Prof. Jingke Li (FAB120-06, lij@pdx.edu), Tue&Thu 12:00-13:15 @SH 212, Lab: Fri 10:00-11:15/11:30-12:45 @EB 103

Lab 8: Variables and Environments

Learning Objectives Upon successful completion, students will be able to:

- Understand the use of environments for propagating program information over an AST.

(*Note:* This lab is adapted from Prof. Andrew Tolmach's earlier version.)

Preparation

Download and unzip the file `lab8.zip`. You'll see a set of sub-directories, `00-03`. We'll walk through many or all of these directories, in sequence. Some directories contain exercises.

00. Expression ASTs

We'd like to see how the tree traversal methods from last week's lab apply to ASTs. We start with a simple language of expressions involving numeric literals, addition, and subtraction, each represented by a different sub-class of class `Exp`. We define an `eval()` function that evaluates an expression tree to an integer; this would be suitable for use in an interpreter for the language.

"from 00/Example.java"

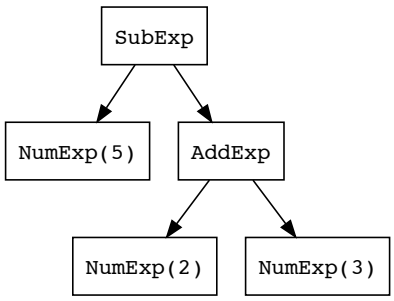
```
7  abstract class Exp {
8    abstract int eval();
9  }
10
11 class NumExp extends Exp {
12   private int num;
13   NumExp(int num) {this.num = num;}
14
15   int eval() { return num; }
16 }
17
18 class AddExp extends Exp {
19   private Exp left;
20   private Exp right;
21   AddExp (Exp left, Exp right) {this.left = left; this.right = right;}
22
23   int eval() { return left.eval() + right.eval(); }
24 }
25
26 class SubExp extends Exp {
27   private Exp left;
28   private Exp right;
29   SubExp (Exp left, Exp right) {this.left = left; this.right = right;}
30
31   int eval() { return left.eval() - right.eval(); }
32 }
```

`eval()` is implemented as just another recursive descent tree traversal, computing its answer bottom-up. In attribute grammar terminology, it corresponds to a synthesized attribute *val*, with the following informal grammar and attribute equations:

$exp \rightarrow num$	$exp.val := num.val$
$exp \rightarrow exp + exp$	$exp.val := exp_1.val + exp_2.val$
$exp \rightarrow exp - exp$	$exp.val := exp_1.val - exp_2.val$

Here we assume that *num* nodes already have a pre-set *val* attribute; in a compiler, this would probably be attached to the numeric literal token returned by the lexical analyzer. As in the previous examples, the code doesn't actually store the *val* attribute at each node, because we are only interested in its value at the root.

The following AST, which corresponds to the definition of `e` in the program shown on the right, should evaluate to 0.



"from 00/Example.java"

```
36    Exp e = new SubExp(new NumExp(5),
37                       new AddExp(new NumExp(2),
38                                  new NumExp(3)));
```

Exercise: Adding a Conditional Expression Extend the expression language with support for `ifnz` expressions and their evaluation. An `ifnz` expression has three sub-expressions *test*, *nz*, and *z*; its concrete syntax might be written (*test* `?` *nz* `:` *z*) to match the conditional expression found in Java, C, and C++. If *test* evaluates to 0, the `ifnz` expression as a whole evaluates to *z*; otherwise it evaluates to *nz*.

For example, the expression `((5-(2+3)) ? 10 : (21+21))` should evaluate to 42.

Hint: You should be able to add support for `ifnz` nodes without changing any of the existing code (except that you'll want to change the test tree).

01. Adding Variables and Environments

Consider what happens when we add *variables* to our expression language. Variables are identifiers (represented here as `Strings`) that carry an associated (integer) value.

- To define variables and give them values, we use a new `LetExp` expression node, which has an associated variable *x* and two sub-expressions *d* and *e*; its concrete syntax might be written `let` *x* `=` *d* `in` *e*. It is evaluated by first evaluating *d*, *binding* the resulting value to variable *x* and then evaluating *e*, whose value becomes the result of the entire `let` form.

  A key point is that the *scope* of *x*, i.e., the part of the program where its definition is visible, is just the sub-expression *e*. Outside the overall `let` expression (and also in *d*) the definition of *x* is not visible.

- Uses of variables become another kind of leaf node (`VarExp`) in expressions. A variable evaluates to its value. In this langauge, the value of a variable cannot be changed after its initial definition (so in fact "constant" might be a better name than "variable.") By arbitrary convention, reference to an undefined variable evaluates to 0.

- To implement the `eval()` function for this language, we give it an *environment* parameter `env`. The environment is a map from variable names to their values. `VarExp` nodes use the environment to look up their value; `LetExp` nodes compute an extended environment with a new binding, which they

pass down to their e subexpression. All other nodes just pass the environment unchanged to their subexpressions.

"from 01/Example.java"

```
25 abstract class Exp {
26   abstract int eval(Env env);
27 }
28
29 class LetExp extends Exp {
30   private String x;
31   private Exp d;
32   private Exp e;
33   LetExp(String x, Exp d, Exp e) {this.x = x; this.d = d; this.e = e;}
34
35   int eval(Env env) {
36     int v = d.eval(env);
37     return e.eval(env.extend(x,v));
38   }
39 }
40
41 class VarExp extends Exp {
42   private String x;
43   VarExp(String x) {this.x = x;}
44
45   int eval(Env env) {
46     Integer v = env.get(x);
47     if (v != null)
48       return v;
49     else
50       return 0;  // default value for undefined variables
51   }
52 }
```

Not surprisingly, the cleanest treatment of environments is again based on immutable maps. The class `Env` defines a type of immutable maps, built on top of the Java library's `HashMap`.

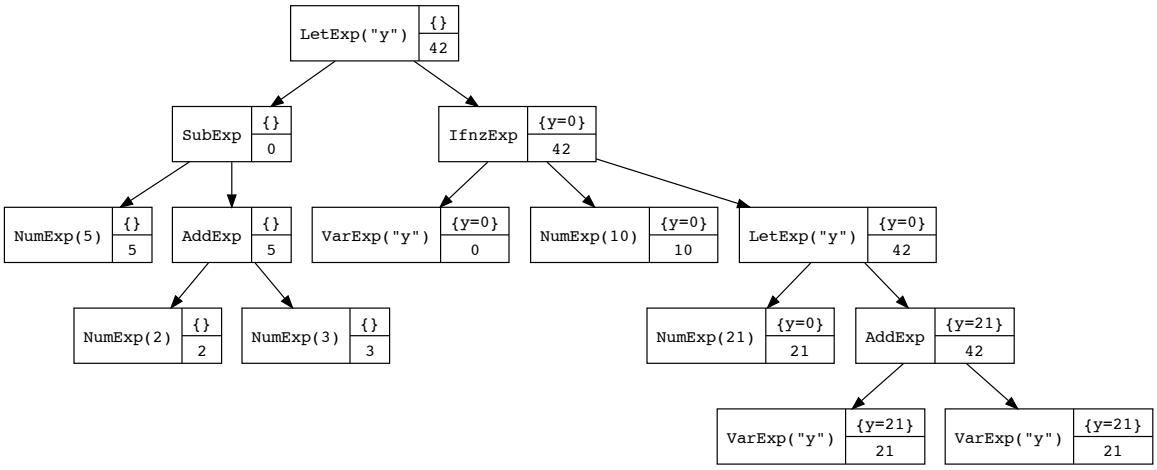
"from 01/Example.java"

```
15 class Env extends HashMap<String,Integer> {
16   Env extend(String s, Integer v) {
17     Env e = new Env();
18     e.putAll(this);
19     e.put(s,v);
20     return e;
21   }
22 }
```

From an attribute grammar perspective, `env` is an *inherited* attribute. We specify this attribute, and give equations for the *val* attribute on the new expression forms, as follows:

$exp \rightarrow$ `let` x `=` exp `in` exp	$exp_1.env := exp.env;\ exp_2.env = extend(exp.env, x, exp_1.val)$ $exp.val := exp_2.val$
$exp \rightarrow x$	$exp.val := lookup(exp.env, x)$
$exp \rightarrow exp + exp$	$exp_1.env := exp.env;\ exp_2.env := exp.env$
$exp \rightarrow exp - exp$	$exp_1.env := exp_2.env := exp.env$ (a shorthand)

It can be enlightening to annotate some example expression trees with the environments and values computed for each node. Here is the test tree from the example file with such annotations. (It can be even more useful to trace the order in which the computations are made and the annotations filled in...)



Exercise: Evaluating a New Expression Add a new testing expression:

```
let x = (let x = 2 in x + x) in (x ? 20 : 30)
```

to the program, and evaluate its value.

02. Imperative Environments

Environment management is a key part of many operations over ASTs, including static analysis, interpretation, code generation, and optimization. Moreover, real compilers make use of environments not just to keep track of variables, but also to manage many other kinds of identifiers, including function names, type names, module names, record field names, etc. Thus environments can get quite large, so handling them efficiently is important. Although it is possible to build quite efficient versions of immutable maps, mutable maps accessed using imperative operations can be more efficient still. Historically, most compilers have used mutable maps and imperative operations to manage environments, which are often called *symbol tables*. Unfortunately, in many languages name binding has characteristics that make imperative implementation complicated. In particular, the limited *scope* of bindings means that we cannot just *add* bindings into a single global environment: we must also *remove* them at the appropriate point in the tree traversal. Moreover, the possibility of shadowing means that removal must be done with care.

This example shows a possible design for an imperative environment implementation that supports retraction of bindings when exiting from a scope.

Instead of passing an (immutable) environment as a parameter to the `eval()` function, we use a single global environment declared as a `static` variable of class `Exp`.

"from 02/Example.java"

```
39 abstract class Exp {
40   static Env env = new Env();
41   abstract int eval();
42 }
```

The evaluator for `LetExps` extends the environment with its binding before evaluating the e subtree, and then explicitly retracts the bound name from the environment again afterwards.

"from 02/Example.java"

```
43
44 class LetExp extends Exp {
45   private String x;
46   private Exp d;
47   private Exp e;
48   LetExp(String x, Exp d, Exp e) {this.x = x; this.d = d; this.e = e;}
49
50   int eval() {
51     int v = d.eval();
52     env.extend(x,v);
53     v = e.eval();
54     env.retract(x);
55     return v;
56   }
57 }
```

We cannot just implement `retract` by removing the name from the environment; there may be an outer binding of the same name that was temporarily shadowed but should now be visible again. To handle the possibility of multiple bindings of the same name, the entries in the environment map are now *stacks* of values. If a name gets rebound in an inner scope, it is pushed onto the stack; the top of the stack is always used as the correct "current" binding; retraction just pops the stack, leaving any outer bindings untouched. We maintain the invariant that the map never contains an empty stack. We assume that `retract` is only called on currently bound identifiers.

"from 02/Example.java"

```
13 class Env extends HashMap<String,Stack<Integer>> {
14   void extend(String s, Integer v) {
15     Stack<Integer> stack = this.get(s);
16     if (stack == null) {
17       stack = new Stack<Integer>();
18       this.put(s, stack);
19     }
20     stack.push(v);
21   }
22
23   void retract(String s) {
24     Stack<Integer> stack = this.get(s);  // should never be null
25     stack.pop();
26     if (stack.empty())
27       this.remove(s);
28   }
29
30   Integer lookup(String s) {
31     Stack<Integer> stack = this.get(s);
32     if (stack == null)
33       return null;
34     return stack.peek();
35   }
36 }
```

Exercise: Tracing the Environment Insert print statements in methods `extend()` and `retract()` to show the before and after `Env` contents. For instance, a possible output for the `Example` program would be:

```
linux> java Example
extend(y,0): {} => {y=[0]}
```

```
extend(y,21): {y=[0]} => {y=[0, 21]}
retract(y): {y=[0, 21]} => {y=[0]}
retract(y): {y=[0]} => {}
```

(*Hint:* Java `HashMap`'s content can be displayed by just printing the object itself.)

03. Adding Dynamically Typed Values

This example extends the toy language presented in previous sections by adding *boolean* values and operators. In particular:

- Values can now be either integers (represented using objects of Java's `Integer` library wrapper class) or boolean (represented using objects of the `Boolean` library wrapper class). The return type of `eval` is now declared as `Object`, allowing either kind of value to be returned; similarly, environments are now `Map`s from `String`s to arbitrary `Object`s.
- (Using `Object` in this way makes the Java type signature of `eval` very uninformative: the body for `eval` will match that signature even if it returns something entirely wrong, like a `String` or an array! An alternative would be to define a new class `Value` with sub-classes `IntValue` and `BoolValue`. This would let us write more precise Java types, but at the expense of writing several new class definitions that largely replicate what is already in the library types, and losing the advantages of Java's autoboxing and unboxing, which are used heavily in this code (where?). Here we preferred the convenience of using the existing classes; this kind of trade-off is quite common.)
- We choose to treat the boolean literals "true" and "false" as pre-defined variables, placed in the environment at line 151 before evaluation of the top-level expression. An alternative would be to make them expression forms of their own, analogous to `NumExp`s. Both these approaches are common in real-world languages. The approach we've picked keeps the language smaller and is a bit simpler to implement, but usually has the consequence that `true` and `false` can be *redefined* in inner scopes, which could lead to a lot of programmer confusion!
- We add new expression forms `AndExp` and `NotExp` as sample boolean operations. (Question: Is this new `And` expression "short-circuiting"?) These two forms expect their operands (subtrees) to evaluate to `Boolean`s, so they downcast these values accordingly. But of course, these downcasts can fail, since nothing stops us writing a nonsense expression such as `AndExp(VarExp "true",NumExp 2)`. To handle such errors gracefully, we catch the low-level built-in `ClassCastException` that Java throws when a downcast fails. (An alternative would be to use `instanceof` to check the subexpression values before downcasting them.) A failure is converted into a polite `RuntimeError` exception with an explanatory message; the main program catches these exceptions and displays the message before halting.
- The existing `eval` code for arithmetic operations has to be modified in a corresponding way, to downcast the values of subtrees to integers. We also modify the `If` expression to expect a boolean rather than an integer as its test expression. Note that the "then" and "else" arms of an `If` can compute values of different types, as illustrated by the test tree `e` in the main program.
- Now that we have a facility for producing checked runtime errors, we change `VarExp` to throw such an error when it encounters an undefined variable, rather than just returning the default value 0.

Exercise: Adding an Leq Operator You may have noticed that language of boolean expressions is rather impoverished, because there is no way to produce a boolean value other than the built-in `true` and `false` variables.

Remedy this situation by extending the language with a less-than-or-equal-to (<=) expression form. This operator should expect two integer operands and return a boolean result. Modify the test expression `e` in `main` to test this new expression type.