Bindings & Substitution

- Even in our simple language, we encounter repeated expressions.

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

- There are several reasons to eliminate duplicated code.

- It introduces a redundant computation.
- Repetition obscures meaning.

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

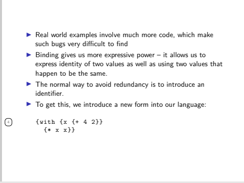
- Duplicating information is always a bad thing.
- Cut and paste leads to bugs:

3
```
{* {+ 4 2} {+ 4 1}}
```

- Real world examples involve much more code, which make such bugs very difficult to find
- Binding gives us more expressive power – it allows us to express identity of two values as well as using two values that happen to be the same.
- The normal way to avoid redundancy is to introduce an identifier.
- To get this, we introduce a new form into our language:

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

2019-02-09



- Real world examples involve much more code, which make such bugs very difficult to find
- Binding gives us more expressive power – it allows us to express identity of two values as well as using two values that happen to be the same.
- The normal way to avoid redundancy is to introduce an identifier.
- To get this, we introduce a new form into our language:

(1)
```
{with {x {+ 4 2}}
  {* x x}}
```

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

- We can reduce this to: `{* 6 6}` by **substituting** 6 for 'x' in the body sub-expression of 'with'.

- A little more complicated example:

5
```
{with {x {+ 4 2}}
  {with {y {* x x}}
    {+ y y}}}
```

- We can do a series of substitutions

6
```
= {with {x 6} {with {y {* x x}}   [add]
                    {+ y y}}}

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

Adding Bindings to AE: The WAE Language

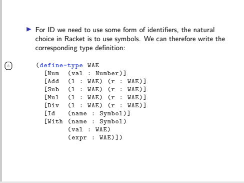
- PLAI (1st ed) Chapter 3. PLAI (2nd ed) Section 5.3.
- To add `with` to our language, we start with the BNF. We now call our language 'WAE' (With+AE):
- Two new rules: one for introducing an identifier, and one for using it.

```
wae: NUMBER
        | { + wae wae }
        | { - wae wae }
        | { * wae wae }
        | { / wae wae }
        | { with { ID wae } wae }
        | ID
```

- 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  (val : Number)]
  [Add  (l : WAE) (r : WAE)]
  [Sub  (l : WAE) (r : WAE)]
  [Mul  (l : WAE) (r : WAE)]
  [Div  (l : WAE) (r : WAE)]
  [Id   (name : Symbol)]
  [With (name : Symbol)
        (val : WAE)
        (expr : WAE)])
```

2019-02-09



- 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  (val : Number)]
  [Add  (l : WAE) (r : WAE)]
  [Sub  (l : WAE) (r : WAE)]
  [Mul  (l : WAE) (r : WAE)]
  [Div  (l : WAE) (r : WAE)]
  [Id   (name : Symbol)]
  [With (name : Symbol)
        (val : WAE)
        (expr : WAE)])
```

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

We can add another case (actually two) to our parser to handle the with form

9

```
(cond
    ⋮
  [(s-exp-symbol? sx) (Id (s-exp->symbol sx))]
  [(s-exp-match? '(with (SYMBOL ANY) ANY) sx)
   (let* ([def (sx-ref sx 1)]
          [id (s-exp->symbol (sx-ref def 0))]
          [val (parse-sx (sx-ref def 1))]
          [expr (parse-sx (sx-ref sx 2))])
     (With id val expr))]
```

Error handling could definitely be improved:

```
(test/exn  (parse-sx `{* 1 2 3}) "parse error")
(test/exn (parse-sx `{foo 5 6}) "parse error")
(test/exn (parse-sx `{with x 5 {* x 8}})
          "parse error")
(test/exn (parse-sx `{with {5 x} {* x 8}})
      "parse error")
(test (parse-sx `{with {x 5} {* x 8}})
      (With 'x (Num 5) (Mul (Id 'x) (Num 8))))
```

Implementing ‘with’ Evaluation

- We will need to do some substitutions.
  To evaluate:

  11
  ```
  {with {id WAE1} WAE2}
  ```

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

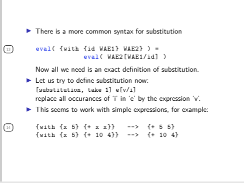
  12
  ```
  eval( {with {id WAE1} WAE2} ) =
        eval( subst(WAE2,id,WAE1) )
  ```

- There is a more common syntax for substitution

13
```
eval( {with {id WAE1} WAE2} ) =
            eval( WAE2[WAE1/id] )
```

  Now all we need is an exact definition of substitution.
- Let us try to define substitution now:
  `[substitution, take 1] e[v/i]`
  replace all occurances of ‘i’ in ‘e’ by the expression ‘v’.
- This seems to work with simple expressions, for example:

14
```
{with {x 5} {+ x x}}    -->  {+ 5 5}
{with {x 5} {+ 10 4}}   -->  {+ 10 4}
```



- There is a more common syntax for substitution

13
```
eval( {with {id WAE1} WAE2} ) =
            eval( WAE2[WAE1/id] )
```

  Now all we need is an exact definition of substitution.

- Let us try to define substitution now:
  `[substitution, take 1] e[v/i]`
  replace all occurances of 'i' in 'e' by the expression 'v'.
- This seems to work with simple expressions, for example:

14
```
{with {x 5} {+ x x}}   -->  {+ 5 5}
{with {x 5} {+ 10 4}}  -->  {+ 10 4}
```

1. 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.

- 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.

- **Binding** Instance: names the identifier in a new binding.
- In our BNF syntax, binding instances are only the ID position of the 'with' form.
- **Scope** region of program text in which instances of the identifier refer to the value bound in the binding instance.
- **Bound** Instance (or **Bound** Occurrence): contained within the scope of a binding instance of its name.
- **Free** Instance (or Free Occurrence): not contained in any binding instance of its name

- **Binding** Instance: names the identifier in a new binding.
- In our BNF syntax, binding instances are only the ID position of the 'with' form.
- **Scope** region of program text in which instances of the identifier refer to the value bound in the binding instance.
- **Bound** Instance (or **Bound** Occurrence): contained within the scope of a binding instance of its name.
- **Free** Instance (or Free Occurrence): not contained in any binding instance of its name

1. Note that this definition of **scope** actually relies on a definition of substitution, because that is what is used to specify how identifiers refer to values.

- the previous definition of substitution failed to distinguish between **bound** instances and **binding** instances.
- So we try to fix this:
  `[substitution, take 2] e[v/i]`
  replace all non-binding instances of ‘i’ with the expression ‘v’.
- The previous working examples still work

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

- and one more

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

- However, if we try this:

18
```
{with {x 5}
  {+ x {with {x 3}
         x}}}
```

- we get:

19
```
-->  {+ 5 {with {x 3} 5}}
-->  {+ 5 5}
-->  10 = wrong
```

- In the above example, we want the inner ‘with’ to **shadow** the outer ‘with’s binding for ‘x’.
  `[substitution, take 3] e[v/i]`
  replace all non-binding instances of ‘i’ not in any nested scope with ‘v’.

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

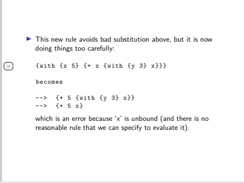
```
{with {x 5} {+ x {with {y 3} x}}}

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).

2019-02-09



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

```
20      {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).

1. 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.

- Finally, maybe correct:
  `[substitution, take 4] e[v/i]`
  replace all instances of non-binding instances of ‘i’ that are not in any nested scope of ‘i’, with the expression ‘v’.
- We can shorten this using the definition of **free**
  `[substitution, take 4b] e[v/i]`
  replace all instances of ‘i’ that are free in ‘e’ with the expression ‘v’.

```
; returns expr[to/from].
; leaves no free occurences of `to'
(define (subst expr from to)
  (type-case WAE expr
    [(Add l r) (Add (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)))]))
```

- this is just as good as writing a formal ”paper version” of the substitution rule.
- Or maybe better. Let’s test a few cases.

22

```
(test (subst
        (Add (Id 'x) (With 'x (Num 3) (Id 'x)))
        'x (Num 5))
      (Add (Num 5) (With 'x (Num 3) (Id 'x))))
(test (subst
        (Add (Id 'x) (With 'y (Num 3) (Id 'x)))
        'x (Num 5))
      (Add (Num 5) (With 'y (Num 3) (Num 5))))
```

- but we still have bugs!
- 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.

evaluating ‘with’

- When we see an expression that looks like:

23
```
{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:

24
```
eval( {with {x E1} E2} )
      = eval( E2[eval(E1)/x] )
```

What about identifiers?

- 'subst' leaves no free instances of the substituted variable around.

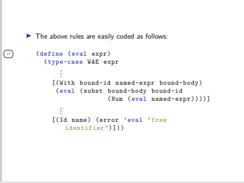
  25
  ```
  {with {x E1} E2} =>  E2[E1/x]
  ```

  If the initial expression is valid (did not contain any free variables), then substition removes all free identifiers.
- We can now extend the eval rule of AE to WAE:

  26
  ```
  eval(...) = ... same as the AE rules ...
  eval({with {x E1} E2}) = eval(E2[eval(E1)/x])
  eval(id) = error!
  ```

- The above rules are easily coded as follows:

```
(define (eval expr)
  (type-case WAE expr
      ⋮
    [(With bound-id named-expr bound-body)
     (eval (subst bound-body bound-id
                  (Num (eval named-expr))))]
      ⋮
    [(Id name) (error 'eval "free
       identifier")]))
```



- The above rules are easily coded as follows:

27

```
(define (eval expr)
  (type-case WAE expr
      ⋮
    [(With bound-id named-expr bound-body)
     (eval (subst bound-body bound-id
                  (Num (eval named-expr))))]
      ⋮
    [(Id name) (error 'eval "free
       identifier")]))
```

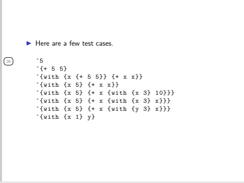
1. If you're paying close attention, you might catch a potential problem in this definition: we're substituting 'eval(E1)' for 'x' in 'E2' – an operation that requires a WAE expression, but 'eval(E1)' is a number. (Look at the type of the 'eval' definition we had for AE, then look at the above definition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

- Here are a few test cases.

28

```
`5
`{+ 5 5}
`{with {x {+ 5 5}} {+ x x}}
`{with {x 5} {+ x x}}
`{with {x 5} {+ x {with {x 3} 10}}}
`{with {x 5} {+ x {with {x 3} x}}}
`{with {x 5} {+ x {with {y 3} x}}}
`{with {x 1} y}
```

2019-02-09



- Here are a few test cases.

(28)

```
`5
`{+ 5 5}
`{with {x {+ 5 5}} {+ x x}}
`{with {x 5} {+ x x}}
`{with {x 5} {+ x {with {x 3} 10}}}
`{with {x 5} {+ x {with {x 3} x}}}
`{with {x 5} {+ x {with {y 3} x}}}
`{with {x 1} y}
```

1. 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.

- What about these cases?

```
(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} {with {y x} y}}}) 5)
(test (run `{with {x 5} {with {x x} x}}}) 5)
```

In expressions like:

30
```
{with {x 5}
  {with {y x}
    y}}
```

we forgot to substitute 'x' in `{with {y x} ...}`

- We need to the recursive substitute in both the with's named-expression as well as its body.

```
;; expr[to/from]
(define (subst expr from to)
  (type-case WAE 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
- 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 substituting.

```
;; expr[to/from]
(define (subst expr from to)
  (type-case WAE expr
        ⋮
    [(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)))]))
```

```
(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/exn (run `{with {x 1} y}) "free identifier")
```