PL: Lecture #6

Tuesday, September 24th

Evaluation of `with`

Now, 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,WAE1) )
```

There is a more common syntax for substitution (quick: what do I mean by this use of “syntax”?):

```
eval( {with {id WAE1} WAE2} )
  = eval( WAE2[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.*

Now all we need is an exact definition of substitution.

> *Note that substitution is not the same as evaluation, it’s only a 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.*

So let us try to define substitution now:

> *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}
```

however, 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 / Identifier): 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 / Identifier): An identifier that is not contained in 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) and 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 reasonable no 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. In fact, that last definition of substitution cannot handle any nested scope.

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`.*

which, finally, is a good definition. This 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 4b): `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.

We still have bugs: but we'll need some more work to get to them.

Before we find the bugs, we need to see when and how substitution is used in the evaluation process.

To modify our evaluator, we will need rules to deal with the new syntax pieces — `with` expressions and identifiers.

When we see an expression that looks like:

```
{with {x E1} E2}
```

we continue by *evaluating* `E1` to get a value `V1`, we then substitute the identifier `x` with the expression `V1` in `E2`, and continue by evaluating this new expression. In other words, we have the following evaluation rule:

```
eval( {with {x E1} E2} )
  = eval( E2[eval(E1)/x] )
```

So we know what to do with `with` expressions. How about identifiers? The main feature of `subst`, as said in the purpose statement, is that it leaves no free instances of the substituted variable around. This means that if the initial expression is valid (did not contain any free variables), then when we go from

```
{with {x E1} E2}
```

to

```
E2[E1/x]
```

the result is an expression that has *no* free instances of `x`. So we don't need to handle identifiers in the evaluator — substitutions make them all go away.

We can now extend the formal definition of AE to that of WAE:

```
eval(...) = ... same as the AE rules ...
eval({with {x E1} E2}) = eval(E2[eval(E1)/x])
eval(id) = error!
```

If you're paying close attention, you might catch a potential problem in this definition: we're substituting `eval(E1)` for `x` in `E2` — an operation that requires a WAE expression, but `eval(E1)` is a number. (Look at the type of the `eval` definition we had for AE, then look at the above definition of `subst`.) This seems like being overly pedantic, but we it will require some resolution when we get to the code. The above rules are easily coded as follows:

```
(: eval : WAE -> Number)
;; evaluates WAE expressions by reducing them to numbers
(define (eval expr)
  (cases expr
    [(Num n) n]
    [(Add l r) (+ (eval l) (eval r))]
    [(Sub l r) (- (eval l) (eval r))]
    [(Mul l r) (* (eval l) (eval r))]
    [(Div l r) (/ (eval l) (eval r))]
    [(With bound-id named-expr bound-body)
     (eval (subst bound-body
                  bound-id
                  (Num (eval named-expr))))] ;***
    [(Id name) (error 'eval "free identifier: ~s" name)]))
```

Note the `Num` expression in the marked line: evaluating the named expression gives us back a number — we need to convert this number into a syntax to be able to use it with `subst`. The solution is to use `Num` to convert the resulting number into a numeral (the syntax of a number). It's not an elegant solution, but it will do for now.

Finally, here are a few test cases. We use a new `test` special form which is part of the course plugin. The way to use `test` is with two expressions and an `=>` arrow — DrRacket evaluates both, and nothing will happen if the results are equal. If the results are different, you will get a warning line, but evaluation will continue so you can try additional tests. You can also use an `=error>` arrow to test an error message — use it with some text from the expected error, `?` stands for any single character, and `*` is a sequence of zero or more characters. (When you use `test` in your homework, the handin server will abort when tests fail.) We expect these tests to succeed (make sure that you understand *why* they should succeed).

```
;; tests
(test (run "5") => 5)
(test (run "{+ 5 5}") => 10)
(test (run "{with {x {+ 5 5}} {+ x x}}") => 20)
(test (run "{with {x 5} {+ x x}}") => 10)
(test (run "{with {x {+ 5 5}} {with {y {- x 3}} {+ y y}}}") => 14)
(test (run "{with {x 5} {with {y {- x 3}} {+ y y}}}") => 4)
(test (run "{with {x 5} {+ x {with {x 3} 10}}}") => 15)
(test (run "{with {x 5} {+ x {with {x 3} x}}}") => 8)
(test (run "{with {x 5} {+ x {with {y 3} x}}}") => 10)
(test (run "{with {x 5} {with {y x} y}}") => 5)
```

```
(test (run "{with {x 5} {with {x x} x}}") => 5)
(test (run "{with {x 1} y}") =error> "free identifier")
```

Putting this all together, we get the following code; trying to run this code will raise an unexpected error…

```
#lang pl

#| BNF for the WAE language:
     <WAE> ::= <num>
             | { + <WAE> <WAE> }
             | { - <WAE> <WAE> }
             | { * <WAE> <WAE> }
             | { / <WAE> <WAE> }
             | { with { <id> <WAE> } <WAE> }
             | <id>
|#

;; WAE abstract syntax trees
(define-type WAE
  [Num  Number]
  [Add  WAE WAE]
  [Sub  WAE WAE]
  [Mul  WAE WAE]
  [Div  WAE WAE]
  [Id   Symbol]
  [With Symbol WAE WAE])

(: parse-sexpr : Sexpr -> WAE)
;; parses s-expressions into WAEs
(define (parse-sexpr sexpr)
  (match sexpr
    [(number: n)    (Num n)]
    [(symbol: name) (Id name)]
    [(cons 'with more)
     (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)]))

(: parse : String -> WAE)
;; parses a string containing a WAE expression to a WAE AST
(define (parse str)
  (parse-sexpr (string->sexpr str)))

(: 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)
  (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
       (With bound-id
             named-expr
             (subst bound-body from to)))]))

(: eval : WAE -> Number)
;; evaluates WAE expressions by reducing them to numbers
(define (eval expr)
  (cases expr
    [(Num n) n]
    [(Add l r) (+ (eval l) (eval r))]
    [(Sub l r) (- (eval l) (eval r))]
    [(Mul l r) (* (eval l) (eval r))]
    [(Div l r) (/ (eval l) (eval r))]
    [(With bound-id named-expr bound-body)
     (eval (subst bound-body
                  bound-id
                  (Num (eval named-expr))))]
    [(Id name) (error 'eval "free identifier: ~s" name)]))

(: run : String -> Number)
;; evaluate a WAE program contained in a string
(define (run str)
  (eval (parse str)))

;; tests
(test (run "5") => 5)
(test (run "{+ 5 5}") => 10)
(test (run "{with {x {+ 5 5}} {+ x x}}") => 20)
(test (run "{with {x 5} {+ x x}}") => 10)
(test (run "{with {x {+ 5 5}} {with {y {- x 3}} {+ y y}}}") => 14)
(test (run "{with {x 5} {with {y {- x 3}} {+ y y}}}") => 4)
(test (run "{with {x 5} {+ x {with {x 3} 10}}}") => 15)
(test (run "{with {x 5} {+ x {with {x 3} x}}}") => 8)
(test (run "{with {x 5} {+ x {with {y 3} x}}}") => 10)
(test (run "{with {x 5} {with {y x} y}}") => 5)
(test (run "{with {x 5} {with {x x} x}}") => 5)
(test (run "{with {x 1} y}") =error> "free identifier")
```

Oops, this program still has problems that were caught by the tests — we encounter unexpected free identifier errors. What's the problem now? In expressions like:

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

we forgot to substitute `x` in the expression that `y` is bound to. We need to the recursive substitute in both the with's body expression as well as its named expression:

```
(: 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)
  (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
       (With bound-id
             (subst named-expr from to)       ;*** new
             (subst bound-body from to)))]))
```

And *still* we have a problem… Now it's

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

that halts with an error, but we want it to evaluate to `5`! Carefully trying out our substitution code reveals the problem: when we substitute `5` for the outer `x`, we don't go inside the inner `with` because it has the same name — but we *do* need to go into its named expression. We need to substitute in the named expression even if the identifier is the *same* one we're substituting:

```
(: 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)
  (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)
     (With bound-id
           (subst named-expr from to)
           (if (eq? bound-id from)
             bound-body
             (subst bound-body from to)))]))
```

The complete (and, finally, correct) version of the code is now:

```
#lang pl

#| BNF for the WAE language:
     <WAE> ::= <num>
             | { + <WAE> <WAE> }
             | { - <WAE> <WAE> }
             | { * <WAE> <WAE> }
             | { / <WAE> <WAE> }
             | { with { <id> <WAE> } <WAE> }
             | <id>
|#
```

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

(: parse-sexpr : Sexpr -> WAE)
;; parses s-expressions into WAEs
(define (parse-sexpr sexpr)
  (match sexpr
    [(number: n)    (Num n)]
    [(symbol: name) (Id name)]
    [(cons 'with more)
     (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)]))

(: parse : String -> WAE)
;; parses a string containing a WAE expression to a WAE AST
(define (parse str)
  (parse-sexpr (string->sexpr str)))

#| Formal specs for `subst':
   (`N' is a <num>, `E1', `E2' are <WAE>s, `x' is some <id>,
   `y' is a *different* <id>)
      N[v/x]                = N
      {+ E1 E2}[v/x]        = {+ E1[v/x] E2[v/x]}
      {- E1 E2}[v/x]        = {- E1[v/x] E2[v/x]}
      {* E1 E2}[v/x]        = {* E1[v/x] E2[v/x]}
      {/ E1 E2}[v/x]        = {/ E1[v/x] E2[v/x]}
      y[v/x]                = y
      x[v/x]                = v
      {with {y E1} E2}[v/x] = {with {y E1[v/x]} E2[v/x]}
      {with {x E1} E2}[v/x] = {with {x E1[v/x]} E2}
|#

(: 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)
  (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)
     (With bound-id
           (subst named-expr from to)
           (if (eq? bound-id from)
             bound-body
             (subst bound-body from to)))]))

#| Formal specs for `eval':
     eval(N)          = N
     eval({+ E1 E2}) = eval(E1) + eval(E2)
     eval({- E1 E2}) = eval(E1) - eval(E2)
     eval({* E1 E2}) = eval(E1) * eval(E2)
     eval({/ E1 E2}) = eval(E1) / eval(E2)
     eval(id)         = error!
     eval({with {x E1} E2}) = eval(E2[eval(E1)/x])
|#

(: eval : WAE -> Number)
;; evaluates WAE expressions by reducing them to numbers
(define (eval expr)
  (cases expr
    [(Num n) n]
    [(Add l r) (+ (eval l) (eval r))]
    [(Sub l r) (- (eval l) (eval r))]
    [(Mul l r) (* (eval l) (eval r))]
    [(Div l r) (/ (eval l) (eval r))]
    [(With bound-id named-expr bound-body)
     (eval (subst bound-body
                  bound-id
                  (Num (eval named-expr))))]
    [(Id name) (error 'eval "free identifier: ~s" name)]))

(: run : String -> Number)
;; evaluate a WAE program contained in a string
(define (run str)
  (eval (parse str)))

;; tests
(test (run "5") => 5)
(test (run "{+ 5 5}") => 10)
(test (run "{with {x 5} {+ x x}}") => 10)
(test (run "{with {x {+ 5 5}} {+ x x}}") => 20)
(test (run "{with {x 5} {with {y {- x 3}} {+ y y}}}") => 4)
(test (run "{with {x {+ 5 5}} {with {y {- x 3}} {+ y y}}}") => 14)
(test (run "{with {x 5} {+ x {with {x 3} 10}}}") => 15)
(test (run "{with {x 5} {+ x {with {x 3} x}}}") => 8)
(test (run "{with {x 5} {+ x {with {y 3} x}}}") => 10)
(test (run "{with {x 5} {with {y x} y}}") => 5)
(test (run "{with {x 5} {with {x x} x}}") => 5)
(test (run "{with {x 1} y}") =error> "free identifier")
```

Reminder:

- We started doing substitution, with a `let`-like form: `with`.
- Reasons for using bindings:
  - Avoid writing expressions twice.
    - More expressive language (can express identity).

    - Duplicating is bad! (“DRY”: *Don’t Repeat Yourself*.)
    - Avoids *static* redundancy.
  - Avoid redundant computations.
    - More than *just* an optimization when it avoids exponential resources.
    - Avoids *dynamic* redundancy.
- BNF:

```
<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.

- 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])
```

- Parser:

```
(: parse-sexpr : Sexpr -> WAE)
;; parses s-expressions into WAEs
(define (parse-sexpr sexpr)
  (match sexpr
    [(number: n)    (Num n)]
    [(symbol: name) (Id name)]
    [(cons 'with more)
     (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) (Mul (parse-sexpr lhs) (parse-sexpr rhs))]
    [else (error 'parse-sexpr "bad syntax in ~s" sexpr)]))
```

- We need to define substitution. Terms:
  1. Binding Instance.
  2. Scope.
  3. Bound Instance.
  4. Free Instance.
- After lots of attempts:

> *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`.*

- Implemented the code, and again, needed to fix a few bugs:

```
(: 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)
  (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)
     (With bound-id
           (subst named-expr from to)
           (if (eq? bound-id from)
             bound-body
             (subst bound-body from to)))]))
```

  (Note that the bugs that we fixed clarify the exact way that our scopes work: in `{with {x 2} {with {x {+ x 2}} x}}`, the scope of the first `x` is the `{+ x 2}` expression.)

- We then extended the AE evaluation rules:

```
eval(...) = ... same as the AE rules ...
eval({with {x E1} E2}) = eval(E2[eval(E1)/x])
eval(id) = error!
```

  and noted the possible type problem.

- The above translated into a Racket definition for an `eval` function (with a hack to avoid the type issue):

```
(: eval : WAE -> Number)
;; evaluates WAE expressions by reducing them to numbers
(define (eval expr)
  (cases expr
    [(Num n) n]
    [(Add l r) (+ (eval l) (eval r))]
    [(Sub l r) (- (eval l) (eval r))]
    [(Mul l r) (* (eval l) (eval r))]
    [(Div l r) (/ (eval l) (eval r))]
    [(With bound-id named-expr bound-body)
     (eval (subst bound-body
                  bound-id
                  (Num (eval named-expr))))]
    [(Id name) (error 'eval "free identifier: ~s" name)]))
```

Formal Specs

Note the formal definitions that were included in the WAE code. They are ways of describing pieces of our language that are more formal than plain English, but still not as formal (and as verbose) as the actual code.

A formal definition of `subst`:

(`N` is a `<num>`, `E1`, `E2` are `<WAE>`s, `x` is some `<id>`, `y` is a *different* `<id>`)

```
N[v/x]                  = N

{+ E1 E2}[v/x]          = {+ E1[v/x] E2[v/x]}

{- E1 E2}[v/x]          = {- E1[v/x] E2[v/x]}

{* E1 E2}[v/x]          = {* E1[v/x] E2[v/x]}

{/ E1 E2}[v/x]          = {/ E1[v/x] E2[v/x]}

y[v/x]                  = y
x[v/x]                  = v

{with {y E1} E2}[v/x] = {with {y E1[v/x]} E2[v/x]}
{with {x E1} E2}[v/x] = {with {x E1[v/x]} E2}
```

And a formal definition of `eval`:

```
eval(N)          = N

eval({+ E1 E2}) = eval(E1) + eval(E2)

eval({- E1 E2}) = eval(E1) - eval(E2)

eval({* E1 E2}) = eval(E1) * eval(E2)

eval({/ E1 E2}) = eval(E1) / eval(E2)

eval(id)         = error!

eval({with {x E1} E2}) = eval(E2[eval(E1)/x])
```