Abstract Syntax Trees

Background: Limitations of Our Parser

In the previous lab, every time we encountered an expression, we evaluated it, and every time we encountered a statement, we executed it. Although this strategy worked fine for our simple grammar, it presents a serious problem if we were faced with executing the following Java statement.

```
boolean isSqrt = (y != 0 && x / y == y);
```

To parse this statement, we need to parse the expression `y != 0` and the expression `x / y == y`. Suppose that `y` does equal 0. If we evaluate every expression we parse, this means we'll find that the value of `y != 0` is false, and then we'll evaluate `x / y == y` and wind up with a division-by-zero error. But according to the Java programming language specification, executing this statement should assign `false` to `isSqrt`, because Java's `&&` operator is only supposed to evaluate its second expression in the event that the first one is true. This property is called *short-circuit boolean*, and it's something our parse-and-evaluate-all strategy can't address.

We just saw an example where a portion of the code should be evaluated once or not at all. By way of contrast, consider the following Java statement.

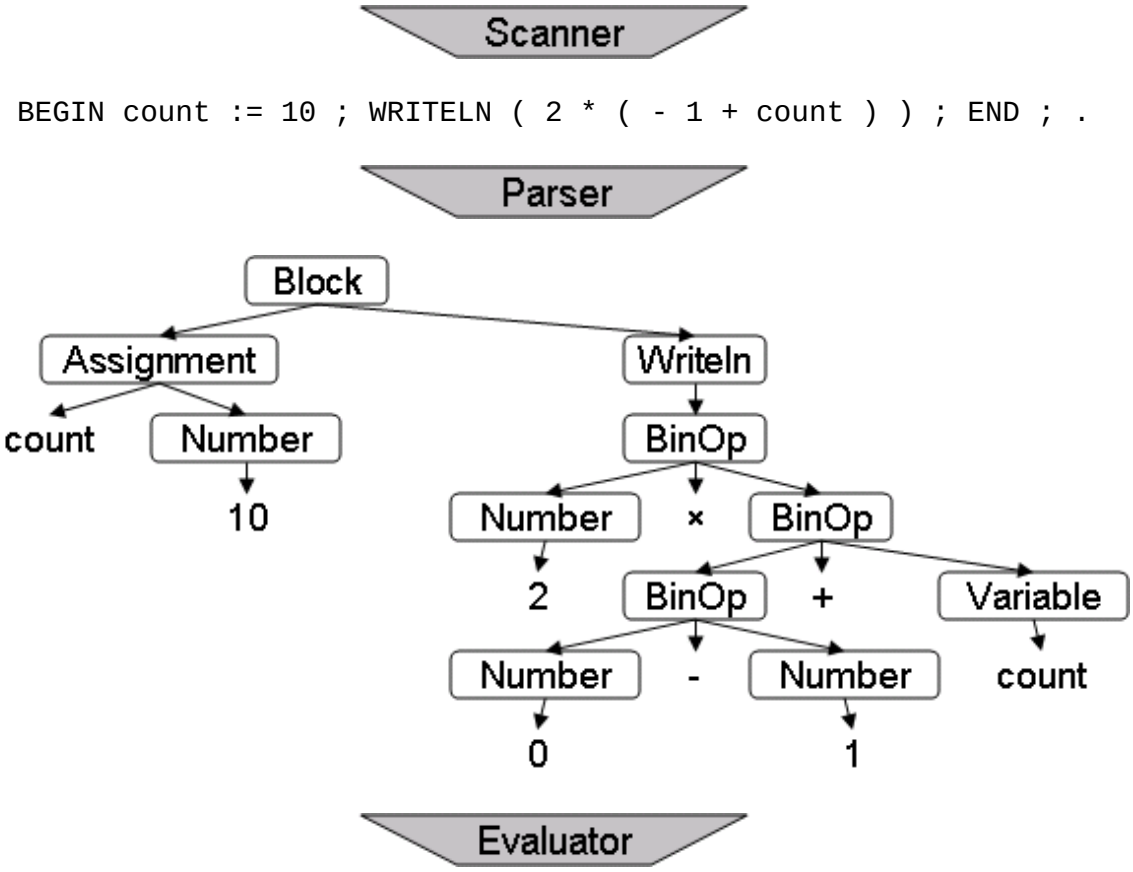
```
while (count > 0)
   count--;
```

We'd like to parse this statement exactly once. However, the expression `count > 0` may be evaluated one or more times, while the statement `count--;` may be evaluated zero or more times. A while-loop is an example of a control structure, governing which portions of code to evaluate at what times. In order for our interpreter to handle control structures correctly, it is necessary for us to divide parsing and evaluation into two separate stages.

Background: Abstract Syntax Trees

The following diagram shows the three stages involved in interpreting a short program.

```
"BEGIN\n\tcount := 10;\n\tWRITELN(2 * (-1 + count));\nEND;\n.\n"
```



18

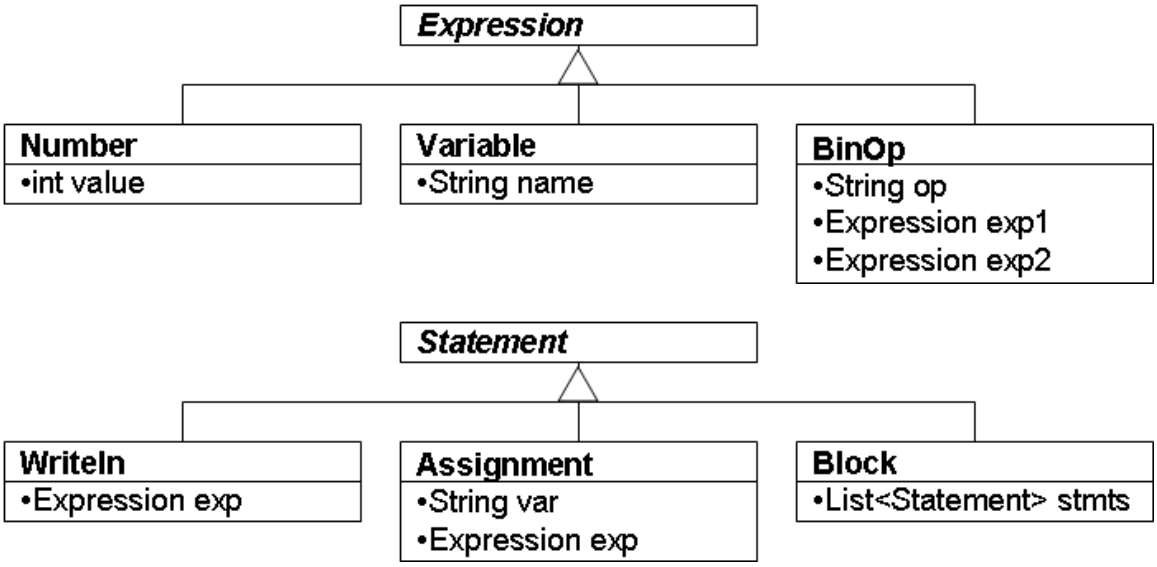
Notice that the output of the parser is now a tree, which is then easily evaluated in the next stage. This tree does not depend on the original programming language syntax. Instead, this *abstract syntax tree* (AST) highlights the abstract relationships between the statements and subexpressions. In a way, we have translated our original PL/0 program into the intermediate language of abstract syntax trees.

One important thing to note about our AST is that it does not necessarily correspond to the choices of terminals and nonterminals in our grammar. For example, the tree captures the order of operations, and therefore there are no parentheses in the tree, nor does the tree need to distinguish between expressions, terms, and factors. Notice also that it may be easier to parse `-1` into the binary operation *0 - 1*, rather than create a new AST element for negating a quantity.

Let's take another look at our simple grammar.

stmt → **WRITELN (** *expr* **) ;** | **BEGIN** *stmts* **END ;** | **id :=** *expr* **;**
stmts → *stmts stmt* | ε
expr → *expr* **+** *term* | *expr* **-** *term* | *term*
term → *term* ***** *factor* | *term* **/** *factor* | *factor*
factor → **(** *expr* **)** | **-** *factor* | **num** | **id**

Here now is the class hierarchy representing the grammar for our abstract syntax trees. Any program described by the first grammar can be parsed into an AST using only the expressions and statements shown here.



Exercise: `Expression` and `Statement`

Back up all the work you've done so far in this course. We will continue to use the `Scanner` class, but we will now be making substantial changes to `Parser` (so be sure to keep a back-up of the old version).

Each of our AST components will require a new class, which means we'll have quite a few classes by the end of the lab. Because future labs will also require many new classes, often with similar names, we'll store our new components in a Java package. Here's what's involved:

- Keep all the AST classes in a package called ast.
- At the top of each AST class, write `package ast;`
- For any class like `Parser`, which is outside of the `ast` package but needs to refer to the `ast` classes, refer to those classes with a fully qualified name (such as `ast.Statement`) , or `import ast.<class>;` where <class> specifies the particular class within the `ast` package at the top of the file. Be sure to follow the style guide for importing.

In the new package, define the classes `Expression` and `Statement`. These classes should be declared abstract, as we will never instantiate them; we'll only instantiate their subclasses (such as `Number` and `Writeln`). For now, we won't need to define any instance variables or methods for `Expression` and `Statement`. For example, the `Statement` class should look like this:

```
package ast;

public abstract class Statement
{
}
```

Exercise: AST Classes

Next, define the `Number`, `Variable`, `BinOp`, `Writeln`, `Assignment`, and `Block` classes in the `ast` package. Each should have appropriate instance variables, and a single constructor for initializing those instance variables. For example, the `Writeln` class might look like this:

```
package ast;

public class Writeln extends Statement
{
   private Expression exp;

   public Writeln(Expression exp)
   {
      this.exp = exp;
   }
}
```

Exercise: Parsing

Now, modify your parser so that each *parseXXXXX* method returns the appropriate component of the abstract syntax tree *without executing/evaluating the parsed text*. For example, here's a portion of the modified `parseStatement` method. Pay close attention to the boldface types. Also notice that we no longer call `System.out.println` in parsing a `WRITELN` statement. We are only parsing the statement here. We'll call `System.out.println` later when (and if!) the statement is executed.

```
public Statement parseStatement()
{
   if (token.equals("WRITELN"))
   {
      eat("WRITELN");
      eat("(");
      Expression exp = parseExpression();
      eat(")");
      eat(";");
      return new Writeln(exp);
   }
   ...
}
```

Test that you can still parse your simple programs without error. We won't know until later exercises whether you have parsed them correctly.

Exercise: Environments

Next we'll need to write the `Environment` class, whose job it will be to remember the values of variables. For now, there will just be a single environment, but later in the lab we'll need to juggle multiple environments. Place your `Environment` code in a package called environment. Your `Environment` class should implement the following two methods.

```
//associates the given variable name with the given value
void setVariable(String variable, int value)

//returns the value associated with the given variable
//name
int getVariable(String variable)
```

Exercise: Evaluating

We're now ready to execute the statements we parse. There are at least two different approaches we could take here. Think about each of these carefully, and decide which approach appeals most to you.

Approach #1: Each AST component knows how to evaluate itself.

In this approach, we define an abstract `exec` method in the `Statement` class and an abstract `eval` method in the `Expression` class, both taking in an `Environment`, as shown below.

```
public abstract class Statement
{
   public abstract void exec(Environment env);
}
```

Each subclass then knows how to execute/evaluate itself when given an `Environment`, often by instructing its children to execute/evaluate, as shown for the `Writeln` class below. Thus the evaluator code is spread out among the many AST components, as shown here.

```
public class Writeln extends Statement
{
   private Expression exp;

   public Writeln(Expression exp)
   {
      this.exp = exp;
   }

   public void exec(Environment env)
   {
      System.out.println(exp.eval(env));
   }
}
```

Approach #2: All the evaluator code appears in a single file.

In this approach, we define a single `Evaluator` class with an `exec` method for each kind of `Statement` and an `eval` method for each kind of `Expression`. Notice how the code below more closely resembles our `Parser` class.

```
public class Evaluator
{
   public void exec(Writeln stmt, Environment env)
   {
      System.out.println(
         eval(stmt.getExpression(), env));
   }

   public int eval(BinOp binop, Environment env)
   ...
}
```

Go ahead and implement your evaluator using one of these approaches, and test that you can run your old simple programs.

Exercise: `IF` Statements

We'll now extend our language to handle `IF` statements like the following.

```
IF val > max THEN max := val;

IF n <> size - 1 THEN
BEGIN
   WRITELN(n);
   n := size - 1;
END;
```

Our new grammar looks as follows.

stmt → **WRITELN (** *expr* **) ;** | **BEGIN** *stmts* **END ;** | **id :=** *expr* **;**
| **IF** *cond* **THEN** *stmt*
stmts → *stmts stmt* | ε
expr → *expr* **+** *term* | *expr* **-** *term* | *term*
term → *term* ***** *factor* | *term* **/** *factor* | *factor*
factor → **(** *expr* **)** | **-** *factor* | **num** | **id**
cond → *expr relop expr*
relop → **=** | **<>** | **<** | **>** | **<=** | **>=**

Notice that the `IF` production does not include a semicolon, as its consequence *stmt* already ends in a semicolon. Also notice that the consequence *stmt* could itself be a `BEGIN`/`END` block. The *cond* nonterminal represents the condition tested by the `IF`. Your condition must handle the 6 relational operators listed (where `<>` means "does not equal").

Add `Condition` and `If` classes with appropriate instance variables and constructors to your `ast` package. Extend your `Parser` (and `Scanner`, if necessary) to correctly parse `IF` statements. Finally, implement the necessary code so that your interpreter evaluates `IF` statements correctly. Test that you can now run programs with `IF`s.

If you feel ambitious, you may choose to handle more complex boolean conditions. Likewise, *in addition* to the simple `IF` construct shown above, you may also choose to handle the more complex `IF`/`ELSE` construct according to the following production.

stmt → **`IF`** *cond* **`THEN`** *stmt* **`ELSE`** *stmt*

Exercise: `WHILE` Loops

We'll now extend our language to handle `WHILE` statements like the following.

```
WHILE count > 0 DO
BEGIN
   WRITELN(count);
   count := count – 1;
END;
```

Our new grammar looks as follows.

stmt → **`WRITELN (`** *expr* **`) ;`** | **`BEGIN`** *stmts* **`END ;`** | **`id :=`** *expr* **`;`**
| **`IF`** *cond* **`THEN`** *stmt* | **`WHILE`** *cond* **`DO`** *stmt*
stmts → *stmts stmt* | ε
expr → *expr* **`+`** *term* | *expr* **`-`** *term* | *term*
term → *term* **`*`** *factor* | *term* **`/`** *factor* | *factor*
factor → **`(`** *expr* **`)`** | **`-`** *factor* | **`num`** | **`id`**
cond → *expr relop expr*
relop → **`=`** | **`<>`** | **`<`** | **`>`** | **`<=`** | **`>=`**

Go ahead and modify your code so that you can now run programs with `WHILE` loops. Be sure to test that your programs run correctly.

If You Finish Early

- Add support for for-loops, using the following syntax.

```
FOR i := 1 TO 10 DO
BEGIN
   sum := sum + i;
   Writeln(sum);
END;
```

- Add support for the `CONTINUE` and `BREAK` statements. The following pointless code segment uses these statements to print the values 1, 1, 2. Hint: You may find throwing and catching exceptions to be useful in your implementation.

```
i := 0;
WHILE 1 + 1 = 2 DO
BEGIN
   i := i + 1;
   IF i = 3 THEN BREAK;
   WRITELN(i);
   IF i = 2 CONTINUE;
   WRITELN(i);
END;
```