Programming Project #4: Abstract Syntax Trees

Due Date: Tuesday, November 1, 2005, Noon

Overview

Modify your parser to build and return an Abstract Syntax Tree representation for the PCAT program as it parses the PCAT source.

Files

The following files can be found via the class web page or FTPed from:

```
~harry/public_html/compilers/p4
```

Main.java
This file has been altered to print the AST returned from **parseProgram()**.

Ast.java
This file is new. It contains the classes used to describe as AST.

PrintAst.java
This file is new. It contains methods to print out an AST and is used only to make sure the AST is correct.

Lexer.class
Token.java
StringTable.java
FatalError.java
LogicError.java
These files are unchanged.

makefile
This file has been altered slightly.

tst
This is a subdirectory containing a number of test files. None of the files in this directory have lexical or syntactic errors.

tst2
This subdirectory contains the same files as p3/tst2. These files had syntax errors. These tests are included to make sure you still print the error messages correctly.

go
go2
run
run2
These files are unchanged.

runAll
runAll2
Same as idea before, but altered slightly for different test file names.

Main.jar
This is the "black box" code, which was used to produce the output files in **tst** and **tst2**.

Do not modify any of the files except for **Parser.java**.

Discussion

Each parsing method will construct and return an abstract syntax tree (AST). The abstract syntax tree returned from **parseProgram** (called from **main**) will represent the entire PCAT source program. The **main** procedure (in project 4) will simply print out the tree in its entirety.

In projects 5 and 6, will we walk the AST looking for semantic errors. (We will also modify the AST in a few places at that time. For example, the expression 2.5 + 1 will be modified to 2.5 + intToReal (1).)

The file **Ast.java** documents the Abstract Syntax Tree data structure.

There are many different kinds of AST node. For example, there is one kind of node to represent assignment statements and another kind to represent expressions involving binary operators.

There is a class for each kind of node. Here are the classes in **Ast.java**.

```
Node                    --  Abstract
  Body
  VarDecl
  TypeDecl
  ProcDecl
  Formal
  TypeName
  CompoundType          --  Abstract
    ArrayType
    RecordType
  FieldDecl
  Stmt                  --  Abstract
    AssignStmt
    CallStmt
    ReadStmt
    WriteStmt
    IfStmt
    WhileStmt
    LoopStmt
    ForStmt
    ExitStmt
    ReturnStmt
  ReadArg
  Expr                  --  Abstract
    IntToReal           --  Not used in project 4
    BinaryOp
    UnaryOp
    FunctionCall
    ArrayConstructor
    RecordConstructor
    IntegerConst
    RealConst
    StringConst
    BooleanConst
    NilConst
    ValueOf
  Argument
  FieldInit
  ArrayValue
  LValue                --  Abstract
    Variable
    ArrayDeref
    RecordDeref
```

The **Ast.java** file contains a class called **Ast** and all the above classes are inside of **Ast**. This allows us to put several classes in a single file, which reduces the number of **.java** files. When you compile this file, it will create a **.class** file for every one of these classes, with names like **Ast$Node.class** and **Ast$Body.class**.

Within your code, you can refer to a class using a name such as **Ast.Node** or **Ast.Body**.

Each AST class has several fields. The document "Abstract Syntax Tree – Class Summary" shows the class layouts / field names in a graphical way which you may find helpful.

The class **Ast.Node** has a single field, called **lineNumber**. Since all classes are subclasses of **Ast.Node**, they will all have a **lineNumber** field. The **lineNumber** will give the source code line number on which the construct appeared in the PCAT source file. (Basically, we just pick up **lexer.lineNumber** when the object is created.) We need to keep the line number around since it must be printed in any semantic error messages that we generate while walking the tree later on.

The following classes are "abstract": **Node**, **CompoundType**, **Stmt**, **Expr**, and **LValue**. The indentation above shows the class-subclass structure. For example, **BinaryOp** is a subclass of **Expr**, which is a subclass of **Node**.

Let's take a look at **BinaryOp**:

```
static class BinaryOp extends Expr {
    int         op;
    Expr        expr1;
    Expr        expr2;
    int         mode;                   // Not used until proj 6
}
```

In addition to the **lineNumber** field, this class has several fields that are particular to it. The **op** field tells which operator is involved and will be something like **Token.PLUS**, **Token.STAR**, or **Token.EQUAL**. (For the **op**, we use the integer of the corresponding token type.) A binary operator expression has two operands, which are its sub-expressions. Thus, a **BinaryOp** node will point to two sub-trees (**expr1** and **expr2**) each of which represents one of the sub-expressions.

As this example shows, the file **Ast.java** also includes some fields we'll be using in later projects. The comments in the file will identify in which project such fields will first be needed; you can ignore these fields until the later projects. For example, you can ignore the field named **mode** for now.

Given the class **Ast.Node**, we can define pointers to nodes and use them to build up larger trees from smaller trees. For example:

```
Ast.Expr e1, e2;
Ast.BinaryOp p;
...
p = new Ast.BinaryOp ();
p.op = Token.PLUS;
p.expr1 = e1;
p.expr1 = e2;
```

The constructor for **Ast.Node** contains code to fill in the **lineNumber** field using the current value, so we don't need to do it here.

[At first, you might think it would be a good idea to create specialized constructors for each class, such as:

```
BinaryOp (int op, Ast.Expr e1, Ast.Expr e2) { ... }
UnaryOp (int op, Ast.Expr e) { ... }
...
```

Then the above code becomes:

```
p = new Ast.BinaryOp (Token.PLUS, e1, e2);
```

We will not do this since (1) it would create a lot of little constructors, (2) often we want to allocate the node before we have the subtrees built (in order to pick up a value of **lineNumber** that is more accurate), and (3) it is often necessary to go back and modify the nodes. Nevertheless, it is a debatable issue of style.]

The nodes in the abstract syntax tree correspond fairly closely to the syntax of PCAT, but there are several differences that need to be discussed.

Statements

There are several places in the PCAT grammar where a list of zero or more statements is expected (like in a **body**, after **"then"**, etc.) To deal with lists of statements, each of the statement classes (**AssignStmt**, **CallStmt**, **ReadStmt**, ..., **ReturnStmt**) has a **next** field. The **next** field is in the **Stmt** class. We will represent a statement list as a linked list of **Stmt** nodes. Thus, this linked list will, in general, contain several different kinds of objects, but they will all be **Stmts**.

You probably have a method **parseStmts()** which will parse zero-or-more statements. This method should return a pointer to a list of statement nodes (or the NULL pointer). **parseStmts()** probably contains a while loop, which calls routines like **parseIfStmt()**, etc. The idea is to modify **parseIfStmt()** so that it will return an **IfStmt** object. Then **parseStmts()** will need to add that node to the end of a growing list of statement nodes, using the **next** field of the previously last node in the list. Note that we don't need to know exactly what kind of a statement was last on the list; we will just treat it as a **Stmt**, which has a **next** field.

If Statements

The IF statement in PCAT contains zero-or-more "**elseif**" clauses but the **IfStmt** class contains only room for a THEN statement list and an ELSE statement list.

```
static class IfStmt extends Stmt {
    Expr        expr;
    Stmt        thenStmts;
    Stmt        elseStmts;
    ...
}
```

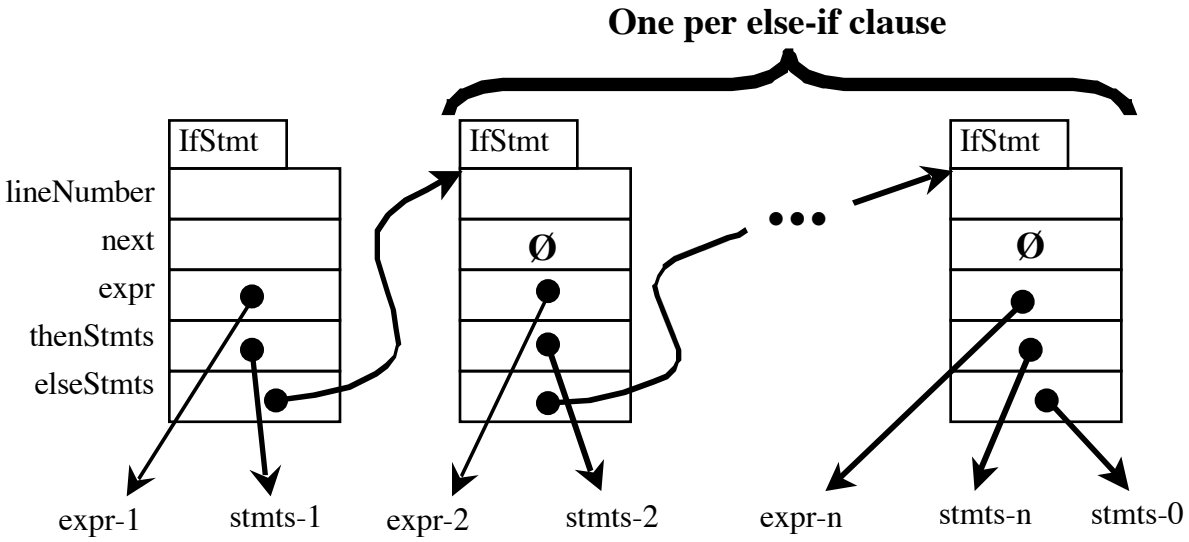
To deal with ELSE-IF clauses, note that:

```
if expr-1 then
      stmts-1
elseif expr-2 then
      stmts-2
...
elseif expr-n then
      stmts-n
else
      stmts-0
end;
```

is equivalent to:

```
if expr-1 then
      stmts-1
else
      if expr-2 then
            stmts-2
      else
            ...
            if expr-n then
                  stmts-n
            else
                  stmts-0
            end;
      end;
end;
```

For the first IF-ELSEIF-ELSEIF-END statement, we will build exactly the same tree that we would build for this set of nested IF-THEN-ELSE statements. The tree is shown next. In that diagram, the "tree" has been visually distorted to suggest that a list of ELSEIF clauses is represented by a linked list of **IfStmt** nodes.



In the document "Abstract Syntax Tree – Class Summary" some of the fields are marked with an asterisk (*). Such a field may be NULL. In an IF statement, for example, the THEN statement list or the ELSE statement list may be empty. In the node constructed for the following code, the **thenStmts** field would be set to NULL.

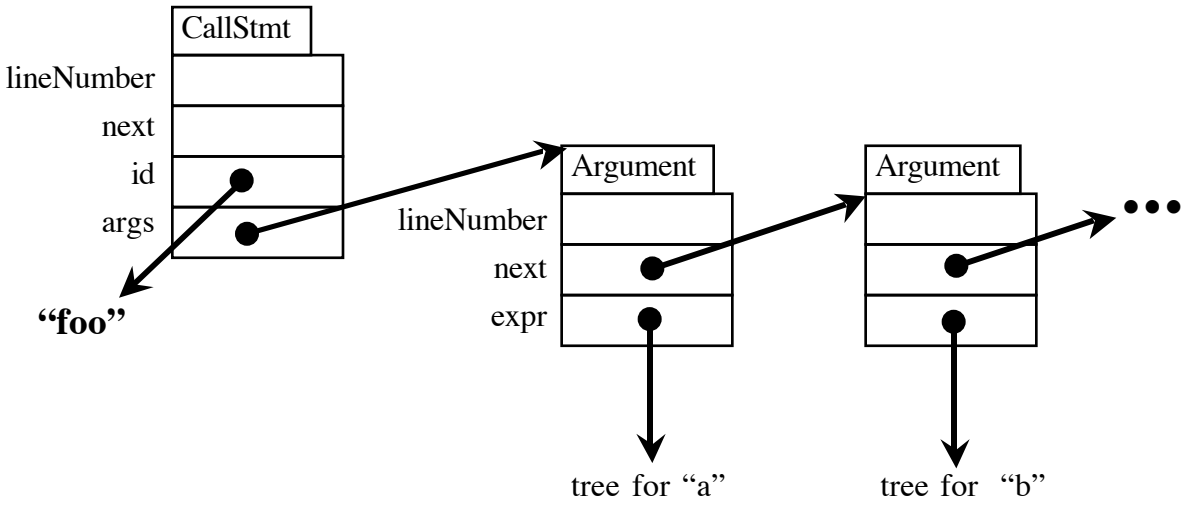
```
if expr-1 then
else
      stmt;
      stmt;
      stmt;
end;
```

Representation of Argument Lists

In the following PCAT statement, a procedure is to be called:

```
foo (a, b, c);
```

To deal with zero-or-more actual parameters, we use a linked list of zero-or-more **Argument** objects. (The term "argument" is synonymous "actual parameter".) This list is pointed to by the **args** field, and is linked through the **next** field in class **Argument**.



Several of the AST classes have a field called **id**. **CallStmt** is an example. In all cases, the **id** field is a pointer to some String returned from the lexer, namely **lexer.sValue.** In a later project, we will need to look this string up in the symbol table to ensure that it was previously defined, but for now we will just store it in the **id** field.

The **FunctionCall** class is similar to the **CallStmt** class in that it, too, will contain a pointer to a list of zero-or-more **Argument** nodes. The difference is that **FunctionCall**s represent a procedure invocation within an expression, as in:

```
x := 3 * foo (a, b, c);
```

while the **CallStmt** represents a procedure invocation at the statement level, as in:

```
foo (a, b, c);
```

Likewise, the **WriteStmt** node will contain a pointer to a list of **Argument** nodes. In WRITE statements—and only in WRITE statements—the PCAT programmer can use string literals, as well as general expressions. For example:

```
write ("hello", (i+3)*x, "world");
```

In PCAT, there are no operations on strings and variables cannot contain strings as values. To represent string literals, use a **StringConst** node, with the **sValue** field pointing to the String associated with the lexer token.

The READ statement expects one-or-more arguments, which may not be general expressions. Instead, each argument must be an l-value. The field **readArgs** in the **ReadStmt** node will point to

a list of one-or-more **ReadArg** nodes, each of which will point to the tree returned after parsing an l-value. In other words, the **lValue** field of the **ReadArg** object will point to an object of one of the **LValue** classes.

(The **LValue** classes are **Variable**, **ArrayDeref**, and **RecordDeref**. Anywhere we see a field called **lValue**, it will contain a pointer to an instance of one of these three classes.)

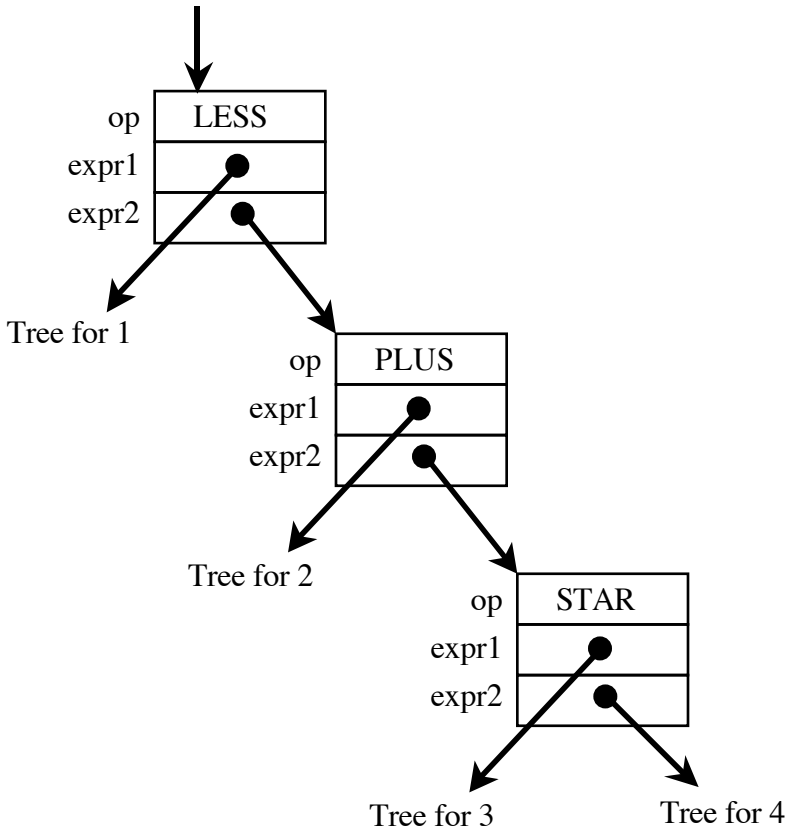
Expressions

BinaryOp nodes have already been mentioned. The tree returned from each call to **parseExpr()** should reflect the correct associativity and precedence of PCAT. Presumably, your parsing routines took care of this, and if you build the trees in the obvious way within the parsing code, you should get the correct grouping.

For example, the expression:

```
1 < 2 + 3 * 4
```

should result in:

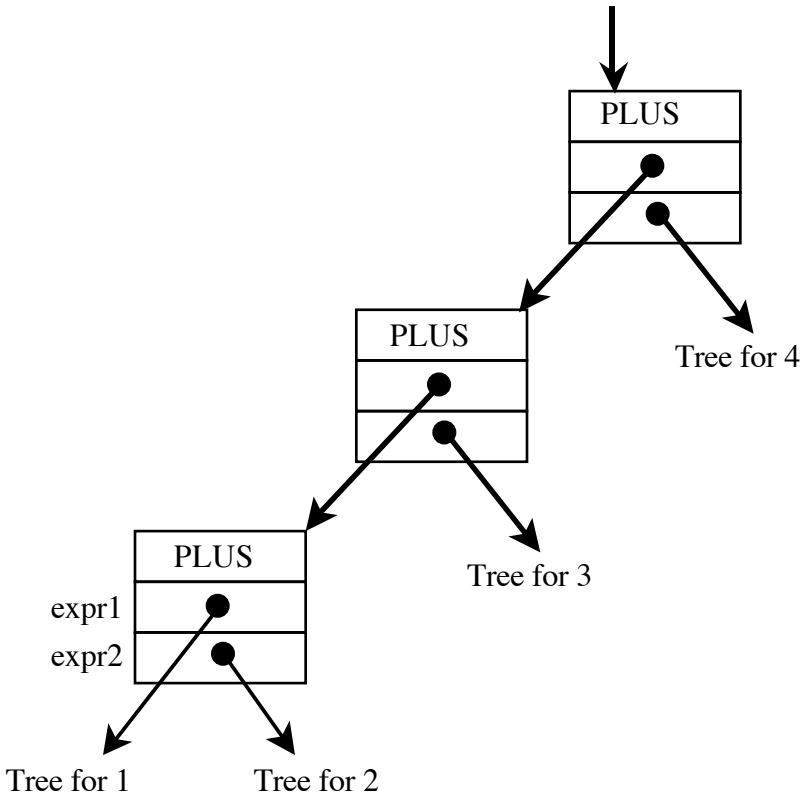


since * has higher precedence than +, which is higher than <.

On the other hand, the expression:

```
1 + 2 + 3 + 4
```

should be represented as:



to reflect the left-associativity of all binary operators. (Note that I am starting to abbreviate the details of the object layouts a little.)

All of this should "come for free" if your parse routines are parsing expressions in a way that reflects the desired associativity and precedence. If your trees are not shaped correctly, you may wish to review the grammar your parser is looking for.

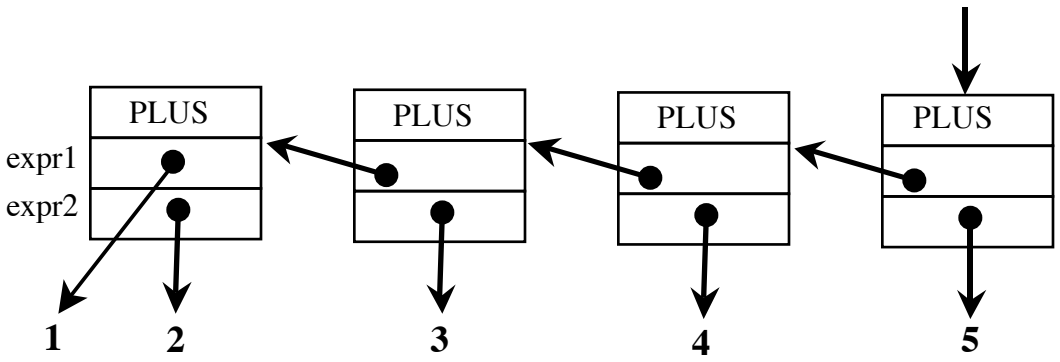
It is likely that your parsing routines implement rules such as:

```
expr2  →  expr3   {   ( '+' | '-' | 'or' )  expr3   }
```

using a while loop: As long as there is a '+' or '-' or 'or' then scan it, pick up an operand, and repeat. So when you are parsing

```
1 + 2 + 3 + 4 + 5
```

you will be building a tree by adding a new structure to the top. Below, I've twisted the diagram to suggest its similarity to a list. Here, the additions are at the front (right end) of the list, using the field **expr1** as a "next" pointer.

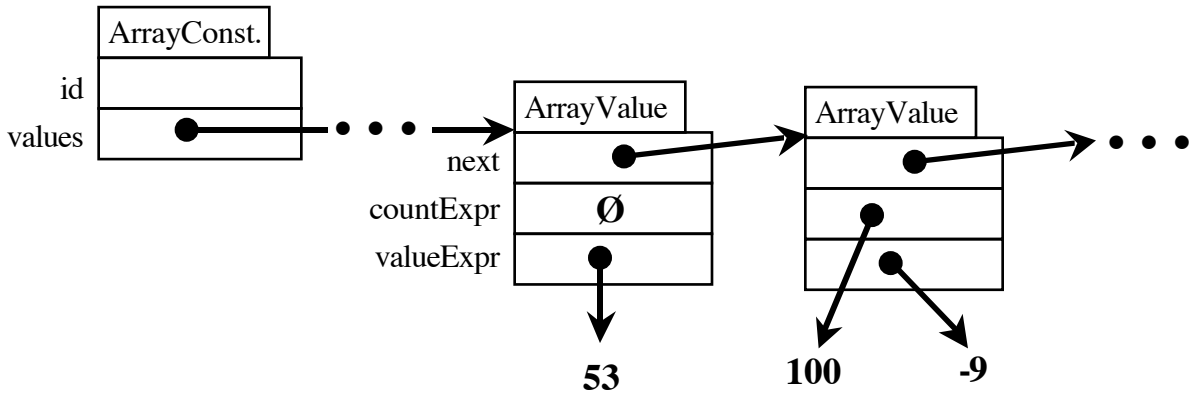


Constructor Expressions

The **values** field in an **ArrayConstructor** node will point to a list of one-or-more **ArrayValues** nodes, which will be linked on their **next** field. If no "count expression" is supplied in the PCAT source code, the **countExpr** field of a **ArrayValues** node will be NULL . For example:

```
MyArr [< ..., 53 , 100 OF -9 , ... >]
```

will result in:



Likewise, the **fieldInits** field in a **RecordConstructor** node will point to a list of one-or-more **FieldInits** nodes, which will be linked on their **next** field.

L-Values and ValueOf

An identifier can appear either in an expression or wherever an l-value is expected. In the following code, **x** appears as an l-value on the left-hand side of an assignment while **y** appears buried within an expression.

```
x := 1 + (y * 4);
```

These two uses are fundamentally different. The **x** is used as an l-value and the **y** is used as an r-value.

What code shall we generate for a variable used as an l-value? For **x** we need to generate code to determine the address of the location to use. In general, this may require some computation since **x** may be a local (dynamically allocated) variable or it might involve record and array dereferencing, as in:

```
a[i+5].f := ...;
```

For r-values, we will compute the address first (again, this may involve arbitrary computation). Then, with address in hand, we can generate a "load" instruction to get the data from the desired location in memory.

We can always use an l-value as an r-value and this is reflected in the PCAT grammar:

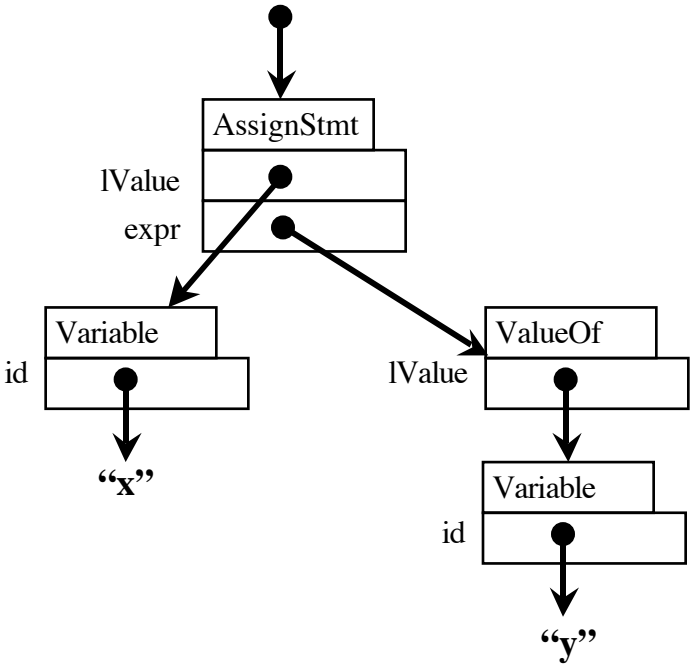
```
Expression  →  LValue
            →  ...
```

Thus, whenever, we encounter an l-value used as an expression, we must insert a **ValueOf** node to indicate that we are converting an l-value to an r-value. Whenever we use an l-value as an l-value (i.e., in a READ statement, on the left-hand side of an ASSIGNMENT, to the left of the '[' in an array indexing expression, to the left of the '.' in a record dereferencing expression, or as the index variable in a FOR statement) we will use it as is.

For example, the following code:

```
x := y;
```

will be represented as:



Later, during code generation, the **ValueOf** node will result in a "load" instruction being generated.

Types

Types can be used in the following contexts:

In a variable declaration, e.g.,
var x: *TypeName* := ...;
As the "element type" in an array type, e.g.
array of *TypeName*
As the type of a field in a record type, e.g.,
record ... myField: *TypeName*; ... **end**
As the return type for a procedure, e.g.,
procedure foo () : *TypeName* **is** ... **end**;
As the type of a parameter to aprocedure, e.g.,
procedure foo (..., p3: *TypeName*, ...) **is** ... **end**;

A *TypeName* may be "integer", "real", or "boolean" or may be a type defined in a type declaration. However, at this stage we will not be checking this.

A *TypeName* will be represented with a **TypeName** object, which has one field of interest at this time. Each **TypeName** object has a field called **id**, which you should set to point to a String. In addition, there is a field called myDef, which will be used in later projects for defined types, i.e., for all types besides "integer", "real", and "boolean". For now, **myDef** will not be set.

A *TypeDecl* defines a new type. Each TypeDecl is represented with a **TypeDecl** object. The new type has a name (stored in the **id** field) and a new type, which is either an array type or a record type. Collectively, array types and record types are called *CompoundTypes*, since they are (in some sense built out of other types, like "integer", "real", "bool" and other compound types. The

TypeDecl object has a field, called **compoundType**, which you should set to point to the definition (i.e., to either an **ArrayType** or **RecordType** object).

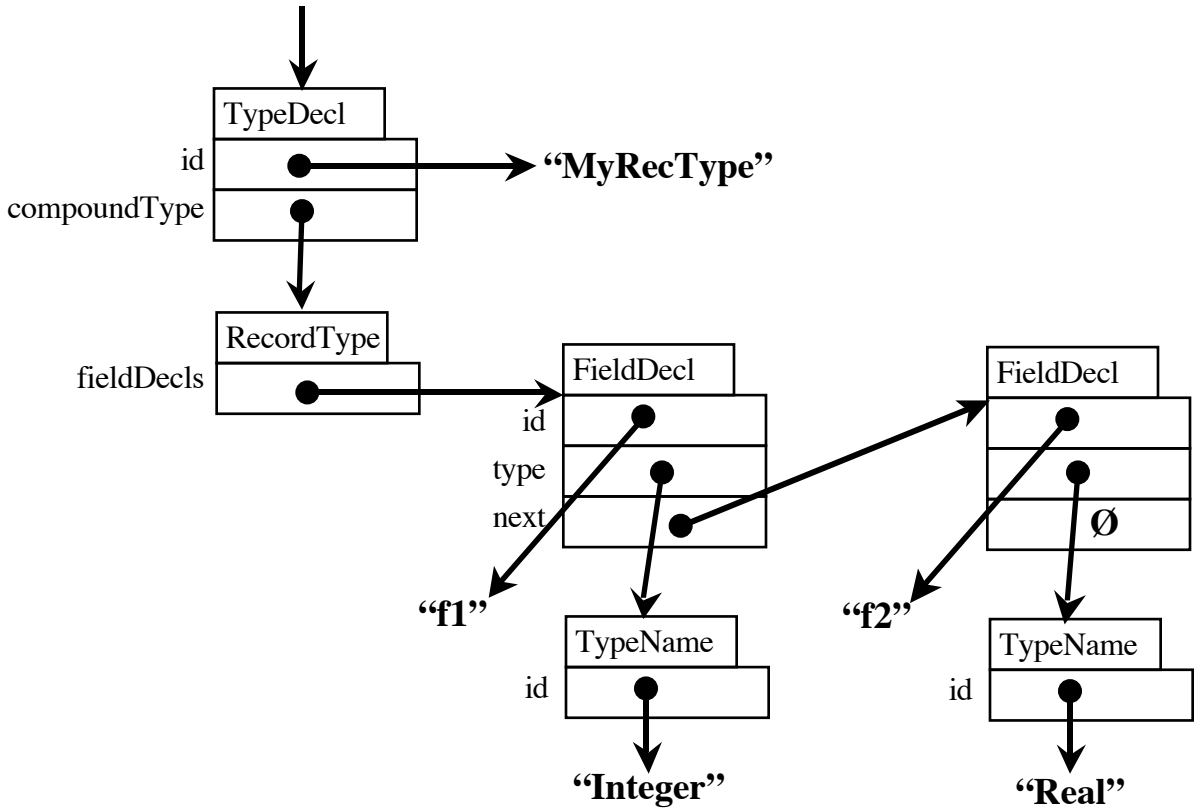
An array type has a base type (or "element type"), which is the type of the elements in the array. The **ArrayType** class has a field called **elementType** which should be set to point to the **TypeName** representing the base type.

A record type will consist of a number of named fields, each of which has a type. The **fieldDecls** field in a **RecordType** node will point to a list of one-or-more **FieldDecl** nodes, which will be linked on their **next** field.

For example the following PCAT fragment:

```
type MyRecType is record
                    f1: integer;
                    f2: real;
                  end;
```

will be represented as:



Bodies and Declarations

A body will be represented with a **Body** node.

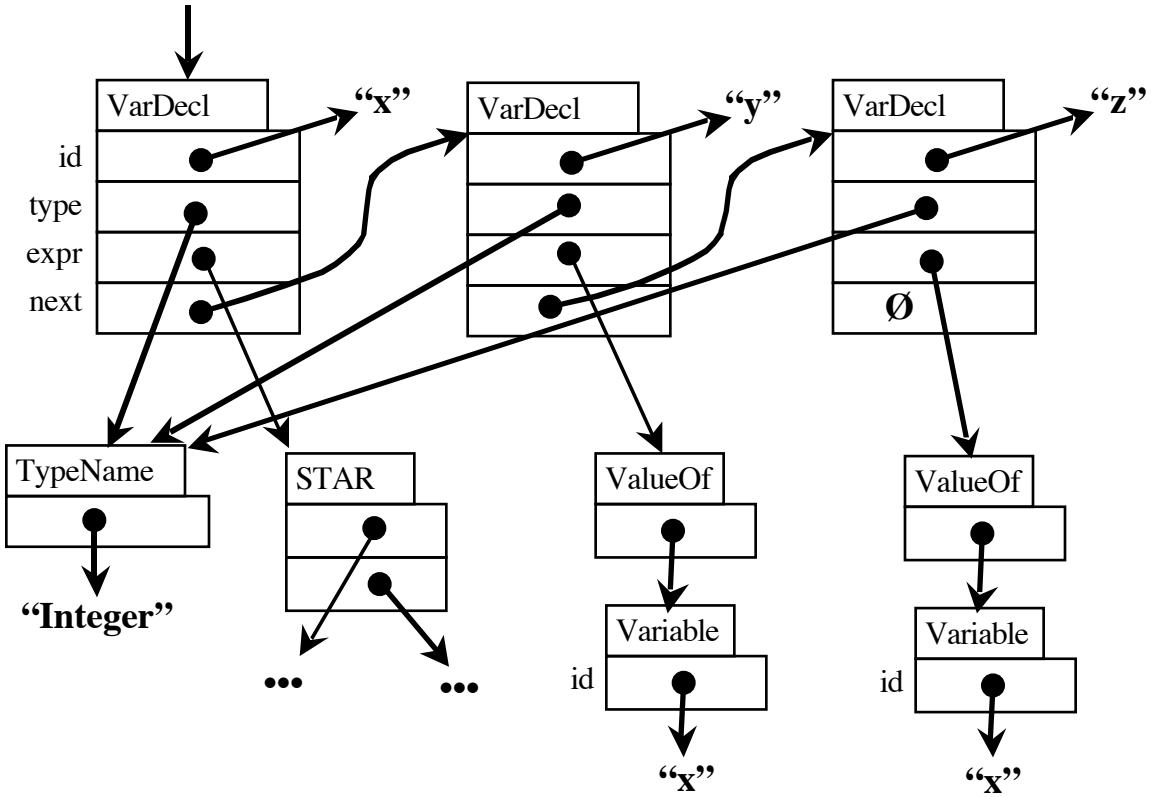
```
static class Body extends Node {
    TypeDecl    typeDecls;
    ProcDecl    procDecls;
    VarDecl     varDecls;
    Stmt        stmts;
    ...
}
```

The **typeDecls** field will point to a list of **TypeDecl** nodes, which will be linked on their **next** fields. The **procDecls** field will point to a list of **ProcDecls** nodes, which will be linked on their **next** fields. The **varDecls** field will point to a list of **VarDecl** nodes, which will be linked on their **next** fields. Finally, the **stmts** field will point to a list of **Stmt** nodes, which will be linked on their **next** fields.

A single *VarDecl* grammatical construct can be used to declare one-or-more variables, as in the following example:

```
var x, y, z: integer := 4 * foo(23);
```

However, each **VarDecl** node can only be used to define a single variable, so the above PCAT fragment must be represented as a linked list of three **VarDecl** nodes. For the above source, you should build an AST that will look like this:



Notice that the node representing the *TypeName* is shared by all three **VarDecl** nodes. (Technically, we are now dealing with DAGs—Directed, Acyclic Graphs—but the distinction may be safely ignored.)

Also notice that the sub-trees representing the initializing expressions for the second and third variables are a little different than what appears in the source. We can't use the same exact expression since it would cause us to call "foo" three times at runtime, which is not the correct semantics for PCAT: we must only evaluate the initializing expression once.

The way we get around it is to build the same structure for this source as we would have built for:

```
var x: integer := 4 * foo(23);
    y: integer := x;
    z: integer := x;
```

As you parse a variable declaration for multiple variables, you will build up a list of **VarDecl** nodes. After you get the type (if present; it is optional) and the initializing expression, you can go back through the list and update the **type** and **expr** fields to point to the correct trees.

The routine which parses a *VarDecl* (I called it **parseVarDecl()**) will then return this linked list.

Similarly, a *TypeDecl* can define one-or-more types at once:

```
type t1 is ...; t2 is ...; t3 is ...;
```

Again, a linked list will result, although there will be no shared sub-trees.

Finally, one-or-more procedures may be defined at once, and a list of **ProcDecl** nodes will result:

```
proc foo1(...) is ...; foo2(...) is ...;
```

In PCAT, we can have variable declarations, type declarations, and procedure definitions all mixed up in any order. For example:

```
proc foo1(...) is ...;
var x, y, z: ...;
type t1 is ...; t2 is ...; t3 is ...;
var a, b: ...;
type t4 is ...;
proc foo2(...) is ...;
```

In the representation, we will group all TYPEs together, all PROCs together, and all VARs together. The above fragment will be represented as if it had been written:

```
type t1 is ...;
     t2 is ...;
     t3 is ...;
     t4 is ...;
proc foo1(...) is ...;
     foo2(...) is ...;
var x: ...;
    y: ...;
    z: ...;
    a: ...;
    b: ...;
```

When we do the semantic checking later, we will want to process the TYPE declarations first (since the VARs and PROCs may include references to types defined in the same scope). We will want to process PROCs second (since the initializing expressions in VARs may include references to functions defined in the same scope) and we will process VARs last. To facilitate this order of processing, we will group the declarations at this stage of the compiler.

To parse and build the AST for declarations and bodies, I used the following parse routines:

```
Ast.Body parseBody () {...}
void parseDecls (Ast.Body body) {...}
Ast.TypeDecl parseTypeDecl () {...}
Ast.ProcDecl parseProcDecl () {...}
Ast.VarDecl parseVarDecl () {...}
```

ParseBody() parses a body. It begins by allocating a **Body** node. (All the lists will be initialized to NULL, since Java initializes all fields to zero.) Then it calls **parseDecls()**, which parses zero-or-more declarations. **ParseDecls()** keeps going as long as the next token is VAR, TYPE, or PROCEDURE, and upon seeing one of those, calls **parseTypeDecl()**, **parseProcDecl()**, or **parseVarDecl()** to get a single declaration. **ParseDecls()** looks like this:

```
while (1) {
   if nextToken == TYPE then
      call parseTypeDecl to get a list of TYPE_DECL nodes
      append this list to the end of the list stored in BODY.typeDecls
   elseIf nextToken == PROC then
      call parseProcDecl to get a list of PROC_DECL nodes
      append this list to the end of the list stored in BODY.procDecls
   elseIf nextToken == VAR then
      call parseVarDecl to get a list of VAR_DECL nodes
      append this list to the end of the list stored in BODY.varDecls
   elseIf nextToken == BEGIN then
      return
   else
      syntaxError
   endIf
}
```

ParseDecls() does not return a tree; instead it is passed a pointer to the **Body** node which represents the body we are currently parsing. It modifies the node by adding to the ends of the existing lists of **TypeDecl**, **ProcDecl**, and **VarDecl** nodes.

The **ProcDecl** node contains a field called **formals**, which points to a linked list of **Formal** nodes. **Formal** nodes are linked on their **next** field. There is one **Formal** node allocated per formal

parameter in a PROCEDURE declaration. For example, the following would result in a list of five **Formal** nodes:

```
procedure foo (x,y,z: integer; a,b: real) is begin ... end;
```

Note that **Formal** nodes may share sub-trees, much like **VarDecl** nodes shared sub-trees. In this example, the first 3 **Formal** nodes will all have their **typeName** fields pointing to a single **TypeName** object representing "integer", while the 4th and 5th nodes on the list will share the node representing "real".

Dealing with "true", "false", and "nil"

In PCAT, the identifiers "true", "false", and "nil" are predefined and must be handled specially. These identifiers can appear in any expression, just like any other variable, but for them, you'll need to construct special nodes.

Note that these special names are not defined in the grammar as keywords, so the lexer will return normal ID tokens when they are encountered.

(By the way, another approach is for the language to dispense with such "predefined identifiers" and make them first-class keywords. I think making "true", "false", and "nil" keywords is a better approach.)

The "nil" variable will stand for a null pointer. (Some languages use "nil" and some use "null". For all intents and purposes, "nil" and "null" mean the same thing.) Whenever "nil" is encountered in an expression, you should build an AST using a **NilConst** object. There are no fields in this object.

For "true" and "false", you'll need to use a **BooleanConst** node. There is a single field called **iValue**. For "true", set **iValue** to 1 and for "false", set **iValue** to 0. Note the similarity between the **BooleanConst** node and the **IntegerConst** node; they'll be used very similarly.

The grammar for expressions has been rewritten to include something like:

```
Expr5    -->  '(' Expr ')'
              | INTEGER
              | REAL
              | IdMods
IdMods   -->    ID '(' Arguments ')'
              | ID '{' FieldInits
              | ID '{' '{'  ArrayValues
              | ID LValueMods
```

You can probably deal with "true", "false", and "nil" most easily by adding code to explicitly check to see which ID you have. If it is one of the predefined IDs, then deal with it directly and don't look for anything after it.

In other words, modify **parseIdMods** so it uses a rule like this:

IdMods --> ID_{true}
| ID_{false}
| ID_{nil}
| ID '(' Arguments ')'
| ID '{' FieldInits
| ID '{' '{' ArrayValues
| ID LValueMods

Note that for every variable occurrence in the source, your compiler must make 3 comparisons to check for "true", "false", and "nil". String comparisons are kind of slow, but remember that all strings returned from the lexer will have been entered into the StringTable. Thus, we can use == for string comparisons, as long as both strings we are checking have been added into the StringTable first.

Perhaps you will want to initialize three values before parsing begins and use them in these comparisons.

```
StringTable.insert ("true", Token.ID);
trueString = StringTable.lookupString ("true");
...
if (id == trueString) { ...
```

The PrintAst.java File

I am providing a file called **PrintAst.java** which is used to print out the AST. It is called from **main** to print whatever tree is returned from your parser. Please read the comment at the beginning of that file to find out more about how ASTs are printed.

Ideas for Getting Started

Begin by creating a **p4** directory and copying your **Parser.java** file. Then edit **Parser.java** and remove all of the "print" statements that we used to verify / check the parser. In other words, remove the printing to **stdout**. Be sure to keep the error reporting stuff that went to **stderr**. Don't just comment them; delete them entirely. (You always have p3/Parser.java to go back to.)

Make sure you can compile cleanly at this stage.

Next, change all the return types on your "parse..." methods from **void** to whatever **Ast** class they will be returning. Also add dummy "return" statements, which will simply return **null**, and get it to compile cleanly.

At this point, you might want to verify that **runAll** and **runAll2** from project 3 still work correctly. Of course, all the output to **stdout** will cause the entire contents of the ***.out.bak** files to be listed in the "diff"s, but your program should still terminate with no errors.

Next, deal with **ParseBody**. Get a **Body** node returned and make sure an AST with one node prints out properly. A dummy program like this should be adequate: "program is begin end;". This is test **p1**, but remember that you can just type PCAT source input on **stdin** during testing.

Next, work on **ParseStmts** and **parseReturnStmt**. Start by just handling the RETURN statement. So that you can handle programs like:

```
program is
    begin
        return;
        return;
        return;
    end;
```

Now you are ready to dig into the parsing of expressions. You can use program like the following to test various expressions:

```
program is
    begin
        return 3 + 4;
    end;
```

At this point, skip **parseIdMods**. Have it return **null** for now. Skip ArrayConstructors and RecordConstructors at this stage.

Get tests **p2**, **p3**, and **p4** working.

Implement **parseLValue** and get test **p5** and **p6** working.

Work on parsing *ArrayConstructors* and *RecordConstructors* and get **p7** and **p8** working.

Work on **parseDecls**. Eliminate **parseDecl** and move it into **parseDecls**. Pass a **Body** down to **parseDecls**.

Implement **parseVarDecl**.

Implement **parseTypeName** and get **p8** working.

Details...

As before, email your completed program as a plain-text attachment to:

`cs321-01@cs.pdx.edu`

Don't forget to use a subject like:

`Proj 4 - John Doe`

DO NOT EMAIL YOUR PROGRAM TO THE CLASS MAILING LIST!!!

Your code should behave in exactly the same way as my code. If there is any question about the exact functionality required,

(1) Use my code (the "black box" **.jar** file) on test files of your own creation, to see how it performs.

(2) *Please ask* or talk to me!!! I will be happy to clarify any of the requirements. If there are any problems with the assignment, I would like to alert other students and/or modify my documents or files. If my test data can be improved, please let me know.

Don't submit multiple times. Be sure to keep an unmodified copy of your file on Sirius with the timestamp intact. Work independently: you must write this program by yourself.