Lecture 4. Varlang

September 14, 2018

Review from Previous ArithLang

- Syntax
  - Design decision
  - Understand the grammar
- Semantics
  - Interpreter Architecture
  - Semantics rules
  - Implementation

Brief Review on ArithLang.g

- $e = exp\,\{\$ast = newProgram(\$e.ast)\}$

Varlang - Variables

What does Variable mean in programming languages?

abstraction

- it can be reused once assigned
- variables provide capability to refer to the (potential complex) definition by referring to the name

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

- easy to understand the code
- make programming scalable

Key terms

- definition, use of a variable
- scoping: static and dynamic scoping
- free, bound variables
- global variables
- environment

x=1;

Definition and Use - Examples

Varlang Programs

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

definition - use of a var

let x be 1 y be 1, the result of x + y is 2

Scoping

- 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

Static Scoping - Example

Please check ScopingExample.ppt for the animation

Definition and Use - Examples

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

Varlang Syntax

VarExp, LetExp

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*

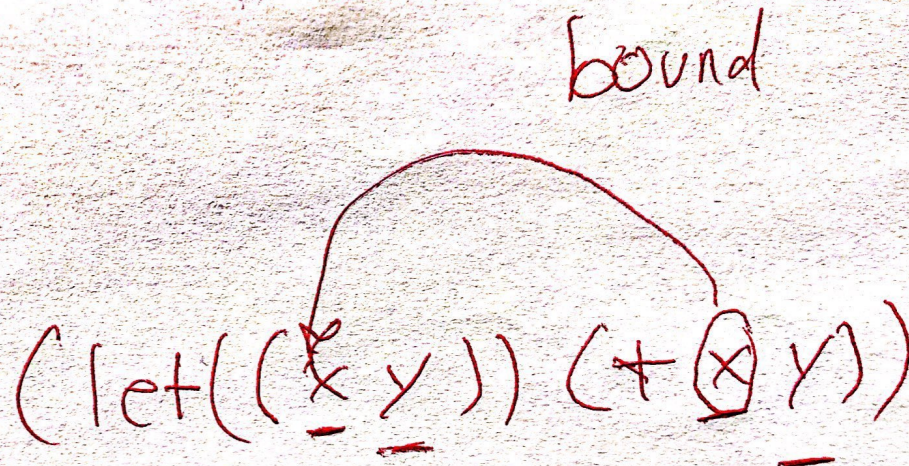
Varlang Syntax

Exercise: Write VarLang Program with 3, 4, 6, 1

Varlang Implementation

```
1  class VarExp extends Exp {
2    String _name;
3    VarExp(String name) { _name = name; }
4    String name() { return _name; }
5    Object accept( Visitor visitor , Env env) {
6      return visitor . visit (this, env);
7    }
8  }
9  class LetExp extends Exp {
10   List<String> _names;
11   List<Exp> _value_exps;
12   Exp _body;
13   LetExp(List<String> names, List<Exp> value_exps, Exp body) {
14     _names = names;
15     _value_exps = value_exps;
16     _body = body;
17   }
18   Object accept( Visitor visitor , Env env) {
19     return visitor . visit (this, env);
20   }
21   List<String> names() { return _names; }
22   List<Exp> value_exps() { return _value_exps; }
23   Exp body() { return _body; }
24 }
```

Free and Bound Variables



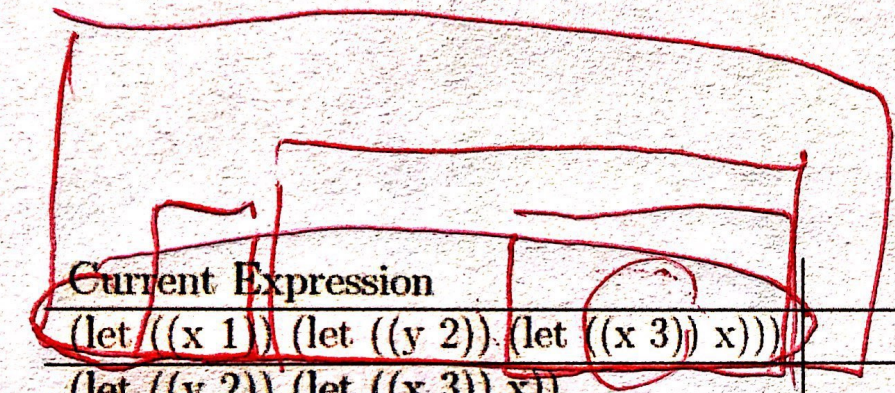
- free variable, a variable occurs free in an expression if it is not defined by an enclosing let expression
- in program x, the variable x occurs free; in program (let ((x 1)) x), x is bound in enclosing let expression, hence x is not free

Varlang: what is the value of Varlang program

Environment

1. An environment is a dictionary that maps variables to values at a program point
2. The value of a variable is the first value from the left found in the environment

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

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 Syntax

Newly Added: varexp

```
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 – Environment Abstraction

An environment is a data type that provides an operation to look up the value of a variable. The definition below models this intent and the implementation in figure 3.6 realizes it.

```
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')
```

Here, `var, var' ∈ Identifier`, the set of identifier, `val ∈ Value`, the set of values in our Varlang language, and `env, env' ∈ Env`, the set of environments. As before, take the notation `(EmptyEnv)` to mean an environment constructed using a constructor of type `EmptyEnv`, and take `EmptyEnv` to mean all such elements, i.e. the entire set. Similarly, `(ExtendEnv var val env)` is an environment constructed using a constructor of type `ExtendEnv`, with `var`, `val`, `env` being values used to construct this environment.

recursive def

env =

[env'
  var val

Varlang: Semantics and their Implementation

In an environment env, the value of a program is the value of its component expression in the same environment env; every expression has a value

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

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

Varlang: Semantics and their 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

Expressions that do not change the environment:

- **Expression that passes the environment to their subexpressions**
- 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.
- **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

Env env

```
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);
}
```

(let((x 1)) x)

Varlang: Semantics of Expressions

a
b

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 of Expressions – VarExp

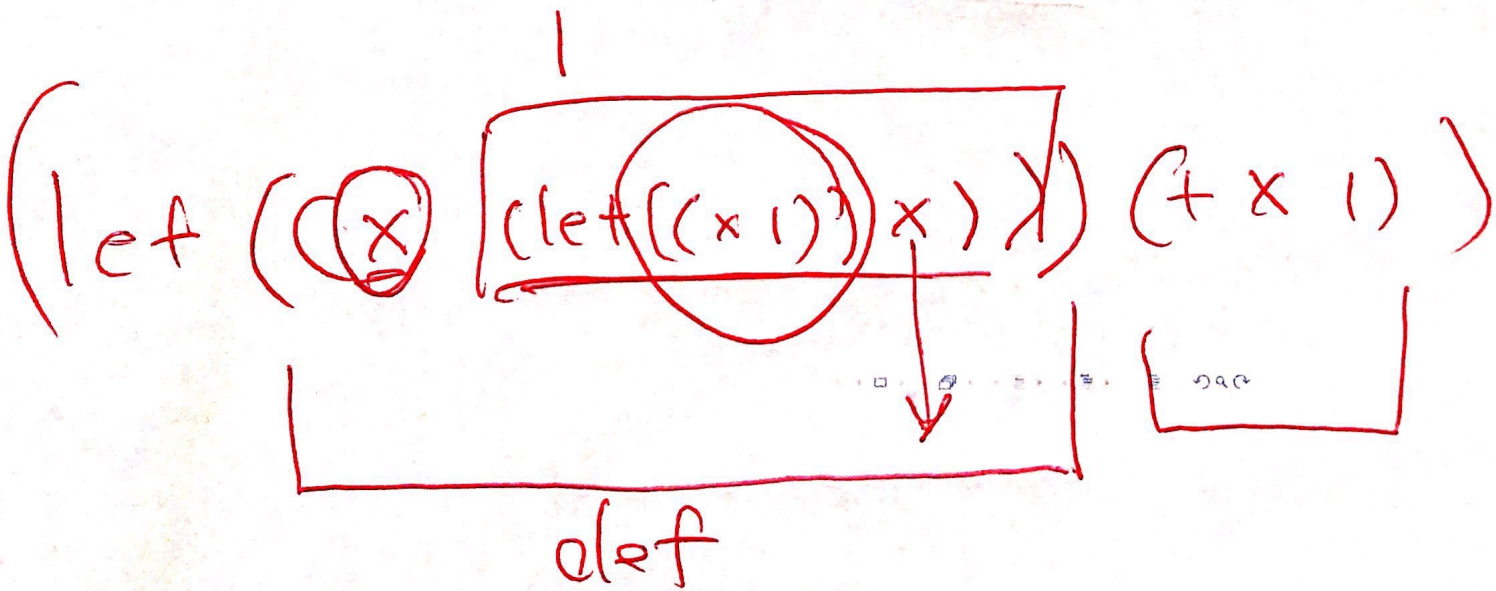
VALUE OF VAREXP

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

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

Varlang: Semantics of Expressions – 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

- 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: Extended Environment

```
1 class ExtendEnv implements Env {
2     private Env _saved_env;
3     private String _var;
4     private Value _val;
5     public ExtendEnv(Env saved_env, String var, Value val){
6       _saved_env = saved_env;
7       _var = var;
8       _val = val;
9     }
10    public Value get (String search_var) {
11      if (search_var.equals(_var))
12        return _val;
13      return _saved_env.get(search_var);
14    }
15 }
```

Figure 3.8: Extended environment for the Varlang Language.

get(env, var')

Varlang: Semantics of Expressions – LetExp

(let ((a 3) (b (- 2))) (+ a b))

VALUE OF LETEXP

$$\frac{\begin{array}{l}\text{value } exp_i\ env_0 = v_i, \text{for } i = 0...k \\ env_{i+1} = (\text{ExtendEnv } var_i\ v_i\ env_i), \text{for } i = 0...k \\ \text{value } exp_b\ env_{k+1} = v\end{array}}{\text{value (LetExp } (var_i\ exp_i), \text{for } i = 0...k\ exp_b)\ env_0 = v}$$

New semantic Rule for Q6 HW3

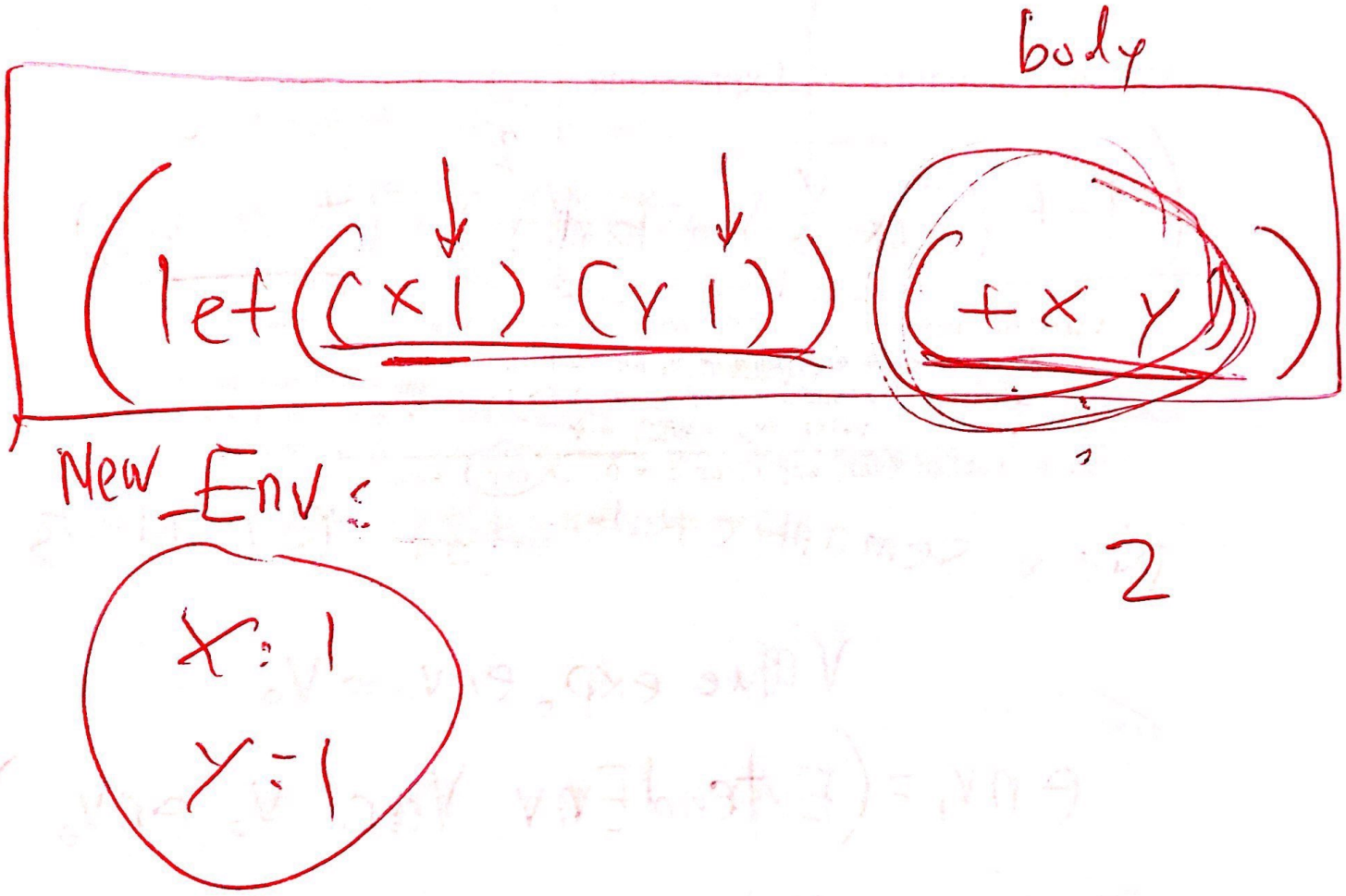
Value $exp_0\ env_0 = V_0$

$env_1 = (\text{ExtendEnv } Var_0\ V_0\ env_0)$

Value $exp_1\ env_1 = V_0$

⋮

Varlang: Semantics of Expressions – LetExp Example



Varlang: Implementation

```
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 :

Key terms

- definition, use of variables
- free bound
- scoping * (design decision)
- syntax varlang, write varlang progm
- semantic design
  1. Value
  2. Exp changes Env
     Exp not changes Env
  3. Semantic rules
     with Implementation * Design
     Decision