Lecture 4. Varlang – variables

September 22, 2019

Overview

- What is a variable?
- Varlang syntax
- Language design decisions related to variables: scoping
- Semantics of Varlang: How to evaluate a program with variables

What does Variable mean in programming languages?

Abstraction: encapsulate the details

- Reuse
- Give a name to the complex definitions

$$\frac{-b \pm \sqrt{b^2 - 4ac}}{2a} * (\frac{-b' \pm \sqrt{b'^2 - 4a'c'}}{2a'} - \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}) \quad \Longrightarrow \quad \text{x * (y - x)}$$

- Easy to understand the code
- Make programming scalable

Some deeper thoughts on variables

In imperative programming languages:

- variable is a name associated with memory location
- you change state by assigning a new value to a variable in a statement

In pure functional programming languages (no side effect at all, given the same input, you get the same output):

- you change state by calling a function
- variables are similar to the math variables, they are symbols
- a function can be assigned to a variable

Varlang programs

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

(let ((x 1) (y 2)) (* y x))

(let ((x 1) (y 2)) (let ((z 3)) (+ x y z)))

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

- let expression consists of two parts: first contains definition of variables, second part is the body of the let expression.
- The body of the let expression could be another expression, it usually contains use of the defined variables.

Varlang Syntax

`Program`	::=	`Exp`	*Program*
`Exp`	::=		*Expressions*
		`Number`	*NumExp*
	|	`(+ Exp Exp`$^+$`)`	*AddExp*
	|	`(- Exp Exp`$^+$`)`	*SubExp*
	|	`(* Exp Exp`$^+$`)`	*MultExp*
	|	`(/ Exp Exp`$^+$`)`	*DivExp*
	|	`Identifier`	***VarExp***
	|	`(let ((Identifier Exp)`$^+$`) Exp)`	***LetExp***
`Number`	::=	`Digit`	*Number*
	|	`DigitNotZero Digit`$^+$	
`Digit`	::=	`[0-9]`	*Digits*
`DigitNotZero`	::=	`[1-9]`	*Non-zero Digits*
`Identifier`	::=	`Letter LetterOrDigit`*	*Identifier*
`Letter`	::=	`[a-zA-Z$_]`	*Letter*
`LetterOrDigit`	::=	`[a-zA-Z0-9$_]`	*LetterOrDigit*

Write Some More Varlang Programs

(+ (let ((x 1)) x) (let ((x 2)) x))

Note. (+ 1 2)

Scoping

- declaration, definition and use of a variable
- static and dynamic scoping
- free, bound variables

Definition and Use - Examples

Expressions	output
1. (let ((x 1)) x)	1
2. (let ((x 1) (y 1)) (+ x y))	2
3. (let ((x 1) (y 1)) (let ((z 1)) (+ x y z)))	3
4. (let ((x 1)) (let ((x 4)) x))	4
5. (let ((x 5)) (let ((y x)) y))	5

Current VarLang Output:
$ (let ((x 5)) (let ((x 1) (y x)) (+ x y)))
6
$ (let ((x 5)) (let ((x 1)) (let ((y x)) (+ x y))))
2

Definition and Use - Examples

```
(let
   ((x
    (let
      ((x 41))
      (+ x 1))))
   x)
```

Scoping

- Keep variables in different parts of program distinct from one another
- Match identifiers' declaration with uses
- Visibility of an entity
- Binding between declaration and uses
- The scope of an identifier is the portion of a program in which that identifier is accessible
- The same identifier may refer to different things in different parts of the program: Different scopes for same name dont overlap
- An identifier may have restricted scope

Lexical or Static Scoping - Varlang

- Effect of variable definition until the next variable definition with the same name
- Scoping rule: variable definitions supersede previous definitions and remain effective until the next variable definition with the same name
- This is a property of the program text and unrelated to the run time call stack.
- Most of the programming languages: like C/C++/Java
- In static scoping, the compiler first searches in the current block, then in the surrounding blocks successively and finally in the global variables.

```c
// A C program to demonstrate static scoping.
#include <stdio.h>
int x = 10;

// Called by g()
int f()
{
        return x;
}

//  g() has its own variable
// named as x and calls f()
int g()
{
        int x = 20;
        return f();
}

int main()
{
        printf("%d", g());
        printf("\n");
        return 0;
}
```

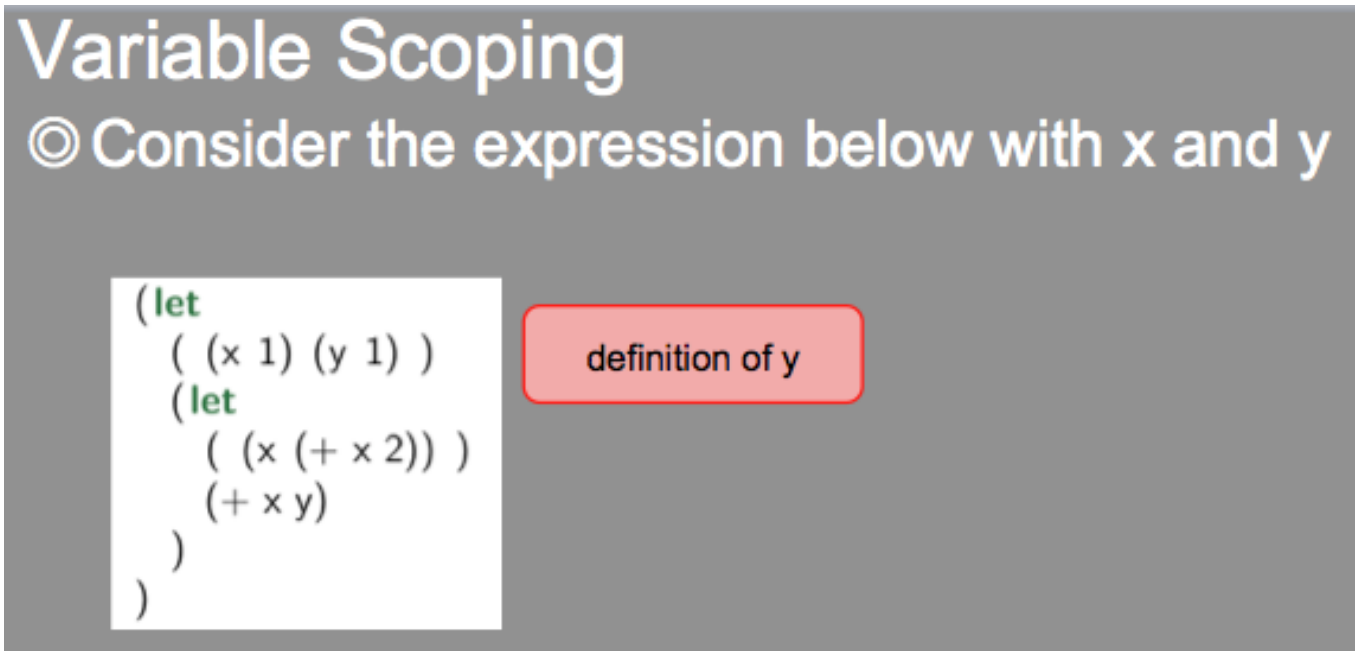
Static Scoping - VarLang Example

Variable Scoping

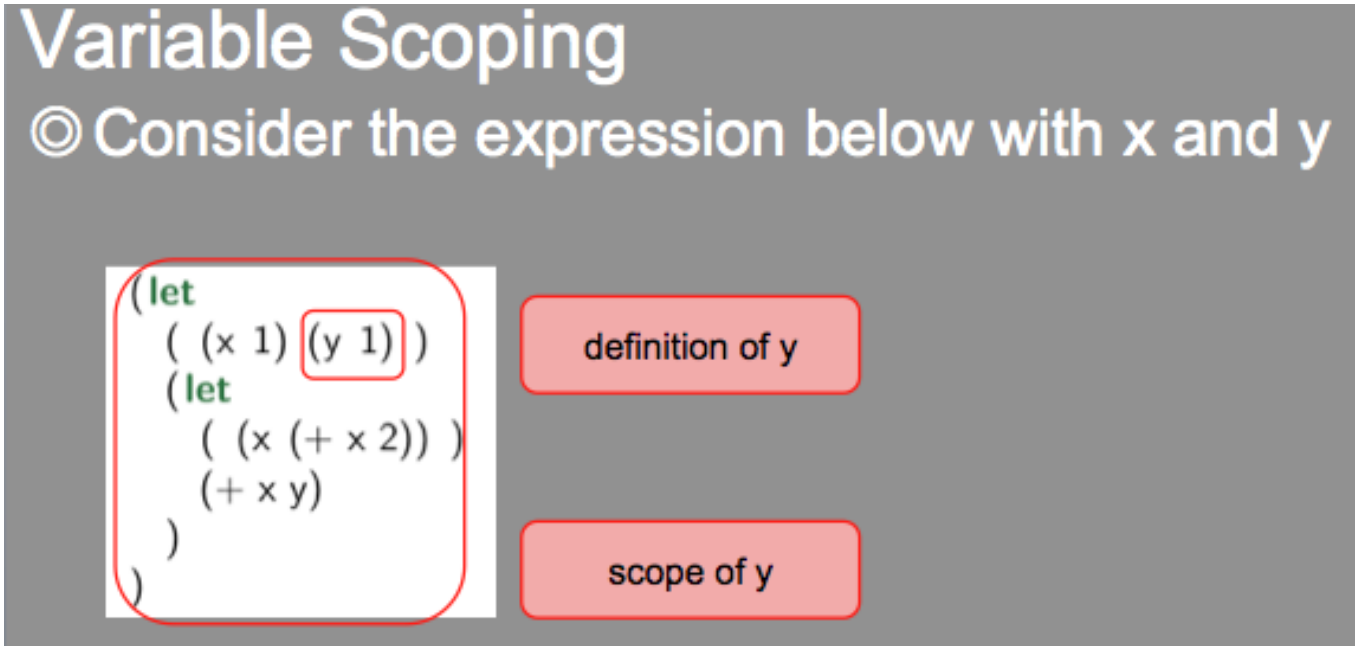
- Consider the expression below with x and y

```
(let
   ( (x 1) (y 1) )
   (let
      ( (x (+ x 2)) )
      (+ x y)
   )
)
```

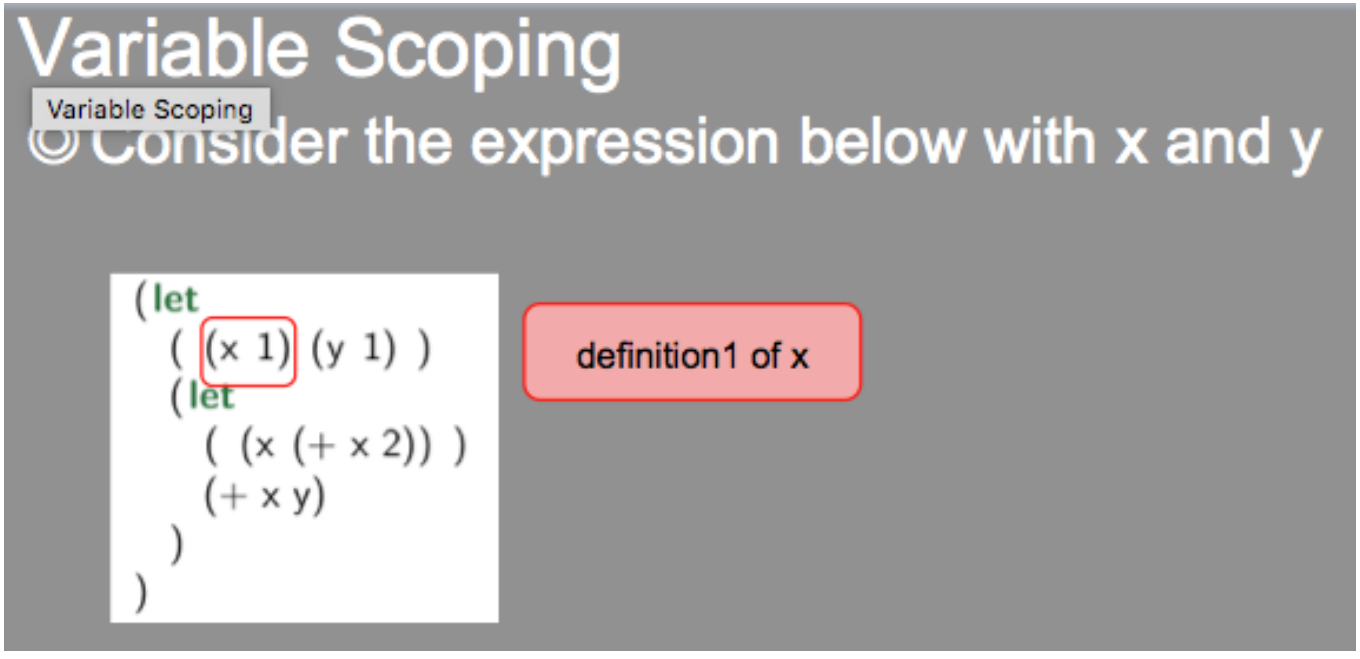
Static Scoping - VarLang Example



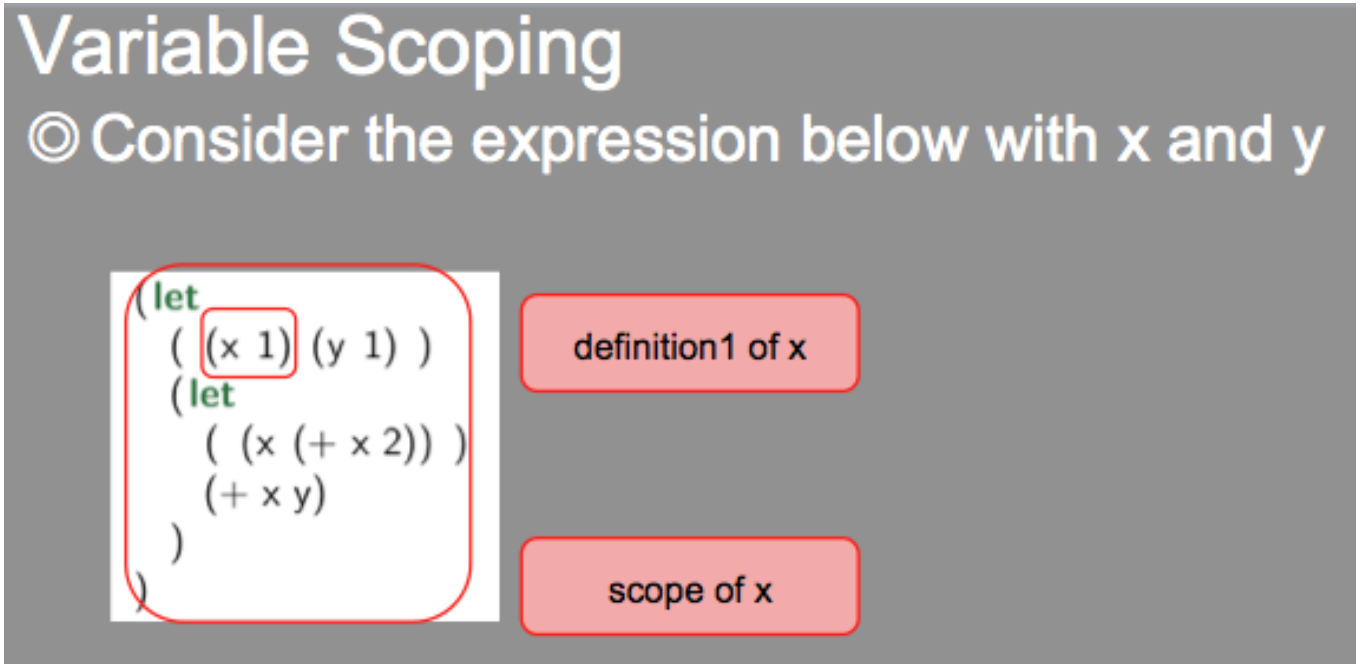
Static Scoping - VarLang Example



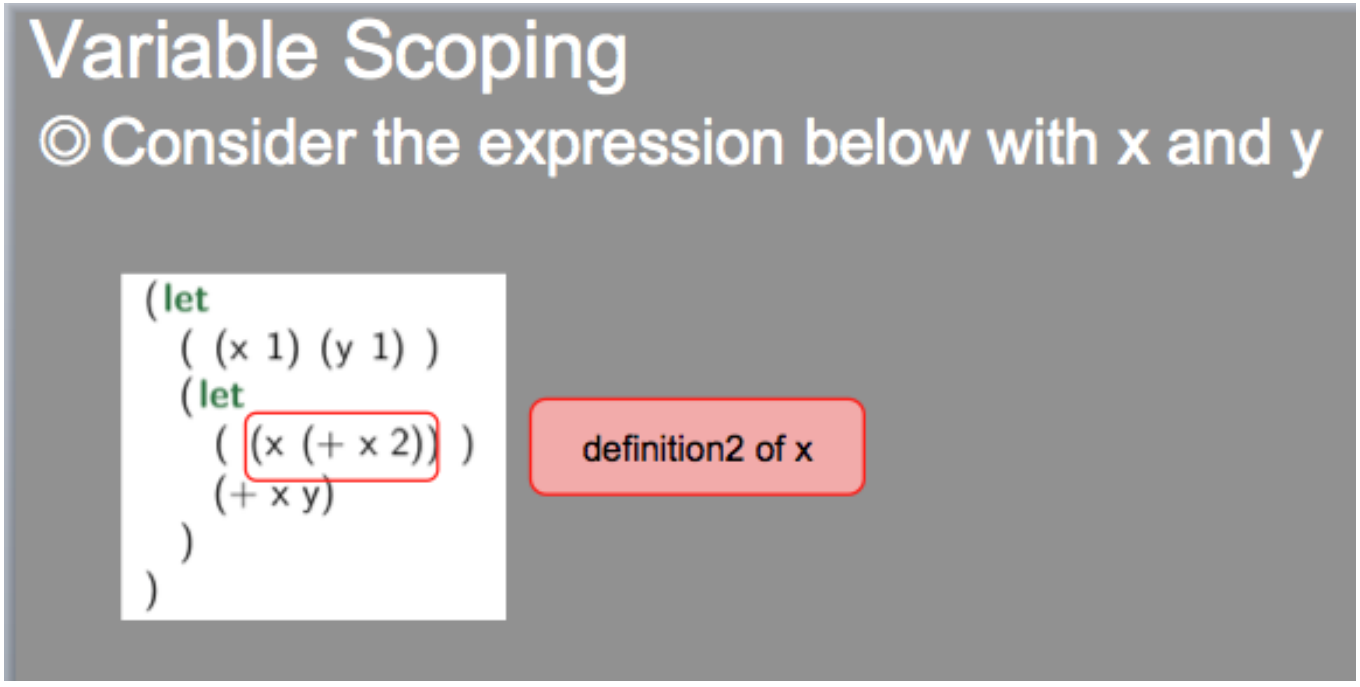
Static Scoping - VarLang Example



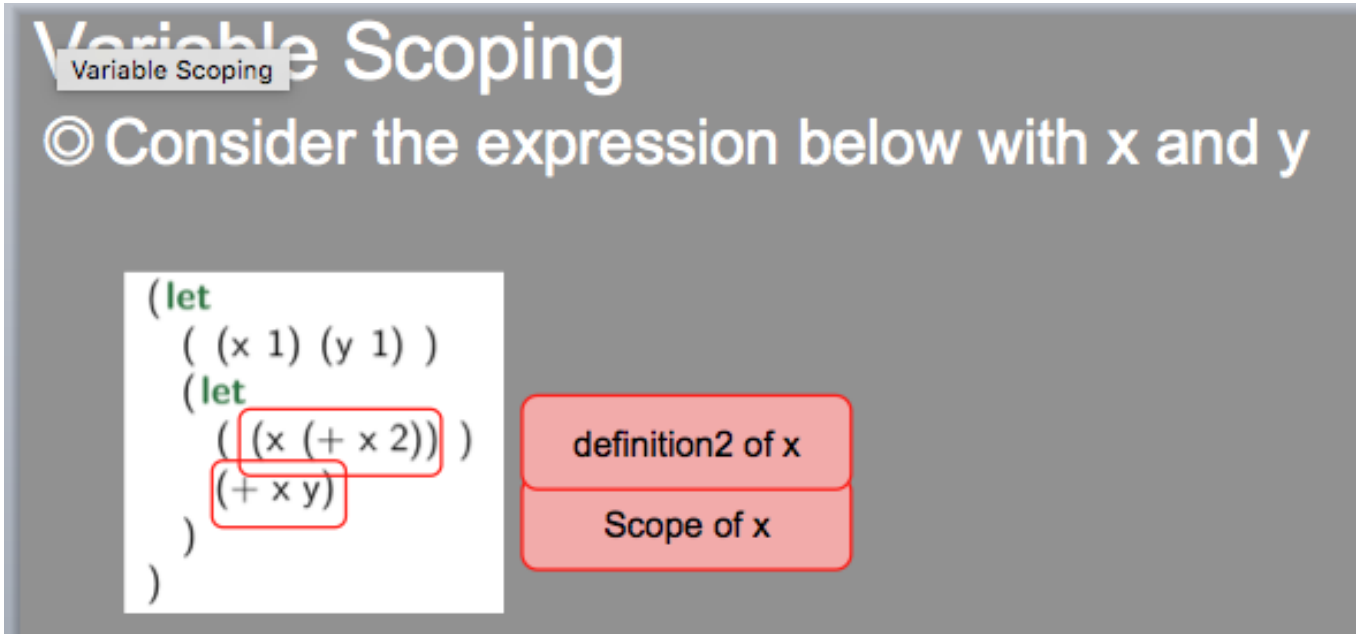
Static Scoping - VarLang Example



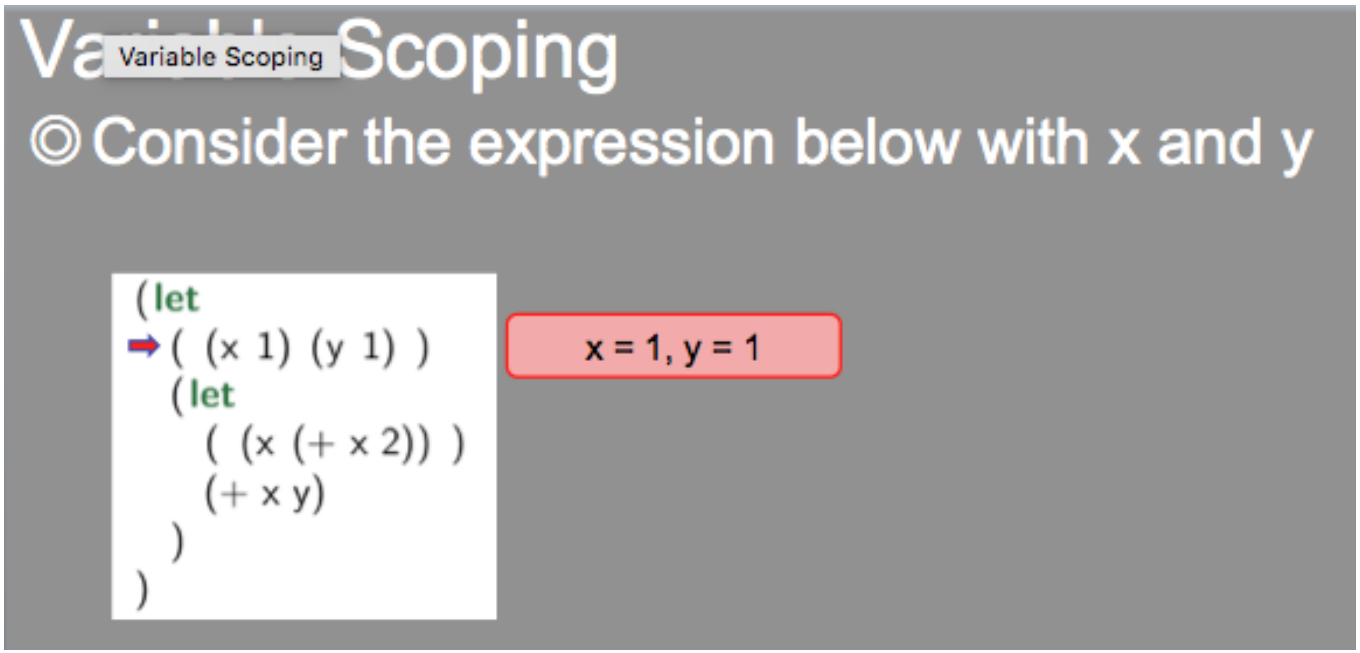
Static Scoping - VarLang Example



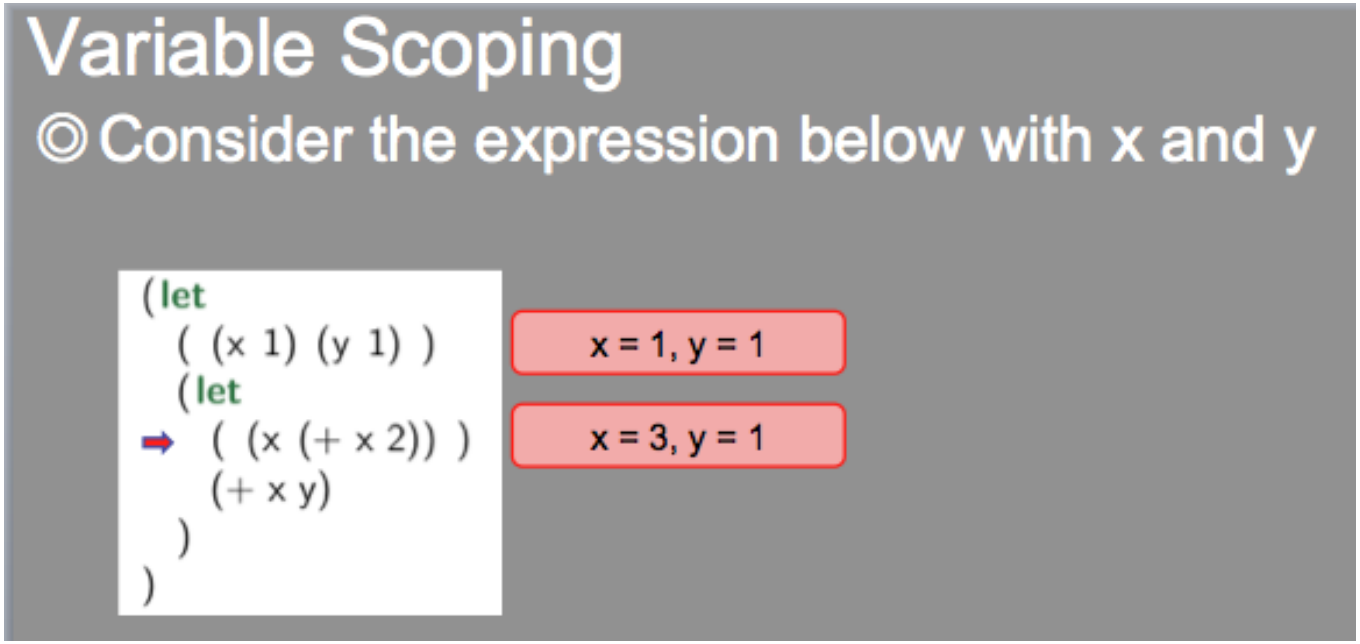
Static Scoping - VarLang Example



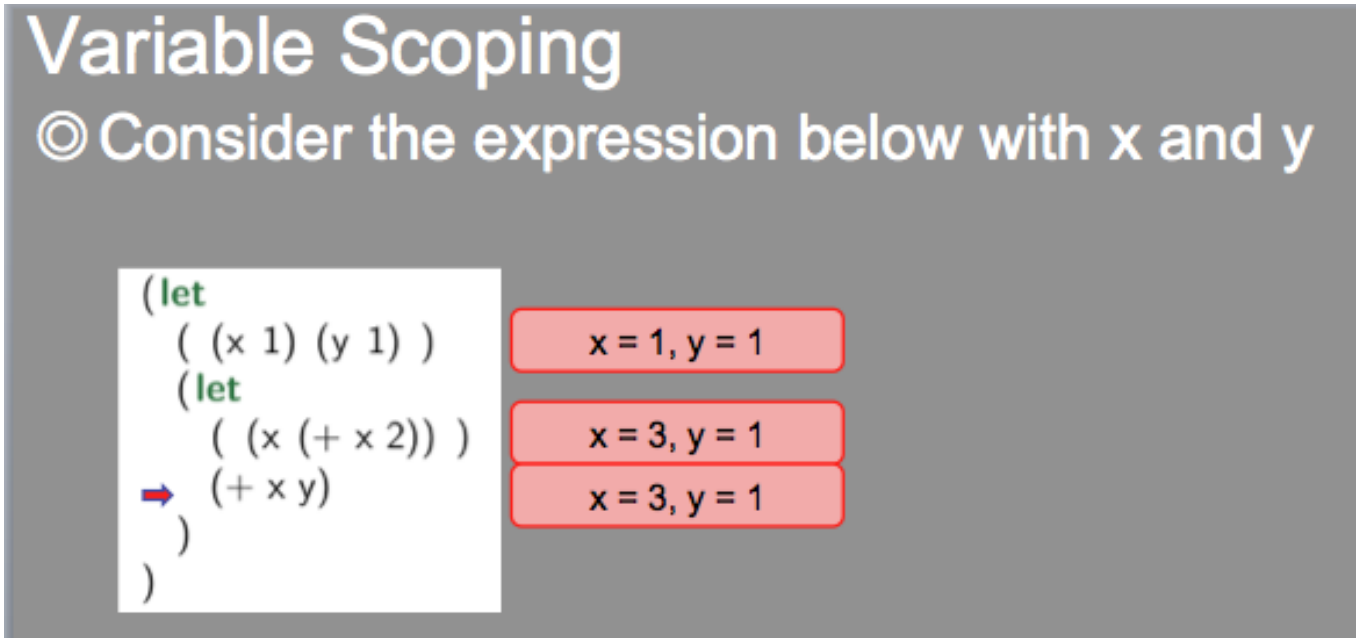
Static Scoping - VarLang Example



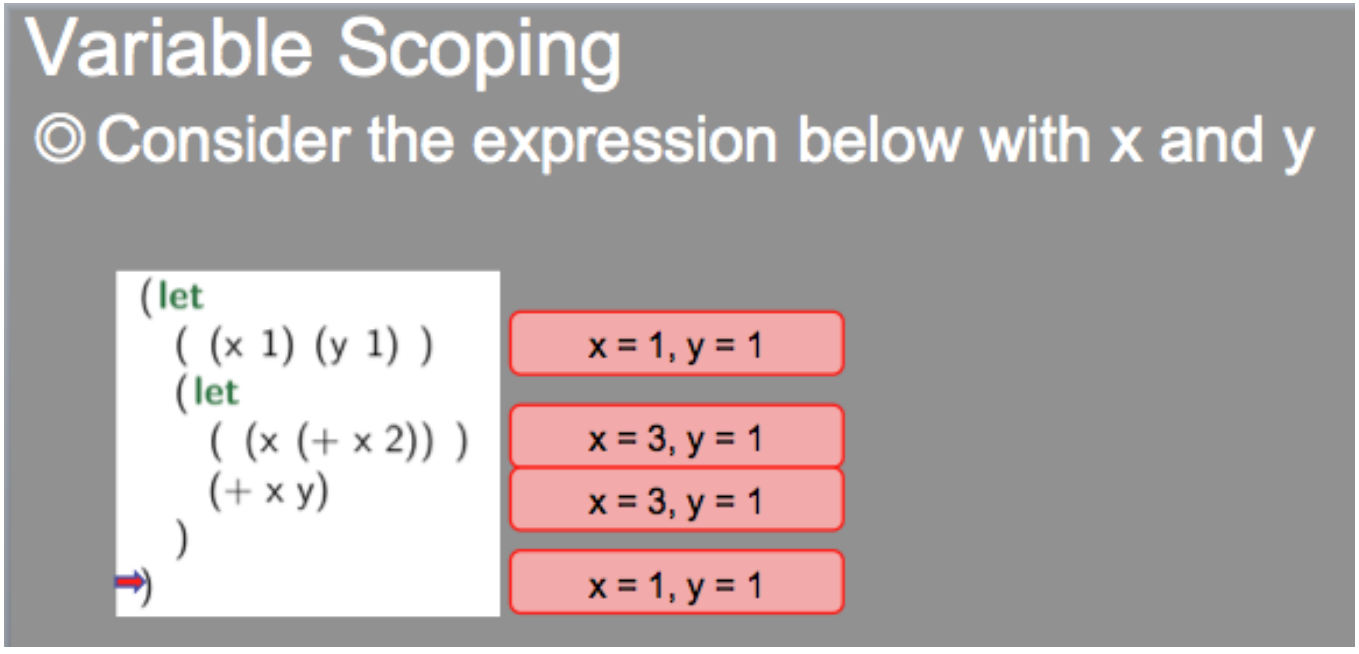
Static Scoping - VarLang Example



Static Scoping - VarLang Example



Static Scoping - VarLang Example



Static Scoping - VarLang Example

A hole in the Scope

```
(let
  ( (x 1) (y 1) )
  (let
    ( (x (+ x 2)) )
    (+ x y)
  )
)
```

Redefine a name creates a "hole":
So the expression (+ x y) is a hole in the scope of the first definition of x

Dynamic Scoping

- Dynamic scoping the compiler first searches the current block and then successively all the calling functions at runtime
- Dynamic scoping does not care how the code is written, but instead how it executes.
- Each time a new function is executed, a new scope is pushed onto the stack.
- Perl has both static and dynamic scoping, using different variables to declare a variable: "my" vs "local"

```
1 // Since dynamic scoping is very uncommon in
2 // the familiar languages, we consider the
3 // following pseudo code as our example. It
4 // prints 20 in a language that uses dynamic
5 // scoping.
6
7 int x = 10;
8
9 // Called by g()
10 int f()
11 {
12         return x;
13 }
14
15 //  g() has its own variable
16 // named as x and calls f()
17 int g()
18 {
19         int x = 20;
20         return f();
21 }
22
23 main()
24 {
25         printf(g());
26 }
```

```perl
# A perl code to demonstrate dynamic scoping
$x = 10;
sub f
{
        return $x;
}
sub g
{
        # Since local is used, x uses
        # dynamic scoping.
        local $x = 20;

        return f();
}
print g(). "\n";
```

Static vs Dynamic Scoping

- Static scoping is easy for programmers to reason about
- Dynamic scoping does not need to explicitly pass the parameters through function calls

Free and Bound Variables

- free variable, a variable occurs free in an expression if it is not defined by an enclosing let expression
- in program x, the variable x occurs free
- in program (let ((x 1)) x), x is bound in enclosing let expression, hence x is not free
- in program (let ((x 1)) (+ x y)), y is free

Varlang: Semantics and Their Implementations

An **environment** is a dictionary that maps variables to values at a program point

Implementation of Varlang for Environment

```
1 public interface Env {
2   Value get (String search_var);
3 }
```

Figure 3.6: Environment Data Type for the Varlang Language.

An empty environment is the simplest kind of environment. It does not define any variables. The listing in figure 3.7 models this behavior.

```
1 class EmptyEnv implements Env {
2     Value get (String search_var) {
3       throw new LookupException("No binding found for: " + search_var);
4     }
5 }

7 class LookupException extends RuntimeException {
8     LookupException(String message){
9       super(message);
10    }
11 }
```

Figure 3.7: Empty environment for the Varlang Language.

Varlang: Use environment to evaluate Varlang program

Current Expression	Current Environment
(let ((x 1)) (let ((y 2)) (let ((x 3)) x)))	Empty
(let ((y 2)) (let ((x 3)) x))	$x \mapsto 1$ :: Empty
(let ((x 3)) x)	$y \mapsto 2$ :: $x \mapsto 1$ :: Empty
x	$x \mapsto 3$ :: $y \mapsto 2$ :: $x \mapsto 1$ :: Empty
3	$x \mapsto 3$ :: $y \mapsto 2$ :: $x \mapsto 1$ :: Empty
(let ((x 3)) 3)	$y \mapsto 2$ :: $x \mapsto 1$ :: Empty
(let ((y 2)) 3)	$x \mapsto 1$ :: Empty
(let ((x 1)) 3)	Empty
3	Empty

The value of a variable is the first value from the left found in the environment

Varlang: Semantics of a Program

- in an environment env, the value of a program is the value of its component expression in the same environment env

$$\frac{\texttt{value e env = v}}{\texttt{value p env = v}}\ \textsc{Value of Program}$$

$$\frac{\texttt{value e env = v, where env} \in \texttt{EmptyEnv}}{\texttt{value p = v}}\ \textsc{Value of Program}$$

Varlang: Implementation

```
1 class Evaluator implements Visitor<Value> {
2   Value valueOf(Program p) {
3     Env env = new EmptyEnv();
4     return (Value) p.accept(this, env);
5   }
6   Value  visit (Program p, Env env) {
7     return (Value) p.e().accept(this, env);
8   }
9   ...
10 }
```

Varlang: Semantics of Expressions

- Every expression has a value
- Expression passes the environment to their subexpressions
- There are expressions that do not change the environment and there are expressions change the environment

Varlang: Semantics of expressions that do not change the environment

- The addition expression neither defines new variable nor removes any existing variable definitions. Therefore, an addition expression should have no direct effects on the environment. (similar for the number, subtraction, division, multiplication expressions)
- All of its subexpressions are evaluated in the same environment.

VALUE OF NUMEXP

value (NumExp n) env = (NumVal n)

VALUE OF ADDEXP

$$\frac{\texttt{value } e_i \texttt{ env} = (\texttt{NumVal } n_i), \text{for i} = 0...\text{k} \qquad n_0 + \ldots + n_k = n}{\texttt{value (AddExp } e_0 \ldots e_k\texttt{) env} = \texttt{(NumVal n)}}$$

Varlang: Implementation

```
Value  visit (AddExp e, Env env) {
   List<Exp> operands = e.all();
   double result  = 0;
   for(Exp exp: operands) {
      NumVal intermediate = (NumVal) exp.accept(this, env);
       result  += intermediate.v();
   }
   return new NumVal(result);
}
```

Revisit Varlang Syntax

```
1  grammar Varlang;
2  program : exp ;
3  exp :   numexp
4          | addexp
5          | subexp
6          | multexp
7          | divexp
8          | varexp
9          | letexp ;
10
11 varexp  :  Identifier  ;
12
13 letexp  : '(' Let '(' ( '(' Identifier exp ')' )+ ')' exp ')' ;

15 Let : 'let' ;

17 Identifier  :  Letter LetterOrDigit *;
```

Figure 3.1: Grammar for the Varlang Language. Non-terminals that are not defined in this grammar are exactly the same as that in Arithlang.

Varlang: Semantics of expressions that change the environment

Var Expressions:

- The meaning of a variable expression in a given environment is dependent on the environment in which we are evaluating that expression
- For example, the value of a var expression x in an environment that maps name x to value 342 would be the numeric value 342. On the other hand, in an environment that maps name x to value 441, the value of the same var expression x would be the numeric value 441

Varlang: Semantics and Implementation – VarExp

VALUE OF VAREXP

```
value (VarExp var) env = get(env, var)
```

```
public Value  visit (VarExp e, Env env) {
  return env.get(e.name());
}
```

```
get(env, var') = Error:  No binding found, if env = (EmptyEnv)
get(env, var') = val, if var = var', env = (ExtendEnv var val env')
                 otherwise get(env', var')
```

Varlang: Semantics – LetExp

- Changes the environment: add new name-value pairs
- A let expression is also serving to combine two expressions exp and exp into a larger expression

Varlang: Semantics of Expressions - LetExp Example

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

New_Env:
x : 1
y : 1

(+ x y) evaluates to 2

Varlang: Semantics – LetExp

- the value of a let expression is the value of its body exp obtained in a newly constructed environment env.
- It is obtained by extended the original environment of the let expression exp with new bindings (variable name to value mapping).

VALUE OF LETEXP

$$\frac{\texttt{value exp env = v'} \qquad \texttt{env' = (ExtendEnv var v'env)} \qquad \texttt{value exp' env' = v}}{\texttt{value (LetExp var exp exp') env = v}}$$

Varlang: Semantics – LetExp

$$\textsc{Value of LetExp}\quad \frac{\begin{array}{c}\texttt{value exp}_i\ \texttt{env}_0 = \texttt{v}_i, \text{for i} = 0...\text{k} \\ \texttt{env}_{i+1} = (\texttt{ExtendEnv var}_i\ \texttt{v}_i\ \texttt{env}_i), \text{for i} = 0...\text{k} \\ \texttt{value exp}_b\ \texttt{env}_{k+1} = \texttt{v}\end{array}}{\texttt{value (LetExp (var}_i\ \texttt{exp}_i\texttt{),for i = 0...k exp}_b\texttt{) env}_0 = \texttt{v}}$$

Note.

- Anything before the line denotes the condition(s), anything under the line denotes the result if the condition is true.
- There could be multiple conditions, these are usually separated by a space or, written in a new line before the straight line that separates the condition.

Varlang: Implementation – LetExp

```
public Value visit (LetExp e, Env env) { // New for varlang.
  List<String> names = e.names();
  List<Exp> value_exps = e.value_exps();
  List<Value> values = new ArrayList<Value>(value_exps.size());

  for(Exp exp : value_exps)
    values.add((Value)exp.accept(this, env));

  Env new_env = env;
  for (int i = 0; i < names.size(); i++)
    new_env = new ExtendEnv(new_env, names.get(i), values.get(i));

  return (Value) e.body().accept(this, new_env);
}
```

Review: VarLang

- What is a variable?
- VarLang Syntax: Write VarLang programs
- Decision for variables: Scoping
  - definition/use of variables,
  - free/bound variable
- VarLang semantics: Environment