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

```
(f 10)  
(f 10)
```

no

not a use of the argument anymore
The Arbitrariness of Identifiers

Are the following two programs equivalent?

\[
\begin{align*}
\text{(define } (f \ x) \ (+ \ y \ 1)) \\
(f \ 10)
\end{align*}
\]

\[
\begin{align*}
\text{(define } (f \ z) \ (+ \ y \ 1)) \\
(f \ 10)
\end{align*}
\]

\text{yes}

argument never used, so almost any name is ok
The Arbitrariness of Identifiers

Are the following two programs equivalent?

\[
\begin{align*}
&\text{(define (f x) (+ y 1))} \\
&(f 10)
\end{align*}
\]

\[
\begin{align*}
&\text{(define (f y) (+ y 1))} \\
&(f 10)
\end{align*}
\]

no

now a use of the argument
The Arbitrariness of Identifiers

Are the following two programs equivalent?

\[
\begin{align*}
(\text{define } (f \ x) ( + \ y \ 1)) & \quad \quad (\text{define } (f \ x) ( + \ z \ 1)) \\
(f \ 10) & \quad \quad (f \ 10)
\end{align*}
\]

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

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

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

\[
(+ \ (\text{local} \ [(\text{define} \ x \ 3)] \ x) \\
(\text{local} \ [(\text{define} \ x \ 4)] \ x))
\]

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

\[
\langle AE \rangle \ ::= \ \langle \text{num} \rangle \\
| \ {+ \ \langle AE \rangle \ \langle AE \rangle} \\
| \ {- \ \langle AE \rangle \ \langle AE \rangle}
\]

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

\[
\text{<AE>} ::= \text{<num>}
\]
\[
| \{ + \text{<AE>} \text{<AE>} \}
\]
\[
| \{ - \text{<AE>} \text{<AE>} \}
\]

; 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

\[<\text{WAE}> ::= <\text{num}>\]
\[
\mid \{+ <\text{WAE}> <\text{WAE}>\}\]
\[
\mid \{- <\text{WAE}> <\text{WAE}>\}\]
\[
\mid \{\text{with} \ {\langle} <\text{id}> <\text{WAE}> \rangle <\text{WAE}>\}\]
\[
\mid <\text{id}>\]
\]

\{\text{with} \ {\langle} x \ {\langle} + 1 2 \rangle \rangle \}
\{+ x x\} \implies 6
With Arithmetic Language

\[<\text{WAE}> ::= \langle\text{num}\rangle\]
\[\lor \quad \{+ \quad <\text{WAE}> \quad <\text{WAE}>\}\]
\[\lor \quad \{- \quad <\text{WAE}> \quad <\text{WAE}>\}\]
\[\lor \quad \{\text{with} \quad \{\langle\text{id}\rangle \quad <\text{WAE}>\} \quad <\text{WAE}>\}\]
\[\lor \quad \langle\text{id}\rangle\]

\[x \quad \Rightarrow \quad \text{error: free identifier}\]
With Arithmetic Language

\[
\begin{align*}
\langle \text{WAE} \rangle & := \phantom{\text{ NEW }} \langle \text{num} \rangle \\
& \quad \mid \{+ \langle \text{WAE} \rangle \langle \text{WAE} \rangle \} \\
& \quad \mid \{- \langle \text{WAE} \rangle \langle \text{WAE} \rangle \} \\
& \quad \mid \{\text{with} \{\langle \text{id} \rangle \langle \text{WAE} \rangle\} \langle \text{WAE} \rangle\} \phantom{\text{ NEW}} \\
& \quad \mid \langle \text{id} \rangle \\
\end{align*}
\]

\[
\begin{align*}
\{+ \{\text{with} \{x \{+ 1 2\}\}\} \\
& \quad \{+ x x\}\} \\
& \quad \{\text{with} \{x \{- 4 3\}\}\} \\
& \quad \{+ x x\}\}\} \Rightarrow 8
\end{align*}
\]
With Arithmetic Language

\[<\text{WAE}> ::= \text{<num>}
\]

| \{+ <\text{WAE}> <\text{WAE}>\}
| \{- <\text{WAE}> <\text{WAE}>\}
| \{\text{with} \{<\text{id}> <\text{WAE}>\} <\text{WAE}>\}
| \{<\text{id}>\}

\{+ \{\text{with} \{x \{+ 1 2\}\}
\{+ x x\}\}
\{\text{with} \{y \{- 4 3\}\}
\{+ y y\}\}\}\} \Rightarrow 8
With Arithmetic Language

\[
\text{<WAE>} ::= \text{<num>}
\]

\[
| \{+ \text{<WAE>} \text{<WAE>}\}
\]

\[
| \{- \text{<WAE>} \text{<WAE>}\}
\]

\[
| \{\text{with} \{\text{id}\} \text{<WAE>}\} \text{<WAE>}\}
\]

\[
| \text{id}
\]

{\text{with} \{x \{+ 1 2\}\}

{\text{with} \{x \{- 4 3\}\}

{\{+ x x\}\}}}

\Rightarrow 2
With Arithmetic Language

\[
<\text{WAE}> ::= \text<num>
| \{+ <\text{WAE}> <\text{WAE}>\}
| \{- <\text{WAE}> <\text{WAE}>\}
| \{\text{with} \{<\text{id}> <\text{WAE}>\} <\text{WAE}>\}
| <\text{id}>
\]

\[
\{\text{with} \{x \{+ 1 2\}\}
\{\text{with} \{y \{- 4 3\}\}
\{+ x x\}\}\} \Rightarrow 6
\]
With Arithmetic Language

\[
\text{<WAE>} ::= \text{<num>}
| \{+ \text{<WAE>} \text{<WAE>}\}
| \{- \text{<WAE>} \text{<WAE>}\}
| \{\text{with} \{\text{id} \text{<WAE>}\} \text{<WAE>}\}
| \text{id}
\]

(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

\[
\text{<WAE>} ::= \text{<num>}
\]

\[
| \{+ \text{<WAE> <WAE>}\}
\]

\[
| \{- \text{<WAE> <WAE>}\}
\]

\[
| \text{with} \{\langle\text{id} \text{<WAE>}\rangle \text{<WAE>}\}
\]

\[
| \langle\text{id}\rangle
\]

; 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

\[
\langle \text{WAE} \rangle ::= \langle \text{num} \rangle \\
| \{+ \langle \text{WAE} \rangle \langle \text{WAE} \rangle \} \\
| \{- \langle \text{WAE} \rangle \langle \text{WAE} \rangle \} \\
| \{\text{with} \ \{\langle \text{id} \rangle \langle \text{WAE} \rangle\} \langle \text{WAE} \rangle\} \\
| \langle \text{id} \rangle
\]

; 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 variable")]])
With Arithmetic Language

\[
\begin{align*}
\text{<WAE>} & \ := \ \text{<num>} \\
& \ | \ \{ + \ \text{<WAE>} \ \text{<WAE>} \} \\
& \ | \ \{ - \ \text{<WAE>} \ \text{<WAE>} \} \\
& \ | \ \{ \text{with} \ \{ \text{id} \ \text{<WAE>} \} \ \text{<WAE>} \} \\
& \ | \ \text{id}
\end{align*}
\]

; 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 variable")]))


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)
      ... (interp body) ...]
    [id (name)
      (error 'interp "free variable")]])
With Arithmetic Language

\[<\text{WAE}> ::= \begin{align*}
\text{<num>} & \mid 
\{ + \ <\text{WAE}> \ <\text{WAE}> \} \\
\{ - \ <\text{WAE}> \ <\text{WAE}> \} & \mid 
\{\text{with} \ \{ \ <\text{id}> \ <\text{WAE}> \} \ <\text{WAE}> \} \ \\
\text{id} & \end{align*} \]

; 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 variable")]))
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)))
; 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 \texttt{with} binding the same name

Conversely, a free variable of a name appears in a \texttt{with} only if the \texttt{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**
  - l.e., what binding is used can be determined lexically
  - l.e., just by looking at the code
  - l.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