.pm

```
package Constraint;
use Equation;
@Constraint::ISA = 'Equation';
```

`Constraint` adds a few utility methods to `Equation` that make more sense in the context of `linogram` than in the general context of equation solving. The most important is `qualify()`. A type like `hline` contains the constraint $start.y - y = 0$. But when considered as part of a `box`, the `hline` has a name like `top` or `bottom`, and the constraint, when translated into the context of the box, turns into $top.start.y - top.y = 0$. `qualify()` takes a constraint and a name prefix and produces a new, transformed constraint:

```
sub qualify {
  my ($self, $prefix) = @_;
  my %result = ("" => $self->constant);
```

```
  for my $var ($self->varlist) {
    $result{"$prefix.$var"} = $self->coefficient($var);
  }
  $self->new(%result);
}
```

Constraint's other methods are simple things. In some places inside linogram, constraints are used as if they were expressions; when there is an expression with an addition in the drawing specification, we have to add together constraints. We'll see this in more detail later; in the meantime, new_constant() manufactures a constraint like $0 = 0$ or $0 = 1$ that plays the role of a constant expression:

```
sub new_constant {
  my ($base, $val) = @_;
  my $class = ref $base || $base;
  $class->new("" => $val);
}
```

add_constant() adds a constant to a constraint, transforming something like $x = 0$ to something like $x = 3$, and mul_constant() multiplies a constraint by a constant, transforming something like $x = 3$ to something like $4x = 12$:

```
sub add_constant {
  my ($self, $v) = @_;
  $self->add_equations($self->new_constant($v));
}

sub mul_constant {
  my ($self, $v) = @_;
  $self->scale_equation($v);
}
```

All the other methods of Constraint are inherited from Equation.

Analogous to Constraint, there is a Constraint_Set class that is derived from Equation::System. It's even simpler than Constraint. It has only one extra method:

```
package Constraint_Set;
@Constraint_Set::ISA = 'Equation::System';

sub constraints {
```

```
  my $self = shift;
  $self->equations;
}

1;
```

9.4.2 Values

In the course of reading and parsing the specification, we'll need to deal with expressions. We saw the parsing end of this in detail in Chapter 8. The question that arises is what the values of the expressions will be; the answer turns out to be quite interesting. Values are not always numbers. For example, consider:

```
point P, Q;
P + (2, 3) = Q;
```

Here we have an expression `P + (2, 3)`. The value of this expression isn't a simple number. It implies parts of two constraints, involving $P.x$ and $P.y$. Later on, these partial constraints must be combined with Q to yield the complete constraints, which are $P.x + 2 = Q.x$ and $P.y + 3 = Q.y$.

One of `linogram`'s main classes is `Value`, which represents the value of an expression. `Value` is where the most interesting arithmetic takes place inside of `linogram`. `Value`s come in three kinds. `Value::Constant` represents a scalar constant value such as 3. `Value::Tuple` represents a lone tuple, such as `(2, 3)`, or a sum of tuples. And `Value::Feature` represents a feature type, even a scalar feature type, such as P or Q or `P + (2, 3)`. `Value` itself is an abstract base class, and doesn't represent anything; it's there only to provide methods that are inherited by the other classes, primarily for doing arithmetic.

`Value` objects have one generic accessor, called `kindof()`, which returns `CONSTANT`, `TUPLE`, or `FEATURE`, depending on what kind of object it is called on. The other methods are arithmetic. The entry to these from the parser is via a quartet of operation methods called `add()`, `sub()`, `mul()`, and `div()`, which are just thin wrappers around the real workhorse, `op()`:

```
sub add { $_[0]->op("add", $_[1]) }
sub sub { $_[0]->op("add", $_[1]->negate) }
sub mul { $_[0]->op("mul", $_[1]) }
sub div { $_[0]->op("mul", $_[1]->reciprocal) }
```

Note that subtraction and division are defined in terms of addition and multiplication, which cuts down on the amount of work we need to do for `op()`.

`op()` itself is driven by a dispatch table because otherwise it would be quite complicated. The dispatch table is indexed by the operation name (either `add` or `mul`) and by the kinds of the two operands. It looks like this:

CODE LIBRARY
Value.pm

```
package Value;

my %op = ("add" =>
          {
           "FEATURE,FEATURE"    => 'add_features',
           "FEATURE,CONSTANT"   => 'add_feature_con',
           "FEATURE,TUPLE"      => 'add_feature_tuple',
           "TUPLE,TUPLE"        => 'add_tuples',
           "TUPLE,CONSTANT"     => undef,
           "CONSTANT,CONSTANT"  => 'add_constants',
           NAME => "Addition",
          },
          "mul" =>
          {
           "FEATURE,CONSTANT"   => 'mul_feature_con',
           "TUPLE,CONSTANT"     => 'mul_tuple_con',
           "CONSTANT,CONSTANT"  => 'mul_constants',
           NAME => "Multiplication",
          },
         );
```

Addition, surprisingly, turns out to be more complicated than multiplication. This is because we've restricted our system to linear operations, which means that multiplication is forbidden, except to multiply by constant values. Given two `Value` objects and an operation tag, `op()` consults the dispatch table, dispatches the appropriate arithmetic function, and returns the result:

```
sub op {
  my ($self, $op, $operand) = @_;
  my ($k1, $k2) = ($self->kindof, $operand->kindof);
  my $method;
  if ($method = $op{$op}{"$k1,$k2"}) {
    $self->$method($operand);
  } elsif ($method = $op{$op}{"$k2,$k1"}) {
```

```
      $operand->$method($self);
    } else {
      my $name = $op{$op}{NAME} || "'$op'";
      die "$name of '$k1' and '$k2' not defined";
    }
  }
```

The two operands are `$self` and `$operand`. `op()` starts by finding out what sorts of values these are, using `kindof`, which returns `CONSTANT` for `Value::Constant` objects, `TUPLE` for `Value::Tuple` objects, and so forth. It then looks in the dispatch table under the operator name (`"add"` or `"mul"`) and the value kinds. If it doesn't find anything, it tries the operands in the opposite order, since a function for adding a tuple to a feature is the same as one for adding a feature to a tuple; this cuts down on the number of functions we have to write. If neither operand order works, then the `op` function fails with a message like `"Addition of 'CONSTANT' and 'TUPLE' not defined"`.

The only other generic methods in `Value` are for `negate()`, which is required for subtraction, and `reciprocal()`, which is required for division. `negate()` passes the buck to a general scaling method, which will be defined differently in each of the various subclasses:

```
sub negate { $_[0]->scale(-1) }
```

`reciprocal()` is even simpler, because in general it's illegal. You're not allowed to divide by a tuple (what would it mean?) or by a feature (since this would mean that the equations were nonlinear; consider $x = 1/y$) so the default `reciprocal()` method dies:

```
sub reciprocal { die "Nonlinear division" }
```

You *are* allowed to divide by a constant, so `Value::Constant::reciprocal()` will override this definition.

CONSTANT VALUES

Of the three kinds of `Value`, we'll look at `Value::Constant` first, because it's by far the simplest. `Value::Constant` objects are essentially numbers. The object is a hash with two members. One is the kind, which is `CONSTANT`; the other is the numeric value. The constructor accepts a number and generates

a Value::Constant value with the number inside it:

```
package Value::Constant;
@Value::Constant::ISA = 'Value';

sub new {
  my ($base, $con) = @_;
  my $class = ref $base || $base;
  bless { WHAT => $base->kindof,
          VALUE => $con,
        } => $class;
}

sub kindof { "CONSTANT" }

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

To perform the scale() operation, we multiply the constant by the argument:

```
sub scale {
  my ($self, $coeff) = @_;
  $self->new($coeff * $self->value);
}
```

Division is defined for constants, so we must override the fatal reciprocal() method with one that actually performs division. The reciprocal of a constant is a new constant with the reciprocal value:

```
sub reciprocal {
  my $self = shift;
  my $v = $self->value;
  if ($v == 0) {
    die "Division by zero";
  }
  $self->new(1/$v);
}
```

Finally, the dispatch table contains two methods for operating on constants. One adds two constants, and the other multiplies them:

```
sub add_constants {
  my ($c1, $c2) = @_;
```

```
  $c1->new($c1->value + $c2->value);
}

sub mul_constants {
  my ($c1, $c2) = @_;
  $c1->new($c1->value * $c2->value);
}
```

TUPLE VALUES

Tuples represent displacements. A tuple like `(2, 3)` represents a displacement of 2 units in the *x* direction (east) and 3 units in the *y* direction (south). As we'll see, `linogram` isn't restricted to two-dimensional drawings, so `(2, 3, 4)` could also be a legal displacement. Although it's unlikely that any four-dimensional beings will be using `linogram`, there's no harm in making it as general as possible, so internally, a tuple is a hash. The keys are component names (*x*, *y*, and so forth) and the values are the components. The tuple `(2, 3)` is represented by the hash `{ x => 2, y => 3 }`. `(2, 3, 4)` is represented by the hash `{ x => 2, y => 3, z => 4 }`. The tuple class itself doesn't care what the component names are, although this version of `linogram` will refuse to generate tuples with any components other than `x`, `y`, and possibly `z`.

One possibly fine point is that tuple components need not be numbers; they might be arbitrary `Values`. A tuple like `(3, hspc)` will have a *y* component that is a `Value::Feature`. It's even conceivable that we could have a tuple whose components are other tuples. We'll take some pains to forbid this last possibility, since it doesn't seem to have any meaning in the context of drawings.

Here is the constructor, which gets a component hash and returns a tuple value object:

```
package Value::Tuple;
@Value::Tuple::ISA = 'Value';

sub kindof { "TUPLE" }

sub new {
  my ($base, %tuple) = @_;
  my $class = ref $base || $base;
  bless { WHAT => $base->kindof,
          TUPLE => \%tuple,
        } => $class;
}
```

It has a few straightforward accessors:

```
sub components { keys %{$_[0]{TUPLE}} }
sub has_component { exists $_[0]{TUPLE}{$_[1]} }
sub component { $_[0]{TUPLE}{$_[1]} }
sub to_hash { $_[0]{TUPLE} }
```

To perform subtraction on tuples, we will need a `scale()` operation that multiplies a tuple by a number. This is done componentwise; `2 * (2, 3)` is `(4, 6)`:

```
sub scale {
    my ($self, $coeff) = @_;
    my %new_tuple;
    for my $k ($self->components) {
      $new_tuple{$k} = $self->component($k)->scale($coeff);
    }
    $self->new(%new_tuple);
}
```

Note that we must use `$self->component($k)->scale($coeff)` rather than `$self->component($k)->value * $coeff`, because the component value might not be a number.

Adding tuples will also be done componentwise. We want to make sure that the user doesn't try to add tuples with different components. It's not clear what `(2, 3) + (2, 3, 4)` would mean, for example. This function takes two tuples and returns true if their component lists are identical:

```
sub has_same_components_as {
  my ($t1, $t2) = @_;
  my %t1c;
  for my $c ($t1->components) {
    return unless $t2->has_component($c);
    $t1c{$c} = 1;
  }
  for my $c ($t2->components) {
    return unless $t1c{$c};
  }
  return 1;
}
```

Adding two tuples is one of the functions from the dispatch table:

```
sub add_tuples {
  my ($t1, $t2) = @_;
  croak("Nonconformable tuples") unless $t1->has_same_components_as($t2);

  my %result ;
  for my $c ($t1->components) {
    $result{$c} = $t1->component($c) + $t2->component($c);
  }
  $t1->new(%result);
}
```

The other dispatch table function that can return a tuple involves multiplying a tuple by a constant. This is a simple application of `scale()`:

```
sub mul_tuple_con {
  my ($t, $c) = @_;

  $t->scale($c->value);
}
```

FEATURE VALUES

The code for handling feature values isn't much longer than the code for handling tuples or constants, but it's more complex, because arithmetic of features is more complex. This is partly because it's not really clear what it should mean to add two boxes together.

What *does* it mean to add two boxes together? Suppose that A and B are `hlines`, and that we have the constraint $A = B$, or, equivalently, $A - B = 0$, which involves a subtraction of two `hline` features. What does this mean?

A contains several intrinsic constraints, including `A.start.x + A.length = A.end.x`, and B similarly contains `B.start.x + B.length = B.end.x`. The end value of $A - B$ must contain both of these constraints. The subtraction won't affect them at all. We will need to carry along all the intrinsic constraints from both input features into the result, but these intrinsic constraints don't otherwise participate in the arithmetic.

But the end value also must include some constraints that relate the two inputs, such as `A.end.y - B.end.y = 0`, `A.end.x - B.end.x = 0`, and so on. We'll call these *synthetic constraints*, because they must be synthesized out of information that we find in the input values.

A feature value has two parts, the *intrinsic constraints* and the *synthetic constraints*. Each is a set of constraints. The intrinsic constraints are those contributed by the definitions of the features themselves, and are internal to particular features. The synthetic constraints are those derived from the structure of the expression and the interactions between the features in the expression. The intrinsic constraints don't participate in arithmetic, while the synthetic constraints do participate in arithmetic.

When we want to add (or subtract) two boxes, we unite their two intrinsic constraint sets into a single set, which becomes the intrinsic constraint set of the result. But to combine the two synthetic constraint sets, we perform arithmetic on *corresponding* synthetic constraints. To keep track of which synthetic constraints correspond, each one is labeled with a string. A synthetic constraint that involves the `start.x` components of two `hlines` will be labeled with the string `start.x` and will be combined with the `start.x` components of any other lines involved in the expression. Synthetic constraint sets will therefore be hashes.

INTRINSIC CONSTRAINTS

Intrinsic constraint sets are represented by the class `Intrinsic_Constraint_Set`. An intrinsic constraint set is a simple container class that holds a list of `Constraint` objects:

```
package Intrinsic_Constraint_Set;

sub new {
  my ($base, @constraints) = @_;
  my $class = ref $base || $base;
  bless \@constraints => $class;
}

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

It has only a few methods. One is a `map`-like function for invoking a callback on each constraint in the set, and returning the set of the results:

```
sub apply {
  my ($self, $func) = @_;
  my @c = map $func->($_), $self->constraints;
  $self->new(@c);
}
```

This is used by `qualify()`, which qualifies all the constraints in the set:

```
sub qualify {
  my ($self, $prefix) = @_;
  $self->apply(sub { $_[0]->qualify($prefix) });
}
```

Last is `union()`, which takes one or more intrinsic constraint sets and generates a new set that contains all the constraints in the input sets:

```
sub union {
  my ($self, @more) = @_;
  $self->new($self->constraints, map {$_->constraints} @more);
}
```

SYNTHETIC CONSTRAINTS

`Synthetic_Constraint_Set` is more interesting, because it supports arithmetic rather than mere aggregation. As mentioned earlier, a synthetic constraint set is represented by a hash, because each constraint in the set has a label that is used to determine which constraints in other sets it will fraternize with. For convenience, the constructor accepts either a regular hash or a reference to a hash:

```
package Synthetic_Constraint_Set;

sub new {
  my $base = shift;
  my $class = ref $base || $base;
  my $constraints;
  if (@_ == 1) {
    $constraints = shift;
  } elsif (@_ % 2 == 0) {
    my %constraints = @_;
    $constraints = \%constraints;
  } else {
    my $n = @_;
    require Carp;
    Carp::croak("$n arguments to Synthetic_Constraint_Set::new");
  }

  bless $constraints => $class;
}
```

It has the usual accessors:

```
sub constraints { values %{$_[0]} }
sub constraint { $_[0]->{$_[1]} }
sub labels { keys %{$_[0]} }
sub has_label { exists $_[0]->{$_[1]} }
```

Also a method for appending another constraint to the set:

```
sub add_labeled_constraint {
  my ($self, $label, $constraint) = @_;
  $self->{$label} = $constraint;
}
```

It has another map-like function that applies a callback to each constraint and returns a new set with the results. This method leaves the labels unchanged:

```
sub apply {
  my ($self, $func) = @_;
  my %result;
  for my $k ($self->labels) {
    $result{$k} = $func->($self->constraint($k));
  }
  $self->new(\%result);
}
```

This function seems to be a good target for currying, but I decided to postpone that change.

Like Intrinsic_Constraint_Set, Synthetic_Constraint_Set also has a method for qualifying all of its constraints:

```
sub qualify {
  my ($self, $prefix) = @_;
  $self->apply(sub { $_[0]->qualify($prefix) });
}
```

Unlike Intrinsic_Constraint_Set, whose constraints are not involved in arithmetic, Synthetic_Constraint_Set has a method for scaling all of its constraints:

```
sub scale {
  my ($self, $coeff) = @_;
```

```
  $self->apply(sub { $_[0]->scale_equation($coeff) });
}
```

Yet another `map`-like function takes *two* synthetic constraint sets and applies the callback function to pairs of corresponding constraints, building a new set of the results:

```
sub apply2 {
  my ($self, $arg, $func) = @_;
  my %result;
  for my $k ($self->labels) {
    next unless $arg->has_label($k);
    $result{$k} = $func->($self->constraint($k),
                          $arg->constraint($k));
  }
  $self->new(\%result);
}
```

This function will be used for addition of features. `apply2()` will be called to add the matching constraints from the sets of its two operands.

This brings up a fine point: What if the labels in the two sets don't match? For example, what if we have:

```
line L;
hline H;
L + H = ... ;
```

Here `H` will have synthetic constraints:

$$
\begin{aligned}
\text{center.x} &\Rightarrow \text{H.center.x} = 0 \\
\text{center.y} &\Rightarrow \text{H.center.y} = 0 \\
\text{end.x} &\Rightarrow \text{H.end.x} = 0 \\
\text{end.y} &\Rightarrow \text{H.end.y} = 0 \\
\text{length} &\Rightarrow \text{H.length} = 0 \\
\text{start.x} &\Rightarrow \text{H.start.x} = 0 \\
\text{start.y} &\Rightarrow \text{H.start.y} = 0 \\
\text{y} &\Rightarrow \text{H.y} = 0
\end{aligned}
$$

but L will be missing a few of these, and will have only:

$$\begin{aligned}
\text{center.x} &\Rightarrow \text{L.center.x} = 0 \\
\text{center.y} &\Rightarrow \text{L.center.y} = 0 \\
\text{end.x} &\Rightarrow \text{L.end.x} = 0 \\
\text{end.y} &\Rightarrow \text{L.end.y} = 0 \\
\text{start.x} &\Rightarrow \text{L.start.x} = 0 \\
\text{start.y} &\Rightarrow \text{L.start.y} = 0
\end{aligned}$$

What happens to H's *length* and *y* constraints? The right thing to do here is to discard them. The result set is:

$$\begin{aligned}
\text{center.x} &\Rightarrow \text{L.center.x} + \text{H.center.x} = 0 \\
\text{center.y} &\Rightarrow \text{L.center.y} + \text{H.center.y} = 0 \\
\text{end.x} &\Rightarrow \text{L.end.x} + \text{H.end.x} = 0 \\
\text{end.y} &\Rightarrow \text{L.end.y} + \text{H.end.y} = 0 \\
\text{start.x} &\Rightarrow \text{L.start.x} + \text{H.start.x} = 0 \\
\text{start.y} &\Rightarrow \text{L.start.y} + \text{H.start.y} = 0
\end{aligned}$$

Thus, the result of adding an `hline` and a `line` is just a `line`. Similarly if we try to equate an `hline` and a `vline`, the resulting expression contains synthetic constraints only for the parts they have in common. The horizontalness and verticalosity are handled by the intrinsic constraint sets instead. There should probably be a check to make sure that the two operands in an addition are of compatible types, but that's something for the next version. In the meantime, the code in `apply2()` silently discards constraints with labels present in one but not both argument sets.

The final method in `Synthetic_Constraint_Set` is a special one for handling arithmetic involving features and tuples. Adding a feature to a tuple is interesting. The trick here is that the tuple's *x* component must be added to all the synthetic constraints that represent *x* coordinates, and similarly for the *y* component. (And similarly also the *z* component in a three-dimensional drawing.) Suppose we had:

```
hline H;
H + (3, 4) = ...
```

The synthetic constraint set for H is:

$$\begin{aligned}
\text{center.x} &\Rightarrow \text{H.center.x} = 0\\
\text{center.y} &\Rightarrow \text{H.center.y} = 0\\
\text{end.x} &\Rightarrow \text{H.end.x} = 0\\
\text{end.y} &\Rightarrow \text{H.end.y} = 0\\
\text{length} &\Rightarrow \text{H.length} = 0\\
\text{start.x} &\Rightarrow \text{H.start.x} = 0\\
\text{start.y} &\Rightarrow \text{H.start.y} = 0\\
\text{y} &\Rightarrow \text{H.y} = 0
\end{aligned}$$

The synthetic constraint set of the sum is:

$$\begin{aligned}
\text{center.x} &\Rightarrow \text{H.center.x} + 3 = 0\\
\text{center.y} &\Rightarrow \text{H.center.y} + 4 = 0\\
\text{end.x} &\Rightarrow \text{H.end.x} + 3 = 0\\
\text{end.y} &\Rightarrow \text{H.end.y} + 4 = 0\\
\text{length} &\Rightarrow \text{H.length} = 0\\
\text{start.x} &\Rightarrow \text{H.start.x} + 3 = 0\\
\text{start.y} &\Rightarrow \text{H.start.y} + 4 = 0\\
\text{y} &\Rightarrow \text{H.y} + 4 = 0
\end{aligned}$$

How do we decide whether a synthetic constraint represents an *x* or a *y* coordinate? `linogram` assumes that any feature named *x* is an *x* coordinate, and that any feature named *y* is a *y* coordinate. The tuple's *x* component should be combined with any synthetic constraint whose label ends in `.x` or is plain `x`. This selective combination is handled by yet another `map`-like function, `apply_hash()`:

```
sub apply_hash {
  my ($self, $hash, $func) = @_;
  my %result;
  for my $c (keys %$hash) {
    my $dotc = ".$c";
```

```
    for my $k ($self->labels) {
      next unless $k eq $c || substr($k, -length($dotc)) eq $dotc;
      $result{$k} = $func->($self->constraint($k), $hash->{$c});
    }
  }
  $self->new(\%result);
}
```

Each component of the argument hash has a label, `$c`. The function scans the labels of the constraints in the set, which are indexed by `$k`. If the constraint label matches the tuple component label, the callback is invoked and its return value is added to the result set. The labels match if they are equal (as with `x` and `x`) or if the constraint label ends with a dot followed by the tuple label (as with `start.x` and `x`.) The dot is important, because we don't want a label like `max` or `box` to match `x`.

FEATURE-VALUE METHODS

Now we can see the methods for operating on feature-value objects. The objects themselves contain nothing more than an intrinsic and a synthetic constraint set:

```
package Value::Feature;
@Value::Feature::ISA = 'Value';

sub kindof { "FEATURE" }

sub new {
    my ($base, $intrinsic, $synthetic) = @_;
    my $class = ref $base || $base;
    my $self = {WHAT => $base->kindof,
                SYNTHETIC => $synthetic,
                INTRINSIC => $intrinsic,
               };
    bless $self => $class;
}
```

There's another very important constructor in the `Value::Feature` class. Instead of building a value from given sets of constraints, it takes a `Type` object, which represents a type such as `box` or `line`, figures out what its constraint sets

should be, and builds a new value with those constraint sets:

```
sub new_from_var {
  my ($base, $name, $type) = @_;
  my $class = ref $base || $base;
  $base->new($type->qualified_intrinsic_constraints($name),
             $type->qualified_synthetic_constraints($name),
            );
}
```

`Value::Feature` naturally has two accessors, one for the intrinsic and one for the synthetic constraint sets:

```
sub intrinsic { $_[0]->{INTRINSIC} }
sub synthetic { $_[0]->{SYNTHETIC} }
```

For its scaling operation, it passes the buck to the synthetic constraint set. The intrinsic constraints don't participate in arithmetic, so they remain the same:

```
sub scale {
  my ($self, $coeff) = @_;
  return
    $self->new($self->intrinsic,
               $self->synthetic->scale($coeff),
              );
}
```

The four other methods are the ones from the dispatch table. To add two features, we unite their intrinsic constraint sets, and add corresponding constraints from their synthetic constraint sets:

```
sub add_features {
  my ($o1, $o2) = @_;
  my $intrinsic = $o1->intrinsic->union($o2->intrinsic);
  my $synthetic = $o1->synthetic->apply2($o2->synthetic,
                                         sub { $_[0]->add_equations($_[1]) },
                                        );
  $o1->new($intrinsic, $synthetic);
}
```

Adding constraints is performed by `add_equations()`, which is inherited from `Equation`.

As with tuples, multiplying a feature by a constant is trivial, since it's the same as `scale()`:

```
sub mul_feature_con {
  my ($o, $c) = @_;
  $o->scale($c->value);
}
```

Adding a feature to a constant isn't hard, once we decide what it should mean. The current version of `linogram` adds the constant to each synthetic constraint. This happens to be correct for features that represent numbers, since, as we'll see, they have a single synthetic constraint with label `""`. But it doesn't make much sense for most other features. Probably this function should contain a type check to make sure that its feature argument represents a scalar, but that isn't present in this version:

```
sub add_feature_con {
  my ($o, $c) = @_;
  my $v = $c->value;
  my $synthetic = $o->synthetic->apply(sub { $_[0]->add_constant($v) });
  $o->new($o->intrinsic, $synthetic);
}
```

Once again, the intrinsic constraints don't participate in arithmetic, so they're unchanged.

The final method is for adding a feature to a tuple. We use the `apply_hash()` function that was specifically intended for adding features to tuples. Its callback argument is complicated by the fact that tuple components might not be simple numbers. If the component *is* a simple number (a `Value::Constant` object), then we use the `add_constant()` method as in the previous function:

```
sub add_feature_tuple {
  my ($o, $t) = @_;
  my $synthetic =
    $o->synthetic->apply_hash($t->to_hash,
                              sub {
                                my ($constr, $comp) = @_;
                                my $kind = $comp->kindof;
                                if ($kind eq "CONSTANT") {
                                  $constr->add_constant($comp->value);
```

If the tuple component is a feature, we assume that it's a scalar, which has only a single constraint, with label "":

```
            } elsif ($kind eq "FEATURE") {
              $constr->add_equations($comp->synthetic->constraint(""));
```

If the tuple component is another tuple, we croak, because that's not allowed. This freak tuple should have been forbidden earlier, but there's little harm in adding more than one check for the same thing:

```
            } elsif ($kind eq "TUPLE") {
              die "Tuple with subtuple component";
            } else {
              die "Unknown tuple component type '$kind'";
            }
          },
        );
  $o->new($o->intrinsic, $synthetic);
}

1;
```

Once again, the intrinsic constraints are unchanged because they don't participate in arithmetic.

9.4.3 Feature Types

Where do the constraints come from? If the equation solver is the heart of `linogram`, then its liver is the parser, which parses the input specification, including the constraint equations. The result of parsing is a hierarchy of feature types such as `box` and `line`. These are Perl objects from the class `Type`. Each type of feature is represented by a `Type` object, which records the sub-features, the constraints, and the other properties of that kind of feature object.

To construct a new type, we call `Type::new`:

CODE LIBRARY
Type.pm

```
package Type;

sub new {
  my ($old, $name, $parent) = @_;
  my $class = ref $old || $old;
```

```
  my $self = {N => $name, P => $parent, C => [],
              O => {}, D => [],
             };
  bless $self => $class;
}
```

`$name` is the name of the new type. `$parent` is optional, and, if present, is a `Type` object representing the type from which the new type is extended. For example, the parent of `vline` is `line`; the parent of `line` is undefined. The parent type is stored under member `P` for "parent"; the name is stored under `N`.

The other members of the `Type` object are:

- **C**: The constraints defined for the object.
- **O**: The sub-features of the type. This is a hash. The keys are the names of the sub-features, and the values are the `Type` objects representing the types of the sub-features.
- **D**: A list of "drawables," either Perl code references or sub-feature names.

SCALAR TYPES

`Type` has a subclass, `Type::Scalar`, which represents trivial types, such as `number`, that have no constraints and no sub-features. `linogram` has no scalar types other than `number`, but a future version might introduce some.

Sometimes these types behave a little differently from compound types such as points and boxes, so it's convenient to put their methods into a separate class. One principal difference is the trivial `is_scalar` method, which returns true for a scalar type object and false for a nonscalar object. `Type::Scalar` also overrides the methods that are used to install constraints and sub-features into type objects:

```
package Type::Scalar;
@Type::Scalar::ISA = 'Type';
sub is_scalar { 1 }

sub add_constraint {
  die "Added constraint to scalar type";
}

sub add_subfeature {
  die "Added subfeature to scalar type";
}
```

We should never be extending scalar types like `number` with sub-features or constraints, so overriding these methods provides us with early warning if something is going terribly wrong.

Type METHODS

The simplest `Type` method says that types are not scalars, except when the method is overridden by the `Type::Scalar` version of the method:

```
package Type;

sub is_scalar { 0 }
```

Many of the accessor methods on `Type` objects are straightforward; for example:

```
sub parent { $_[0]{P} }
```

But in some cases, an accessor needs to be referred up the derivation chain to the parent type. For example, a `vline` has a sub-feature named `start`, but it's not stored in the type object for `vline`; it's inherited from `line`. So if we want find out about the type of the `start` sub-feature of `vline`, we must search in `line`. Moreover, a `vline` has a sub-feature named `start.x`, which is the `x` sub-feature of the `start` sub-feature. The `subfeature` method handles all of these situations:

```
sub subfeature {
  my ($self, $name, $nocroak) = @_;
  return $self unless defined $name;
  my ($basename, $suffix) = split /\./, $name, 2;
  if (exists $self->{O}{$basename}) {
    return $self->{O}{$basename}->subfeature($suffix);
  } elsif (my $parent = $self->parent) {
    $parent->subfeature($name);
  } elsif ($nocroak) {
    return;
  } else {
    Carp::croak("Asked for nonexistent subfeature '$name' of type '$self->{N}'");
  }
}
```

`$type->subfeature($name)` returns the type of the sub-feature of `$type` with name `$name`. If `$name` is a compound name, which contains a dot, it is split into a `$basename` (the component before the first dot) and a `$suffix` (everything after the first dot); the `$basename` is looked up directly, and the `$suffix` is referred to a recursive call to `subfeature`. If the specified type does not contain a sub-feature with the appropriate basename, then its parent object is consulted instead. If there is no parent type, then the requested sub-feature doesn't exist, and the function croaks. This is because the error is most likely to be caused by an incorrect specification in the drawing, asking for a nonexistent sub-feature. To disable the croaking behavior, the user of the function can pass the optional third parameter, which makes the function return false instead. An example of this is the simple `has_subfeature` method, which returns true if the target has a sub-feature of the specified name, and false if not:

```
sub has_subfeature
 {
  my ($self, $name) = @_;
  defined($self->subfeature($name, "don't croak"));
}
```

The recursion in `subfeature()` is in two different directions. Sometimes we recurse from a feature to one of its sub-features, and sometimes we recurse up the type inheritance tree to the parent type. Suppose `$box`, `$hline`, `$line`, `$point`, and `$number` are the `Type` objects that represent the indicated types. Let's see how the call `$box->subfeature("top.start.x")` is resolved:

```
$box->subfeature("top.start.x")
```

`$box` has a sub-feature called `"top"`, which is an `hline`, so the call is referred to the sub-feature type:

```
$hline->subfeature("start.x");
```

`$hline` has no sub-feature called `"start"`, so the call is referred to the parent type:

```
$line->subfeature("start.x");
```

`$line` does have a sub-feature called `"start"`, which is a `point`, so the call is referred to the sub-feature type:

```
$point->subfeature("x");
```

`$point` does have a sub-feature called `"x"`, which is a `number`, so the call is referred to the sub-feature type:

```
$number->subfeature(undef);
```

The call reaches the base case and returns `$number`, which is indeed the type of the `top.start.x` feature of `box`.

A similar process occurs in the `Type::constraints` method, which delivers an array of all the constraints of a type, including those implied by the sub-features and the parent type:

```
sub constraints {
  my $self = shift;
```

First the function obtains the constraints inherent in the type itself:

```
  my @constraints = @{$self->{C}};
```

Then it obtains the constraints that are inherited from the parent type, and, via recursion, from all the ancestor types:

```
  my $p = $self->parent;
  if (defined $p) { push @constraints, @{$p->constraints} }
```

Then it obtains the constraints that it gets from its sub-features, including any constraints that *they* inherit from their ancestor types:

```
  while (my ($name, $type) = each %{$self->{O}}) {
    my @subconstraints = @{$type->constraints};
    push @constraints, map $_->qualify($name), @subconstraints;
  }
  \@constraints;
}
```

`constraint_set()` is the same, except that it returns a `Constraint_Set` object instead of a raw array reference:

```
sub constraint_set {
  my $self = shift;
  Constraint_Set->new(@{$self->constraints});
}
```

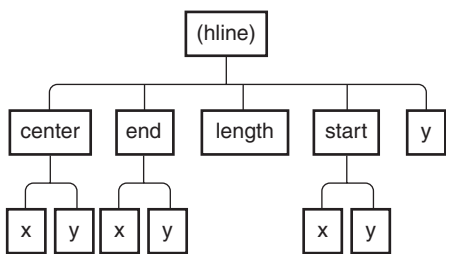


FIGURE 9.5 The subfeatures of a feature form a tree structure.

These constraints are precisely the intrinsic constraints that are used by `Value::Feature` objects, so we have:

```
sub intrinsic_constraints {
  my $constraints = $_[0]->constraints;
  Intrinsic_Constraint_Set->new(@$constraints);
}
```

The `new_from_type` method of `Value::Feature` actually wants the *qualified* intrinsic constraints:

```
sub qualified_intrinsic_constraints {
  $_[0]->intrinsic_constraints->qualify($_[1]);
}
```

As usual, the synthetic constraints for a type are rather more interesting. In the absence of any other information, an expression like `P` is interpreted as the constraint $P = 0$. Later, the $P = 0$ might be combined with a $Q = 0$ to produce $P + Q = 0$ or $P - Q = 0$, and we'll see that we can treat $P = Q$ as if it were $P - Q = 0$. So figuring out the synthetic constraints for a type like `point` involves locating all the scalar type subfeatures of `point`, and then setting each one to 0.

The recursive auxiliary method `all_leaf_subfeatures()` recovers the names of all the scalar sub-features of the given type (see Figure 9.5). Its name refers to the fact that the sub-feature relation makes each type into a tree. The scalar sub-features are the leaves of the tree.

```
sub all_leaf_subfeatures {
  my $self = shift;
  my @all;
  my %base = $self->subfeatures;
  while (my ($name, $type) = each %base) {
```

```
    push @all, map {$_ eq "" ? $name : "$name.$_"}
      $type->all_leaf_subfeatures;
  }
  @all;
}
```

The function starts by getting all the direct sub-features. These include those defined directly by the target type and also those defined by its ancestor types. Some of these sub-features might be compound features and have sub-features of their own, and some might be leaves. The function loops over them to do the recursion on each one. It qualifies the names appropriately and adds the information to the result array. The special case in the `map` is to avoid extra periods from appearing at the end of the key names in some cases.

To build the synthetic constraint set for a particular type, we locate all the scalar sub-features and make a constraint for each one. If *name* is the name of a scalar subfeature, we introduce the synthetic constraint that has *name* = 0 with label *name*:

```
sub synthetic_constraints {
  my @subfeatures = $_[0]->all_leaf_subfeatures;
  Synthetic_Constraint_Set->new(map {$_ => Constraint->new($_ => 1)}
                                @subfeatures
                               );
}

sub qualified_synthetic_constraints {
  $_[0]->synthetic_constraints->qualify($_[1]);
}
```

All but one of the remaining `Type` methods are accessors, most of them fairly simple:

```
sub add_drawable {
  my ($self, $drawable) = @_;
  push @{$self->{D}}, $drawable;
}
```

`subfeatures()` returns all the direct sub-features of a type, but not the sub-sub-features. For `box`, it will return `top` and `nw`, but not `top.center` or `nw.y`:

```
sub subfeatures {
  my $self = shift;
```

```
  my %all;
  while ($self) {
    %all = (%{$self->{O}}, %all);
    $self = $self->parent;
  }
  %all;
}
```

The function that retrieves the list of drawable sub-features and drawing functions for a type recurses up the type inheritance tree using `subfeatures()`. It doesn't need to recurse into the sub-features, because the drawing method will do that itself. We'll see the drawing method later; here's the `drawables()` method, which returns a list of the drawables:

```
sub drawables {
  my ($self) = @_;
  return @{$self->{D}} if $self->{D} && @{$self->{D}};
  if (my $p = $self->parent) {
    my @drawables = $p->drawables;
    return @drawables if @drawables;
  }
  my %subfeature = $self->subfeatures;
  my @drawables = grep ! $subfeature{$_}->is_scalar, keys %subfeature;
  @drawables;
}
```

If the type definition contains an explicit drawable list, the method returns it. If not, it uses the drawable list of its parent object, if it has one. If the type has no parent type, the method generates and returns the default, which is a list of all the sub-features that aren't scalars. There's no point returning scalars, since they're not drawable, so they're filtered out.

New sub-features are installed into a type with `add_subfeature()`. Its arguments are a name and a sub-feature type:

```
sub add_subfeature {
  my ($self, $name, $type) = @_;
  $self->{O}{$name} = $type;
}
```

Similarly, new constraints are installed into a type with `add_constraints()`. Its arguments are `Value::Feature` objects. The method extracts the constraints

from the values and inserts them into the Type object:

```
sub add_constraints {
  my ($self, @values) = @_;
  for my $value (@values) {
    next unless $value->kindof eq 'FEATURE';
    push @{$self->{C}},
      $value->intrinsic->constraints,
      $value->synthetic->constraints;
  }
}
```

I've left the most important Type method for the end. It's the most important method in the entire program, because it's the method that actually draws the picture. Its primary argument is a Type object. When invoked for the root type, it draws the entire picture. It's a little longer than the other methods, so we'll see it a bit at a time:

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

The primary argument, $self, is the type to draw. The other argument is an *environment*, which belongs to an Environment class we didn't see. The environment is nothing more than a hash with the names and values of the solutions of the constraints.[1] The initial call to draw(), which draws the root feature, omits the environment, because the equations haven't been solved yet; the missing $env parameter triggers draw() to solve the equations:

```
unless ($env) {
  my $equations = $self->constraint_set;
  my %solutions = $equations->values;
  $env = Environment->new(%solutions);
}
```

The rest of the function does the actual drawing. It scans the list of drawables for the feature being drawn. If the drawable is a reference to an actual drawing

1 In an earlier version of this program, the environment parameter was more interesting. Features could contain local variables that didn't participate in the constraint solving (and which therefore didn't have to be linear) and parameters passed in from the containing feature. In the interests of clear exposition, I trimmed these features out.

function, the function is invoked, and is passed the environment:

```
    for my $name ($self->drawables) {
      if (ref $name) {            # actually a coderef, not a name
        $name->($env);
```

Otherwise, the drawable is the name of a sub-feature on which the `draw()` method is recursively called. The function recovers the type of the sub-feature. It also uses the `Environment::subset()` method to construct a new environment that contains only the variables relevant to that sub-feature:

```
      } else {
        my $type = $self->subfeature($name);
        my $subenv = $env->subset($name);
        $type->draw($subenv);
      }
    }
}

1;
```

For completeness, here is `Environment::subset()`:

CODE LIBRARY
Environment.pm

```
package Environment;
sub subset {
  my ($self, $name) = @_;
  my %result;
  for my $k (keys %$self) {
    my $kk = $k;
    if ($kk =~ s/^\Q$name.//) {
      $result{$kk} = $self->{$k};
    }
  }
  $self->new(%result);
}
```

9.4.4 The Parser

We're now ready to see the core of `linogram`, which is the parser that parses drawing specifications. First, the lexer, which is straightforward:

CODE LIBRARY
linogram.pl

```
use Parser ':all';
use Lexer ':all';
```

```
my $input = sub { read INPUT, my($buf), 8192 or return; $buf };

my @keywords = map [uc($_), qr/\b$_\b/],
  qw(constraints define extends draw);

my $tokens = iterator_to_stream(
  make_lexer($input,
             @keywords,
             ['ENDMARKER',  qr/__END__.*/s,
              sub {
                my $s = shift;
                $s =~ s/^__END__\s*//;
                ['ENDMARKER', $s]
              } ],
             ['IDENTIFIER', qr/[a-zA-Z_]\w*/],
             ['NUMBER', qr/(?: \d+ (?: \.\d*)?
                           | \.\d+)
                           (?: [eE]  \d+)? /x ],
             ['FUNCTION',   qr/&/],
             ['DOT',        qr/\./],
             ['COMMA',      qr/,/],
             ['OP',         qr|[-+*/]|],
             ['EQUALS',     qr/=/],
             ['LPAREN',     qr/[(]/],
             ['RPAREN',     qr/[)]/],
             ['LBRACE',     qr/[{]/],
             ['RBRACE',     qr/[}]\n*/],
             ['TERMINATOR', qr/;\n*/],
             ['WHITESPACE', qr/\s+/, sub { "" }],
  ));
```

Only a few of these need comment. `IDENTIFIER` is a simple variable name, such as `box` or `start`. Compound names like `start.x` will be assembled later, by the parser.

`ENDMARKER` consists of the sequence `__END__` and *all* the following text up to the end of the file. The lexer preprocesses this to delete the `__END__` itself, leaving only the following text.

Several similar definitions for the `CONSTRAINTS`, `DEFINE`, `EXTENDS`, and `DRAW` tokens are generated programmatically, and are inserted at the beginning of the lexer definition via the `@keywords` array.

Whitespace, as in earlier parsers, is discarded.

PARSER EXTENSIONS

The parser module used in `linogram` is based on our functional parser library of Chapter 8, with some additions. Suppose that `$A` and `$B` are parsers. Recall the following features supplied by the parser of Chapter 8:

- `empty()` is a parser that consumes no tokens and always succeeds.
- `$A - $B` ("*A*, then *B*") is a parser that matches whatever `$A` matches, consuming the appropriate tokens, and then applies `$B` to the remaining input, possibly consuming more tokens. It succeeds only if both `$A` and `$B` succeed in sequence.
- `$A | $B` ("*A* or *B*") is a parser that tries to apply `$A` to its input, and, if that doesn't work, tries `$B` instead. It succeeds if either of `$A` or `$B` succeeds.
- `star($A)` matches zero or more occurrences of whatever `$A` matches; it is equivalent to `empty() | $A - star($A)`.
- `_(...)` is a synonym for `lookfor([...])`, which builds a parser that looks for a single token of the indicated kind. If the next token is of the correct kind, it is consumed and the parser succeeds; otherwise the parser fails.
- `$A >> $coderef` is a synonym for `T($A, $coderef)`, a parser that applies `$A` to its input stream, and then uses `$coderef` to transform the result returned by `$A` into a different form. It assumes that `$A` is a concatenation of other parsers.

To these operations, we'll add a few extras:

- `option($item)` indicates that the syntax matched by the `$item` parser is optional. It builds a parser equivalent to:

```
$item | empty()
```

- `labeledblock($label, $contents)` is for matching labeled blocks like:

```
draw {
  ...
}
```

and:

```
define line {
  ...
}
```

It's equivalent to:

```
$label - _('LBRACE') - star($contents) -_('RBRACE')
  >> sub { [ $_[0], @{$_[2]} ] }
```

- `commalist($item, $separator)` is for matching comma-separated lists of items. The `$separator` defaults to `_('COMMA')`. It is otherwise equivalent to:

```
$item - star($separator - $item >> sub { $_[1] })
      - option($separator)
  >> sub { [ $_[0], @{$_[1]} ] }
```

  The first `sub` throws away the values associated with the separators, leaving only the values of the items. The second `sub` accumulates all the item values into a single array, which is the value returned by the `commalist` parser.

- `$parser > $coderef` is like `$parser >> $coderef`, except that it doesn't assume that `$parser` is a concatenation. Instead of assuming that the value returned by `$parser` is an array reference, and passing the elements of the array to the coderef, it passes the value returned by `$parser` directly to `$coderef` as a single argument.

- `$parser / $condition` is like `$parser`, with a side condition on the result. It runs `$parser` as usual, and then passes the resulting value to the coderef in `$condition`. If the condition returns true, the parser succeeds, and the final result is the same value originally returned by `$parser`. If the coderef returns false, the parser fails.

%TYPES

The main data structure in `linogram` is `%TYPES`, which is a hash that maps known type names to the `Type` objects that represent them. When the program starts, `%TYPES` is initialized with two predefined types:

```
my $ROOT_TYPE = Type->new('ROOT');
my %TYPES = ('number' => Type::Scalar->new('number'),
             'ROOT'   => $ROOT_TYPE,
            );
```

Initially, `linogram` knows about the type `number`, which is a trivial type with no sub-features and no constraints, and the type `ROOT`, which represents the entire diagram.

PROGRAMS

A program in linogram is a series of subtype definitions and feature and constraint declarations which together define the root type. As subtype definitions are encountered, the corresponding Type objects are manufactured and installed in %TYPES. As feature and constraint declarations are encountered, they are installed into the root type object.

The top-level parser looks like this:

```
$program = star($Definition
              | $Declaration
                > sub { add_declarations($ROOT_TYPE, $_[0]) }
              )
         - option($Perl_code) - $End_of_Input
  >> sub {
    $ROOT_TYPE->draw();
  };
```

The $definition parser will take care of manufacturing new type objects and installing them into %TYPES. When a declaration is parsed, add_declarations() will install it into the root type object $ROOT_TYPE. The program may be followed with an optional section of plain Perl code, which is a convenient place to stick auxiliary functions like draw_line. When the parser finishes parsing the entire specification, it invokes the draw method on the root type object, drawing the entire diagram.

$perl_code is an optional section at the end of the drawing specification. It's an arbitrary segment of Perl code, separated from the rest of the specification with the endmarker __END__:

```
$perl_code = _("ENDMARKER") > sub { eval $_[0];
                                    die if $@;
                                  };
```

The lexer has already trimmed off the endmarker itself. The Perl code is then passed to eval, which compiles the Perl code and installs it into the program.

DEFINITIONS

$definition is a parser for a block of the form:

```
define point { ... }
```

or:

```
define hline extends line { ... }
```

We use the `labeledblock` function to construct this parser:

```
$definition = labeledblock($Defheader, $Declaration)
  >> sub { ... } ;
```

`$declaration` is the parser for a declaration, which will see shortly. `$defheader` is the part of the definition block before the curly braces:

```
$defheader = _("DEFINE") - _("IDENTIFIER") - $Extends
  >> sub { ["DEFINITION", @_[1,2] ]};

$extends = option(_("EXTENDS") - _("IDENTIFIER") >> sub { $_[1] }) ;
```

The value from the `$definition` parser is passed to a postprocessing function that is responsible for constructing a new `Type` object and installing it into `%TYPES`; the code is all straightforward. For a definition that begins `define hline extends line`, `$name` is `hline` and `$extends` is `$line`:

```
$definition = labeledblock($defheader, $Declaration)
  >> sub {
     my ($defheader, @declarations) = @_;
     my ($name, $extends) = @$defheader[1,2];
     my $parent_type = (defined $extends) ? $TYPES{$extends} : undef;
     my $new_type;

     if (exists $TYPES{$name}) {
       lino_error("Type '$name' redefined");
     }
     if (defined $extends && ! defined $parent_type) {
       lino_error("Type '$name' extended from unknown type '$extends'");
     }

     $new_type = Type->new($name, $parent_type);

     add_declarations($new_type, @declarations);

     $TYPES{$name} = $new_type;
  };
```

DECLARATIONS

A declaration takes one of three forms. One is the declaration of one or more sub-features:

```
hline top, bottom;
```

Two others are `constraints` and `draw` sections:

```
constraints { ... }
draw { ... }
```

Here's the declaration parser:

```
$declaration = $Type - commalist($Declarator) - _("TERMINATOR")
                  >> sub { ... }
             | $Constraint_section
             | $Draw_section
             ;
```

A `$type` is the same as an identifier, with the side condition that it must be mentioned in the `%TYPES` hash:

```
$type = lookfor("IDENTIFIER",
                sub {
                 exists($TYPES{$_[0][1]}) || lino_error("Unrecognized type '$_[0][1]'");
                 $_[0][1];
                }
               );
```

A declaration might declare more than one variable, as with:

```
hline top, bottom;
```

Each of the sub-parts of the declaration is called a *declarator*; the preceding declaration has two declarators. In its simplest form, a declarator is nothing more than a variable name:

```
$declarator = _("IDENTIFIER")
            - option(_("LPAREN")  - commalist($Param_Spec) - _("RPAREN")
                     >> sub { $_[1] }
                    )
```

```
>> sub {
  { WHAT => 'DECLARATOR',
    NAME => $_[0],
    PARAM_SPECS => $_[1],
  };
};
```

The optional section in the middle is for a parenthesis-delimited list of "parameter specifications." A declarator might look like this:

```
... F(ht=3, wd=boxwid), ...
```

which is equivalent to:

```
... F, ...
F.ht = 3;
F.wd = boxwid;
```

The `sub { $_[1] }` discards the parentheses; the parameter specifications are packaged into the resulting value under the key `PARAM_SPECS`. The format of a parameter specification is simple:

```
$param_spec = _("IDENTIFIER") - _("EQUALS") - $Expression
  >> sub {
    { WHAT => "PARAM_SPEC",
      NAME => $_[0],
      VALUE => $_[2],
    }
  }
  ;
```

Thus the value manufactured for the declarator `F(ht=3, wd=boxwid)` looks like this:

```
{ WHAT => 'DECLARATOR',
  NAME => 'F',
  PARAM_SPECS =>
    [ { WHAT => 'PARAM_SPEC',
        NAME => 'ht',
        VALUE => (expression representing constant 3),
      },
```

```
      { WHAT => 'PARAM_SPEC',
        NAME => 'wd',
        VALUE => (expression representing variable 'boxwid'),
      },
    ]
}
```

We haven't yet seen the representation for expressions.

The `$declaration` parser gets a type name and a list of declarators and manufactures a declaration value; later on, the `add_declarations()` function will install this declaration into the appropriate `Type` object. The declaration value is manufactured as follows:

```
$declaration = $Type - commalist($Declarator) - _("TERMINATOR")
                 >> sub { my ($type, $decl_list) = @_;
                          unless (exists $TYPES{$type}) {
                            lino_error("Unknown type name '$type' in declaration '@_'\n");
                          }
                          for (@$decl_list) {
                            $_->{TYPE} = $type;
                            check_declarator($TYPES{$type}, $_);
                          }
                          {WHAT => 'DECLARATION',
                           DECLARATORS => $decl_list };
                        }

 ....

| $Constraint_section
| $Draw_section
;
```

The construction function checks to make sure the type used in the declaration actually exists. It then installs the type into each declarator value, transforming:

```
{ WHAT => 'DECLARATOR',
  NAME => 'F',
  PARAM_SPECS => [ ... ],
}
```

into:

```
{ WHAT => 'DECLARATOR',
  NAME => 'F',
  PARAM_SPECS => [ ... ],
  TYPE => $type,
}
```

Each declarator is also checked to make sure the names in its parameter specifications are actually the names of sub-features of its type. `box F(ht=3)` passes the check, but `box F(age=34)` fails, because boxes don't have ages. This check is performed by `check_declarator()`:

```
sub check_declarator {
  my ($type, $declarator) = @_;
  for my $pspec (@{$declarator->{PARAM_SPECS}}) {
    my $name = $pspec->{NAME};
    unless ($type->has_subfeature($name)) {
      lino_error("Declaration of '$declarator->{NAME}' "
               . "specifies unknown subfeature '$name' "
               . "for type '$type->{N}'\n");
    }
  }
}
```

Declarator values are combined into declaration values; a typical declaration value, for the declaration `box C, F(ht=3, wd=boxwid);`, looks like this:

```
{ WHAT => 'DECLARATION',
  DECLARATORS =>
    [ { WHAT => 'DECLARATOR',
        NAME => 'C',
        PARAM_SPECS => [],
        TYPE => 'box',
      },
      { WHAT => 'DECLARATOR',
        NAME => 'F',
        PARAM_SPECS =>
          [ { WHAT => 'PARAM_SPEC',
              NAME => 'ht',
```

```
                VALUE => (expression representing constant 3),
              },
              { WHAT => 'PARAM_SPEC',
                NAME => 'wd',
                VALUE => (expression representing variable 'boxwid')
              },
            ]
          TYPE => 'box',
        },
      ]
}
```

The other two kinds of declarations we've seen before have been constraint and draw sections, which have their own productions in the grammar:

```
$declaration = ...
             | $Constraint_section
             | $Draw_section
             ;
```

The overall structure of a constraint section is a block, labeled with the word `constraints`:

```
$constraint_section = labeledblock(_("CONSTRAINTS"), $Constraint)
  >> sub { shift;
           { WHAT => 'CONSTRAINTS', CONSTRAINTS => [@_] }
         };
```

A constraint is simply an equation, which is a pair of expressions with an equal sign between them:

```
$constraint = $Expression - _("EQUALS") - $Expression - _("TERMINATOR")
  >> sub { Expression->new('-', $_[0], $_[2]) } ;
```

The value of the constraint is not actually a `Constraint` object, but rather an `Expression` object. Since the constraint $A = B$ is semantically equivalent to $A - B = 0$, we compile it into an expression that represents `A - B` and leave it at that. The finished value for a constraint section, say for:

```
constraints { start.x = end.x;
              start.x = x;
              start.y + height = end.y;
}
```

is the hash:

```
{ WHAT => 'CONSTRAINTS',
  CONSTRAINTS =>
    [ (expression representing start.x - end.x),
      (expression representing start.x - x),
      (expression representing start.y + height - end.y),
    ]
}
```

The third sort of declaration is a `draw` section, which might look like this:

```
draw { &draw_line; }
```

or like this:

```
draw { top; bottom; left; right; }
```

Once again, it is a labeled block, very similar to the definition of the constraint section:

```
$draw_section = labeledblock(_("DRAW"), $Drawable)
  >> sub { shift; { WHAT => 'DRAWABLES', DRAWABLES => [@_] } };
```

Since there are two possible formats for a drawable, however, the definition of `$drawable` is a little more complicated than the definition of `$constraint`:

```
$drawable = $Name - _("TERMINATOR")
                >> sub { { WHAT => 'NAMED_DRAWABLE',
                           NAME => $_[1],
                         }
                       }
          | _("FUNCTION") - _("IDENTIFIER") - _("TERMINATOR")
                 >> sub { my $ref = \&{$_[1]};
                          { WHAT => 'FUNCTIONAL_DRAWABLE',
                            REF => $ref,
                            NAME => $_[1],
                          };
                        };
```

The first clause handles the case where the drawable is the name of a subfeature of the feature being defined, say `top;`. In this case we construct

the value:

```
{ WHAT => 'NAMED_DRAWABLE',
  NAME => 'top',
}
```

The other clause handles the case where the drawable is the name of a Perl function, say `&draw_line;`. In this case we construct the value:

```
{ WHAT => 'FUNCTIONAL_DRAWABLE',
  NAME => 'draw_line',
  REF => \&draw_line,
}
```

The `NAME` member here is just for debugging purposes; only the reference is actually used. Drawables of both types may be mixed in the same `draw` section. A draw section like `draw { top; &draw_line; }` turns into the value:

```
{ WHAT => 'DRAWABLES',
  DRAWABLES => [ { WHAT => 'NAMED_DRAWABLE',
                   NAME => 'TOP',
                 },
                 { WHAT => 'FUNCTIONAL_DRAWABLE',
                   NAME => 'draw_line',
                   REF => \&draw_line,
                 },
               ]
}
```

When a complete type definition has been parsed, several values will be available: the type name; the name of the parent type, if there is one; and the list of declarations. The parser function manufactures a new type object from class `Type`, and calls `add_declarations()` to install the declarations into the new object.

`add_declarations()` is rather complicated, because it has many different branches to handle the different kinds of declarations. Each branch individually is simple, which argues for a dispatch table structure:

```
my %add_decl = ('DECLARATION' => \&add_subfeature_declaration,
                'CONSTRAINTS' => \&add_constraint_declaration,
                'DRAWABLES' => \&add_draw_declaration,
```

```
                'DEFAULT' =>  sub {
                  lino_error("Unknown declaration kind '$[1]{WHAT}'");
                },
               );

sub add_declarations {
  my ($type, @declarations) = @_;

  for my $declaration (@declarations) {
    my $decl_kind = $declaration->{WHAT};
    my $func = $add_decl{$decl_kind} || $add_decl{DEFAULT};
    $func->($type, $declaration);
  }
}
```

Sub-feature declarations to `Type` objects are added by this function, which loops over the declarators, adding them one at a time:

```
sub add_subfeature_declaration {
  my ($type, $declaration) = @_;
  my $declarators = $declaration->{DECLARATORS};
  for my $decl (@$declarators) {
    my $name = $decl->{NAME};
    my $decl_type = $decl->{TYPE};
    my $decl_type_obj = $TYPES{$decl_type};
```

`$decl_type` is the name of the type of the sub-feature being declared; `$decl_type_obj` is the `Type` object that represents that type. The first thing the function does is record the name and the type of the new sub-feature:

```
    $type->add_subfeature($name, $decl_type_obj);
```

Unless the declarator came with parameter specifications, the function is done. If there were parameter specifications, the function turns them into constraints and adds them to the type's list of constraints:

```
    for my $pspec (@{$decl->{PARAM_SPECS}}) {
      my $pspec_name = $pspec->{NAME};
      my $constraints = convert_param_specs($type, $name, $pspec);
      $type->add_constraints($constraints);
    }
  }
}
```

convert_param_specs() turns the parameter specifications into constraints. We'll see this function later, after we've discussed the way in which expressions are turned into constraints.

```
sub add_constraint_declaration {
  my ($type, $declaration) = @_;
  my $constraint_expressions = $declaration->{CONSTRAINTS};
  my @constraints
    = map expression_to_constraints($type, $_),
          @$constraint_expressions;
  $type->add_constraints(@constraints);
}
```

This function is invoked to install a constraints block into a type object. The contents of the constraints block have been turned into Expression objects, but these objects are still essentially abstract syntax trees, and haven't yet been turned into constraints. The function expression_to_constraints() performs that conversion. add_constraints() then inserts the new constraints into the type object's constraint list. We'll see expression_to_constraints() later, along with the other functions that deal with expressions.

The third sort of declaration is a draw section, whose contents are drawables. These are installed into a type object by add_draw_declaration():

```
sub add_draw_declaration {
  my ($type, $declaration) = @_;
  my $drawables = $declaration->{DRAWABLES};

  for my $d (@$drawables) {
    my $drawable_type = $d->{WHAT};
    if ($drawable_type eq "NAMED_DRAWABLE") {
      unless ($type->has_subfeature($d->{NAME})) {
        lino_error("Unknown drawable feature '$d->{NAME}'
      }
      $type->add_drawable($d->{NAME});
    } elsif ($drawable_type eq "FUNCTIONAL_DRAWABLE") {
      $type->add_drawable($d->{REF});
    } else {
      lino_error("Unknown drawable type '$type'");
    }
  }
}
```

There are two branches here, for the two kinds of drawables. One is a functional drawable, typified by `&draw_line`; here we insert a reference to the Perl `draw_line` function into the drawables list. The other kind of drawable is a named drawable, which is the name of a sub-feature; here we insert the name into the drawables list. The only real difference in handling is that we make sure that the name of a named drawable is already known.

EXPRESSIONS

The expression parser is similar to the ones we saw in Chapter 8. Its output is essentially an abstract syntax tree, blessed into the `Expression` class. Expressions appear in constraints and on the right-hand sides of parameter specifications. The grammar is:

```
$expression = operator($Term,
                       [_('OP', '+'), sub { Expression->new('+', @_) } ],
                       [_('OP', '-'), sub { Expression->new('-', @_) } ],
                      );

$term = operator($Atom,
                        [_('OP', '*'), sub { Expression->new('*', @_) } ],
                        [_('OP', '/'), sub { Expression->new('/', @_) } ],
                );
```

which is nothing new. Expressions, as mentioned before, are nothing more than abstract syntax trees. `Expression::new()` is trivial:

```
package Expression;

sub new {
  my ($base, $op, @args) = @_;
  my $class = ref $base || $base;
  bless [ $op, @args ] => $class;
}
```

The $atom parser accepts the usual numbers and parenthesized compound expressions. But there are a few additional atoms of interest:

```
package main;

$atom = $Name
      | $Tuple
      | lookfor("NUMBER", sub { Expression->new('CON', $_[0][1]) })
      | _('OP', '-') - $Expression
          >> sub { Expression->new('-', Expression->new('CON', 0), $_[1]) }
      | _("LPAREN") - $Expression - _("RPAREN") >> sub {$_[1]};
```

The _('OP', '-') production handles unary minus expressions such as -A; this is compiled as if it had been written 0-A.

$name is a variable name, possibly a compound variable name containing dots; it is turned into an expression object containing ['VAR', $varname]:

```
$name = $Base_name
      - star(_("DOT") - _("IDENTIFIER") >> sub { $_[1] })
      >> sub { Expression->new('VAR', join(".", $_[0], @{$_[1]})) }
      ;

$base_name = _"IDENTIFIER";
```

Similarly, a number is turned into an expression object containing ['CON', $number]. (CON is an abbreviation for "constant.")

$tuple is a tuple expression, which we saw before in connection with the constraint:

```
plus = F + (hspc, 0);
```

The (hspc, 0) is a tuple expression. Syntactically, a tuple is a parenthesized, comma-separated list of expressions. But its parser has some interesting features:

```
$tuple = _("LPAREN")
       - commalist($Expression) / sub { @{$_[0]} > 1 }
       - _("RPAREN")
```

The side condition sub { @{$_[0]} > 1 } requires that the comma-separated list have more than one value in it. This prevents something like (3) from ever being parsed as a 1-tuple.

The value of the tuple expression is generated as follows:

```
>> sub {
  my ($explist) = $_[1];
  my $N = @$explist;
  my @axis = qw(x y z);
  if ($N == 2 || $N == 3) {
    return [ 'TUPLE',
             { map { $axis[$_] => $explist->[$_] } (0 .. $N-1) }
           ];
  } else {
    lino_error("$N-tuples are not supported \n");
  }
} ;
```

This does two things. First, it checks to make sure that the tuple has exactly two or three elements. For two-dimensional diagrams, only 2-tuples make sense.

3-tuples are supported because `linogram` might as easily be used for three-dimensional diagrams. One would have to write another standard library, including definitions like:

```
define point { number x, y, z; }
```

and with replacement drawing functions that understood about perspective. But once this was done, `linogram` would handle three-dimensional diagrams as well as it handles two-dimensional ones. Many of the standard library definitions would remain exactly the same. For example, the definition of `line` would not need to change; a line is determined by its two endpoints, regardless of whether those endpoints are considered to be points in two or three dimensions. *n*-tuples for *n* larger than three are forbidden until someone thinks of a use for them.

The value returned from the tuple parser for a tuple such as `(5, 12)` is:

```
[ 'TUPLE',
  { x => 5,
    y => 12,
  }
]
```

For 3-tuples, there is an additional `z` member of the hash. The special treatment of the names `x`, `y`, and `z` comes ultimately from here.

The result of parsing an expression, as mentioned before, is an abstract syntax tree. For the expression `x + 2 * y`, the tree is:

```
[ '+', ['VAR', 'x'],
       ['*', ['CON', 2],
             ['VAR', 'y'],
       ],
]
```

which should be familiar.

When constraint and parameter declarations are processed, they contain these raw `Expression` objects. Later, expressions need to be converted to constraints. This is probably the most complicated part of the program. The process of conversion is essentially evaluation, except that instead of producing a number result, the result is an object from class `Value`. This evaluation is performed by the function `expression_to_constraints()`:

```
sub expression_to_constraints {
  my ($context, $expr) = @_;
```

Variables in an expression have associated types, and to map from a variable's name to its type we need a context. To see why, consider the following example:

```
define type_A {
  number age;
  age = 4;
}

define type_B {
  box age;
  age = 4;
}
```

The constraint `age = 4` in the first definition makes sense, but the same constraint in the second definition does not make sense because 4 is not a box. More generally, the meaning of a constraint might depend in a complex way on the types of the variables it contains. So `expression_to_constraints` requires a context that maps variable names to their types. This is nothing more than a `Type` object; the mapping is performed by `Type::subfeature()`.

Continuing with the evaluation function:

```
unless (defined $expr) {
  Carp::croak("Missing expression in 'expression_to_constraints'");
}
my ($op, @s) = @$expr;
```

Here we break up the top-level expression into an operator `$op` and zero or more subexpressions, `@s`. We then switch on the operator type. It might be a variable, a constant, a tuple, or some binary operator such as + or *:

```
if ($op eq 'VAR') {
  my $name = $s[0];
  return Value::Feature->new_from_var($name, $context->subfeature($name));
```

If it's a variable, we build a new `Value::Feature` object of the indicated name and type. `new_from_var()`, which we saw earlier, is responsible for manufacturing the appropriate set of constraints.

```
} elsif ($op eq 'CON') {
  return Value::Constant->new($s[0]);
```

If the expression is a constant, the code is simple; we build a `Value::Constant` object.

Tuples are where things start to get interesting. As we saw earlier, tuples are *not* required to be constants; `(hspc + 3, 2 * top.start.y)` is a perfectly legitimate tuple. Since the components of a tuple may be arbitrary expressions, we call `expression_to_constraints()` recursively:

```
} elsif ($op eq 'TUPLE') {
  my %components;
  for my $k (keys %{$s[0]}) {
    $components{$k} = expression_to_constraints($context, $s[0]{$k});
  }
  return Value::Tuple->new(%components);
}
```

There should probably be a check here to make sure that the resulting component values are not themselves tuples. At present, `((1, 2), (3, 4))`, which is illegal, is not diagnosed until later, when the malformed tuple participates in an arithmetic operation.

If the argument expression was neither a tuple, a variable, nor a constant, then it's a compound expression. We start by evaluating the two operands:

```
my $e1 = expression_to_constraints($context, $s[0]);
my $e2 = expression_to_constraints($context, $s[1]);
```

We then dispatch an appropriate method to combine the two operands into a single expression. When the operator is +, we use the `add` method, and so on:

```
my %opmeth = ('+' => 'add',
              '-' => 'sub',
              '*' => 'mul',
              '/' => 'div',
             );

my $meth = $opmeth{$op};
if (defined $meth) {
  return $e1->$meth($e2);
} else {
  lino_error("Unknown operator '$op' in AST");
}
```

This is what connects the parser with the arithmetic functions from class `Value`.

The one important function we haven't seen is `convert_param_specs()`, which takes the parameter specifications in a declaration like `hline L(end=Q+R)` and converts them to constraints. The arguments are a context, the sub-feature type (`hline` in the example), and a parameter specification value, something like:

```
{ WHAT => 'PARAM_SPEC',
  NAME => 'end',
  VALUE => [ '+', ['VAR', 'Q'],
                  ['VAR', 'R'],
           ],
}
```

The only fine point here is that parameter specifications are asymmetric. The name `end` on the left side is interpreted as a sub-feature of `L`, but the named `Q` and `R` on the right side are interpreted as sub-features of the outer context in which `L` is being defined. `convert_param_specs()` builds a new `Value::Feature` object for the left side by making two calls to `subfeature()`, one to find

the type of the feature that's being defined, `L` in the example, and then one more to find the type of the parameter name, `end` in the example. It uses the `expression_to_constraints()` function to convert the right-hand side, and then subtracts right from left to produce the final constraint:

```
sub convert_param_specs {
  my ($context, $subobj, $pspec) = @_;
  my @constraints;
  my $left = Value::Feature->new_from_var("$subobj." . $pspec->{NAME},
                                          $context->subfeature($subobj)
                                          ->subfeature($pspec->{NAME})
                                         );
  my $right = expression_to_constraints($context, $pspec->{VALUE});
  return $left->sub($right);
}
```

Our walk through the code is now complete.

9.4.5 Missing Features

`linogram` is missing a few valuable features. Some are easier to fix than others. It doesn't support varying thickness lines, colored lines, or filled boxes. These are easy to add, and in fact an earlier version of `linogram` supports them; I took the feature out for pedagogical reasons. The technical support for the feature was to allow "parameter" declarations, like this:

```
define line {
  point x, y;
  param number thickness = 1;
  param string color = "black";
  draw { &draw_line; }
}
```

A parameter is just another sub-feature, except that it doesn't participate in the system of linear equations. Like any other sub-feature, it may be constrained by the root feature or some other feature that includes it. The following root feature definition draws a vertical black line crossed by a horizontal red line:

```
vline v;
hline h(color="red");
constraints { v.center = h.center; }
```

The `color="red"` parameter specification overrides the default of `"black"`. The parameter values are then included in the environment hash that is passed to the drawing functions. When `draw_line` sees that the color is specified as `"red"` it is responsible for drawing a red line instead of a black one.

With the parameter feature, we can support the placement of objects that contain text:

```
define text extends box {
  param string text = "";
  param number font_size = 9;
  param string font = "courier";
  draw { &draw_text }
}
```

and now we have something that has a top, bottom, left, northwest corner, and so forth, like a box, but whose four sides are invisible. Instead, the `draw_text` function is responsible for placing the text appropriately, or for issuing an error message if it doesn't fit.

The value of a parameter must be completely determined before the constraint system is solved, either by a declaration like `hline h(color="red")`, or by a specified default. If neither is present, it is a fatal error.

Parameters can be used for other applications:

```
define marked_line extends vline {
  hline mark;
  param number markpos = 50;
  constraints {
    mark.length = 0.02;
    mark.center = (center.x, start.y + markpos/100 * height);
  }
}
```

This defines a feature that is a vertical line with a horizontal tick mark across it. By default, the tick mark is halfway up the line, but this depends on the value of *markpos*, which can be between 0 and 100 to indicate a percentage of the way to the end of the `vline`. If *markpos* is 100, the tick mark is at the end of the `vline`; if *markpos* is 75, the tick mark is one-quarter of the way from the end.

If *markpos* were not a `param`, the definition would be illegal, because the expression `markpos/100 * height` is nonlinear. But parameters do not participate in linear-equation solving. The rules for parameters say that *markpos* must be specified somewhere before the equation solving begins. Suppose it has been

specified to be 75. Then the constraint is effectively:

```
mark.center = (center.x, start.y + 75/100 * height);
```

which *is* linear. This feature lends a great deal of flexibility to the system.

One major feature that is missing is splines. A spline is a curved line whose path is determined by one or more control points. The spline wiggles along, starting at its first control point and heading towards the second, then veering off toward the third, and so on, until it ends at the last control point. The main impediment here is that unlike the other features we've seen, the number of control points of a spline isn't known in advance. We could conceivably get around this by defining a series of spline types:

```
define spline2 {
  point p1, p2;
  draw { &draw_spline; }
}

define spline3 extends spline2 {
  point p3;
}

define spline4 extends spline3 {
  point p4;
}

...
```

but this is awfully clumsy. What `linogram` really needs to support features like splines and polygons is a way to specify a parametrizable array of features and their associated constraints, perhaps something like this:

```
define polygon(N) {
  point v[N];
  line s[N];
  constraints {
    when j is 1 .. N   { s[j].start = v[j]; }
    when j is 1 .. N-1 { s[j].end   = v[j+1]; }
    s[N].end = v[1];
  }
}
```

There are a few missing syntactic features. A declaration like:

```
number hsize = 12;
```

would be convenient, as would equations with multiple equals signs:

```
A.sw = B.n = C.s;
```

9.5 CONCLUSION

`linogram` is a substantial application, one that might even be useful. I have been using the venerable `pic` system, developed at Bell Labs, for years, and it convinced me that defining diagrams by writing a text file of constraints is a good general strategy. But I've never been entirely happy with `pic`, and I wanted to see what else I could come up with.

I also wanted to finish the book with a serious example that would demonstrate how the techniques we've studied could be integrated into real Perl programs. `linogram` totals about 1,300 lines of code, counting the parsing system we developed in Chapter 8, but not counting comments, whitespace, curly braces, and the like. It would have been very difficult to build without the techniques of earlier chapters. The parsing system itself was essential; the clean design of the parsing system depends heavily on the earlier work on lazy streams and iterators. We used recursion and dispatch tables throughout to reduce and reorganize the code. Although the program doesn't use any explicit currying or memoization, there are several places where the code would probably be improved by its introduction — the functions based on `apply()`, and the `subfeature()` function spring to mind.