Chapter 3

Varlang: A Language with Variables

Our goal in this chapter is to learn about variable definition, usage, and related concepts such as environment, variable scope, entering and leaving scope, holes in variable scope, substitution-based model, etc. These concepts are often a key ingredient of many programming language features such as functions, procedures, closures, classes, and modules.

3.1 Variables as means of abstraction

One of most important functionalities in computer programming languages is their *means of abstraction*, i.e. the ability to create a proxy for certain program elements that hides certain concrete details while preserving those deemed important. Most elementary form of abstraction is variable definition, almost directly borrowed from mathematical notion of variables. In a variable definition, name of the variable is a proxy for the actual definition. When we define a variable, we abstract away from its concrete definition and give the clients of the variable the capability to refer to the (potential complex) definition by referring to the name of the variable. For example, the variables `x` and `y` below stand for the definitions on the right.

$$x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a} \qquad y = \frac{-b' \pm \sqrt{b'^2 - 4a'c'}}{2a'}$$

which allows us to write $x * (y - x)$ instead of the following more complex

form.

$$\frac{-b \pm \sqrt{b^2 - 4ac}}{2a} * (\frac{-b' \pm \sqrt{b'^2 - 4a'c'}}{2a'} - \frac{-b \pm \sqrt{b^2 - 4ac}}{2a})$$

Abstractions are important for scalability of programming as an intellectual activity as well as for efficiency of programs. Both benefits can be seen in the previous example. Since we can understand the shorter form $x * (y - x)$ much easier compared to the more complex form, our ability to understand mathematical expressions is more effectively leveraged. Also, notice that in the shorter form, the quadratic root x is only computed once and used twice, whereas in the longer form it is computed twice. That can be more efficient in some languages. It is for these reasons that variable definitions and usage are important components of most computer programming languages.

There are two kinds of variable definitions that are commonly included in programming language designs. These differ in terms of *scope*, i.e. the extent during which the variable definition is valid. In this chapter, we study the former kind where once defined a variable definition has a limited scope. Then, in next chapter we will look at variable definitions that once defined, are perpetually valid, unless redefined.

3.2 Variable Definition and Usage

We will get started by designing and implementing a simple programming language *Varlang*. Varlang is an extension of Arithlang, the language developed in the previous chapter. When we say that a language extends another, we will mean that the former has all the features of the latter and more.

Goal of Varlang will be to allow its programmer to write programs that can perform arithmetics using both numbers and defined variables. For example, a user of this programming language might write the following expressions.

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

4. (let ((x 1)) (let ((x 4)) x))

5. (let ((x 5)) (let ((y x)) y))

These users can then expect such programs to produce values 1, 2, 3, 4, and 5 respectively. Notice that, like Arithlang, the programs in Varlang are also written in the prefix notation, as opposed to the infix notation that languages like Java use. To recall, in a prefix notation, operators such as +, -, *, /, %, `let` appear before their operands. To be unambiguous, we delineate start and end of a Varlang expression with '(' and ')'.

These programs show two main features: variable definition and variable usage. Variable definition is done using the "let" expression, which is similar to variable definitions in mathematics where we often write statements like.

```
Let x be 1, and y be 1.  Then, the result is x + y.
```

In the syntax of Varlang, the let expression starts with open parenthesis '(' and the keyword 'let' and ends with close parenthesis ')'. The rest of the let expression has two parts: open parenthesis '(', followed by one or more variables that are defined, followed by close parenthesis ')', and the expression for which the definition would be effective, referred to as the scope of variable definition. Each variable definition has the form open parenthesis '(' followed by the name given to the variable, followed by the expression whose value will be given to the variable, followed by close parenthesis ')'. So in Varlang, the informal statement above will be written as the following expression `(let ((x 1) (y 1)) (+ x y))`.

```
(let
  ( (x 1) (y 1) ) //List of definitions .
  (+ x y)  // Body of the let expression
)
```

Let expressions can be nested inside each other. For example, we can also define `x` and `y` as follows:

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

In this example, first let expression declares variable `x` and the second let expression declares variable `y`. Let us look at another example.

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

In this example, first let expression is assigning the value of the entire second let expression to `x`. The value of the entire let expresion is the value of `(+ x 1)` in that context, which is `42`. Therefore, the value of this program is `42`.

Exercise

3.2.1. Convert following arithmetic expressions written in an informal notation to the notation of Varlang. Also, check and write down the value of each expression.

1. `Let x be 1. Then, the result is x.`
2. `Let x be 4 and y be 2. Then, the result is x - y.`
3. `Let x be 1. Let y be 2. Then, the result is x + y.`
4. `Let x be 7, y be 1, and z be 2. Then, the result is x - y - z.`
5. `Let x be 10, y be 2. Then, the result is x / y.`

3.2.2. Write five different arithmetic expressions using the syntax of Varlang such that each expression produces the value 342. Each expression must use all four arithmetic operators +, -, *, and / and the new let expression.

3.2.3. Building on the last problem, write five different arithmetic expressions using the syntax of Varlang such that each expression produces the value 342. Each expression must use all four arithmetic operators +, -, *, and / and the new let expression. First expression should use one let expression, second expression should use two nested let

expressions, third expression should use three nested let expressions, and so on.

3.2.4. Write a let expression that declares a variable `pi` with value `3.1415` and then uses that variable in its body to compute the area of a circle of radius `2`.

3.3 Variable Scoping

When thinking about variable definitions their validity span is an important concern. It is important to understand the extent to which the same variable definition would be in effect, and if later variable definitions can supersede prior definitions. For example, consider the statement: "Let x be 1. Then, the result is x + 1.". Here we can see that in the second sentence the meaning of x is the same as that given by the first sentence. In an different statement "Let x be 1. Let x be 42. Then, the result is x + 1.", the meaning of x in the third sentence is not the same as that given by the first sentence. In general, in writing, variable definitions supersede previous definitions and remain effective until the next variable definition with the same name or the end of discussion, whichever comes earlier. The previous sentence is an example of *scoping rule*.

Now consider a sequence of statements in ALGOL family of languages like C, C++, Java, etc... `int x = 1; { int x = 42; res = x + 1; }`. For these languages, the validity span of a variable definition is usually marked with an open brace '{' and the end of validity span is marked with a close brace '}'. So one can say that the definition of `x` which assigns `42` will be effective for the statement `res = x + 1;`. This allows programmers to look at the code and determine whether a variable definitions would have effect without running the program. This kind of scoping rule is called *lexical scoping* or *static scoping*.

An alternative scoping rule is called *dynamic scoping*. It is available in some languages such as Perl and Logo. To understand dynamic scoping consider these two pieces of text. "Let s be a student. To understand how to grade their work see the section on grading.". "Section: grading. If s is an undergraduate student then we grade section 1, otherwise we grade sections 1 and 2." In these two text segments when we transition from reading the first section to reading the section on grading, we carry with us

the definition of s as a student, even though the text section on grading may be in a completely different location compared to the previous segment.

In Varlang, we adopt lexical scoping rules. The name defined in a let expression would be effective only within the body of the let expression, which can be found by examining the let expression. In other words, by observing the static code of a Varlang program it will be possible to determine the extent of each variable definition. For example, consider the following program in Varlang that has two let expressions.

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

The scope of y's definition in this program is the entire body of the first let expression, whereas the scope of first x's definition is only the expression `(+ x 2)`. After evaluating this expression, the value of x is overridden with a new definition. The scope of this new definition is the expression `(+ x y)`. Redefining a name creates a *hole in the scope* of the original scope. So the expression `(+ x y)` is a hole in the scope of the first definition of x.

Related to variable definition and scoping is the concept of a *free variable* and a *bound variable*. A variable occurs free in an expression if it is not defined by an enclosing let expression. For example, in program "`x`" the variable x occurs free. In Varlang program "`(let ((x 1)) x)`" the variable x is bound because it is defined by an enclosing let expression. So x is not a free variable in this program.

Exercise

3.3.1. Write down all free and bound variables in each of the following Varlang programs. If a program has none of a kind, state None.

1. (let ((x 1)) x)
2. (let ((x 1) (y 1)) (+ x y z))
3. (let ((c 3) (l p)) (let ((s 2)) (+ c l s)))
4. (let ((x 1)) (let ((x 4)) x))

5. (let ((q 5)) (let ((r q)) s))

3.3.2. Are there holes in the scope of top-level variable definitions in each of the following Varlang programs? If a program has none, state None. Otherwise, write the portions of the program that represents the hole.

1. (let ((x 1)) x)
2. (let ((x 1)) (let ((x 4)) x))
3. (let ((x 1) (y 1)) (+ x y z))
4. (let ((c 3) (l 4)) (let ((c 2)) (+ c l)))
5. (let ((q 5)) (let ((r q)) r))

In the rest of this chapter, we will understand different aspects of the semantics of variable definitions and usage by building an implementation of Varlang. As with Arithlang, we will build an interpreter as opposed to a compiler. Fortunately, we have the Arithlang implementation to work with. So the interpreter discussed in this chapter will build on that.

3.4 Reading Let and Var Expressions

Recall that an interpreter typically consists of three steps: read, eval, and print. The read step consists of reading a program in the programming language as a string, dividing the string into smaller, atomic pieces called tokens, organizing the tokens according the grammar rules of the language into a parse tree, and finally converting the parse tree to an abstract syntax tree. We have already looked at the grammar rule for the Arithlang programming language in the previous chapter. Since Varlang has few more expressions, we will need to extend those rules. This extended grammar is given in figure 3.1.

The rule for programs is the same – a program can be an expression in Varlang. The rule for expression (`exp`) has two additional alternatives (highlighted): `varexp` (short for variable expression) and letexp. The rule for `varexp` says that a variable name can be an `Identifier`. An `Identifier` can be a letter followed by zero or more letter or digits, e.g. `a1` is a valid identifier but `1a` is not a valid identifier. As discussed earlier, the rule for `letexp` says that a let expression begins with an open parenthesis and 'let'

`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*

Figure 3.1: Grammar for the Varlang Language

keyword and ends with a close parenthesis, and contains one or more variable definitions `‘(’ ( ‘(’ Identifier exp ‘)’ )+ ‘)’` followed by the body `exp`. In our grammars when we say (x y)+, it means that a sequence of x followed by y repeated one or more times, e.g. `( ‘(’ Identifier exp ‘)’ )+`.

Similar to Arithlang, we can write syntax derivations for example programs in Letlang. One such example derivation is shown in figure 3.2.

Exercise

3.4.1. Write down the leftmost derivations for the following programs in Varlang.

1. (let ((c 3)) c)
2. (let ((c 3)(d 4)) d)
3. (let ((c 3)(d 4)) (+ c d))

3.4.2. Modify the grammar of the Varlang language so that the let expression can declare exactly one variable. Is this design less expressive

1. *program* →
2. *exp* →
3. *letexp* →
4. '(' 'let' '(' *('(' Identifier exp ')')+* ')' exp ')' →
5. '(' 'let' '(' '(' *Identifier* exp ')' ')' exp ')' →
6. '(' 'let' '(' '(' *Letter* exp ')' ')' exp ')' →
7. '(' 'let' '(' '(' 'x' *exp* ')' ')' exp ')' →
8. '(' 'let' '(' '(' 'x' *numexp* ')' ')' exp ')' →
9. '(' 'let' '(' '(' 'x' *Number* ')' ')' exp ')' →
10. '(' 'let' '(' '(' 'x' *DIGIT* ')' ')' exp ')' →
11. '(' 'let' '(' '(' 'x' '1' ')' ')' *exp* ')' →
12. '(' 'let' '(' '(' 'x' '1' ')' ')' *varexp* ')' →
13. '(' 'let' '(' '(' 'x' '1' ')' ')' *Identifier* ')' →
14. '(' 'let' '(' '(' 'x' '1' ')' ')' *Letter* ')' →
15. '(' 'let' '(' '(' 'x' '1' ')' ')' 'x' ')'

Figure 3.2: A Leftmost Syntax Derivation for Program `(let ((x 1)) x)`

compared to the previous design where multiple variables could be declared?

3.5 AST Nodes for Let and Var

The read phase of the interpreter produces an abstract syntax tree representation of the program for other phases. In the abstract syntax tree representation of Arithlang we didn't have suitable data structures for stor-

```
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 }
```

Figure 3.3: New AST nodes for the Varlang Language.

ing variable and let expressions. So we extend the AST data structure to include two new kinds of AST nodes: `VarExp` representing variables and `LetExp` representing let expressions.

These new node kinds are shown in figure 3.3. Since both kinds of nodes are also expressions, their implementation extends the `Exp` class. Internally `VarExp` stores the name of the variable, a string. The `LetExp` stores the names of defined variables, their initial values, and the *let body*, an expression during which variable definitions will have effect (the scope).

Since our overall interpreter framework uses the visitor design pattern,

these new AST nodes also implement the delegating method `accept`. Recall that the role of the method `accept` is to invoke the method with signature `visit(LetExp e, Env env)` declared in any subclass of the `Visitor` class.

```
1  public interface Visitor <T> {
2    public T visit (AST.AddExp e, Env env);
3    public T visit (AST.NumExp e, Env env);
4    public T visit (AST.DivExp e, Env env);
5    public T visit (AST.ErrorExp e, Env env);
6    public T visit (AST.MultExp e, Env env);
7    public T visit (AST.Program p, Env env);
8    public T visit (AST.SubExp e, Env env);
9    public T visit (AST.VarExp e, Env env); // New for the Varlang
10   public T visit (AST.LetExp e, Env env); // New for the Varlang
11 }
```

Figure 3.4: The Visitor Interface for the Varlang Language.

Finally, the interface `Visitor` is extended to support these two new expressions (new methods on lines 9-10 in figure 3.4). You will also notice that each existing method is extended to include an additional formal parameter of type `Env` that stands for environment. We will discuss the notion of environment shortly in section 3.7. Since `Formatter` is a kind of visitor this change requires updating it also.

To summarize, to extend the frontend of the Arithlang language to support new expressions for variable definition and usage we have taken the following steps:

- extended the language's grammar to support parsing new expressions,
- extended the abstract syntax tree data structure to add new classes for each new kind of expression,
- extended the visitor structure to support new kind of expressions, and
- updated existing visitors to support new kind of expressions.

In the rest of this book, by *extending the frontend* we will refer to these steps.

3.6 Lexically Scoped Variable Definitions

What is the value of `(let ((x 1)) x)`? For this simple expression it is easy to guess that it may be `1`. We can reason about the value of this expression as follows "Let x be 1. Then, result is x. Since x is 1, therefore result is 1." So we might venture a guess and claim that the value of a let expression is the value of the let body. So the value of `(let ((x 3)(y 4)) (+ x y))` is the value of `(+ x y)`. We have realized the semantics of addition expression, so based on that we can say that the value of the let expression is the value `x` added to the value of `y`, but how do we find the value of `x` and `y`?

There are two ways of thinking about it. First method, called the *substitution-based semantics* works as follows. The value of `(let ((x 3)(y 4)) (+ x y))` is the value of a new expression created from the original let body `(+ x y)` by replacing `x` with `3` and `y` with `4`. According to this approach the value of the let expression is the value of `(+ 3 4)` that is `7`. In this style, we do not need to define a meaning of "value of a variable" because we never need to find it. To further illustrate, the value of `(let ((x 3)) (let ((y 4)) (+ x y)))` is the value of a new expression created from the original let body `(let ((y 4)) (+ x y))` by replacing `x` with `3`, i.e. value of `(let ((y 4)) (+ 3 y))`. The value of `(let ((y 4)) (+ 3 y))` is the value of a new expression created from the let body `(+ 3 y)` by replacing `y` with `4`.

Second method, called the *environment-based semantics* works as follows. The value of `(let ((x 3)(y 4)) (+ x y))` is the value of the original let body `(+ x y)` in the presence of a dictionary, called the *environment*, which maps `x` to `3` and `y` to `4`. Value of `x` is found by looking up `x` in that dictionary, and similarly value of `y` is found by looking up `y` in that dictionary.

Both strategies have some distinct advantages. In theoretical treatment of programming languages, the former approach is preferred. In this book, we will follow the environment-based style because it is closer to concrete implementations of programming languages.

In a nutshell, in an environment-based semantics all expressions have access to an environment. An environment is a dictionary that contains mapping from variable names to values. A let expression adds new mappings to the environment. These new mappings exist while running the let body. Afterward these new mappings are discarded. We now illustrate this

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

Figure 3.5: Illustrating Environment for the Varlang Language.

using an example in figure 3.5.

Notice that new mappings from variables to values such as $x \mapsto 1$ are placed at the beginning of the environment. Also, observe from the evolution of the expression that the value of a variable is the first value from the left found in the environment. To implement this semantics we will first build data structures for the environment.

3.7 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'` $\in$ `Identifier`, the set of identifier, `val` $\in$ `Value`, the set of values in our Varlang language, and `env`, `env'` $\in$ `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.

This definition provides a single operation `get` to lookup a variable from an environment. The definition says that looking up any variable in an empty environment leads to error. Furthermore, it says that looking up a variable in an extended environment constructed using `var`, `val`, `env` is `val` if variable being searched is the same as `var`, otherwise it is the same as the value obtained by looking up the variable in `env`.

```
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.

Since this empty environment does not define any variables, for every variable lookup it tells its client that the variable cannot be found by signaling an exception.

Another kind of environment is one that stores a mapping from a single variable to its value. This behavior is modeled by the environment class `ExtendEnv` defined in figure 3.8. When a variable is looked up using the `get` operation in this kind of environment, it checks whether the variable

```
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.

being searched is the same as the variable stored. If so, it returns the stored value. Otherwise, it delegates.

The delegation behavior is interesting in that it allows environments to be chained together in a singly linked list. The field `_saved_env` stores the next environment in the chain and on line 13 in figure 3.8 we delegate to it.

Using these data types, we can create an environment that maps a variables `a` to value `3`, `b` to value `4`, and `c` to value `1` as follows.

```
Env env0 = new EmptyEnv();
Env env1 = new ExtendEnv(env0, "a", new NumVal(3));
Env env2 = new ExtendEnv(env1, "b", new NumVal(4));
Env env3 = new ExtendEnv(env2, "c", new NumVal(1));
```

Notice that since an empty environment marks the end of the list, each variable lookup that is not found in the environment will result in `LookupException`. We can also have two mappings for the same name in a chain of environments such as in the environment `env4` below that has two mappings for the name "c".

```
Env env4 = new ExtendEnv(env3, "c", new NumVal(2));
env4.get("c"); //Result: new NumVal(2)
```

```
env4.get("b"); //Result: new NumVal(4)
env4.get("foo"); //Result: LookupException("No binding found for: foo")
```

We can lookup some names in this created environment as shown above. For each lookup the reader is encouraged to trace through the listings for extended environment and empty environment to understand the exhibited behavior noted in comments.

Exercise

3.7.1. *[ExtendEnvList]* Design and implement a new kind of environment `ExtendEnvList` that also implements the `Env` interface presented in this section. Unlike `ExtendEnv`, this kind of environment should allow storing multiple name-to-value mapping.

3.7.2. *[EmptyEnvNice]* Design and implement a new kind of empty environment `EmptyEnvNice` that also implements the `Env` interface presented in this section. Unlike `EmptyEnv` that throws an exception when a name, say "`x`" is looked up, this kind of environment extends itself to add a mapping from "`x`" to the default value "`0`". With this environment, could we distinguish between names that were given the default value "`0`" from undefined names?

3.8 Environment-passing Interpreters

Now since we have a datatype for representing environments, we can use it in our evaluator. Since environments can change when new variables are defined, and when the scope of variable definitions end, we need to pass around environments from parent expressions to subexpressions during expression evaluation. In this section, we enhance the interpreter of Arithlang with this functionality.

Value of a program in an environment

First things first, when a program starts running, what is the value of the environment?

Recall that a program consists of an expression, and previously we have stated that the semantics of a program as "the value of a program is the

value of its component expression." Let `Program` be the set of all programs in Varlang, and `Exp` be the set of all expressions in Varlang. Also, let `p` be a program, i.e it is in set `Program` and `e` be an expression, i.e. it is in set `Exp` such that `e` is the inner expression of `p`. With these assumptions, we can write the statement "value of a program is the value of its expression" more precisely as the following mathematical relation, where `Program` is the set of all programs, `Exp` is the set of all expressions, and `Value` is the set of all values.

VALUE OF PROGRAM
$$\frac{\texttt{value e = v}}{\texttt{value p = v}}$$

In the presence of environments, this changes sligtly. We will state the semantics of a program in Varlang as "In an environment `env`, the value of a program is the value of its component expression in the same environment `env`." Here, `env` $\in$ `Env` the set of all environments.

VALUE OF PROGRAM
$$\frac{\texttt{value e env = v}}{\texttt{value p env = v}}$$

To illustrate consider the program `(+ x y)`. Here, `p` is `(+ x y)` and `e` is an addition expression with two subexpressions `x` and `y`. Now, imagine that this program is being run in an environment `env` that maps `x` to a value `300` and `y` to a value `42`. The semantics above says that the value of `p` in `env` is going to be the same as that of `e` computed using the same `env`.

The case of program in the Evaluator (figure 3.9) implements this new meaning. Notice that the same environment `env` is passed forward to evaluate the component expression.

Intuitively, when a program starts running, no variables have been defined yet. Therefore, we can start evaluating every program in an empty environment. This intended semantics is modeled on line 3 in figure 3.9 and defined below.

VALUE OF PROGRAM
$$\frac{\texttt{value e env = v, where env} \in \texttt{EmptyEnv}}{\texttt{value p = v}}$$

The reader is encouraged to think about environments that predefine certain constants for ease of programming. We will revisit this topic in chapter 4.

```
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 }
```

Figure 3.9: Evaluating a Program to Value in the Varlang Language.

Recall from section 3.5 that to pass around environments, signatures of `accept` methods for each AST node were extended to take the environment as an extra parameter. Similarly, signatures of `visit` methods for the case of each AST node were extended to take the environment as an extra parameter.

Expressions that do not directly change environment

Some expressions do not depend on the environment, e.g. `NumExp`.

VALUE OF NUMEXP
`value (NumExp n) env = (NumVal n)`

Some other expressions do not change the environment directly. They simply pass forward environments to their subexpressions. To illustrate consider the case for addition expression.

VALUE OF ADDEXP

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

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.

To illustrate consider the addition expression `(+ x y)` with two subexpressions `x` and `y`. Now, imagine that this expression is being evaluated in an environment `env` that maps `x` to a value `300` and `y` to a value `42`. The semantics above says that the environment in which we will compute the value of `x` and that of `y` should be the same as the environment of the parent expression `(+ x y)`.

```
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);
}
```

The semantics and implementation of subtraction, multiplication, and division expressions are similar to the addition expression. The reader is encouraged to review concrete implementation of these expressions in the companion code before proceeding further.

Interesting cases that directly interact with the environment are variable and let expressions.

3.9 Value of a Var Expression

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`.

The value relation below models this semantics.

VALUE OF VAREXP

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

It says that the value of a variable expression is the value obtained by looking up that variable name in the current environment.

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

The case for variable expression in the evaluator implementation also realizes this semantics as shown above.

3.10 Value of a Let Expression

A let expression is the only expression in Varlang that changes the environment by adding new names to it. First, consider a let expression that can only define a single name-to-value mapping. The meaning of this expression can be given as:

VALUE OF LETEXP

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

First and foremost, this definition says that the value of a let expression is the value of its body `exp'` obtained in a newly constructed environment `env'`. This new environment isn't just any environment. It is obtained by extended the original environment of the let expression `exp` with new bindings (variable name to value mapping). In particular, we are mapping the name `var` to the value of `exp` that we have previously computed.

Notice that besides variable definition a let expression is also serving to combine two expressions `exp` and `exp'` into a larger expression. Therefore, it serves both as a *means of combination and abstraction* in our Varlang programming language.

We can now extend this definition to support the full let expression that allows defining several variables.

VALUE OF LETEXP

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

Similar to the case of a single definition, value of a let expression is the value of its body expression $\texttt{exp}_b$ in a new environment $\texttt{env}_{k+1}$, which

is created by extending the original environment `env` with bindings from variables $\texttt{var}_i$ to their value $\texttt{v}_i$ obtained by evaluating them.

We have considered several examples of let expressions in previous sections. For the purpose of the discussion in this section consider the let expression `(let ((a 3)(b 4)(c 2)) (+ a b c))`. For this let expression names that are being defined are "a", "b", and "c", and they are being defined to have the values of expressions "3", "4" and "2" which are all constant expressions. In other words, names "a", "b", and "c" are defined to have values `new NumVal(3)`, `new NumVal(4)`, and `new NumVal(2)`. The body of the let expression is `(+ a b c)`.

Against this background, consider the implementation of the `LetExp` case which models the semantics of the let expression.

```
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);
}
```

In summary, to support variable definition and usage we included a new abstraction environment, added support for variable and let expressions in the read phase, supported new AST nodes to store variable and let expressions, changed the evaluator to pass around environments from parent expressions to subexpressions, and added new cases for variable and let expressions in the evaluator. Those are the only components of the interpreter that need to be modified to provide support for the Varlang programming language.

We can play with this interpreter to see how these features works. In the *interaction log* below $ is the prompt of the interpreter, text after $ is

the program that the user writes and the text on the next line is the value of this program.

```
$ (let ((a 3)) a)
3
$ (let ((a 3) (b 4)) a)
3
$ (let ((a 3) (b 4) (c 2)) (+ a b c))
9
$ (let ((a 3)) (let ((a 4)) (let ((a 2)) a)))
2
```

Exercise

3.10.1. *[Initial Environment]* Extend the Varlang programming language such that all programs start running in a non-empty environment that predefines roman numerals for 1 - 10 to their numeric value. Following interaction log illustrates the intended semantics of this change.

```
$ i
1
$ v
5
$ x
10
```

3.10.2. *[Expressive Environment]* Extend the environment interface and data types to define following additional functionalities.

1. A predicate method `isEmpty` that returns `true` when the environment is empty and false otherwise.
2. A predicate method `hasBinding` that accepts a variable name and returns `true` when the environment has binding for that name and false otherwise.

3.10.3. *[isFree]* A variable occurs free in an expression, if it is not bound by an enclosing `let` expression. Implement this predicate as a method `isFree` for each AST node such that given a variable name of type `String`, the method returns `true` if that name occurs free in that AST node, and false otherwise.

3.10.4. *[Substitution-based Let Expression]* Extend the Varlang programming language from previous question to implement a *substitution-based variation* of the let expression, say `lets` expression. Recall that a substitution-based semantics works as follows. The value of `(lets ((x 3)(y 4)) (+ x y))` is the value of a new expression created from the original let body `(+ x y)` by replacing `x` with `3` and `y` with `4`. According to the substitution-based semantics the value of the let expression is the value of `(+ 3 4)` that is `7`.

- The grammar of this new language feature should be exactly the same as the grammar of the `let` expression in the Varlang language, except for the keyword **lets**.
- Implement substitution as a method `subst` for each AST node such that given a list of variable names, and a list of values the `subst` method returns a copy of current AST node with each free variable name substituted with corresponding value.

3.10.5. *[Unique Let Expression]* Notice from the semantics and from the implementation of the let expression that Varlang doesn't place any restriction on defining the same variable two or more times in the same let expression. In fact, a variable can be defined any number of times and only the rightmost definition would have effect in the body of the let expression. So the program `(let ((a 3) (a 4) (a 2) (a 342)) a)` would give the answer `342`. Modify the semantics of the Varlang programming language so that all variable names defined in a let expression must be unique.

3.10.6. *[Disallow Hole in Scope]* Current semantics of the let expression in the Varlang language allows variable definitions to create a hole in the scope of the outer definition. Modify the semantics of the Varlang programming language so that redefinition of variables is prohibited and results in a dynamic error.

3.10.7. *[Encrypted Environment]* In some operating systems when programs deallocate memory pages those memory pages can be allocated to other programs without any changes. In such cases, other malicious programs can read the value stored in memory pages, which can cause information leakage. To avoid such information leak due to environment storage we can augment the Varlang language with a `lete`

(encoded let) expression that encodes the value before storing in environment and `dec` expression that decodes it prior to using it. Extend the Varlang programming language to support these two expression. Implement an encrypted let (lete for let encrypted), which is similar to let but takes an additional parameter key and a `dec` expression that is similar to `VarExp`. All values are stored by encrypting them with key, and read by decrypting them with key.

```
> (lete 42 ((x 1)) x)
43

> (lete 42 ((x 1)) (dec 42 x))
1

> (lete 10 ((y 12)) y)
22

> (lete  10 ((y 12)) (dec 10 y))
12
```

3.11 Further Reading

Some recent languages have added the notion of implicit variable definition, i.e. the variable is defined implicitly, given a type and an initial value dependent of its usage. Such definitions are convenient, but they can complicate program understanding. Consider adding such variables to Varlang.