Chapter 3

Substitution

Even in a simple arithmetic language, we sometimes encounter repeated expressions. For instance, the Newtonian formula for the gravitational force between two objects has a squared term in the denominator. We'd like to avoid redundant expressions: they are annoying to repeat, we might make a mistake while repeating them, and they waste computational cycles.

The normal way to avoid redundancy is to introduce an *identifier*.[1] As its name suggests, an identifier names, or identifies, (the value of) an expression. We can then use its name in place of the larger computation. Identifiers may sound exotic, but you're used to them in every programming language you've used so far: they're called *variables*. We choose not to call them that because the term "variable" is semantically charged: it implies that the value associated with the identifier can change (vary). Since our language initially won't offer any way of changing the associated value, we use the more conservative term "identifier". For now, they are therefore just names for computed constants.

Let's first write a few sample programs that use identifiers, inventing notation as we go along:

```
{with {x {+ 5 5}} {+ x x}}
```

We can reduce this to

```
{+ 10 10}
```

by substituting `10` for `x`. The existing rules of evaluation determine that this term's value is `20`. Here's a more elaborate example:

```
  {with {x {+ 5 5}}
    {with {y {- x 3}}
      {+ y y}}}                                  [+ operation]
= {with {x 10} {with {y {- x 3}} {+ y y}}}       [substitution]
= {with {y {- 10 3}} {+ y y}}                    [- operation]
= {with {y 7} {+ y y}}                           [substitution]
= {+ 7 7}                                        [+ operation]
= 14
```

[1] As the authors of *Concrete Mathematics* say: "Name and conquer".

(*En passant*, observe that the act of reducing an expression to a value requires more than just substitution.) Now let's define the language more formally.

To honor the addition of identifiers, we'll give our language a new name: WAE, short for "`with` with arithmetic expressions". Its BNF is:

```
<WAE> ::= <num>
        | {+ <WAE> <WAE>}
        | {- <WAE> <WAE>}
        | {with {<id> <WAE>} <WAE>}
        | <id>
```

Notice that we've had to add *two* rules to the BNF: one for associating values with identifiers and another for actually using the identifiers. The nonterminal `<id>` stands for some suitable syntax for identifiers (usually a sequence of alphanumeric characters).

To write programs that process WAE terms, we need a data definition to represent those terms. Most of WAE carries over unchanged from AE, but we must pick some concrete representation for identifiers. Fortunately, Scheme has a primitive type called the symbol, which serves this role admirably.[2] Therefore, the data definition is

(**define-type** WAE
  [num (*n number?*)]
  [add (*lhs WAE?*) (*rhs WAE?*)]
  [sub (*lhs WAE?*) (*rhs WAE?*)]
  [with (*name symbol?*) (*named-expr WAE?*) (*body WAE?*)]
  [id (*name symbol?*)])

We'll call the expression in the *named-expr* field the *named expression*, since `with` lets the name in the id field stand in place of that expression.

3.1 Defining Substitution

Without fanfare, we used substitution to explain how `with` functions. We were able to do this because substitution is not unique to `with`: we've studied it for years in algebra courses, because that's what happens when we pass arguments to functions. For instance, let $f(x,y) = x^3 + y^3$. Then

$$f(12,1) = 12^3 + 1^3 = 1728 + 1 = 1729$$

$$f(10,9) = 10^3 + 9^3 = 1000 + 729 = 1729$$ [3]

Nevertheless, it's a good idea to pin down this operation precisely.

Let's make sure we understand what we're trying to define. We want a crisp description of the process of *substitution*, namely what happens when we replace an identifier (such as *x* or `x`) with a value (such as 12 or `5`) in an expression (such as $x^3 + y^3$ or `{+ x x}`).

[2] In many languages, a string is a suitable representation for an identifier. Scheme does have strings, but symbols have the salutary property that they can be compared for equality in constant time.

[3] What's the next smallest such number?

Observe that substitution is *not* the same as evaluation. Looking back at the sequence of steps in the evaluation example above, some of them invoke substitution while the rest are evaluation as defined for AE. For now, we're first going to pin down substitution. Once we've done that, we'll revisit the related question of evaluation. But it'll take us a few tries to get substitution right!

Definition 1 (Substitution) *To* substitute *identifier i in e with expression v, replace all identifiers in e that have the name i with the expression v.*

Beginning with the program

```
{with {x 5} {+ x x}}
```

we would need to apply substitution to eliminate the `with` and be left with an arithmetic expression. The definition of substitution above certainly does the trick: substituting `5` for `x` in the body of the `with` yields the program

```
{+ 5 5}
```

as we would want. Likewise, it correctly substitutes `x` with `5` in

```
{+ 10 4}
```

to

```
{+ 10 4}
```

(since there are no instances of `x` in the expression, no substitutions occur). Consider the same substitution in

```
{+ x {with {x 3} 10}}
```

The rules reduce this to

```
{+ 5 {with {5 3} 10}}
```

Huh? Our substitution rule converted a perfectly reasonable program (whose value is `15`) into one that isn't even *syntactically* legal, i.e., it would be rejected by a parser, because the program contains `5` where the BNF tells us to expect an identifier. We definitely don't want substitution to have such an effect! It's obvious that the substitution algorithm is too naïve. To state the problem with the algorithm precisely, though, we need to introduce a little terminology.

Definition 2 (Binding Instance) *A* binding instance *of an identifier is the instance of the identifier that gives it its value. In* WAE*, the* `<id>` *position of a* `with` *is the only binding instance.*

Definition 3 (Scope) *The* scope *of a binding instance is the region of program text in which instances of the identifier refer to the value bound by the binding instance.*

Definition 4 (Bound Instance) *An identifier is* bound *if it is contained within the scope of a binding instance of its name.*

Definition 5 (Free Instance) *An identifier not contained in the scope of any binding instance of its name is said to be* free.

With this terminology in hand, we can now state the problem with the first definition of substitution more precisely: it failed to distinguish between bound instances (which should be substituted) and binding instances (which should not). This leads to a refined notion of substitution.

Definition 6 (Substitution, take 2) *To substitute identifier i in e with expression v, replace all identifiers in e which are not binding instances that have the name i with the expression v.*

A quick check reveals that this doesn't affect the outcome of the examples that the previous definition substituted correctly. In addition, substituting `x` with `5`, this definition of substitution reduces

```
{+ x {with {x 3} 10}}
```

to

```
{+ 5 {with {x 3} 10}}
```

Let's consider a closely related expression with the same substitution:

```
{+ x {with {x 3} x}}
```

Hopefully we can agree that the value of this program is `8` (the left `x` in the addition evaluates to `5`, the right `x` is given the value `3` by the inner `with`, so the sum is `8`). The refined substitution algorithm, however, converts this expression into

```
{+ 5 {with {x 3} 5}}
```

which, when evaluated, yields `10`.

What went wrong here? Our substitution algorithm respected binding instances, but not their scope. In the sample expression, the `with` introduces a new scope for the inner `x`. The scope of the outer `x` is *shadowed* or *masked* by the inner binding. Because substitution doesn't recognize this possibility, it incorrectly substitutes the inner `x`.

Definition 7 (Substitution, take 3) *To substitute identifier i in e with expression v, replace all non-binding identifiers in e having the name i with the expression v, unless the identifier is in a scope different from that introduced by i.*

While this rule avoids the faulty substitution we've discussed earlier, it has the following effect: after substituting for `x`, the expression

```
{+ x {with {y 3} x}}
```

whose value should be that of `{+ 5 5}`, or `10`, becomes

```
{+ 5 {with {y 3} x}}
```

which, when evaluated, halts with an error, because `x` has no value. Once again, substitution has changed a correct program into an incorrect one!

Let's understand what went wrong. Why didn't we substitute the inner `x`? Substitution halts at the `with` because, by definition, every `with` introduces a new scope, which we said should delimit substitution. But this `with` contains an instance of `x`, which we very much want substituted! So which is it—substitute within nested scopes or not? Actually, the two examples above should reveal that our latest definition for substitution, which may have seemed sensible at first blush, is too draconian: it rules out substitution within *any* nested scopes.

Definition 8 (Substitution, take 4) *To substitute identifier i in e with expression v, replace all non-binding identifiers in e having the name i with the expression v, except within nested scopes of i.*

Finally, we have a version of substitution that works. A different, more succint way of phrasing this definition is

Definition 9 (Substitution, take 5) *To substitute identifier i in e with expression v, replace all free instances of i in e with v.*

Recall that we're still defining substitution, not evaluation. Substitution is just an algorithm defined over expressions, independent of any use in an evaluator. It's the calculator's job to invoke substitution as many times as necessary to reduce a program down to an answer. Therefore, substituting `x` with `5` in

```
{+ x {with {y 3} x}}
```

results in

```
{+ 5 {with {y 3} 5}}
```

Reducing this to an actual value is the task of the rest of the calculator.

Phew! Just to be sure we understand this, let's express it in the form of a function.

```
;; subst : WAE symbol WAE → WAE
;; substitutes second argument with third argument in first argument,
;; as per the rules of substitution; the resulting expression contains
;; no free instances of the second argument

(define (subst expr sub-id val)
  (type-case WAE expr
    [num (n) expr]
    [add (l r) (add (subst l sub-id val)
                    (subst r sub-id val))]
    [sub (l r) (sub (subst l sub-id val)
```

```
                (subst r sub-id val))]
    [with (bound-id named-expr bound-body)
          (if (symbol=? bound-id sub-id)
              expr
              (with bound-id
                    named-expr
                    (subst bound-body sub-id val)))]
    [id (v) (if (symbol=? v sub-id) val expr)]))
```

3.2 Calculating with `with`

We've finally defined substitution, but we still haven't specified how we'll use it to reduce expressions to answers. To do this, we must modify our calculator. Specifically, we must add rules for our two new source language syntactic constructs: `with` and identifiers.

- To evaluate `with` expressions, we calculate the named expression, then substitute its value in the body.
- How about identifiers? Well, any identifier that is in the scope of a `with` is replaced with a value when the calculator encounters that identifier's binding instance. Consequently, the purpose statement of *subst* said there would be no free instances of the identifier given as an argument left in the result. In other words, *subst* replaces identifiers with values before the calculator ever "sees" them. As a result, any as-yet-unsubstituted identifier must be free in the whole program. The calculator can't assign a value to a free identifier, so it halts with an error.

```
;; calc : WAE → number
;; evaluates WAE expressions by reducing them to numbers

(define (calc expr)
  (type-case WAE expr
    [num (n) n]
    [add (l r) (+ (calc l) (calc r))]
    [sub (l r) (- (calc l) (calc r))]
    [with (bound-id named-expr bound-body)
          (calc (subst bound-body
                       bound-id
                       (num (calc named-expr))))]
    [id (v) (error 'calc "free identifier")]))
```

One subtlety: In the rule for `with`, the value returned by `calc` is a number, but `subst` is expecting a WAE expression. Therefore, we wrap the result in (num ···) so that substitution will work correctly.

Here are numerous test cases. Each one should pass:

```
(test (calc (parse '5)) 5)
```

(*test* (*calc* (*parse* '{+ 5 5})) 10)
(*test* (*calc* (*parse* '{with {*x* {+ 5 5}} {+ *x* *x*}})) 20)
(*test* (*calc* (*parse* '{with {*x* 5} {+ *x* *x*}})) 10)
(*test* (*calc* (*parse* '{with {*x* {+ 5 5}} {*with* {*y* {- *x* 3}} {+ *y* *y*}}})) 14)
(*test* (*calc* (*parse* '{with {*x* 5} {*with* {*y* {- *x* 3}} {+ *y* *y*}}})) 4)
(*test* (*calc* (*parse* '{with {*x* 5} {+ *x* {*with* {*x* 3} 10}}})) 15)
(*test* (*calc* (*parse* '{with {*x* 5} {+ *x* {*with* {*x* 3} *x*}}})) 8)
(*test* (*calc* (*parse* '{with {*x* 5} {+ *x* {*with* {*y* 3} *x*}}})) 10)
(*test* (*calc* (*parse* '{with {*x* 5} {*with* {*y* *x*} *y*}})) 5)
(*test* (*calc* (*parse* '{with {*x* 5} {*with* {*x* *x*} *x*}})) 5)

3.3 The Scope of `with` Expressions

Just when we thought we were done, we find that several of the test cases above (can you determine which ones?) generate a free-identifier error. What gives?

Consider the program

```
{with {x 5}
  {with {y x}
    y}}
```

Common sense would dictate that its value is `5`. So why does the calculator halt with an error on this test case?

As defined, *subst* fails to correctly substitute in this program, because we did not account for the named sub-expressions in `with` expressions. To fix this problem, we simply need to make `subst` treat the named expressions as ordinary expressions, ripe for substitution. To wit:

```
(define (subst expr sub-id val)
  (type-case WAE expr
    [num (n) expr]
    [add (l r) (add (subst l sub-id val)
                    (subst r sub-id val))]
    [sub (l r) (sub (subst l sub-id val)
                    (subst r sub-id val))]
    [with (bound-id named-expr bound-body)
          (if (symbol=? bound-id sub-id)
              expr
              (with bound-id
                    (subst named-expr sub-id val)
                    (subst bound-body sub-id val)))]
    [id (v) (if (symbol=? v sub-id) val expr)]))
```

The boxed expression shows what changed.

Actually, this isn't quite right either: consider

```
{with {x 5}
  {with {x x}
    x}}
```

This program should evaluate to `5`, but it too halts with an error. This is because we prematurely stopped substituting for `x`. We should substitute in the named expression of a `with` even if the `with` in question defines a new scope for the identifier being substituted, because its named expression is still in the scope of the enclosing binding of the identifier.

We finally get a valid programmatic definition of substitution (relative to the language we have so far):

```
(define (subst expr sub-id val)
  (type-case WAE expr
    [num (n) expr]
    [add (l r) (add (subst l sub-id val)
                    (subst r sub-id val))]
    [sub (l r) (sub (subst l sub-id val)
                    (subst r sub-id val))]
    [with (bound-id named-expr bound-body)
          (if (symbol=? bound-id sub-id)
              (with bound-id
                    (subst named-expr sub-id val)
                    bound-body)
              (with bound-id
                    (subst named-expr sub-id val)
                    (subst bound-body sub-id val)))]
    [id (v) (if (symbol=? v sub-id) val expr)]))
```

Observe how the different versions of *subst* have helped us refine the scope of `with` expressions. By focusing on the small handful of lines that change from one version to the next, and studying how they change, we progressively arrive at a better understanding of scope. This would be much harder to do through mere prose; indeed, our prose definition has not changed at all through these program changes, and translating the definition into a program has helped us run it and determine whether it matches our intuition.

Exercise 3.3.1 *What's the value of*

```
{with {x x} x}
```

? What should *it be, and what does your calculator say it* is*? (These can be two different things!)*

3.4 What Kind of Redundancy do Identifiers Eliminate?

We began this material motivating the introduction of `with`: as a means for eliminating redundancy. Let's revisit this sequence of substitutions:

```
  {with {x {+ 5 5}} {with {y {- x 3}} {+ y y}}}
= {with {x 10} {with {y {- x 3}} {+ y y}}}
= {with {y {- 10 3}} {+ y y}}
= {with {y 7} {+ y y}}
= {+ 7 7}
= 14
```

Couldn't we have also written it this way?

```
  {with {x {+ 5 5}} {with {y {- x 3}} {+ y y}}}
= {with {y {- {+ 5 5} 3}} {+ y y}}
= {+ {- {+ 5 5} 3} {- {+ 5 5} 3}}
= {+ {- 10 3} {- {+ 5 5} 3}}
= {+ {- 10 3} {- 10 3}}
= {+ 7 {- 10 3}}
= {+ 7 7}
= 14
```

In the first sequence of reductions, we first reduce the named expression to a number, then substitute that number. In the second sequence, we perform a "textual" substitution, and only when we have no substitutions left to do do we begin to perform the arithmetic.

Notice that this shows there are really two interpretations of "redundancy" in force. One is a purely *static*[4] notion of redundancy: `with` exists solely to avoid writing an expression twice, even though it will be evaluated twice. This is the interpretation taken by the latter sequence of reductions. In contrast, the former sequence of reductions manifests both static and *dynamic*[5] redundancy elimination: it not only abbreviates the program, it also avoids re-computing the same value during execution.

Given these two sequences of reductions (which we will call *reduction regimes*, since each is governed by a different set of rules), which does our calculator do? Again, it would be hard to reason about this verbally, but because we've written a program, we have a concrete object to study. In particular, the lines we should focus on are those for `with`. Here they are again:

```
...
          [with (bound-id named-expr bound-body)
                (calc (subst bound-body
                             bound-id
                             (num (calc named-expr))))]
...
```

The boxed portion tells us the answer: we invoke *calc* before substitution (because the *result* of *calc* is what we supply as an argument to *subst*). This model of substitution is called *eager*: we "eagerly" reduce the named expression to a value before substituting it. This is in contrast to the second sequence of reductions above, which we call *lazy*, wherein we reduce the named expression to a value only when we need to (such as at the application of an arithmetic primitive).

[4] Meaning, referring only to program text.

[5] Meaning, referring to program execution.

At this point, it may seem like it doesn't make much difference which reduction regime we employ: both produce the same answer (though they may take a different number of steps). But do keep this distinction in mind, for we will see a good deal more on this topic in the course of our study.

Exercise 3.4.1 *Can you* prove *that the eager and lazy regimes will always produce the same answer for the* WAE *language?*

Exercise 3.4.2 *In the example above, the eager regime generated an answer in fewer steps than the lazy regime did. Either prove that that will always be the case, or provide a counterexample.*

Exercise 3.4.3 *At the beginning of this section, you'll find the phrase*

> *an identifier names, or identifies, (the value of) an expression*

Why the parenthetical phrase?

3.5 Are Names Necessary?

A lot of the trouble we've had with defining substitution is the result of having the same name be bound multiple times. To remedy this, a computer scientist named Nicolaas de Bruijn had a good idea.[6] He asked the following daring question: Who needs names at all? De Bruijn suggested that instead, we replace identifiers with numbers. The number dictates how many enclosing scopes away the identifier is bound. (Technically, we replace identifiers not with numbers but *indices* that indicate the binding depth. A number is just a convenient *representation* for an index. A more pictorially pleasing representation for an index is an arrow that leads from the bound to the binding instance, like the ones DrScheme's Check Syntax tool draws.)

The idea is easy to explain with an example: intead of writing

```
{with {x 5} {+ x x}}
```

we would write

```
{with 5 {+ <0> <0>}}
```

Notice that two things changed. First, we replaced the bound identifiers with indices (to keep indices separate from numbers, we wrap each index in pointy brackets). We've adopted the convention that the current scope is zero levels away. Thus, `x` becomes `<0>`. The second change is that, because we no longer care about the names of identifiers, we no longer need keep track of the `x` as the bound identifier. The presence of `with` indicates that we've entered a new scope; that's enough. Similarly, we convert

```
{with {x 5}
  {with {y 3}
    {+ x y}}}
```

[6] De Bruijn had *many* great ideas, particularly in the area of using computers to solve math problems. The idea we present here was a small offshoot of that much bigger project, but as so happens, this is the one many people know him for.

into

```
{with 5
  {with 3
    {+ <1> <0>}}}
```

Let's consider one last example. If this looks incorrect, that would suggest you may have misunderstood the scope of a binding. Examining it carefully actually helps to clarify the scope of bindings. We convert

```
{with {x 5}
  {with {y {+ x 3}}
    {+ x y}}}
```

into

```
{with 5
  {with {+ <0> 3}
    {+ <1> <0>}}}
```

De Bruijn indices are useful in many contexts, and indeed the de Bruijn form of a program (that is, a program where all identifiers have been replaced by their de Bruijn indices) is employed by just about every compiler. You will sometimes find compiler texts refer to the indices as *static distance coordinates*. That name makes sense: the coordinates tell us how far away statically—i.e., in the program text— an identifier is bound. I prefer to use the less informative but more personal moniker as a form of tribute.

```
(define-type WAE
  [num (n number?)]
  [add (lhs WAE?) (rhs WAE?)]
  [sub (lhs WAE?) (rhs WAE?)]
  [with (name symbol?) (named-expr WAE?) (body WAE?)]
  [id (name symbol?)])

;; subst : WAE symbol WAE → WAE
(define (subst expr sub-id val)
  (type-case WAE expr
    [num (n) expr]
    [add (l r) (add (subst l sub-id val)
                    (subst r sub-id val))]
    [sub (l r) (sub (subst l sub-id val)
                    (subst r sub-id val))]
    [with (bound-id named-expr bound-body)
          (if (symbol=? bound-id sub-id)
              (with bound-id
                    (subst named-expr sub-id val)
                    bound-body)
              (with bound-id
                    (subst named-expr sub-id val)
                    (subst bound-body sub-id val)))]
    [id (v) (if (symbol=? v sub-id) val expr)]))

;; calc : WAE → number
(define (calc expr)
  (type-case WAE expr
    [num (n) n]
    [add (l r) (+ (calc l) (calc r))]
    [sub (l r) (- (calc l) (calc r))]
    [with (bound-id named-expr bound-body)
          (calc (subst bound-body
                       bound-id
                       (num (calc named-expr))))]
    [id (v) (error 'calc "free identifier")]))
```

Figure 3.1: Calculator with `with`