Bindings & Substitution

We now get to an important concept: **substitution.**

Even in our simple language, we encounter **repeated expressions**. For example, if we want to compute the square of some expression:

```
{* {+ 4 2} {+ 4 2}}
```

- Why would we want to get rid of the repeated sub-expression?
  - **Redundant computation** -- In this example, we want to avoid computing the same sub-expression a second time.
  - **More complicated computation** -- computation can be simpler without the repetition. Compare the above with:

    ```
    x = {+ 4 2},
      {* x x}
    ```

  - **Duplicating information is always a bad thing** -- Among other bad consequences, it can even **lead to bugs** that could not happen if we wouldn't duplicate code. A toy example is "fixing" one of the numbers in one expression and forgetting to fix the corresponding one:

    ```
    {* {+ 4 2} {+ 4 1}}
    ```

    Real world examples involve much more code, which make such bugs very difficult to find, but they still follow the same principle.

  - **More expressive power** -- We don't just say that we want to multiply two expressions that both *happen to be* `{+ 4 2}`, we say that we multiply the `{+ 4 2}` expression by ***itself***. It allows us to express identity of two values as well as using two values that happen to be the same.

Identifiers

The normal way to avoid redundancy is to introduce an identifier. Even when we speak, we might say: "***let x be 4 plus 2, multiply x by x***".

(These are often called "variables", but we will try to avoid this name: what if the identifier does not change (vary)?)

To get this, we introduce a new form into our language:

```
{with {x {+ 4 2}}
  {* x x}}
```

We expect to be able to reduce this to:

```
{* 6 6}
```

by substituting '`x`' by `6` in the body sub-expression of '`with`'.

A little more complicated example:

```
{with {x {+ 4 2}}
  {with {y {* x x}}
    {+ y y}}}

[add]   = {with {x 6} {with {y {* x x}} {+ y y}}}
[subst]= {with {y {* 6 6}} {+ y y}}
[mul]   = {with {y 36} {+ y y}}
[subst]= {+ 36 36}
[add]   = 72
```

Adding Bindings to AE: The WAE Language

[[[PLAI Chapter 3]]]

To add this to our language, we start with the BNF. We now call our language '**WAE**' (With+AE):

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

Note that we had to introduce **two new rules**:

- **One for introducing an identifier, and**
- **One for using it.**

This is common in many language specifications, for example '**define-type**' introduces a new type, and it comes with '**cases**' that allows us to destruct its instances.

For <id> we need to use some form of identifiers, the natural choice in Racket is to use symbols. We can therefore write the corresponding type definition:

```
(define-type WAE
  [Num  Number]
  [Add  WAE WAE]
  [Sub  WAE WAE]
  [Mul  WAE WAE]
  [Div  WAE WAE]
  [Id   Symbol]
  [With Symbol WAE WAE])
```

The parser is easily extended to produce these syntax objects:

```
  (: parse-sexpr : Sexpr -> WAE)
  ;; to convert s-expressions into WAEs

  (define (parse-sexpr sexpr)
    (match sexpr
      [(number: n)    (Num n)]
      [(symbol: name) (Id name)]
      [(list 'with (list (symbol: name) named) body)
       (With name (parse-sexpr named) (parse-sexpr body))]
      [(list '+ lhs rhs) (Add (parse-sexpr lhs) (parse-sexpr rhs))]
      [(list '- lhs rhs) (Sub (parse-sexpr lhs) (parse-sexpr rhs))]
      [(list '* lhs rhs) (Mul (parse-sexpr lhs) (parse-sexpr rhs))]
      [(list '/ lhs rhs) (Div (parse-sexpr lhs) (parse-sexpr rhs))]
      [else (error 'parse-sexpr "bad syntax in ~s" sexpr)]))
```

But note that this parser is inconvenient -- if any of these expressions:

```
   {* 1 2 3}
   {foo 5 6}
   {with x 5 {* x 8}}
   {with {5 x} {* x 8}}
```

would result in a "**bad syntax**" error, which is not very informative. To make things better, we can add another case for 'with' expressions that are malformed, and give a more specific message in that case:

```
  (: parse-sexpr : Sexpr -> WAE)
  ;; to convert s-expressions into WAEs
  (define (parse-sexpr sexpr)
    (match sexpr
      [(number: n)    (Num n)]
      [(symbol: name) (Id name)]
      [(list 'with (list (symbol: name) named) body)
       (With name (parse-sexpr named) (parse-sexpr body))]
      [(cons 'with more)
       (error 'parse-sexpr "bad 'with' syntax in ~s" sexpr)]
      [(list '+ lhs rhs) (Add (parse-sexpr lhs) (parse-sexpr rhs))]
      [(list '- lhs rhs) (Sub (parse-sexpr lhs) (parse-sexpr rhs))]
      [(list '* lhs rhs) (Mul (parse-sexpr lhs) (parse-sexpr rhs))]
      [(list '/ lhs rhs) (Div (parse-sexpr lhs) (parse-sexpr rhs))]
      [else (error 'parse-sexpr "bad syntax in ~s" sexpr)]))
```

Finally, to group all of the parsing code that deals with 'with' expressions (both valid and invalid ones), we can use a single case for both of them:

```
  (: parse-sexpr : Sexpr -> WAE)
  ;; to convert s-expressions into WAEs
  (define (parse-sexpr sexpr)
    (match sexpr
      [(number: n)     (Num n)]
      [(symbol: name) (Id name)]
      [(cons 'with more)
       ;; go in here for all sexpr that begin with a 'with
       (match sexpr
         [(list 'with (list (symbol: name) named) body)
          (With name (parse-sexpr named) (parse-sexpr body))]
         [else (error 'parse-sexpr "bad `with' syntax in ~s" sexpr)])]
      [(list '+ lhs rhs) (Add (parse-sexpr lhs) (parse-sexpr rhs))]
      [(list '- lhs rhs) (Sub (parse-sexpr lhs) (parse-sexpr rhs))]
      [(list '* lhs rhs) (Mul (parse-sexpr lhs) (parse-sexpr rhs))]
      [(list '/ lhs rhs) (Div (parse-sexpr lhs) (parse-sexpr rhs))]
      [else (error 'parse-sexpr "bad syntax in ~s" sexpr)]))
```

And now we're done with the syntactic part of the '**with** ' extension.

Quick note – the difference between 'With' and 'with':

- We would indent 'With' like a normal function in code like this

```
  (With 'x
        (Num 2)
        (Add (Id 'x) (Num 4)))
```

instead of an indentation that looks like a 'let'

```
  (With 'x (Num 2)
    (Add (Id 'x) (Num 4)))
```

The reason for this is that the second indentation looks more a binding construct (eg, how a 'let' is indented), but '**With**' is ***not*** a binding form -- it's a plain function because it's at the Racket level. You should therefore keep in mind the huge difference between that '**With**' and the '**with**' that appears in WAE programs:

```
  {with {x 2}
    {+ x 4}}
```

Another way to look at it: imagine that we intend for the language to be used by Spanish speakers. In this case we would translate "**with**":

```
  {con {x 2}
    {+ x 4}}
```

but we will not do the same for '**With**'.)

Implementing Evaluation of 'with' (evaluation with substitutions)

To make this work, we will need to do some **substitutions**.

We basically want to say that to evaluate:

```
  {with {id WAE1} WAE2}
```

we need to evaluate **WAE2** with id substituted by **WAE1**. Formally:

```
  eval( {with {id WAE1} WAE2} ) = eval( subst(WAE2,id,eval(WAE1)) )
```

There is a more common syntax for substitution (quick: what do I mean by this "**syntax**"?):

```
  eval( {with {id WAE1} WAE2} ) = eval( WAE2[eval(WAE1)/id] )
```

(Side note: this syntax originates with logicians who used '[x/v]e', and later there was a convention that mimicked the more natural order of arguments to a function with 'e[x->v]', and eventually both of these got combined into 'e[v/x]' which is a little confusing in that the left-to-right order of the arguments is not the same as for the 'subst' function.)

Substitutions:

Now all we need is an exact definition of substitution. (Note that **substitution is not the same as evaluation**, only part of the evaluation process. In the previous examples, when we evaluated the expression we did substitutions as well as the usual arithmetic operations that were already part of the AE evaluator. In this last definition there is still a missing evaluation step, see if you can find it.)

Defining substitutions:

[***substitution***, take 1] **e[v/i]**
To substitute an identifier '**i**' in an expression '**e**' with an expression '**v**', **replace all identifiers** in '**e**' that have the **same name** '**i**' **by the expression** '**v**'.

This seems to work with simple expressions, for example:

```
  {with {x 5} {+ x x}}   -->  {+ 5 5}
  {with {x 5} {+ 10 4}}  -->  {+ 10 4}
```

Problem: we crash with an invalid syntax if we try:

```
  {with {x 5} {+ x {with {x 3} 10}}}  -->  {+ 5 {with {5 3} 10}} ???
```

-- we got to an invalid expression.

To fix this, we need to distinguish "**normal**" occurrences of identifiers, and ones that are used as **new bindings**. We need a few new terms for this:

1. **Binding Instance**: a binding instance of an identifier is one that is used to name it in a new binding. In our **<WAE>** syntax, binding instances are only the **<id>** position of the '**with**' form.

2. **Scope**: the scope of a binding instance is the region of program text in which instances of the identifier refer to the value bound in the binding instance. (Note that this definition actually relies on a definition of substitution, because that is what is used to specify how identifiers refer to values.)

3. **Bound Instance (or Bound Occurrence)**: an instance of an identifier is bound if it is contained within the scope of a binding instance of its name.

4. **Free Instance (or Free Occurrence)**: An identifier that is not contained in the scope of any binding instance of its name is said to be free.

Using this we can say that the problem with the previous definition of substitution is that **it failed to distinguish between bound instances** (which should be substituted) from **binding instances** (which should not).

So, we try to fix this:

[***substitution***, take 2] **e[v/i]**
To substitute an identifier '**i**' in an expression '**e**' with an expression '**v**', replace all instances of '**i**' that are **not themselves binding instances** with the expression '**v**'.

First of all, check the previous examples:

```
  {with {x 5} {+ x x}}    -->  {+ 5 5}
  {with {x 5} {+ 10 4}}   -->  {+ 10 4}
```

still work, and

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

also works. However, if we try this:

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

we get:

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

but we want that to be **8**: the inner '**x**' should be bound by the closest '**with**' that binds it.

The problem is that the new definition of substitution that we have respects binding instances, but it **fails to deal with their scope.** In the above example, we want the inner '**with**' to ***shadow*** the outer '**with**'s binding for '**x**'.

[***substitution***, take 3] **e[v/i]**
To substitute an identifier '**i**' in an expression '**e**' with an expression '**v**', replace all instances of '**i**' that are **not themselves binding instances**, and that are **not in any nested scope**, with the expression '**v**'.

This avoids bad substitution above, but it is now doing things too carefully:

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

becomes

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

which is an **error** because '**x**' is unbound (and there is no reasonable rule that we can specify to evaluate it).

The problem is that our substitution halts at every new scope, in this case, it stopped at the new '**y**' scope, but it shouldn't have because it uses a different name.

Revise again:

[***substitution***, **take 4**] **e[v/i]**
To substitute an identifier '**i**' in an expression '**e**' with an expression '**v**', replace all instances of '**i**' that **are not themselves binding instances**, and that **are not in any nested scope of** '**i**', with the expression '**v**'.

Finally, this is a good definition. It is just a little too mechanical. Notice that we actually refer to all instances of '**i**' that are not in a scope of a binding instance of '**i**', which simply means all ***free occurrences*** of '**i**' -- free in '**e**' (why? -- remember the definition of "free"?):

[***substitution***, **take 4a**] **e[v/i]**
To substitute an identifier '**i**' in an expression '**e**' with an expression '**v**', replace all instances of '**i**' that **are free in** '**e**' with the expression '**v**'.

Based on this we can finally write the code for it:

```
(: subst : WAE Symbol WAE -> WAE)
;; substitutes the second argument with the third argument in the
;; first argument, as per the rules of substitution; the resulting
;; expression contains no free instances of the second argument
(define (subst expr from to) ; returns expr[to/from]
  (cases expr
    [(Num n) expr]
    [(Add l r) (Add (subst l from to) (subst r from to))]
    [(Sub l r) (Sub (subst l from to) (subst r from to))]
    [(Mul l r) (Mul (subst l from to) (subst r from to))]
    [(Div l r) (Div (subst l from to) (subst r from to))]
    [(Id name) (if (eq? name from) to expr)]
    [(With bound-id named-expr bound-body)
     (if (eq? bound-id from)
       expr           ; <-- don't go in!
       (With bound-id
             named-expr
             (subst bound-body from to)))]))
```

... and this is just the same as writing a formal "paper version" of the substitution rule.

... but we still have bugs! ………