

Variables and Binding

The Arbitrariness of Identifiers

The “Are the following two programs equivalent?” game

The Arbitrariness of Identifiers

Are the following two programs equivalent?

```
(define (f x) (+ x 1))  
(f 10)
```

```
(define (f y) (+ y 1))  
(f 10)
```

yes

argument is consistently renamed

The Arbitrariness of Identifiers

Are the following two programs equivalent?

```
(define (f x) (+ x 1))  
(f 10)
```

```
(define (f x) (+ y 1))  
(f 10)
```

no

not a use of the argument anymore

The Arbitrariness of Identifiers

Are the following two programs equivalent?

```
(define (f x) (+ x 1))  
(f 10)
```

```
(define (f y) (+ x 1))  
(f 10)
```

no

not a use of the argument anymore

The Arbitrariness of Identifiers

Are the following two programs equivalent?

```
(define (f x) (+ y 1))  
(f 10)
```

```
(define (f z) (+ y 1))  
(f 10)
```

yes

argument never used, so almost any name is ok

The Arbitrariness of Identifiers

Are the following two programs equivalent?

```
(define (f x) (+ y 1))  
(f 10)
```

```
(define (f y) (+ y 1))  
(f 10)
```

no

now a use of the argument

The Arbitrariness of Identifiers

Are the following two programs equivalent?

```
(define (f x) (+ y 1))  
(f 10)
```

```
(define (f x) (+ z 1))  
(f 10)
```

no

still an unbound identifier, but a different one

The Arbitrariness of Identifiers

Are the following two programs equivalent?

```
(define (f x)
  (local [(define y 10)]
    (+ x y)))
(f 0)
```

```
(define (f z)
  (local [(define y 10)]
    (+ z y)))
(f 0)
```

yes

argument is consistently renamed

The Arbitrariness of Identifiers

Are the following two programs equivalent?

```
(define (f x)
  (local [(define y 10)]
    (+ x y)))
(f 0)
```

```
(define (f x)
  (local [(define z 10)]
    (+ x z)))
(f 0)
```

yes

local identifier is consistently renamed

The Arbitrariness of Identifiers

Are the following two programs equivalent?

```
(define (f x)
  (local [(define y 10)]
    (+ x y)))
(f 0)
```

```
(define (f x)
  (local [(define x 10)]
    (+ x x)))
(f 0)
```

no

local identifier now shadows (hides) the argument

The Arbitrariness of Identifiers

Are the following two programs equivalent?

```
(define (f x)
  (local [(define y 10)]
    (+ x y)))
(f 0)
```

```
(define (f y)
  (local [(define y 10)]
    (+ y y)))
(f 0)
```

no

local identifier now shadows the argument

Free and Bound Identifiers

An identifier for the argument of a function or the name of a local identifier is a **binding occurrence**.

```
(define (f x y) (+ x y z))
```

```
(local [(define a 3)
        (define c 4)]
  (+ a b c))
```

Free and Bound Identifiers

A use of a function argument or a local identifier is a **bound occurrence**.

```
(define (f x y) (+ x y z))
```

```
(local [(define a 3)
        (define c 4)]
  (+ a b c))
```

Free and Bound Identifiers

A use of an identifier that is not a function argument or a local identifier is a **free identifier**.

```
(define (f x y) (+ x y z))
```

```
(local [(define a 3)
        (define c 4)]
  (+ a b c))
```

Shadowing

Shadowing happens when a binding occurrence of an identifier occurs in a context where that identifier is already bound (i.e., there was a prior binding occurrence).

```
(define (f x y)
  (local [(define x 3)]
    (+ x y)))
```


Shadowing

Shadowing happens when a binding occurrence of an identifier occurs in a context where that identifier is already bound (i.e., there was a prior binding occurrence).

```
(define (f x y)
  (local [(define x 3)]
    (+ z y)))
```

This is **still** an example of shadowing; two binding occurrences for **x** even though **x** is not used.

Shadowing

Shadowing happens when a binding occurrence of an identifier occurs in a context where that identifier is already bound (i.e., there was a prior binding occurrence).

```
(+ (local [(define x 3)] x)
   (local [(define x 4)] x))
```

This is **not** an example of shadowing; the two binding occurrences have non-overlapping scopes.

Homework 2

- Out now
- Your job will be to write functions to distinguish between the different kinds of identifiers

Arithmetic Language

```
<AE> ::= <num>
      | { + <AE> <AE> }
      | { - <AE> <AE> }
```

```
(define-type AE
  [num (n number?)]
  [add (lhs AE?)
       (rhs AE?)]
  [sub (lhs AE?)
       (rhs AE?)])
```



Arithmetic Language

```
<AE> ::= <num>
      | {+ <AE> <AE>}
      |- {- <AE> <AE>}
```

```
; interp : AE? -> number?
(define (interp an-ae)
  (type-case AE an-ae
    [num (n) n]
    [add (l r) (+ (interp l) (interp r))]
    [sub (l r) (- (interp l) (interp r))]))
```

No identifiers to help us study binding...

With Arithmetic Language

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

```
{with {x {+ 1 2}}
  {+ x x}} ⇒ 6
```

With Arithmetic Language

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



NEW

NEW

x \Rightarrow *error: free identifier*

With Arithmetic Language



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

```
{+ {with {x {+ 1 2}}
    {+ x x}}
  {with {x {- 4 3}}
    {+ x x}}}} ⇒ 8
```


With Arithmetic Language

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



 

```
{+ {with {x {+ 1 2}}
    {+ x x}}
  {with {y {- 4 3}}
    {+ y y}}}
```

\Rightarrow 8

With Arithmetic Language



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

```
{with {x {+ 1 2}}
  {with {x {- 4 3}}
    {+ x x}}}} ⇒ 2
```

With Arithmetic Language

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

```
{with {x {+ 1 2}}
  {with {y {- 4 3}}
    {+ x x}}}
```

\Rightarrow 6

With Arithmetic Language

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



NEW

NEW

```
(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?)])
```

With Arithmetic Language



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



```
; interp : WAE? -> number?
(define (interp a-wae)
  (type-case WAE a-wae
    [num (n) n]
    [add (l r) (+ (interp l) (interp r))]
    [sub (l r) (- (interp l) (interp r))]
    [with (name named-expr body)
          ...]
    [id (name)
         ...])))
```

With Arithmetic Language



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

```
; interp : WAE? -> number?
(define (interp a-wae)
  (type-case WAE a-wae
    [num (n) n]
    [add (l r) (+ (interp l) (interp r))]
    [sub (l r) (- (interp l) (interp r))]
    [with (name named-expr body)
          ...]
    [id (name)
      (error 'interp "free identifier")]))
```

With Arithmetic Language

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

```
; interp : WAE? -> number?
(define (interp a-wae)
  (type-case WAE a-wae
    [num (n) n]
    [add (l r) (+ (interp l) (interp r))]
    [sub (l r) (- (interp l) (interp r))]
    [with (name named-expr body)
          ... (interp named-expr) ... ]
    [id (name)
         (error 'interp "free identifier")]))
```

With Arithmetic Language



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

NEW

NEW

```
; interp : WAE? -> number?
(define (interp a-wae)
  (type-case WAE a-wae
    [num (n) n]
    [add (l r) (+ (interp l) (interp r))]
    [sub (l r) (- (interp l) (interp r))]
    [with (name named-expr body)
          ... (interp named-expr)
          ... (interp body) ... ]
    [id (name)
         (error 'interp "free identifier")]))
```


With Arithmetic Language

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

```
; interp : WAE? -> number?
(define (interp a-wae)
  (type-case WAE a-wae
    [num (n) n]
    [add (l r) (+ (interp l) (interp r))]
    [sub (l r) (- (interp l) (interp r))]
    [with (name named-expr body)
           (interp (subst body name
                           (interp named-expr)))]
    [id (name)
         (error 'interp "free identifier")]))
```

Substitution

```
; subst : WAE? symbol? number? -> WAE?  
(define (subst a-wae sub-id val)  
  (type-case WAE a-wae  
    [num (n) ...]  
    [add (l r) ...]  
    [sub (l r) ...]  
    [with (name named-expr body)  
          ...]  
    [id (name) ...])))
```

Let's make examples/tests first...

Example Substitutions

```
; 10 for x in {+ 1 x} ⇒ {+ 1 10}
(test (subst (add (num 1) (id 'x)) 'x 10)
      (add (num 1) (num 10)))
```

```
; 10 for x in x ⇒ 10
(test (subst (id 'x) 'x 10)
      (num 10))
```

```
; 10 for x in y ⇒ y
(test (subst (id 'y) 'x 10)
      (id 'y))
```

```
; 10 for y in {- x 1} ⇒ {- x 1}
(test (subst (sub (id 'x) (num 1)) 'y 10)
      (sub (id 'x) (num 1)))
```

Substitution

```
; subst : WAE? symbol? number? -> WAE?  
(define (subst a-wae sub-id val)  
  (type-case WAE a-wae  
    [num (n) a-wae]  
    [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 (name named-expr body)  
          ...]  
    [id (name) (if (symbol=? name sub-id)  
                  (num val)  
                  a-wae) ]))
```

Example Substitutions

```
; 10 for x in {with {y 17} x} ⇒ {with {y 17} 10}
(test (subst (with 'y (num 17) (id 'x)) 'x 10)
      (with 'y (num 17) (num 10)))
```

```
; 10 for x in {with {y x} y} ⇒ {with {y 10} y}
(test (subst (with 'y (id 'x) (id 'y)) 'x 10)
      (with 'y (num 10) (id 'y)))
```

```
; 10 for x in {with {x y} x} ⇒ {with {x y} x}
(test (subst (with 'x (id 'y) (id 'x)) 'x 10)
      (with 'x (id 'y) (id 'x)))
```

Substitution

Substitution replaces

- free identifiers with the same name
- no binding identifiers
- no bound identifiers

An identifier is bound when it appears in the body of a **with** binding the same name

Conversely, a free identifier with a given name appears in a **with** only if the **with** doesn't bind the name

Substitution

```
; subst : WAE? symbol? number? -> WAE?  
(define (subst a-wae sub-id val)  
  (type-case WAE a-wae  
    ...  
    [with (name named-expr body)  
          (with name  
              (subst named-expr sub-id val)  
              (if (symbol=? name sub-id)  
                  body  
                  (subst body sub-id val))))]  
    ...))
```

The Bigger Picture

- This scoping mechanism is called **lexical scope**
 - I.e., what binding is used can be determined lexically
 - I.e., just by looking at the code
 - I.e., no need to execute the program
- Used almost universally by programming languages, with few exceptions
 - Emacs Lisp, LaTeX (dynamic scope)
 - Python, old JavaScript (almost lexical scope, but some issues)
- Note: not tied to substitution!
 - That just happens to be how we implemented it
 - We'll see another way next week