

Part I: Introduction

Marco T. Morazán

Seton Hall University

Outline

- 1 Introduction
- 2 Local Variables
- 3 Scoping
- 4 Lexical Analysis

Recursively Specified Data

Introduction

Local Variables

Scoping

Lexical Analysis

- When writing a function we must know precisely what kinds of values may occur as arguments

Recursively Specified Data

Introduction

Local Variables

Scoping

Lexical Analysis

- When writing a function we must know precisely what kinds of values may occur as arguments
- Data of arbitrary size requires an inductive data definition
- To specify a set, S , inductively, define it as the smallest set satisfying two properties:
 - Some specific values are in S
 - If certain values are in S , then certain other values are in S

Recursively Specified Data

Introduction

Local Variables

Scoping

Lexical Analysis

- When writing a function we must know precisely what kinds of values may occur as arguments
- Data of arbitrary size requires an inductive data definition
- To specify a set, S , inductively, define it as the smallest set satisfying two properties:
 - Some specific values are in S
 - If certain values are in S , then certain other values are in S
- - $0 \in S$
 - $x+3 \in S$, where $x \in S$
- What has been defined?

Introduction

- Data of arbitrary size can also be defined using inference rules

Introduction

Introduction

Local Variables

Scoping

Lexical Analysis

- Data of arbitrary size can also be defined using inference rules
- - $\overline{0 \in S}$
 - $\frac{x \in S}{x+3 \in S}$

Introduction

Introduction

Local Variables

Scoping

Lexical Analysis

- Data of arbitrary size can also be defined using inference rules
- - $\overline{0 \in S}$
 - $\frac{x \in S}{x+3 \in S}$
- Define a list of numbers

Introduction

Local Variables

Scoping

Lexical Analysis

- Data of arbitrary size can also be defined using inference rules

- - $\overline{0 \in S}$
 - $\frac{x \in S}{x+3 \in S}$
- Define a list of numbers
- - $\overline{() \in (\text{listofnumber})}$
 - $\frac{L \in (\text{listofnumber})}{(\text{consnumber}L) \in S}$

Introduction

Introduction

Local Variables

Scoping

Lexical Analysis

- A third way to specify data is using BNF (Backus Naur Form)
- Grammar rules used to specify programming languages

Introduction

Introduction

Local Variables

Scoping

Lexical Analysis

- A third way to specify data is using BNF (Backus Naur Form)
- Grammar rules used to specify programming languages
- Elements of rules

Terminal symbols Characters in the external representation

Nonterminal Symbols Names of sets being defined usually in angle brackets (syntactic categories)

Productions Rules with a RHS and a LHS

- $LHS ::= RHS$
- Read as: LHS can be/is RHS

Introduction

- List of numbers
 - $\langle \text{lon} \rangle ::= ' ()$
 - $\langle \text{lon} \rangle ::= (\text{cons } \langle \text{number} \rangle \langle \text{lon} \rangle)$

Introduction

Local Variables

Scoping

Lexical Analysis

- List of numbers
 - $\langle \text{lon} \rangle ::= ' ()$
 - $\langle \text{lon} \rangle ::= (\text{cons } \langle \text{number} \rangle \langle \text{lon} \rangle)$
- Syntactic Derivation: (2 4)
 - $\langle \text{lon} \rangle ::= (\langle \text{number} \rangle \langle \text{list-of-numbers} \rangle)$
 - $::= (2 \langle \text{list-of-numbers} \rangle)$
 - $::= (2 \langle \text{number} \rangle \langle \text{list-of-numbers} \rangle)$
 - $::= (2 \text{ 4 } \langle \text{list-of-numbers} \rangle)$
 - $::= (2 \text{ 4})$
- Order of substitutions does not matter

Introduction

Introduction

Local Variables

Scoping

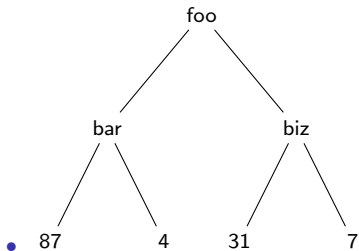
Lexical Analysis

- Symbolic Lists
- - $\langle \text{slist} \rangle ::= (\langle \text{sexp} \rangle^*)$
 - $\langle \text{sexp} \rangle ::= \langle \text{symbol} \rangle \mid \langle \text{slist} \rangle$

- Symbolic Lists
 - $\langle \text{slist} \rangle ::= (\langle \text{sexp} \rangle^*)$
 - $\langle \text{sexp} \rangle ::= \langle \text{symbol} \rangle \mid \langle \text{slist} \rangle$
 - Derivation of (a (b c) d)
 - $\langle \text{slist} \rangle ::= (\langle \text{sexp} \rangle^*)$
 - $::= (\langle \text{sexp} \rangle \langle \text{sexp} \rangle \langle \text{sexp} \rangle)$
 - $::= (a \langle \text{sexp} \rangle \langle \text{sexp} \rangle)$
 - $::= (a \langle \text{sexp} \rangle d)$
 - $::= (a \langle \text{slist} \rangle d)$
 - $::= (a (\langle \text{sexp} \rangle^*) d)$
 - $::= (a (\langle \text{sexp} \rangle \langle \text{sexp} \rangle) d)$
 - $::= (a (b \langle \text{sexp} \rangle) d)$
 - $::= (a (b c) d)$

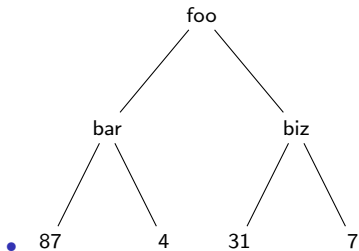
Introduction

- Define a binary tree with numeric leaves



Introduction

- Define a binary tree with numeric leaves



- - $\langle \text{bintree} \rangle ::= \langle \text{number} \rangle$
 - $\langle \text{bintree} \rangle ::= (\langle \text{symbol} \rangle \langle \text{bintree} \rangle \langle \text{bintree} \rangle)$

Introduction

- The Lambda Calculus
- A simple language used to study the theory of programming languages
- Contains:
 - variable references
 - lambda expressions with a single parameter
 - application expressions

Introduction

- The Lambda Calculus
- A simple language used to study the theory of programming languages
- Contains:
 - variable references
 - lambda expressions with a single parameter
 - application expressions
- - $\langle \text{exp} \rangle ::= \langle \text{id} \rangle$
 - $::= (\text{lambda } (\langle \text{id} \rangle) \langle \text{exp} \rangle)$
 - $::= (\langle \text{exp} \rangle \langle \text{sexp} \rangle)$

Introduction

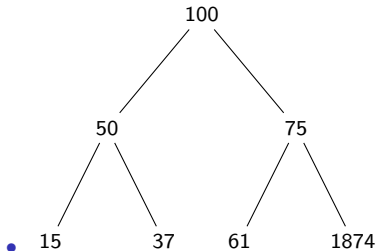
- HOMEWORK: 1.1, 1.3

Introduction

- BNF grammars are also known as context-free grammars
- Rules may be used regardless of the context in any order

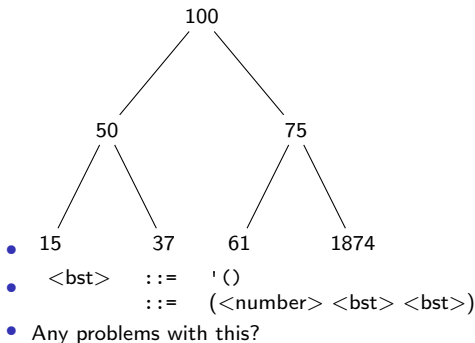
Introduction

- BNF grammars are also known as context-free grammars
- Rules may be used regardless of the context in any order
- Not all sets are context-free
- Consider defining the set of BSTs of numbers



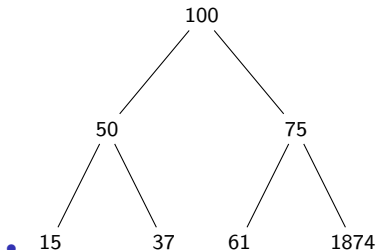
Introduction

- BNF grammars are also known as context-free grammars
- Rules may be used regardless of the context in any order
- Not all sets are context-free
- Consider defining the set of BSTs of numbers



Introduction

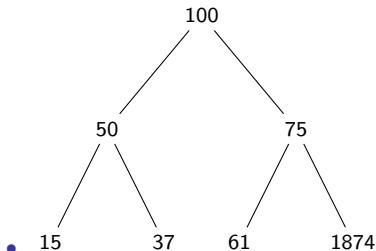
- BNF grammars are also known as context-free grammars
- Rules may be used regardless of the context in any order
- Not all sets are context-free
- Consider defining the set of BSTs of numbers



- $\langle \text{bst} \rangle ::= '()$
- $\quad ::= (\langle \text{number} \rangle \langle \text{bst} \rangle \langle \text{bst} \rangle)$
- Any problems with this?
- Does not capture: everything in the LST must be \leq to the root number and everything in the RST
- Such constraints are context-sensitive: valid LSTs and RSTs depend on the root value

Introduction

- BNF grammars are also known as context-free grammars
- Rules may be used regardless of the context in any order
- Not all sets are context-free
- Consider defining the set of BSTs of numbers



- $\langle \text{bst} \rangle ::= '()$
- $\quad ::= (\langle \text{number} \rangle \langle \text{bst} \rangle \langle \text{bst} \rangle)$
- Any problems with this?
- Does not capture: everything in the LST must be \leq to the root number and everything in the RST
- Such constraints are context-sensitive: valid LSTs and RSTs depend on the root value
- Arise in some PLs: variables must be declared before using them

Introduction

- Proving properties of recursive data is done using induction

Introduction

Local
Variables

Scoping

Lexical
Analysis

Introduction

- Proving properties of recursive data is done using induction
- Prove that $T \in \text{bintree} \Rightarrow T$ has an odd number of nodes

Introduction

Local
Variables

Scoping

Lexical
Analysis

Introduction

Introduction

Local Variables

Scoping

Lexical Analysis

- Proving properties of recursive data is done using induction
- Prove that $T \in \text{bintree} \Rightarrow T$ has an odd number of nodes
- Proof by induction on, h , the height of T
- Base case: $h = 0$
 - $h = 0 \Rightarrow T$ is a number
 - $\Rightarrow T$ has one node
 - $\Rightarrow T$ has an odd number of nodes

Introduction

Introduction

Local
Variables

Scoping

Lexical
Analysis

- Proving properties of recursive data is done using induction
- Prove that $T \in \text{bintree} \Rightarrow T$ has an odd number of nodes
- Proof by induction on, h , the height of T
- Base case: $h = 0$

$h = 0 \Rightarrow T$ is a number
 $\Rightarrow T$ has one node
 $\Rightarrow T$ has an odd number of nodes

- Inductive case

Assume: T has an odd number for nodes for $h = k$

Prove: T has an odd number for nodes for $h = k+1$

$h = k+1 \Rightarrow T$ is not a number
 $\Rightarrow T$ has a root number and two sub-bintrees: lst and rst

Both lst and rst have height at most k
 \Rightarrow each has an odd number of nodes (by IH)
 $\Rightarrow \text{lst}$ has $2*i+1$ nodes and rhs has $2*j+1$ nodes
 $\Rightarrow T$ has $2*i+1 + 2*j+1 + 1$ nodes $= 2(i+j)+3$ nodes
 $\Rightarrow T$ has an odd number of nodes \square

Introduction

- Write a program to count the number of nodes in a bintree

Introduction

Local Variables

Scoping

Lexical Analysis

Introduction

Introduction

Local Variables

Scoping

Lexical Analysis

- Write a program to count the number of nodes in a bintree
- $\begin{array}{ll} \langle \text{bst} \rangle & ::= '()' \\ & ::= (\langle \text{number} \rangle \langle \text{bst} \rangle \langle \text{bst} \rangle) \end{array}$

Introduction

Introduction

Local
Variables

Scoping

Lexical
Analysis

- Write a program to count the number of nodes in a bintree
- ```
<bst> ::= '()'
 ::= (<number> <bst> <bst>)
```
- ```
#lang eopl
(require rackunit)
#|
bintree ::= number
          ::= (number bintree bintree)

|#
```


Introduction

- Write a program to count the number of nodes in a bintree
- ```
<bst> ::= '()'
 ::= (<number> <bst> <bst>)
```
- ```
#lang eopl
(require rackunit)
#|
bintree ::= number
         ::= (number bintree bintree)

|#

;; bintree → odd-natnum
;; Purpose: Count the nodes the given bintree
(define (cnt-nodes s)
```

Introduction

- Write a program to count the number of nodes in a bintree

- ```
<bst> ::= '()'
 ::= (<number> <bst> <bst>)
```

- ```
#lang eopl
(require rackunit)
#|
bintree ::= number
         ::= (number bintree bintree)

|#

;; bintree → odd-natnum
;; Purpose: Count the nodes the given bintree
(define (cnt-nodes s)
```

- ```
(check-equal? (cnt-nodes 23) 1)
(check-equal? (cnt-nodes (list 45 88 6561)) 3)
(check-equal? (cnt-nodes (list 45 (list 88 11 99)
 (list 6561 -6 42)))
```

7)

# Introduction

- Write a program to count the number of nodes in a bintree
- ```
<bst> ::= '()'
      ::= (<number> <bst> <bst>)
```
- ```
#lang eopl
(require rackunit)
#|
bintree ::= number
 ::= (number bintree bintree)

|#

;; bintree → odd-natnum
;; Purpose: Count the nodes the given bintree
(define (cnt-nodes s)

 (if (number? s)
 1

 (check-equal? (cnt-nodes 23) 1)
 (check-equal? (cnt-nodes (list 45 88 6561)) 3)
 (check-equal? (cnt-nodes (list 45 (list 88 11 99)
 (list 6561 -6 42)))
 7)

)
)
```

# Introduction

## Introduction

### Local Variables

### Scoping

### Lexical Analysis

- Write a program to count the number of nodes in a bintree
- ```
<bst> ::= '()'
      ::= (<number> <bst> <bst>)
```
- ```
#lang eopl
(require rackunit)
#|
bintree ::= number
 ::= (number bintree bintree)

|#

;; bintree → odd-natnum
;; Purpose: Count the nodes the given bintree
(define (cnt-nodes s)

 (if (number? s)
 1
 (+ 1 (cnt-nodes (cadr s)) (cnt-nodes (caddr s)))))

(check-equal? (cnt-nodes 23) 1)
(check-equal? (cnt-nodes (list 45 88 6561)) 3)
(check-equal? (cnt-nodes (list 45 (list 88 11 99)
 (list 6561 -6 42)))
 7)
```

# Introduction

- **Follow the Grammar**
- At least one function for each syntactic category (i.e., nonterminal)
- Processing an instance of a syntactic category is done by calling a function to process instances of the syntactic category

# Introduction

- Write a predicate of a lon
- - $\langle \text{lon} \rangle ::= ' ()$
  - $\langle \text{lon} \rangle ::= (\text{cons } \langle \text{number} \rangle \langle \text{lon} \rangle)$

# Introduction

- Write a predicate of a lon
- - `<lon> ::= '()`
  - `<lon> ::= (cons <number> <lon>)`
- `#lang eopl`  
`(require rackunit)`  
  
`#|`  
`lon ::= ()`  
`::= (number lon)`  
`|#`

# Introduction

- Write a predicate of a lon
  - $\langle \text{lon} \rangle ::= ' ()$
  - $\langle \text{lon} \rangle ::= (\text{cons } \langle \text{number} \rangle \langle \text{lon} \rangle)$
- `#lang eopl`  
`(require rackunit)`  
  
`#|`  
`lon ::= ()`  
`::= (number lon)`  
`|#`
- `;; (listof X) → Boolean`  
`;; Purpose: Determine if the given input is a lon`  
`(define (lon? l)`



# Introduction

- Write a predicate of a lon
  - `<lon> ::= '()`
    - `<lon> ::= (cons <number> <lon>)`
- `#lang eopl`  
`(require rackunit)`  
  
`#|`  
`lon ::= ()`  
`::= (number lon)`  
`|#`
- `;; (listof X) → Boolean`  
`;; Purpose: Determine if the given input is a lon`  
`(define (lon? l)`
- `(check-equal? (lon? '(a b c d)) #f)`  
`(check-equal? (lon? '()) #t)`  
`(check-equal? (lon? '(1 2 3)) #t)`

# Introduction

- Write a predicate of a lon
  - `<lon> ::= ' ( )`
  - `<lon> ::= (cons <number> <lon>)`
- `#lang eopl`  
`(require rackunit)`  
  
`#|`  
`lon ::= (`  
`::= (number lon)`  
`|#`
- `;; (listof X) → Boolean`  
`;; Purpose: Determine if the given input is a lon`  
`(define (lon? l)`
  - `(if (null? l)`  
`#t`
- `(check-equal? (lon? '(a b c d)) #f)`  
`(check-equal? (lon? '()) #t)`  
`(check-equal? (lon? '(1 2 3)) #t)`

# Introduction

- Write a predicate of a lon
  - `<lon> ::= ' ()`
  - `<lon> ::= (cons <number> <lon>)`
- `#lang eopl`  
`(require rackunit)`  
  
`#|`  
`lon ::= ()`  
`::= (number lon)`  
`|#`
- `;; (listof X) → Boolean`  
`;; Purpose: Determine if the given input is a lon`  
`(define (lon? l)`
  - `(if (null? l)`  
`#t`
  - `(and (number? (car l)) (lon? (cdr l))))`
- `(check-equal? (lon? '(a b c d)) #f)`  
`(check-equal? (lon? '()) #t)`  
`(check-equal? (lon? '(1 2 3)) #t)`

# Introduction

## Introduction

### Local Variables

### Scoping

### Lexical Analysis

- ```
;; (listof X) → Boolean
;; Purpose: Determine if the given list is a lon
(define (lon? l)
  (if (null? l)
      #t
      (and (number? (car l)) (lon? (cdr l))))))
```

Introduction

Introduction

Local Variables

Scoping

Lexical Analysis

- ```
;; (listof X) → Boolean
;; Purpose: Determine if the given list is a lon
(define (lon? l)
 (if (null? l)
 #t
 (and (number? (car l)) (lon? (cdr l)))))
```
- Prove that the function is correct

## Introduction

### Local Variables

### Scoping

### Lexical Analysis

- ```
;; (listof X) → Boolean
;; Purpose: Determine if the given list is a lon
(define (lon? l)
  (if (null? l)
      #t
      (and (number? (car l)) (lon? (cdr l))))))
```
- Prove that the function is correct
- Proof by induction on, k , $|l|$

Introduction

Introduction

Local Variables

Scoping

Lexical Analysis

- `;; (listof X) → Boolean`
`;; Purpose: Determine if the given list is a lon`
`(define (lon? l)`
 `(if (null? l)`
 `#t`
 `(and (number? (car l)) (lon? (cdr l))))`
- Prove that the function is correct
- Proof by induction on, k , $||$
- Base case: $k = 0$
 $k = 0 \Rightarrow l$ is null
 \Rightarrow `(lon? l)` returns `#t`,
 which correct because l is a lon with 0 numbers

Introduction

- ```
;; (listof X) → Boolean
;; Purpose: Determine if the given list is a lon
(define (lon? l)
 (if (null? l)
 #t
 (and (number? (car l)) (lon? (cdr l)))))
```

- Inductive Step

Assume:  $(\text{lon? } l)$  works for  $|l|=k$

Show:  $(\text{lon? } l)$  works for  $|l|=k+1$

$|l|=k+1$

$\Rightarrow l = (\text{cons } X (\text{listof } X))$

$\Rightarrow (\text{lon? } (\text{cdr } l))$  returns  $\#t$  if  $(\text{cdr } l)$  is a lon and  $\#f$  otherwise, by IH

$(\text{lon? } (\text{cdr } l)) = \#t$

$\Rightarrow (\text{lon? } l)$  returns  $(\text{and } (\text{number? } (\text{car } l)) \#t)$

$\Rightarrow (\text{lon? } l)$  returns  $\#t$  when  $(\text{number? } (\text{car } l))$  holds, which is correct because  $l$  is a lon

$\wedge \#f$  when  $(\text{not } (\text{number? } (\text{car } l)))$

holds, which is correct as  $l$  is not a lon □



# Introduction

## Introduction

### Local Variables

### Scoping

### Lexical Analysis

- •  $\langle \text{listof } X \rangle ::= '() \mid (\text{cons } X (\text{listof } X))$
- Write a function to extract the  $n^{\text{th}}$  element

# Introduction

## Introduction

### Local Variables

### Scoping

### Lexical Analysis

- $\langle (\text{listof } X) \rangle ::= '() \mid (\text{cons } X (\text{listof } X))$

- Write a function to extract the  $n^{\text{th}}$  element

- `#lang eopl`  
`(require rackunit)`

```
;; (listof X) ::= '() | (cons X (listof X))
```

```
;; (listof X) natnum \rightarrow X
```

```
;; Purpose: Extract the nth element of the given list
```

```
(define (nthelem l n)
```

```
 (cond [(null? l)
```

```
 (eopl:error 'nthelem "List too short by ~s elems" (+ n 1))
```

```
 [(zero? n) (car l)]
```

```
 [else (nthelem (cdr l) (- n 1))])
```

```
(check-equal? (nthelem '(0 1 2 3) 3) 3)
```

```
(check-equal? (nthelem '(a b c d e f g h) 5) 'f)
```

# Introduction

- Write a function to substitute in a list all occurrences of a symbol with another symbol

# Introduction

- Write a function to substitute in an slist all occurrences of a symbol with another symbol

- ```
#| <slist> ::= (<s-exp>*)  
  <sexp> ::= <symbol> | <s-list>  
|#  
;; slist → slist  
;; Purpose: Substitute old with new in given slist  
(define (subst-slist old new an-slist)  
  (map (lambda (s) (subst-sexp old new s)) an-slist))
```

- ```
(check-equal? (subst-slist 'a 'b '()) '())
(check-equal? (subst-slist 'c 'c '(b c c c)) '(b c c c))
(check-equal? (subst-slist 'a 'z '(a (b x a) (f (g a a) b)))
 '(z (b x z) (f (g z z) b))))
```

# Introduction

- Write a function to substitute in an slist all occurrences of a symbol with another symbol
- ```
#| <slist> ::= (<s-exp>*)  
  <sexp> ::= <symbol> | <s-list>  
|#  
;; slist → slist  
;; Purpose: Substitute old with new in given slist  
(define (subst-slist old new an-slist)  
  (map (lambda (s) (subst-sexp old new s)) an-slist))
```
- ```
(define (subst-sexp old new a-sexp)
 (if (symbol? a-sexp)
 (if (eq? old a-sexp) new a-sexp)
 (subst-slist old new a-sexp)))
```
- ```
(check-equal? (subst-slist 'a 'b '()) '())  
(check-equal? (subst-slist 'c 'c '(b c c c)) '(b c c c))  
(check-equal? (subst-slist 'a 'z '(a (b x a) (f (g a a) b)))  
  '(z (b x z) (f (g z z) b))))
```
- ```
(check-equal? (subst-sexp 'a 'b 'a) 'b)
(check-equal? (subst-sexp 'b 'z 'c) 'c)
(check-equal? (subst-sexp 'i 'j '(u h (i (j h s (i i i) s) y)))
 '(u h (j (j h s (j j j) s) y)))
```

# Local Variables

Introduction

Local  
Variables

Scoping

Lexical  
Analysis

- let-expressions allow you to define local variables
- ```
(let ((a 10)  
      (b 20)  
      (c 30))  
    (+ a b c))
```
- The value of a let-expression is the value of its body

Local Variables

- let-expressions allow you to define local variables
- ```
(let ((a 10)
 (b 20)
 (c 30))
 (+ a b c))
```
- The value of a let-expression is the value of its body
- The definitions are not mutually recursive
- What is the value of this expression?

```
(define x 10)
```

```
(let ((x 1)
 (y (+ x 1)))
 (+ x y))
```

# Local Variables

- let-expressions allow you to define local variables
- ```
(let ((a 10)
      (b 20)
      (c 30))
    (+ a b c))
```
- The value of a let-expression is the value of its body
- The definitions are not mutually recursive
- What is the value of this expression?

```
(define x 10)
```

```
(let ((x 1)
      (y (+ x 1)))
    (+ x y))
```

- $(+ 1 11) = 12$

Local Variables

- For mutually recursive definitions use a letrec-expression

Local Variables

- For mutually recursive definitions use a letrec-expression
- What is the value of this expression?

```
(define x 10)
```

```
(let ((x 1)  
      (y (+ x 1)))  
  (+ x y))
```

Local Variables

- For mutually recursive definitions use a letrec-expression
- What is the value of this expression?

```
(define x 10)
```

```
(let ((x 1)
      (y (+ x 1)))
  (+ x y))
```

- $(+ 1 2) = 3$

Local Variables

- HOMEWORK: 1.4, 1.5, 1.8, 1.11, 1.12, 1.13, 1.15, 1.16, 1.17, 1.24, 1.25, 1.27, 1.28, 1.29

Scoping

Introduction

Local
Variables

Scoping

Lexical
Analysis

- In most PLs variables must appear in two places:
 - declarations
 - references

Scoping

- In most PLs variables must appear in two places:
 - declarations
 - references
- In $(f \times y)$, f , x , and y are references

- In most PLs variables must appear in two places:
 - declarations
 - references
- In $(f \times y)$, f , x , and y are references
- In $(\text{lambda } (x) (\dots))$, x is a declaration

- In most PLs variables must appear in two places:
 - declarations
 - references
- In $(f \times y)$, f , x , and y are references
- In $(\text{lambda } (x) (\dots))$, x is a declaration
- The value of a variable is also called its *denotation*
- The denotation must come from some declaration and that bounds the declaration

- In most PLs variables must appear in two places:
 - declarations
 - references
- In $(f \times y)$, f , x , and y are references
- In $(\text{lambda } (x) (\dots))$, x is a declaration
- The value of a variable is also called its *denotation*
- The denotation must come from some declaration and that bounds the declaration
- In most PLs, variables have limited scope allowing for multiple uses of the same variable
- The scope of the variable is the part of the program where it is valid

Scoping

- In most PLs, the relationship between a variable reference and its declaration is static
- This relationship can be determined by analyzing the text of the program
- Languages with this property are *statically scoped*

- In most PLs, the relationship between a variable reference and its declaration is static
- This relationship can be determined by analyzing the text of the program
- Languages with this property are *statically scoped*
- In some PLs, the declaration to which a variable reference refers can not be determined until the program is executed
- These languages are *dynamically scoped*

- An extended lambda calculus:
 $\langle \text{exp} \rangle ::= \langle \text{number} \rangle$
 $::= \langle \text{Boolean} \rangle$
 $::= \langle \text{id} \rangle$
 $::= (\text{lambda } (\langle \text{id} \rangle^*) \langle \text{exp} \rangle)$
 $::= (\langle \text{exp} \rangle \langle \text{exp} \rangle^*)$

Scoping

- An extended lambda calculus:
 - $\langle \text{exp} \rangle ::= \langle \text{number} \rangle$
 - $::= \langle \text{Boolean} \rangle$
 - $::= \langle \text{id} \rangle$
 - $::= (\text{lambda } (\langle \text{id} \rangle^*) \langle \text{exp} \rangle)$
 - $::= (\langle \text{exp} \rangle \langle \text{exp} \rangle^*)$
- Binding rule for the extended lambda calculus
 - In $(\text{lambda } (\langle \text{id} \rangle^*) \langle \text{exp} \rangle)$, the $\langle \text{id} \rangle$ s are declarations
 - These declarations bind all occurrences of these variables in $\langle \text{exp} \rangle$ unless there are intervening declarations of any of the same variables
- In an expression, a variable *occurs bound* if it is referenced and it is bound by a declaration
- In an expression, a variable *occurs free* if it is referenced and it is not bound by a declaration

Scoping

- $((\text{lambda } (x \ z) \ x) \ y \ y)$
 - x occurs bound
 - y occurs free (twice)
 - z occurs neither bound nor free

Scoping

- $((\text{lambda } (x\ z)\ x)\ y\ y)$
 - x occurs bound
 - y occurs free (twice)
 - z occurs neither bound nor free
- $(\text{lambda } (y)\ ((\text{lambda } (x\ z)\ x)\ y\ y))$
 - x and y occur bound
 - z occurs neither bound nor free
- $(\text{lambda } (y)\ ((\text{lambda } (x\ z)\ x)\ y\ y))$: A function that takes one input and returns the value of its input

- $((\text{lambda } (x\ z)\ x)\ y\ y)$
 - x occurs bound
 - y occurs free (twice)
 - z occurs neither bound nor free
- $(\text{lambda } (y)\ ((\text{lambda } (x\ z)\ x)\ y\ y))$
 - x and y occur bound
 - z occurs neither bound nor free
- The meaning of expressions without free variable is fixed and are called *combinators*
- $(\text{lambda } (y)\ ((\text{lambda } (x\ z)\ x)\ y\ y))$: A function that takes one input and returns the value of its input

Scoping

- A variable x occurs free in an extended lambda calculus expression E iff:
 - E is a variable reference and E is the same as x
 - E is of the form $(\text{lambda } (y) E1)$, where $y \neq x$ and x occurs free in $E1$
 - E is of the form $(E1 E2)$ and x occurs free in $E1$ or $E2$

Scoping

- A variable x occurs free in an extended lambda calculus expression E iff:
 - E is a variable reference and E is the same as x
 - E is of the form $(\text{lambda } (y) E1)$, where $y \neq x$ and x occurs free in $E1$
 - E is of the form $(E1 E2)$ and x occurs free in $E1$ or $E2$
- We can write a function to determine if x occurs free in E :

```
#lang eopl
```

```
(require rackunit "../eopl-extras.rkt")
```

```
;; symbol exp  $\rightarrow$  Boolean
```

```
;; Purpose: Determine if the given variable occurs free in the given
```

```
(define (occurs-free? x exp)
```

- ```
(check-equal?(occurs-free? 'x 187) #f)
(check-equal?(occurs-free? 'a #t) #f)
(check-equal?(occurs-free? 'a 'b) #f)
(check-equal?(occurs-free? 'a '(lambda (a b) (f (g a b)))) #f)
(check-equal?(occurs-free? 'a '(lambda (a b) (f (g b b)))) #f)
(check-equal?(occurs-free? 'b '((lambda (a x) (times 2 x a)) b)) #t)
(check-equal?(occurs-free? 'a '(lambda (x b) (f (g a b x)))) #t)
(check-equal?(occurs-free? 'a '((lambda (z x) (times 2 x a)) b)) #t)
```

## Scoping

- A variable  $x$  occurs free in an extended lambda calculus expression  $E$  iff:
  - $E$  is a variable reference and  $E$  is the same as  $x$
  - $E$  is of the form  $(\text{lambda } (y) E1)$ , where  $y \neq x$  and  $x$  occurs free in  $E1$
  - $E$  is of the form  $(E1 E2)$  and  $x$  occurs free in  $E1$  or  $E2$

- We can write a function to determine if  $x$  occurs free in  $E$ :

```
#lang eopl
```

```
(require rackunit "../eopl-extras.rkt")
```

```
;; symbol exp → Boolean
```

```
;; Purpose: Determine if the given variable occurs free in the given
```

```
(define (occurs-free? x exp)
```

- ```
(cond [(or (number? exp) (boolean? exp)) #f]
```

- ```
(check-equal?(occurs-free? 'x 187) #f)
```

```
(check-equal?(occurs-free? 'a #t) #f)
```

```
(check-equal?(occurs-free? 'a 'b) #f)
```

```
(check-equal?(occurs-free? 'a '(lambda (a b) (f (g a b)))) #f)
```

```
(check-equal?(occurs-free? 'a '(lambda (a b) (f (g b b)))) #f)
```

```
(check-equal?(occurs-free? 'b '((lambda (a x) (times 2 x a)) b)) #t)
```

```
(check-equal?(occurs-free? 'a '(lambda (x b) (f (g a b x)))) #t)
```

```
(check-equal?(occurs-free? 'a '((lambda (z x) (times 2 x a)) b)) #t)
```

## Scoping

- A variable  $x$  occurs free in an extended lambda calculus expression  $E$  iff:
  - $E$  is a variable reference and  $E$  is the same as  $x$
  - $E$  is of the form  $(\text{lambda } (y) E1)$ , where  $y \neq x$  and  $x$  occurs free in  $E1$
  - $E$  is of the form  $(E1 E2)$  and  $x$  occurs free in  $E1$  or  $E2$

- We can write a function to determine if  $x$  occurs free in  $E$ :

```
#lang eopl
```

```
(require rackunit "../eopl-extras.rkt")
```

```
;; symbol exp → Boolean
```

```
;; Purpose: Determine if the given variable occurs free in the given
```

```
(define (occurs-free? x exp)
```

- ```
(cond [(or (number? exp) (boolean? exp)) #f]
```
- ```
 [(symbol? exp) (eqv? x exp)]
```

- ```
(check-equal?(occurs-free? 'x 187) #f)
(check-equal?(occurs-free? 'a #t) #f)
(check-equal?(occurs-free? 'a 'b) #f)
(check-equal?(occurs-free? 'a '(lambda (a b) (f (g a b)))) #f)
(check-equal?(occurs-free? 'a '(lambda (a b) (f (g b b)))) #f)
(check-equal?(occurs-free? 'b '((lambda (a x) (times 2 x a)) b)) #t)
(check-equal?(occurs-free? 'a '(lambda (x b) (f (g a b x)))) #t)
(check-equal?(occurs-free? 'a '((lambda (z x) (times 2 x a)) b)) #t)
```

Scoping

- A variable x occurs free in an extended lambda calculus expression E iff:

- E is a variable reference and E is the same as x
- E is of the form $(\text{lambda } (y) E1)$, where $y \neq x$ and x occurs free in $E1$
- E is of the form $(E1 E2)$ and x occurs free in $E1$ or $E2$

- We can write a function to determine if x occurs free in E :

```
#lang eopl
```

```
(require rackunit "../eopl-extras.rkt")
```

```
;; symbol exp → Boolean
```

```
;; Purpose: Determine if the given variable occurs free in the given
```

```
(define (occurs-free? x exp)
```

- (cond [(or (number? exp) (boolean? exp)) #f]

- [(symbol? exp) (eqv? x exp)]

- [(eqv? (car exp) 'lambda)
 (and (not (member x (cadr exp)))
 (occurs-free? x (caddr exp)))]

- (check-equal?(occurs-free? 'x 187) #f)

```
(check-equal?(occurs-free? 'a #t) #f)
```

```
(check-equal?(occurs-free? 'a 'b) #f)
```

```
(check-equal?(occurs-free? 'a '(lambda (a b) (f (g a b)))) #f)
```

```
(check-equal?(occurs-free? 'a '(lambda (a b) (f (g b b)))) #f)
```

```
(check-equal?(occurs-free? 'b '(((lambda (a x) (times 2 x a)) b)) #t)
```

```
(check-equal?(occurs-free? 'a '(lambda (x b) (f (g a b x)))) #t)
```

```
(check-equal?(occurs-free? 'a '(((lambda (z x) (times 2 x a)) b)) #t)
```

Scoping

- A variable x occurs free in an extended lambda calculus expression E iff:
 - E is a variable reference and E is the same as x
 - E is of the form $(\text{lambda } (y) E1)$, where $y \neq x$ and x occurs free in $E1$
 - E is of the form $(E1 E2)$ and x occurs free in $E1$ or $E2$

- We can write a function to determine if x occurs free in E :

```
#lang eopl
```

```
(require rackunit "../eopl-extras.rkt")
```

```
;; symbol exp → Boolean
```

```
;; Purpose: Determine if the given variable occurs free in the given
```

```
(define (occurs-free? x exp)
```

- ```
(cond [(or (number? exp) (boolean? exp)) #f]
```
- ```
      [(symbol? exp) (eqv? x exp)]
```
- ```
 [(eqv? (car exp) 'lambda)
```

```
 (and (not (member x (cadr exp)))
```

```
 (occurs-free? x (caddr exp))))
```
- ```
      [else (or (occurs-free? x (car exp))
```

```
                  (ormap (lambda (e) (occurs-free? x e)) (cdr exp))])
```
- ```
(check-equal?(occurs-free? 'x 187) #f)
```

```
(check-equal?(occurs-free? 'a #t) #f)
```

```
(check-equal?(occurs-free? 'a 'b) #f)
```

```
(check-equal?(occurs-free? 'a '(lambda (a b) (f (g a b)))) #f)
```

```
(check-equal?(occurs-free? 'a '(lambda (a b) (f (g b b)))) #f)
```

```
(check-equal?(occurs-free? 'b '((lambda (a x) (times 2 x a)) b)) #t)
```

```
(check-equal?(occurs-free? 'a '(lambda (x b) (f (g a b x)))) #t)
```

```
(check-equal?(occurs-free? 'a '((lambda (z x) (times 2 x a)) b)) #t)
```

# Scoping

- A variable  $x$  occurs bound in a lambda calculus expression  $E$  iff
  - $E$  is of the form  $(\text{lambda } (y) E1)$ , where  $x$  occurs bound in  $E1 \vee y = x \wedge y$  occurs free in  $E1$
  - $E$  is of the form  $(E1 E2)$  and  $x$  occurs bound in  $E1$  or  $E2$

# Scoping

- A variable  $x$  occurs bound in a lambda calculus expression  $E$  iff
  - $E$  is of the form  $(\text{lambda } (y) E1)$ , where  $x$  occurs bound in  $E1 \vee y = x \wedge y$  occurs free in  $E1$
  - $E$  is of the form  $(E1 E2)$  and  $x$  occurs bound in  $E1$  or  $E2$
- `;; symbol exp  $\rightarrow$  Boolean`  
`;; Purpose: Determine if given variable occurs bound in given exp`  
`(define (occurs-bound? x exp)`



## Scoping

- A variable  $x$  occurs bound in a lambda calculus expression  $E$  iff
  - $E$  is of the form  $(\text{lambda } (y) E1)$ , where  $x$  occurs bound in  $E1 \vee y = x \wedge y$  occurs free in  $E1$
  - $E$  is of the form  $(E1 E2)$  and  $x$  occurs bound in  $E1$  or  $E2$
- `;; symbol exp  $\rightarrow$  Boolean`  
`;; Purpose: Determine if given variable occurs bound in given exp`  
`(define (occurs-bound? x exp)`
  - `(check-equal? (occurs-bound? 'x 187) #f)`  
`(check-equal? (occurs-bound? 'a #t) #f)`  
`(check-equal? (occurs-bound? 'a 'b) #f)`  
`(check-equal? (occurs-bound? 'a '(lambda (a b) (f (g b b)))) #f)`  
`(check-equal? (occurs-bound? 'a '(lambda (x b) (f (g a b x)))) #f)`  
`(check-equal? (occurs-bound? 'b '((lambda (a x) (times 2 x a)) b))`  
`(check-equal? (occurs-bound? 'a '(lambda (a b) a)) #t)`  
`(check-equal? (occurs-bound? 'a '(lambda (a b) (f (g a b)))) #t)`  
`(check-equal? (occurs-bound? 'z '((lambda (z x) (times 2 x z)) b))`

## Scoping

- A variable  $x$  occurs bound in a lambda calculus expression  $E$  iff
  - $E$  is of the form  $(\text{lambda } (y) E1)$ , where  $x$  occurs bound in  $E1 \vee y = x \wedge y$  occurs free in  $E1$
  - $E$  is of the form  $(E1 E2)$  and  $x$  occurs bound in  $E1$  or  $E2$
- `;; symbol exp  $\rightarrow$  Boolean`  
`;; Purpose: Determine if given variable occurs bound in given exp`  
`(define (occurs-bound? x exp)`
  - `(cond [(or (number? exp) (boolean? exp) (symbol? exp)) #f]`
- `(check-equal? (occurs-bound? 'x 187) #f)`  
`(check-equal? (occurs-bound? 'a #t) #f)`  
`(check-equal? (occurs-bound? 'a 'b) #f)`  
`(check-equal? (occurs-bound? 'a '(lambda (a b) (f (g b b)))) #f)`  
`(check-equal? (occurs-bound? 'a '(lambda (x b) (f (g a b x)))) #f)`  
`(check-equal? (occurs-bound? 'b '((lambda (a x) (times 2 x a)) b))`  
`(check-equal? (occurs-bound? 'a '(lambda (a b) a)) #t)`  
`(check-equal? (occurs-bound? 'a '(lambda (a b) (f (g a b)))) #t)`  
`(check-equal? (occurs-bound? 'z '((lambda (z x) (times 2 x z)) b))`

## Scoping

- A variable  $x$  occurs bound in a lambda calculus expression  $E$  iff
  - $E$  is of the form  $(\text{lambda } (y) E1)$ , where  $x$  occurs bound in  $E1 \vee y = x \wedge y$  occurs free in  $E1$
  - $E$  is of the form  $(E1 E2)$  and  $x$  occurs bound in  $E1$  or  $E2$
- `;; symbol exp → Boolean`  
`;; Purpose: Determine if given variable occurs bound in given exp`  
`(define (occurs-bound? x exp)`
  - `(cond [(or (number? exp) (boolean? exp) (symbol? exp)) #f]`
  - `[(eqv? (car exp) 'lambda)`  
`(or (occurs-bound? x (caddr exp))`  
`(and (member x (cadr exp))`  
`(occurs-free? x (caddr exp))))]`
- `(check-equal? (occurs-bound? 'x 187) #f)`  
`(check-equal? (occurs-bound? 'a #t) #f)`  
`(check-equal? (occurs-bound? 'a 'b) #f)`  
`(check-equal? (occurs-bound? 'a '(lambda (a b) (f (g b b)))) #f)`  
`(check-equal? (occurs-bound? 'a '(lambda (x b) (f (g a b x)))) #f)`  
`(check-equal? (occurs-bound? 'b '((lambda (a x) (times 2 x a)) b))`  
`(check-equal? (occurs-bound? 'a '(lambda (a b) a)) #t)`  
`(check-equal? (occurs-bound? 'a '(lambda (a b) (f (g a b)))) #t)`  
`(check-equal? (occurs-bound? 'z '((lambda (z x) (times 2 x z)) b))`

## Scoping

- A variable  $x$  occurs bound in a lambda calculus expression  $E$  iff
  - $E$  is of the form  $(\text{lambda } (y) E1)$ , where  $x$  occurs bound in  $E1 \vee y = x \wedge y$  occurs free in  $E1$
  - $E$  is of the form  $(E1 E2)$  and  $x$  occurs bound in  $E1$  or  $E2$
- `;; symbol exp  $\rightarrow$  Boolean`  
`;; Purpose: Determine if given variable occurs bound in given exp`  
`(define (occurs-bound? x exp)`
  - `(cond [(or (number? exp) (boolean? exp) (symbol? exp)) #f]`
  - `[(eqv? (car exp) 'lambda)`  
`(or (occurs-bound? x (caddr exp))`  
`(and (member x (cadr exp))`  
`(occurs-free? x (caddr exp))))]`
  - `[else (or (occurs-bound? x (car exp))`  
`(occurs-bound? x (cadr exp))))]`
- `(check-equal? (occurs-bound? 'x 187) #f)`  
`(check-equal? (occurs-bound? 'a #t) #f)`  
`(check-equal? (occurs-bound? 'a 'b) #f)`  
`(check-equal? (occurs-bound? 'a '(lambda (a b) (f (g b b)))) #f)`  
`(check-equal? (occurs-bound? 'a '(lambda (x b) (f (g a b x)))) #f)`  
`(check-equal? (occurs-bound? 'b '((lambda (a x) (times 2 x a)) b))`  
`(check-equal? (occurs-bound? 'a '(lambda (a b) a)) #t)`  
`(check-equal? (occurs-bound? 'a '(lambda (a b) (f (g a b)))) #t)`  
`(check-equal? (occurs-bound? 'z '((lambda (z x) (times 2 x z)) b))`

# Lexical Analysis

- How is each variable reference associated with its declaration?

# Lexical Analysis

- How is each variable reference associated with its declaration?
- A PL associates a variable declaration with the program's region where it is valid

# Lexical Analysis

- How is each variable reference associated with its declaration?
- A PL associates a variable declaration with the program's region where it is valid
- Many PLs allow these regions to be nested (nested lambdas)

# Lexical Analysis

- How is each variable reference associated with its declaration?
- A PL associates a variable declaration with the program's region where it is valid
- Many PLs allow these regions to be nested (nested lambdas)
- Such languages are called *block-structured*



# Lexical Analysis

- How is each variable reference associated with its declaration?
- A PL associates a variable declaration with the program's region where it is valid
- Many PLs allow these regions to be nested (nested lambdas)
- Such languages are called *block-structured*
- (lambda (x)

```
(lambda (y) (lambda (x) (x y) x))
```

- Regions are searched from innermost to outermost for a parameter list that declares the referenced variable
- If no parameter list searched contains the referenced variable then the variable is free

# Lexical Analysis

- How is each variable reference associated with its declaration?
- A PL associates a variable declaration with the program's region where it is valid
- Many PLs allow these regions to be nested (nested lambdas)
- Such languages are called *block-structured*
- (lambda (x)

(lambda (y) (lambda (x) (x y) x) ) )

The diagram illustrates the lexical environment for the expression (lambda (y) (lambda (x) (x y) x) ) ). It uses nested boxes to show the regions where variables are looked up:

- A red box encloses the entire expression, representing the global environment.
- A blue box encloses the body of the first lambda (lambda (x) (x y) x) ), representing the environment where y is bound.
- A green box encloses the body of the second lambda (x y), representing the environment where x is bound.

Within these regions, the variables are labeled: 'y' is blue, 'x' is green, and 'x' is red. The 'x' in (x y) is green, indicating it is found in the innermost environment. The 'x' at the end of the second lambda's body is red, indicating it is found in the environment where the first lambda was defined.

- Regions are searched from innermost to outermost for a parameter list that declares the referenced variable
- If no parameter list searched contains the referenced variable then the variable is free
- The number of regions crossed is called the *lexical or static depth*
- The position of a variable is the position in the parameter list that declares it
- A variable's lexical depth and position are known as its *lexical address*

# Lexical Analysis

- A variable reference may be replaced by its lexical address

# Lexical Analysis

- A variable reference may be replaced by its lexical address
- ```
(lambda (x y)
  ((lambda (a)
    (x (a y)))
   x))
```

Lexical Analysis

- A variable reference may be replaced by its lexical address
- ```
(lambda (x y)
 ((lambda (a)
 (x (a y)))
 x))
```
- ```
(lambda 2
  (lambda 1
    ((: 1 0) ((: 0 0) (: 1 1))))
  (: 0 0))
```

Lexical Analysis

- QUIZ: 1.34, 1.35

Lexical Analysis

- QUIZ: 1.34, 1.35
- Due in 1 week