Expressions and Environments: `envlab`

CS 321 Languages and Compiler Design I, Winter Term 2015
Department of Computer Science, Portland State University*

Learning Objective

Upon successful completion, students will be able to understand and extend environment-based interpreters and type checkers for simple languages using local computations over ASTs.

Instructions

Download and unzip the file `envlab.zip`. You'll see a set of sub-directories, 08–16. (The numbering reflects the fact that the examples in these directories repeat (08-10) and then continue on (11-16) from the closing steps in the previous "`treelab`".) Each contains a Java program called `Example.java`; some directories contain other Java files as well. To compile and run the contents of each directory, `cd` to it and type

```
$ javac Example.java
$ java Example
```

We'll walk through many or all of these directories, in sequence. Some directories contain exercises, with solutions hidden in a subdirectory.

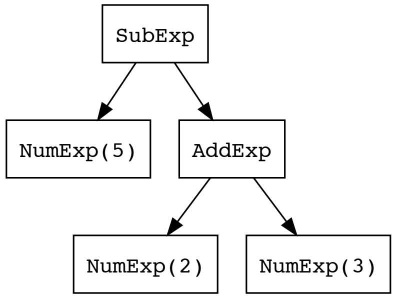
As some of the later code examples are now getting quite large, they are not folded into this document, but you should still look at the `Example.java` files while reading the notes associated with each one.

Expression ASTs (08)

In the previous lab, we discussed for methods for traversing and computing values of attributes on different types of tree structure. Now let's see how these methods 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.

- The following AST, which corresponds to the definition of `e` at lines 35–37, should evaluate to 0.

*The original version of this lab was developed by Andrew Tolmach; the modified version presented here was developed by Mark Jones, who takes responsibility (and apologizes) for any errors or infelicities that you may find!



- `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:

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

  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.

- Many other computations on ASTs can be implemented using a similar structure.

Exercise: Adding a Conditional Expression (09)

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).

Adding Variables and Environments (10)

Now consider what happens when we add *variables* to our expression language.

- Variables are identifiers (represented here as `String`s) that carry an associated (integer) value.
- Uses of variables become another kind of leaf node (`VarExp`) in expressions. A variable evaluates to its 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.

- The value of a variable cannot be changed after its initial definition (so in fact "constant" might be a better name than "variable.")
- 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.
- It is possible for the same variable name to be bound more than once. If one of these bindings is nested within the e part of another, the inner binding takes precedence. (We say that the inner binding *shadows* the outer one.)
- By arbitrary convention, reference to an undefined variable evaluates to 0.
- Here are a number of small examples of `let` expressions, each of which evaluates to 42:

```
let x = 21 in
    x + x
```

```
let x = 20 in
    let y = x + 2 in
        x + y
```

```
(let x = 1 in
     x + x) +
(let y = 20 in
     y + y)
```

```
let x = 20 in
    let x = 21 in
        x + x
```

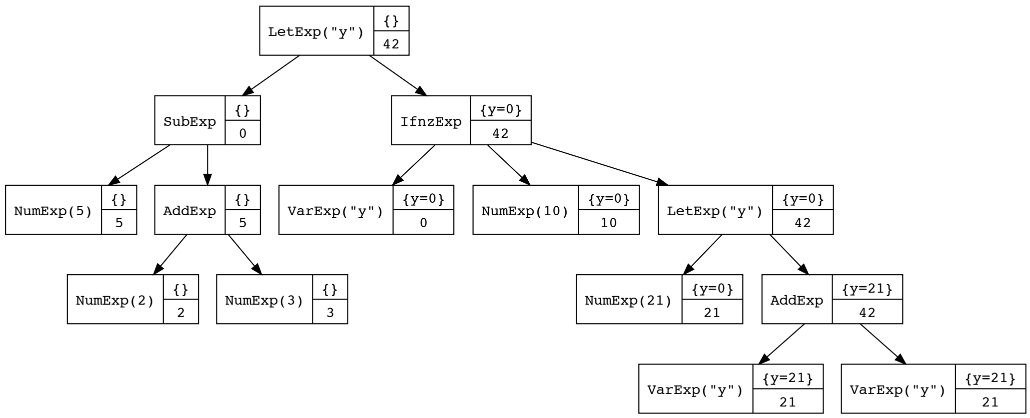
```
let x = 20 in
    let x = x + 1 in
        x + x
```

```
let x = 40 in
    (let x = 1 in
         x + x) +
    x
```

- This is just about the simplest way to introduce a variable declaration and initialization mechanism into a language. (`let` expressions occur in real functional languages such as Scheme, ML and Haskell.) Imperative languages, in which variables refer to mutable locations that themselves contain values, can be modeled in similar, but more complicated, ways.
- 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.
- One of the simplest ways to implement environments is to use a linked list (or what, in the particular form that we use here, is sometimes called an *association list*). The code in `ValueEnv` meets our need for this. Environment management is an 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. Our `ValueEnv` implementation may not be a very efficient implementation for work with large programs but it is simple and good enough for present purposes.
- 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...)



Adding Dynamically Typed Values (11)

This example extends the toy language presented in Example 10 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 lines 131–132 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 (12)

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.

Adding Static Types (13)

In this example, we introduce a notion of *static* types and provide code for *typechecking* an expression before attempting to evaluate it. The goal of typechecking is to convert potential (checked) runtime errors into *static errors*, which can be reported to the programmer early in the development cycle, before the code is run (or released to a customer!)

- We define an abstract class `Type` to describe static types, with two sub-classes, `IntType` and `BoolType`. The only important operation on types is `equals`. (Because these types have no instance variables, there is no real point in creating multiple instances of them, so by convention we create just one instance of each and use the `static` names `Type.INT` and `Type.BOOL` to refer to these.)
- We introduce a new method, `check`, which traverses the expression to determine if it is well-typed, and, if so, returns the corresponding `Type`. If the expression is not well-typed, `check` throws a `StaticError` exception.
- As in most static typing systems, we attach a definite type to each variable and to each expression. We insist that all definitions and uses of a variable are consistent with its type. To keep track of the types of variables, the `check` method is parameterized by a `TypeEnv` environment that maps variable names to types: this environment is extended when a variable is bound in a `LetExp` and consulted when a variable used in a `VarExp`. The `TypeEnv` and V̂alueEnv implementations that we have used here are very similar. For a more sophisticated approach, it might make sense for these to be implemented as instances of a generic class. We choose instead to present them as completely separate structures, in part to reinforce the separation between static, or compile time uses of environments (like `TypeEnv`) and dynamic, or runtime uses of environments (like `ValueEnv`).
- From an attribute grammar perspective, we can specify the typechecker using a synthesized attribute *typ*, an inherited attribute $tenv$, and the following equations:

$exp \to$ `let` x `=` exp `in` exp	$exp_1.tenv := exp.tenv;\ exp_2.tenv = extend(exp.tenv, x, exp_1.typ)$ $exp.typ :=$ *if* $exp_1.typ = ERROR$ *then* $ERROR$ *else* $exp_2.typ$
$exp \to x$	$exp.typ := lookup(exp.typ, x)$
$exp \to num$	$exp.typ := INT$
$exp \to$ `if` exp `then` exp `else` exp	$exp_1.env := exp_2.tenv := exp_3.tenv := exp.tenv$ $exp.typ :=$ *if* $exp_1.typ = BOOL$ *and* $exp_2.typ = exp_3.typ$ *then* $exp_2.typ$ *else* $ERROR$
$exp \to exp + exp$	$exp_1.tenv := exp_2.tenv := exp.tenv$ $exp.typ :=$ *if* $exp_1.typ = exp_2.typ = INT$ *then* INT *else* $ERROR$
$exp \to exp - exp$	$exp_1.tenv := exp_2.tenv := exp.tenv$ $exp.typ :=$ *if* $exp_1.typ = exp_2.typ = INT$ *then* INT *else* $ERROR$
$exp \to exp$ `&&` exp	$exp_1.tenv := exp_2.tenv := exp.tenv$ $exp.typ :=$ *if* $exp_1.typ = exp_2.typ = BOOL$ *then* $BOOL$ *else* $ERROR$
$exp \to$ `!`exp	$exp_1.tenv := exp.tenv$ $exp.typ :=$ *if* $exp_1.typ = BOOL$ *then* $BOOL$ *else* $ERROR$

(Compare these to the attribute equations for evaluation given in Examples 08 and 10.) We assume that the inherited value of $tenv$ at the root of a tree representing a whole program is $\{\texttt{true} \mapsto BOOL, \texttt{false} \mapsto BOOL\}$. Notice that we describe errors by allowing the typ attribute to take on a special value $ERROR$, which then must be explicitly propagated through the equations; there is no direct attribute grammar equivalent of throwing an exception.

- For comparison, here is a presentation of the same typing rules as formal judgments $TE \vdash e : t$ in the style presented in lecture, where TE represents a typing environment, or a `TypeEnv` value in our implementation.

$$\frac{TE \vdash e_1 : t_1 \quad TE + \{x \mapsto t_1\} \vdash e_2 : t_2}{TE \vdash \texttt{let } x \texttt{ = } e_1 \texttt{ in } e_2 : t_2}\ \text{(Let)} \qquad \frac{TE(x) = t}{TE \vdash x : t}\ \text{(Var)}$$

$$\frac{}{TE \vdash \mathit{num} : \mathit{int}}\ \text{(Num)} \qquad \frac{TE \vdash e_1 : \mathit{bool} \quad TE \vdash e_2 : t \quad TE \vdash e_3 : t}{TE \vdash \texttt{if } e_1 \texttt{ then } e_2 \texttt{ else } e_3 : t}\ \text{(If)}$$

$$\frac{TE \vdash e_1 : \mathit{int} \quad TE \vdash e_2 : \mathit{int}}{TE \vdash e_1 \texttt{ + } e_2 : \mathit{int}}\ \text{(Add)} \qquad \frac{TE \vdash e_1 : \mathit{int} \quad TE \vdash e_2 : \mathit{int}}{TE \vdash e_1 \texttt{ - } e_2 : \mathit{int}}\ \text{(Sub)}$$

$$\frac{TE \vdash e_1 : \mathit{bool} \quad TE \vdash e_2 : \mathit{bool}}{TE \vdash e_1 \texttt{ \&\& } e_2 : \mathit{bool}}\ \text{(And)} \qquad \frac{TE \vdash e : \mathit{bool}}{TE \vdash \texttt{!}e : \mathit{bool}}\ \text{(Not)}$$

Again, we assume that the TE for an expression representing a whole program is $\{\texttt{true} \mapsto \mathit{bool}, \texttt{false} \mapsto \mathit{bool}\}$. Note that in this presentation, places where the typechecker implementation throws an exceptions correspond to *missing rules*, i.e., situations where no valid typing judgment can be constructed.

- Observe the close correspondence between the code for `eval` and the code for `check` in each expression class, and between the contents of the corresponding environments at each point. In fact, if you think of types as very coarse *approximations* of values (i.e., all integer values are approximated by type `INT` and all boolean values by type `BOOL`) then you can view `check` as just another kind of evaluation that works over the approximations rather than the real values.

- But an important difference between checking and evaluation is that the former must explore *every* possible path through the expression, whereas the latter explores only one. This difference is exposed most clearly by the checking code for `IfExps`. Whereas the `eval` code evaluates either the `t` or `f` sub-expression, the `check` code checks *both* sub-expressions, to make sure that no errors will occur no matter which way the conditional branches at runtime. (Question: is there another expression in the language for which `check` explores more of the tree than `eval`?)

- Moreover, the `check` method of `IfExp` requires that the `t` and `f` sub-expressions have the *same* type. This ensures that it is possible to give a single definite type to the value of the whole `IfExp`, regardless of which way the conditional branches. But it means that `check` will reject some programs that would in fact `eval` without error, e.g. the test expression `e` in the `main` method of Example 11. (Try it!) This is one of the inherent drawbacks of static checking: like most static analyses, a checker must *approximate* the program's runtime behavior, and it usually cannot do so precisely. If we design the checker to ensure that every program it accepts is free of runtime errors, we will typically also have to let it reject some programs that actually have no runtime errors. That is, using a checker effectively reduces the number of legal programs we are allowed to write in the language.
- Another drawback of static typing disciplines is that they often require the programmer to specify type information in the text of the program, e.g., by declaring the types of variables before they are used. This increased overhead for the programmer is modest for large, long-lived code (where, moreover, type declarations are independently useful as a form of internal documentation). But it is harder to justify for short, "run once" programs, e.g. for scripting tasks, where speed of coding is of paramount importance. Our example code does not yet illustrate this overhead, because we are able to *infer* the types of variables bound in `LetExp`s from their initializing expressions. (Indeed, type inference is often used in real-world languages to lower the declaration burden for the programmer.) But we will see this point arise again in a later example.
- Finally, one of the major *benefits* of using a statically typed language is that it can make programs run faster because there is no need to perform runtime checks; moreover, it is often possible to use more compact and efficient runtime representations for values when their types do not need to be checked. But our example code does not illustrate this benefit: because we are interpreting our little language *within* the strongly typed Java language, we cannot easily avoid performing checks again in `eval`, even though we know that they are unnecessary for any expression that has successfully passed `check`. The performance benefits of knowing types statically will become clearer when we study code generation in CS 322.

Exercise: Typing the Leq Operator (14)

Extend your solution to Example 12 to include type-checking code for the less-than-or-equal expression. Hint: As usual, `eval` and `check` have very similar structure! Test your checker on both valid and invalid example programs.

Also write down an attribute grammar rule and a typing judgment for this expression in the style shown above for Example 13.

Adding Functions (15)

This example illustrates a few of the issues that arise when we scale up the evaluator and checker to include more elaborate features, in this case a mechanism for defining and calling top-level functions.

- We now have several different *syntactic classes* (kinds of entities) in the AST. A `Program` consists of a set of `Function` definitions and a top-level `Exp` representing what the program as a whole should compute.

- Each `Function` is represented by its name, a (single) formal argument, the argument type, a body, which is an `Exp`, and the return type (i.e. the type of the body). A function is called using a `CallExp`, which specifies the function name and an expression for the actual argument value to pass.
- We choose to keep the *name space* of functions separate from that of variables, i.e. it is possible for a function and a variable to have the same name without clashing or shadowing each other. (We can do this because variable names and function names never appear in the same position in the AST.) Functions are stored in a separate `FunctionEnv` environment mapping `String`s to `Function` AST nodes. (This further variant on our previous `ValueEnv` and `TypeEnv` classes provides an even more compelling argument for the use of a generic type!) This environment is built just once, by the `Program` constructor, when the AST is first built. To avoid duplication of information, we actually store the function environment as well as the list or array of `Function`s in the `Program` node itself. The function environment is passed to both `check` and `eval` methods, since it contains all the information needed for either typing or evaluation of the function. (Because the environment never changes, we could have made it a global variable instead of passing it down everywhere, but the class structure we've been using doesn't provide a convenient spot to define such a variable.)
- To evaluate a `CallExp`, we look up the function by name in the function environment, evaluate the argument expression, and pass the result to the function's `eval` method, which evaluates the function's body in an environment that binds (just) the function argument to its value.
- To typecheck a `CallExp`, we look up the function by name in the function environment, calculate the type of the argument expression, and make sure that its type matches the function argument's expected type; if so, we produce the function's return type.
- To typecheck a program, we must typecheck all the function definitions as well as the top-level expression. Typechecking a function definition just means checking that its body expression has the declared return type, in an environment containing (just) the function argument, bound to its declared type. There is just one subtlety here. All functions are in scope everywhere; they can be recursive or mutually recursive. This means that in order to typecheck a function body, we need to know the types of all functions already! Fortunately, we only need to know their *declared* types, which are recorded in the function environment; ultimately, we will check that every function definition does indeed match its declared type. Incidentally, this is why we need to include the return type as an explicit part of the function definition, rather than just inferring it from the body expression as we do with `let` expressions. (We also give the function argument's type explicitly, in order to be able to typecheck the body in isolation from its calls. In principle, we could infer the argument's type from how the argument variable is used within the body expression, and some languages do this.)

Exercise: Adding Pairs (16)

The simple language of Example 15 lacks any way to build data structures. Remedy this by extending the language, evaluator, and typechecker to include *pair* values. A pair is a very simple kind of record with two fields, each of which can contain a value of any type (integer, boolean, or a further pair). A pair value is created by evaluating a `PairExp`, which takes the field values as operands; the field values can be extracted from a pair value by using `FstExp` and `SndExp` expressions. Pairs are *immutable*: once a pair has been constructed, there is no way to change the contents of its fields. The values of the fields should be computed *before* the pair is built. For example (using ad-hoc concrete notation), the expression

```
let p =
  (let x = 22 in
       pair(x + 20,x <= 23)) in
  if (snd p) then (fst p) else 99
```

should evaluate to 42.

To help get you started, the example file already contains:

- A definition of a new type sub-class `PairType`. Unlike the existing types, the pair type has instance variables representing the types of its first and second components. For this reason, we sometimes describe pair as a "type constructor" (it builds new types out of existing types) rather than just a "type."
- `Pair` containing two `Object` fields, which can be used to hold pair values at runtime. (Unfortunately, the Java library doesn't include a class like this already.) Another alternative would be to use a two-element Java array.

To complete the task, you will need to define new expression sub-classes `PairExp`, `FstExp`, and `SndExp`, and give implementations for `eval` and `check` for each of them. Note that the definitions for `FstExp` and `SndExp` are almost identical: you can use cut-and-paste, but, as always, be careful!

Exercise: Functions and Pairs (16a)

Pairs are particularly useful for passing multiple arguments to (or obtaining multiple results from) a function. To exercise this facility, write (i) a function that implement an equality comparison on integers and (ii) a function that implements logical OR on booleans. Then use these functions to build some larger computation of your choice.

Question: Is your OR function "short-circuiting"? If not, can you change it to be?