A Review of the Abstract Syntax Tree
How a PCAT Program is Represented

Harry H. Porter III
January 21, 2006

Introduction

This document discusses the Abstract Syntax Tree (AST) representation. It can serve as a refresher for students who took CS-321 from me or an introduction for students who took CS-321 elsewhere.

It is assumed that you have already read the PCAT language specification and are familiar with the language. If you have not used PCAT in CS-321, then you should also look at some example programs. There are other documents written specifically for students who have taken CS-321 from Andrew Tolmach, which may be good for these students.

You know what a parse tree is. A parse tree exactly captures the PCAT source program and represents it exactly as specified in the grammar. There is a one-to-one correspondence between the grammar and the parse tree. More precisely, each interior node in the parse tree is labeled with a non-terminal symbol from the grammar and each leaf node is labeled with a terminal symbol (i.e., a token) from the grammar. From a parse tree, it would be possible to recreate the original source input, except that the spacing of tokens is lost and the comments (which were eliminated by the lexer) were lost.

Often, there are rules in the grammar that are there for "syntactic" reasons. For example, the grammar rules may be written to enforce a certain associativity or precedence in the operators, or the grammar rules may have been manipulated to make it possible to use a certain parsing algorithm.

Unfortunately, the shape of the parse tree is exactly determined by the grammar rules and is not always the best for use in later stages of the compiler. The parse tree will often have too much information and unnecessary nodes.

On the other hand, an "abstract syntax tree" represents the source PCAT program in a more efficient way, in a way that will make code generation easier. So, in CS-321 we built an AST ("abstract syntax tree"), not a parse tree. The AST is similar to a parse tree and captures more-or-less the same information. The AST contains some simplifications that make it impossible to recover the original source PCAT program exactly. However, it captures all that is semantically necessary.

Trees, DAGs, and Directed Graphs

Formally, in a "tree" each node (except the root) has a single parent. In other words, each node is pointed to by exactly one node (its parent) and there are no cycles. Every node is either an interior node or a leaf.

Walking (or "traversing") a tree is straightforward, using a recursive routine. As part of visiting a node, this routine will visit each of the children recursively. You can use such a recursive routine to print the tree. If there are different sorts of nodes in the tree, you might have a different routine for each type of node, but these routines will call each other. They are "mutually" recursive.

A "directed, acyclic graph" (or DAG) is like a tree except that a node may have more than one parent. In a DAG there are still no cycles, but now we have the possibility that some child is shared by several parents: more than one node may point to a given node, but there is still a concept of direction. "Down" is from parent to child, following pointers. "Up" is from child to parent, following the edges backward.

If you naïvely use the same recursive method to print a DAG it will work... more-or-less. However, a child which is shared by several parents will be printed several times, but since there are no cycles, there will not be any infinite looping.

A "directed graph" is similar to a tree and a DAG, except there is no notion of "up" or "down". Although some node might be distinguished as a "root", or entry into the graph, there may be cycles. Walking a graph with cycles can be tricky; you don't want to get caught in a cycle and keep recursing infinitiely.

Adding Semantic Information

The AST is initially produced during parsing and contains only grammatical, syntactic information. During type-checking in CS-321, our compiler gathered additional information about the program. This information was stored in the AST for use this term during code generation. We will first discuss the main fields of the AST, which were filled in during parsing. We can call these the "syntactic fields".

Then we will discuss several more fields that were added to the AST and filled in during type-checking. We can call these additional fields the "semantic fields". Many of the semantic fields contain pointers to other nodes in the AST. When we consider all the fields in an AST it is no longer a tree; the AST is a directed graph. Following tradition, we continue to call it a "abstract syntax TREE", although it is not technically a tree.

When the AST was first created by the parser, it was a tree. It was easy to traverse and print. In fact, there is a program available to you, called **PrettyPrint.java**, which walks the AST and prints it. This code looks only at the syntactic fields. The **PrettyPrint** code prints the AST in a format that is very similar to the original PCAT source. This makes it fairly easy to check the syntactic fields and see

exactly what your AST looks like. The **PrettyPrint** code also has some hooks in it to allow you to print selected semantic fields in such a way that cycles do not cause infinite output.

The AST Classes

In a textbook tree or directed graph there is only one sort of node, which can accommodate zero or more outgoing edges.

In our AST, we have many different kinds of node. For example, to represent an "if" statement, we will use an **IfStmt** node. To represent a variable declaration, we'll use a **VarDecl** node. For each type of node, we have a class. There is a class called **IfStmt** and a class called **VarDecl**. All these classes are included in the file called **Ast.java**, which you should study.

So our AST is made of nodes. Each node is an object and there are many kinds of nodes. For each kind of node, there is a different class. There are 43 different classes; here they are in alphabetic order:

Argument
ArrayConstructor
ArrayDeref
ArrayType
ArrayValue
AssignStmt
BinaryOp
Body
BooleanConst
CallStmt
CompoundType
ExitStmt
Expr
FieldDecl
FieldInit
ForStmt
Formal
FunctionCall
IfStmt
IntToReal
IntegerConst
LValue
LoopStmt
NilConst
Node
ProcDecl
ReadArg

ReadStmt
RealConst
RecordConstructor
RecordDeref
RecordType
ReturnStmt
Stmt
StringConst
TypeDecl
TypeName
UnaryOp
ValueOf
VarDecl
Variable
WhileStmt
WriteStmt

Each class of node has a number of fields. For example, the class **IfStmt** is defined as:

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

It uses the classes **Stmt** and **Expr**, which we'll describe later.

During parsing, for each "if" statement in the source program, the parser will create one **IfStmt** node. Every **IfStmt** node corresponds to an "if" statement in the source program being compiled. Likewise, there is a class called **WhileStmt** to represent "while" statements and a class called **Variable** to represent each appearance or use of a variable, and so on for each of the important kinds of things we might see in a PCAT program.

We can see that a node of class **IfStmt** will contain pointers to three other nodes. The idea is that the field **expr** will point to the node representing the boolean conditional expression that occurs in every "if" statement. The field called **thenStmts** will point to the node representing the sequence of the "then" statements and the **elseStmts** field will point to the sequence of "else" statements.

Some fields may be null, while others will never be null. Either the **thenStmts** or the **elseStmts** or both may be null, but the **expr** field will never be null. The **IfStmt** class contains only syntactic fields; there are no semantic fields.

Note that the **IfStmt** class has no methods. This is true all the **Ast** classes, except for a constructor in the root superclass. The AST classes can be considered to be data-representation classes and the methods that operate on the AST data structures are placed in other classes, which will be discussed in the next section.

There is a class-subclass relationship between many of the AST classes. We say that subclasses like **IfStmt** extend superclasses like **Stmt**. For example, the class **Stmt** extends **Node**. **Node** is a superclass of **Stmt** and **Stmt** is a subclass of **Node**.

Node is the root class; every kind of AST node extends **Node**, either directly (like **Stmt**) or indirectly (like **IfStmt**).

All of the AST classes are actually included as members of a class called **Ast**. If you are unfamiliar with putting one class inside another class—it is pretty weird, after all—the only thing you really need to know is that you will be referring in your code to the classes by using names like **Ast.IfStmt**, **Ast.Expr**, **Ast.Stmt**, **Ast.Node**, etc. In this document, I'll often omit the "**Ast.**" part and say just **IfStmt**, **Expr**, Stmt, **Node**, etc.

The Compiler Classes

There are several other classes we use in our compiler.

Class	Created in...	Provided by...
Main		Porter
Lexer	proj 2	
Parser	proj 3,4	
Checker	proj 5,6	
Generator	proj 8,9,10	
FatalError		Porter
LogicError		Porter
Token		Porter
PrettyPrint		Porter
PrintAst		Porter
StringTable		Porter
SymbolTable		Porter
IR		Porter

Each of these classes is in a **.java** file with the same name.

These classes are used to group and organize all the methods. In other words, the code lives in these classes. By their names, you can probably guess what tasks their methods are used for. We will not really be concerned with creating objects for these classes. Each of these classes will have at most one object, i.e., at most there will be one "instance" of each class.

For example, the compiler will create a single **Lexer** object. This object will then be used to do the lexical analysis. Any local, temporary data needed only during lexing will be kept in the fields of this object. Every time the parser needs a new token, it will invoke a method on the lexer object.

Likewise, there will be a single **Parser** object and it (or more exactly, its methods) will do the parsing and creation of the AST. There will be one **Checker** object and the methods here will be used to perform type-checking and fill in the semantic fields.

The class **Main** has only a single static method, called **main**, which will create the **Parser** and **Checker** objects and will use them first to create the AST and then to check it. The **main** method will then create a **Generator** object and invoke methods on it to perform intermediate code generation.

The **Lexer**, **Parser**, and **Checker** classes were written by students last term. You will not be given the source code for these classes, so you will not be able to see the code there. But you should not need to look over that code. You will be given compiled **.jar** files for the **Lexer**, **Parser**, and **Checker** classes, so that the AST can be built and checked.

In project 8, you will create the **Generator** class. The **main** method will create a single instance of the **Generator** class and will then invoke a method, which you should name **generateIR**, to generate the intermediate (IR) code. The main method will pass a pointer to the AST to the **generateIR** method.

The **FatalError** and **LogicError** clases extend **Exception** and are used to deal with error conditions that must halt the compiler dead in its tracks. The **main** method catches these errors and terminates with an error message.

The class **Token** is used during lexical analysis. The **Token** class is never instantiated, i.e., there will never be any **Token** objects. Instead, **Token** just contains some useful constant values. Here are some constants from **Token.java** which you'll be needing in the back-end:

```
final static int
    AND          = 0,
    ...
    DIV          = 4,
    ...
    MOD          = 14,
    NOT          = 15,
    ...
    OR           = 17,
    ...
    PLUS         = 33,   // +
    MINUS        = 34,   // -
    STAR         = 35,   // *
    SLASH        = 36,   // /
    LESS         = 37,   // <
    GREATER      = 38,   // >
    EQUAL        = 39,   // =
    ...
    LEQ          = 51,   // <=
    GEQ          = 52,   // >=
    NEQ          = 53,   // <>
```

You'll need to use constants (such as **Token.PLUS** and **Token.NOT**) when you look at a **BinaryOp** or **UnaryOp** node to see what kind of operator it is.

Printing an AST can be done in two ways, called **PrintAst** and **PrettyPrint**.

PrintAst produces an exhaustive dump of the AST and it is used by the grader to make sure your AST is exactly what it should be. **PrintAst** produces a lot of output and is rather difficult to read, except for the smallest programs. All the methods in class **PrintAst** are static, so no object is created. The **PrintAst** code (which is provided by Porter) is designed to be as bullet-proof as possible. The algorithm is designed to avoid infinite looping, even if a tree is in error and actually contains cycles or bad data.

The **PrettyPrint** class, which is also provided by Porter, walks the tree using a much simpler (recursive) approach. It prints the AST is a form closely resembling the original source PCAT. The **PrettyPrint** code can be used as a starting framework, as you begin to code your **Generator** class. The primary method in **PrettyPrint** is called **prettyPrintAst**. This method is static. It creates a **PrettyPrint** object and then invokes the relevant method on it. There are a number of methods in the **PrettyPrint** class; but only the **PrettyPrintAst** method is static. Thus, there will be a single **PrettyPrint** instance, whenever the AST is printed.

The **StringTable** class is used to deal with Java Strings. Its methods are all static, so there will be no **StringTable** objects.

The **SymbolTable** class is used by the **Checker** class during type checking. You can safely ignore this class, since it will not be used at all during the back-end (i.e., during code generation). All the methods in **SymbolTable** are static. No **SymbolTable** object is created, although the class contains a nested class, called **Bucket**. There are many instances of **Bucket**. During the compilation, there is only one symbol table and the information representing it is kept in the static variables of class **SymbolTable**.

The **IR** class is new to Project 8. I am providing this class to facilitate the code generation process. The IR class is a combination of data representation and code. There will be many **IR** objects created during code generation: one **IR** object will be created for each intermediate instruction. This class contains many static methods to make working with IR instruction objects easier. The **IR** class is discussed in more detail in the Project 8 assignment.

Dealing With Java Strings

In Java, two String objects may have the same characters yet not be the same object. In other words, a program can easily create two String objects that are "equal" in the sense of having the same characters, but not "equal" in the sense of being different objects. To compare Strings in Java, you really ought to use a method that goes through the Strings, character by character, comparing every byte. Unfortunately, this is time-consuming and we would much rather be able to simply compare pointers.

To allow us to compare strings quickly and safely by simply comparing pointers, we have gone to some trouble in the front-end to make sure that, for any sequence of characters, there is only one String with those characters. In other words, we are making sure that every String is unique.

We do this by keeping a table containing all Strings; this is the purpose of the **StringTable**. During lexical analysis, the lexer will read in some characters and build a string. This occurs every time an identifier in scanned. The lexer then checks the **StringTable** to see if we have already seen that same sequence of characters before. If so, the lexer will throw away the new String and use a pointer to the old String instead.

In the lexer, we used code that looked something like this:

```
newString = ...scan a bunch of characters...
if (StringTable.lookupToken (newString) == ...notFound...) {
    StringTable.insert (newString, ...);
}
str = StringTable.lookupString (newString);
```

This way, even if the string "myOwnVar" appears in the program 361 times, there will only be one String object with the characters "m y O w n V a r".

Such a shared common version of a String is called the "canonical version" of the String. Of all the 361 String objects with characters "m y O w n V a r", this one canonical object is the representative. It will be used everywhere and the other 360 versions will be ignored.

Note that a number of nodes in the AST will contain pointers to Strings. For example, when a variable is declared, there will be a **VarDecl** node in the AST. The **VarDecl** node contains a field, called **id**, which points to the String for the identifier. Whenever the variable is used, say in some expression, there will be a **Variable** node in the AST. The **Variable** node also contains a field, named **id**, which will point to the same String.

When I say "the same String", I mean "the very same String object". So, in some sense, we can observe right now that the AST was never really a tree, but is a DAG, since one object (like the String object for "m y O w n V a r") has several other objects pointing to it.

The bottom line is that in the back-end, if you should need to compare two strings from the AST, you can use the == operation safely.

An AST Subtree for an Expression

Let's imagine that the PCAT syntax contains some grammar rules for expressions, such as these.

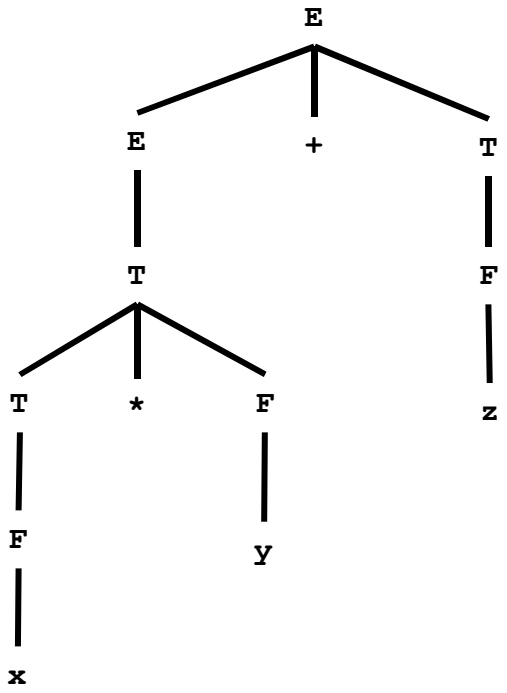
E → E + T
E → T
T → T * F
T → F
F → **id**
F → (E)

(The actual rules you used in CS-321 were a little more complex, of course.)

Now let's say some PCAT input source program contains an expression such as

x * y + z

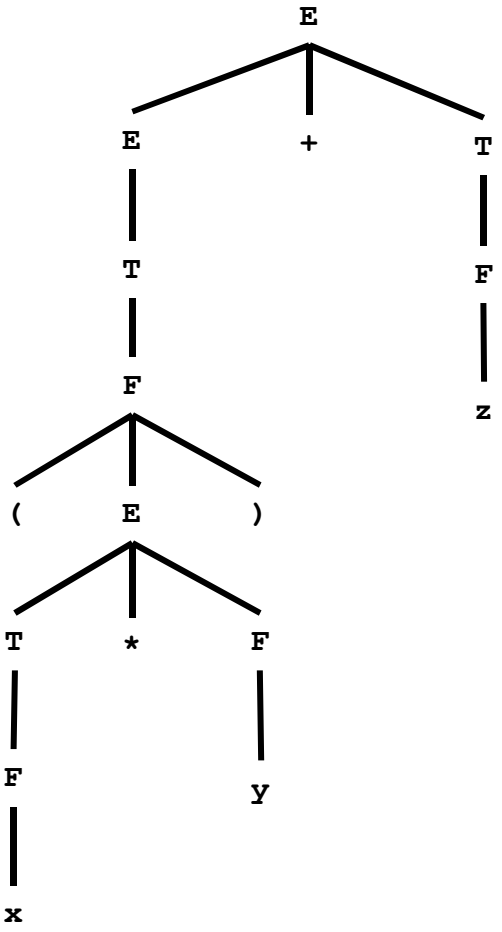
Using this grammar fragment, we could build this parse tree for the expression:



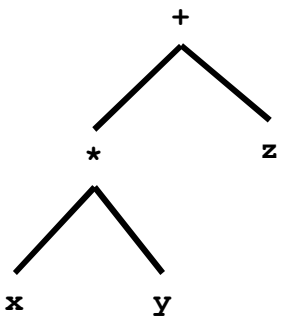
Next consider a similar expression:

(x * y) + z

For this we would construct this parse tree:



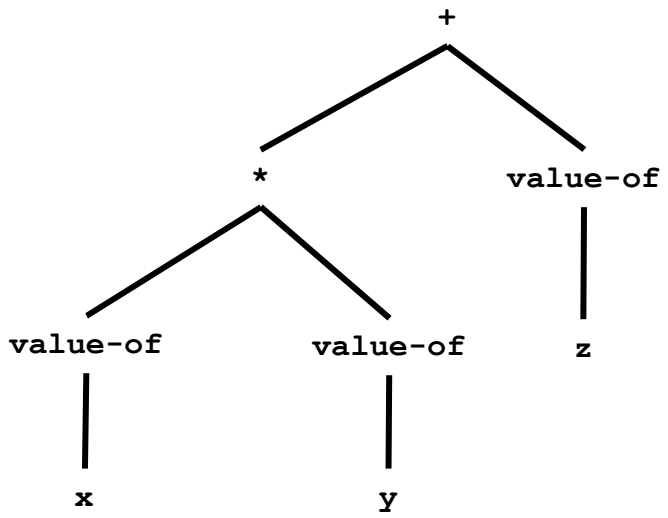
Note that both of these expressions are semantically equivalent and we want to generate the same IR code for both. The benefit of the AST approach is that we will "abstract away" all the irrelevant syntactic information. For both of these expressions, our parser will build the same AST. It will build an AST that looks like this:



This tree captures the essence of the expression; it's a simple tree with a lot fewer nodes than a parse tree and will be easier to work with.

Recall the difference between "L-Values" and "R-Values". An L-Value corresponds to a variable's address and an R-Value corresponds to a variable's value. When a variable is used in an expression, we want its value. To get a variable's value, we'll first need to get its address. Then, at runtime we can go to that address in memory and fetch its value. So, in some sense, an R-Value requires an L-Value; if you have an L-Value, you can turn it into an R-Value by adding an additional memory fetch operation.

We need to modify our AST to reflect this. We'll assume that a variable by itself represents an L-Value and we'll use a node called "value-of" to reflect the fact that, in an expression, we want the R-Value.



Next, we have to ask, how will an AST be represented with Java objects? The answer is that every node in the tree will be represented with one Java object and every edge will be represented with one Java reference (i.e., with a "pointer") from the parent to the child.

There are three kinds of nodes involved here. The nodes representing the binary operators point down to two children. The nodes representing variables are leaves and do not have any children. The nodes representing the conversion of an L-Value into an R-Value have a single child.

To represent the AST for **x*y+z** using Java objects, we'll use three classes: **Ast.BinaryOp**, **Ast.Variable**, and **Ast.ValueOf**. We'll discuss each of these classes in turn.

Here is the **Ast.BinaryOp** class:

```
static class BinaryOp extends Expr {
    int         op;
    Expr        expr1;
    Expr        expr2;
    int         mode;
}
```

There are several kinds of binary operators. The **op** field tells which operator it is. Here are the possible values for **op**:

	Meaning	
Token.PLUS	**+**	
Token.MINUS	**-**	
Token.STAR	*	
Token.SLASH	**/**	(real division)
Token.DIV	**div**	(integer division)
Token.MOD	**mod**	(remainder after integer division)
Token.AND	**and**	
Token.OR	**or**	
Token.LESS	**<**	
Token.LEQ	**<=**	
Token.GREATER	**>**	
Token.GEQ	**>=**	
Token.EQUAL	**=**	
Token.NEQ	**<>**	(not equal)

While we're at it, here are the choices for the **op** field for class **Ast.UnaryOp**:

	Meaning	
Token.PLUS	unary **+**	(nop)
Token.MINUS	unary **-**	
Token.NOT	**not**	

The **expr1** field will point to the left operand. More precisely, the **expr1** field will point to the root of an AST sub-tree representing the sub-expression that occurs to the left of the operator. Likewise, the **expr2** field will point to the right sub-tree.

The operators can work on several kinds of data. For example, **PLUS** can mean integer addition or floating point addition and we'll have to generate different machine code depending on which it is. The **mode** field will give us this information.

There are several "modes". Here are the legal modes:

```
static final int INTEGER_MODE = 1;
static final int REAL_MODE    = 2;
static final int STRING_MODE  = 3;
static final int BOOLEAN_MODE = 4;
```

In **BinaryOp** and **UnaryOp**, the **mode** field will have one of two possible values:

INTEGER_MODE
REAL_MODE

The other two modes are used elsewhere.

For the following operators, the mode may be either **INTEGER_MODE** or **REAL_MODE**:

Token.PLUS (either as a **BinaryOP** or as a **UnaryOP**)
Token.MINUS (either as a **BinaryOP** or as a **UnaryOP**)
Token.STAR
Token.LESS
Token.LEQ
Token.GREATER
Token.GEQ
Token.EQUAL
Token.NEQ

For **Token.SLASH**, it will only be **REAL_MODE**. The mode will be **INTEGER_MODE** for all other operators.

Note that some of the operators (like **EQUAL**) can deal with boolean values or even arrays and records. For these, the **mode** will still be **INTEGER_MODE**. In other words, **mode**=**REAL_MODE** just signals when we need to use floating-point operations. All other times, we will use integer instructions.

Here is the class **Ast.Variable**:

```
static class Variable extends LValue {
    String   id;
    Node     myDef;
    int      currentLevel;
}
```

For each variable, we capture a String in **id** giving its spelling during the parse. The fields named **myDef** and **currentLevel** are semantic fields and are filled in during type-checking.

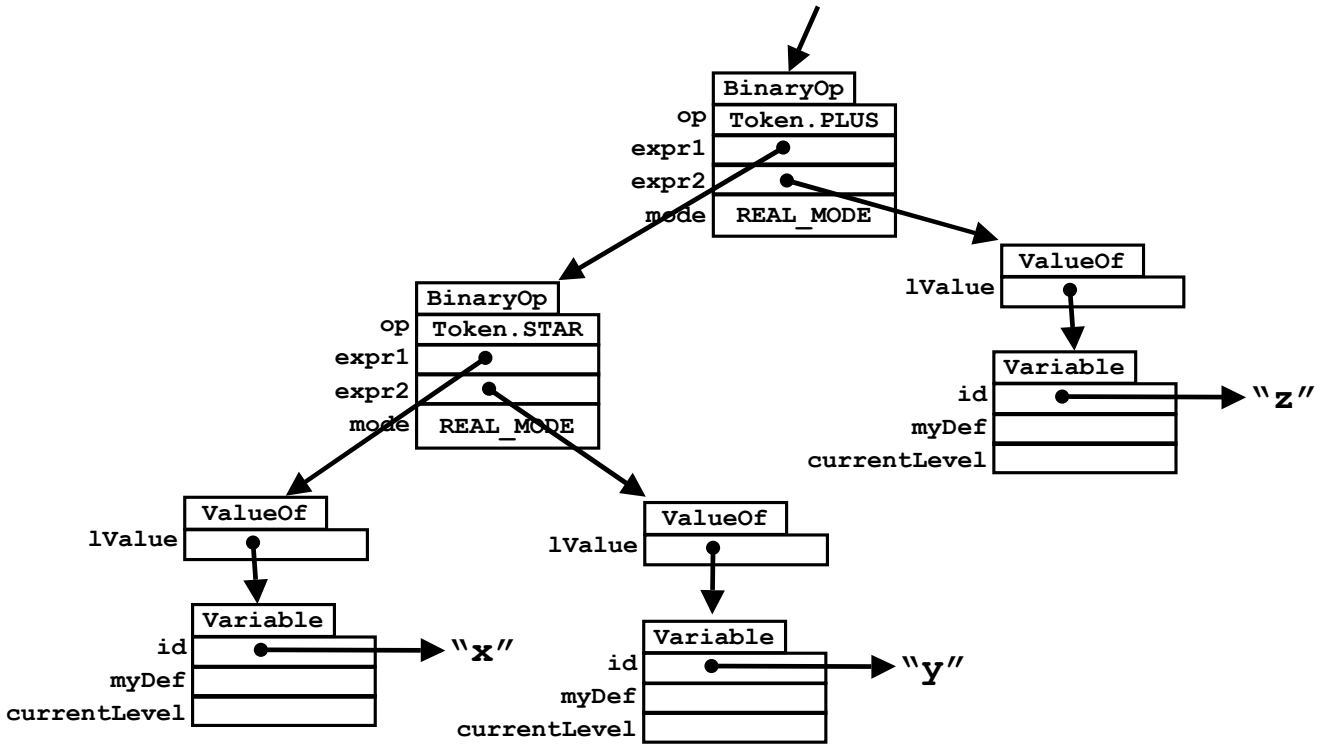
During type-checking, we looked the string up in the **SymbolTable** and made sure it was properly declared. At that time, we saved this information, which will come in handy during code generation, in the **myDef** field. The **myDef** field was set to point to the node that describes the declaration of that variable. A variable can be defined either in a variable declaration or as a formal parameter to a procedure. The **myDef** field will point to either an **Ast.VarDecl** or **Ast.Formal** node.

An **Ast.Variable** node is an L-Value; it is not an expression. The purpose of the **Ast.ValueOf** node is to "convert" an L-Value into an R-Value so it can be used in an expression.

Here is the class **Ast.ValueOf**:

```
static class ValueOf extends Expr {
    LValue   lValue;
}
```

Given these classes, we can now show how the AST is represented in Java. In the picture below, each object has a label on top showing its class. The fields are labeled to the left of the object.



The fields named **myDef** and **currentLevel** (in the **Variable** nodes) are semantic fields and are not shown here.

The original expression

x * y + z

occurs somewhere within a PCAT program, perhaps in an assignment statement. The objects shown above form a sub-tree, and the root of this sub-tree would be pointed to by some other node in the AST, perhaps by an **Ast.AssignStmt** node. Or perhaps this expression is a sub-expression in a larger expression like

a – (x * y + z) / 2

in which case this subtree would be pointed to by another **Ast.BinaryOp** node.

The AST Classes

Be sure to print out the document called "Ast Summary". It may be found in the project 4 directory, or at

```
www.cs.pdx.edu/~harry/compilers/p4/ASTSummary.pdf
```

This document shows all the syntactic fields in every kind of object. Although, it does not show the semantic fields, it is a good guide to understanding the AST.

Let's begin with the class hierarchy. The indentation show the class-subclass relationship. For example, **Stmt** extends **Node** (or we say "**Stmt** is a subclass of **Node**"). Likewise, **AssignStmt** is a kind of **Stmt**, so **AssignStmt** is a subclass of **Stmt**.

- **Node**
  - **Body**
  - **VarDecl**
  - **TypeDecl**
  - **TypeName**
  - **ProcDecl**
  - **Formal**
  - **CompoundType**
    - **ArrayType**
    - **RecordType**
  - **FieldDecl**
  - **Stmt**
    - **AssignStmt**
    - **CallStmt**
    - **ReadStmt**
    - **WriteStmt**
    - **IfStmt**
    - **WhileStmt**
    - **LoopStmt**
    - **ForStmt**
    - **ExitStmt**
    - **ReturnStmt**
  - **ReadArg**
  - **Expr**
    - **BinaryOp**
    - **UnaryOp**
    - **IntToReal**
    - **FunctionCall**

ArrayConstructor
RecordConstructor
IntegerConst
RealConst
StringConst
BooleanConst
NilConst
ValueOf
Argument
ArrayValue
FieldInit
LValue
Variable
ArrayDeref
RecordDeref

Node is an abstract class, which means there are no objects which are **Nodes**, without being something more specific, like a **Body** or an **AssignStmt** node.

Notice that there are many kinds of statements. **Stmt** is also an abstract class, with a subclass for each kind of statement that can appear in a PCAT program. For example, a "return" statement would be represented with a **ReturnStmt** node.

Also notice that there are many kinds of expression. **Expr** is an abstract class. For example, an integer can, by itself, appear anywhere an expression can appear. The integer would be represented with an **IntegerConst** object.

The class **LValue** is also abstract. There are three kinds of L-Value: a variable, an array dereference (such as "a[i]") and a record dereference (such as "myRec.name").

Finally, **CompoundType** is abstract. There are two kinds of compound type: **ArrayType** and **RecordType**.

Many of the classes correspond to syntactic categories in the PCAT grammar. For example, there is a one-to-one correspondence between the kinds of PCAT statements shown in the grammar rules and the subclasses of **Stmt**.

Take a look at the grammar on the last page of the PCAT Reference Manual.

For example, here are the rules for LValue:

```
LValue   → ID
         → LValue '[' Expression ']'
         → LValue '.' ID
```

and notice that the class **LValue** has 3 subclasses: **Variable**, **ArrayDeref**, and **RecordDeref**.

The Fields in the AST Classes

Next we give a summary of all the fields in the classes. (I believe the following information is accurate, but the file **Ast.Java** is the ultimate reference.)

The *** indicates a semantic field, which is filled in after parsing.

Node
lineNumber: int
Body
typeDecls: TypeDecl
procDecls: ProcDecl
varDecls: VarDecl
stmts: Stmt
VarDecl
id: String
typeName: TypeName
expr: Expr
next: VarDecl
lexLevel: int ***
TypeDecl
id: String
compoundType: CompoundType
next: TypeDecl
TypeName
id: String
myDef: CompoundType ***
ProcDecl
id: String
retype: TypeName
formals: Formal
body: Body
next: ProcDecl
lexLevel: int ***
Formal
id: String
typeName: TypeName
next: Formal
lexLevel: int ***

CompoundType
ArrayType
elementType: TypName
RecordType
fieldDecls: FieldDecl
FieldDecl
id: String
typeName: TypeName
next: FieldDecl
Stmt
next: Stmt
AssignStmt
lValue: LValue
expr: Expr
CallStmt
id: String
args: Argument
myDef: ProcDecl ***
ReadStmt
readArgs: ReadArg
WriteStmt
args: Arguments
IfStmt
expr: Expr
thenStmts: Stmt
elseStmts: Stmt
WhileStmt
expr: Expr
stmts: Stmt
LoopStmt
stmts: Stmt
ForStmt
lValue: LValue
expr1: Expr
expr2: Expr
expr2: Expr
stmts: Stmt
ExitStmt
myLoop: Stmt ***
ReturnStmt
expr: Expr
myProc: ProcDecl

Expr
BinaryOp
    op: int
    expr1: Expr
    expr2: Expr
    mode: int ***
UnaryOp
    op: int
    expr: Expr
    mode: int ***
IntToReal ***
    expr: Expr
FunctionCall
    id: String
    args: Argument
    myDef: ProcDecl ***
ArrayConstructor
    id: String
    values: ArrayValues
    myDef: TypeDecl ***
RecordConstructor
    id: String
    values: FieldInits
    myDef:TypeDecl ***
IntegerConst
    iValue: int
RealConst
    rValue: double
StringConst
    sValue: String
BooleanConst
    iValue: int
NilConst
ValueOf
    lValue: LValue
Argument
    expr: Expr
    mode: int ***
    next: Argument
ArrayValue
    countExpr: Expr
    valueExpr: Expr
    next: ArrayValue

FieldInit
id: String
expr: Expr
myFieldDecl: FieldDecl ***
next: FieldInit
LValue
Variable
id: String
myDef: Node (either VarDecl or Formal) ***
currentLevel: int ***
ArrayDeref
lValue: LValue
expr: Expr
RecordDeref
lVlaue: LValue
id: String
myFieldDecl: FieldDecl ***

Representing Lists

In PCAT, there are a number of syntactic constructs that repeat. For example, following the ELSE keyword, there can be zero or more statements. As another example, there can be zero or more formal parameters in a procedure declaration.

Here is a PCAT procedure declaration, with eight formal parameters:

```
procedure foo (a,b,c,d,e: int; f,g,h: real) is
  begin
    ...
  end;
```

Sequences are represented with linked lists in the AST. For example, here is a portion of the class **ProcDecl**. For a procedure like "foo", there will be a single **ProcDecl** node and it will point to a linked list of objects, one for each formal parameter.

```
static class ProcDecl extends Node {
  ...
  Formal      formals;
  ...
}
```

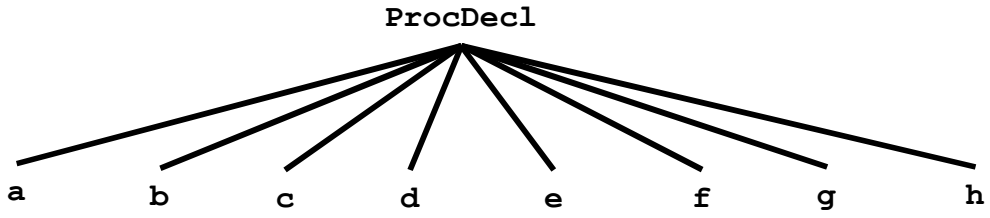
Each formal parameter is represented by an **Formal** object. Each **Formal** object contains a **next** pointer.

```
static class Formal extends Node {
  ...
  Formal      next;
}
```

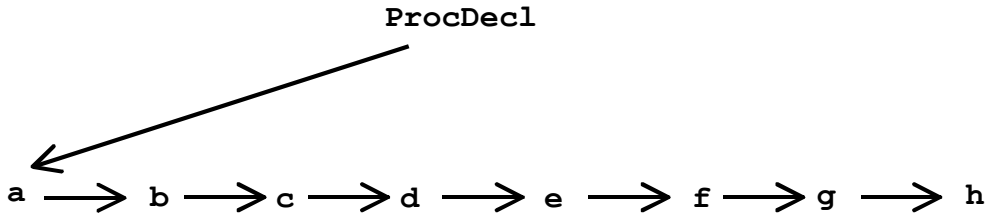
To go though a linked list, we might use code like this:

```
for (Formal f = formals; f = f.next; f) {
  ... f ...
}
```

Conceptually, we might think of a single **ProcDecl** node which is capable of accommodating zero-or-more children, like this:



To represent such a node (with a variable number of children), we'll build a structure more like this:



[Another design approach to dealing with multiple children is to represent sequences with arrays. In fact, Andrew Tolmach uses arrays for representing sequences.]

Our AST will use linked lists to represent sequences of "zero-or-more" occurrences for several sorts of sequences. The following classes all contain a field called **next** and their objects can be parts of lists:

VarDecl	variable declarations
TypeDecl	type declarations
ProcDecl	procedure declarations
Formal	formal parameters to procedures
FieldDecl	fields within a record
Stmt	statements
ReadArg	arguments to a "read" statement
Argument	arguments to a procedure invocation
ArrayValue	count-value pairs in an array initialization
FieldInit	initial values for fields in a record initialization

Statement Sequencing

In a number of places in the PCAT syntax, there can be zero-or-more statements. For example, a "while" loop can contain zero-or-more statements in its body and an "if" statement can have zero or more statements in its "then" part or in its "else" part.

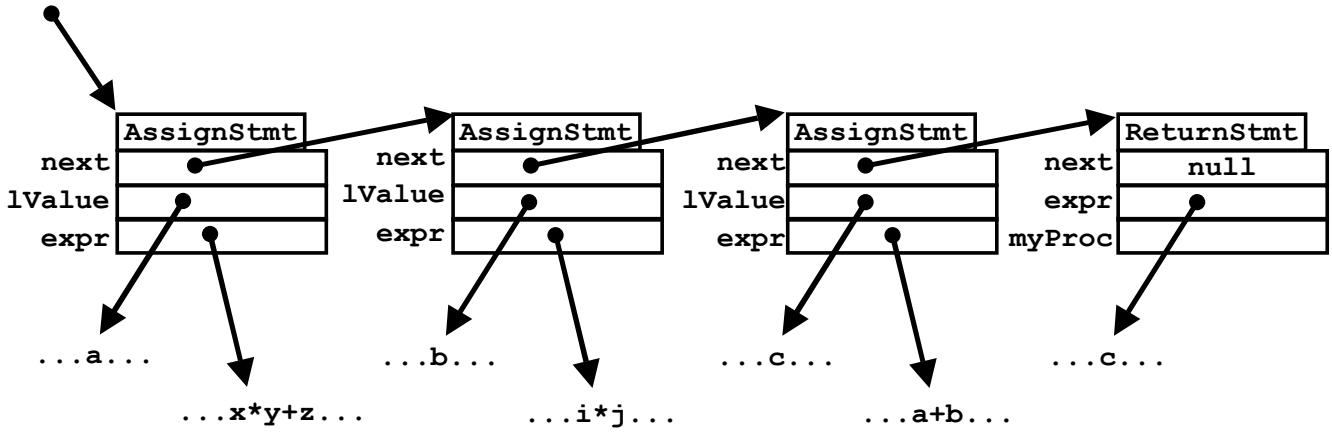
For example, here is a sequence of 4 statements:

```
a := x * y + z;
b := i * j;
c := a + b;
return c;
```

Every statement node has a **next** field, so each statement can be a member of a linked list.

```
abstract static class Stmt extends Node {
  Stmt next;
}
...
static class AssignStmt extends Stmt {
  LValue lValue;
  Expr   expr;
}
...
```

Here is how we would represent the sequence of 4 statements. (The sub-trees are not shown in detail and the **myProc** pointer in the **ReturnStmt** node is a semantic pointer that points back upward into the AST and it is not shown.)



Sequences of statements can occur in many places. For example, the general form of an "if" statement in PCAT is...

```
if ...expression... then
  ...list of "then" statements...
else
  ...list of "else" statements...
end;
```

Here is **IfStmt**:

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

The **thenStmts** field points to a linked list of statements and the **elseStmts** field points to another linked list of statements. Of course, either of these lists could be null. Here is an example where **elseStmts** would be null:

```
if a=4 then
  x := b;
  y := c;
  z := d;
end;
```

Here is an example where **thenStmts** would be null.

```
if x<9 then
else
  y := 43;
end;
```

In fact, both could be null, as in the following program fragment:

```
if n < 0 then
    (* n := 0; *)
else
    (* not yet implemented *)
end;
```

In the AST classes, whenever a field points to a linked list of nodes, the name of the field is pluralized. In this example, the fields **thenStmts** and **elseStmts** end with an "s" indicating that they point to a linked list of **Stmt** nodes.

By the way, the "if" statement in the PCAT grammar contains zero-or-more "elseif" clauses. Here are the relevant grammar rules. (In the grammar rules, keywords are in boldface and the braces mean "zero-or-more occurrences". The brackets mean "optional".)

```
Statement  → ...
           → if Expression then {Statement}
             {elseif Expression then {Statement}}
             [else {Statement}] end ';'
           → ...
```

Here is an example:

```
if tmp < 0 then
    desc := 1;
elseIf tmp < 32 then
    desc := 2;
elseIf tmp < 55 then
    desc := 3;
elseIf temp < 80 then
    desc := 4;
else
    desc := 5;
end;
```

Unfortunately, the **IfStmt** class contains only room for a THEN statement list and an ELSE statement list, but nothing that could obviously accommodate ELSEIF.

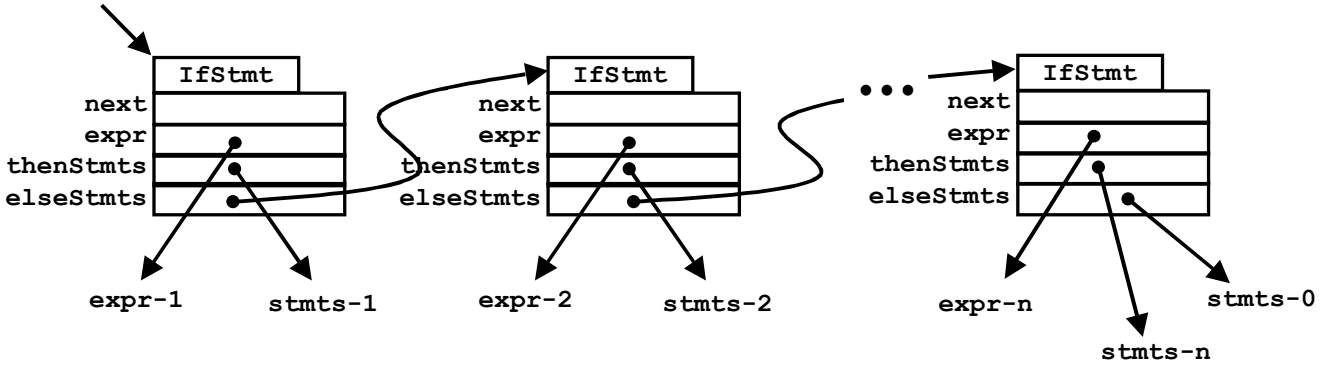
To deal with ELSE-IF clauses, note that:

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

is semantically 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 a program containing the first IF-ELSEIF-ELSEIF-END statement, the parser will build exactly the same tree as it would for the set of nested IF-THEN-ELSE statements. The tree it constructs is shown next. In that diagram, the "tree" has been visually distorted to suggest that a list of ELSEIF clauses can be viewed as a linked list of **IfStmt** nodes, linked on the **elseStmts** field.



The "main" Method

Here is the **main** method of our compiler (somewhat simplified):

```
Ast.Body ast;
Parser parser;
Checker checker;
Generator generator;
...
// Parse the source and return the AST...
parser = new Parser (args);
ast = parser.parseProgram ();

// Check the AST...
checker = new Checker ();
checker.checkAst (ast);

// Generate the IR...
generator = new Generator ();
generator.generateIR (ast);

// Print the IR...
IR.printIR ();
```

First, the code creates a **Parser** object and then invokes the **parseProgram** method on it to scan the source and build a AST. Then the code creates a **Checker** object and then invokes the **checkAst** method on it.

Next, it creates a **Generator** object and invokes the code you will write in Project 8 to generate intermediate instructions. Finally, it invokes the **printIR** method, which I am providing, to print out the results.

There is also some error checking in **main**, which is not shown here. Basically, if any errors occur during the front-end processing, the **main** method will abort before invoking the **generateIR** method.

Dealing With Errors

In Project 8, you should not encounter any errors, since the source PCAT program was thoroughly checked during the front-end processing. Any lexical, syntactic, or semantic errors will have been caught by the **Lexer**, the **Parser**, and the **Checker** code. The AST passed to the generator will be correct.

Errors in the front-end are handled in two ways. If the error is recoverable, then a message is printed on **stderr** and processing continues. If the error is unrecoverable, then an exception is thrown. If any exceptions are thrown, they are caught way up in **main** and the compiler will terminate after printing the error message.

The compiler uses two exceptions: **FatalError** and **LogicError**.

You may want to throw **LogicError** in the code you write. As you are coding, when there is some condition you expect to be true—but a part of you wonders whether it will really always be true—it is often a good idea to add some code to check it to make sure. If this condition turns out to be not true sometime, then it means something is dreadfully wrong. Either your code contains a bug or you have somehow misunderstood the problem. The thing to do is throw **LogicError**.

For example, I said previously that the **BinaryOp** node contains an **op** field, which tells which operator it represents. I said that the **op** should have certain values, and your code might contain a "switch" on the **op** field to handle each value differently. A common mistake is to forget some case. Another problem occurs if the program is changed and a new case is added, but you forget to update the "switch".

So here is how you might want to code the "switch" statement:

```
switch (binNode.op) {
    case Token.PLUS:
        ...
    case Token.MINUS:
        ...
    ...
    case Token.GREATER:
        ...
    case Token.GEQ:
        ...
    default:
        throw new LogicError ("Unknown binNode.op in genBinaryOp");
}
```

Since many of our methods are mutually recursive and will call other methods in complex patterns, dependent on the exact shape on an AST, it means that an exception can arise when just about any method is active. Java requires you to say, in the header of a method, which exceptions might arise while the method is active. The easiest thing to do is to include **throws FatalException** in every method. For example:

```
Ast.Node genBinaryOp (Ast.BinaryOp binNode, ...)
    throws FatalError
{
    ...
    switch (binNode.op) {
        ...
```

This works since **LogicError** is a kind of **FatalError**.

The Superclass "Node"

The class **Node** has one field, called **lineNumber**, which means that every node will inherit a **lineNumber** field. This integer gives the source code line number on which the corresponding source construct appeared.

For example, an **IfStmt** node might have a **lineNumber** of 46, indicating that the "if" statement it represents was found on—or at least began on—line 46.

Having a line number for every node is useful when printing error messages during type checking. To print an error message, we can supply the relevant AST node to the error routine, and the error handling code will extract the line number and use it in printing the error message.

Here is a message that might be printed during type checking:

> Error on line 46: Conditional expr after IF, ELSEIF, or WHILE is not BOOLEAN

Since we do not expect errors in the back-end, you can safely ignore the **lineNumber** field.

The "Body" Node

A PCAT program consists of a "body". Here are the relevant grammar rules:

```
Program  → program is Body ';'
Body     → {Declaration} begin {Statement} end
```

Here is an example program:

```
program is
    var x,y: integer;
    begin
        x := 5;
        y := x * 3;
    end;
```

A procedure also contains a "body":

```
ProcedureDecl  → ID FormalParams [':' TypeName] is Body ';'
```

Here is an example procedure declaration, which might be found inside a larger program:

```
...
procedure foo (a,b,c: integer) : boolean is
    var x,y: integer;
    begin
        x := 5;
        y := x * 3;
    end;
...
```

In both examples, the body contains a variable declaration and a sequence of two statements.

There are three kinds of declarations that can appear in a body: variable, type, and procedure. A body can have any number of declarations. Our example defines two variables. A body can also include several type declarations and several procedure declarations.

Here is **Body**:

```
static class Body extends Node {
    TypeDecl    typeDecls;
    ProcDecl    procDecls;
    VarDecl     varDecls;
    Stmt        stmts;
    int         frameSize;              // Not used until proj 9
}
```

Each of the fields **typeDecls**, **procDecls**, **varDecls**, and **stmts** will point to a linked list. If there are no types, then **typeDecls** will be null. Likewise, **procDecls**, **varDecls** and **stmts** may be null.

This example shows that the **Ast.java** file contains a few fields (e.g., **frameSize**) that will be used later in the class, and these are commented appropriately. Just ignore **frameSize** until project 9, when we'll talk about it. Also, some of the semantic fields were added during projects 5 and 6, so a few of the fields will be commented like this:

```
static class Variable extends LValue {
    String      id;
    Node        myDef;                  // Not used until proj 5
    ...
}
```

(It turns out that you'll need to use the **myDef** field in project 8.)

[Elsewhere in this document, I am replacing fields that are not needed until later with "..." to keep things simpler.]

Variable Declarations

Here are the grammar rules concerning variable declarations:

```
Declaration   → var VarDecl {VarDecl}
              → type TypeDecl {TypeDecl}
              → procedure ProcedureDecl {ProcedureDecl}
VarDecl       → ID { ',' ID} [ ':' TypeName ] ':=' Expression ';'
TypeName      → ID
```

Here are three examples of variable declarations you might find at the beginning of a "body":

Example #1:

```
var x, y, z: integer := 0;
var a, b, c: integer := 2;
```

Example #2:

```
var x, y, z: integer := 0;
    a, b, c: integer := 2;
```

Example #3:

```
var x: integer := 0;
    y: integer := x;
    z: integer := x;
    a: integer := 2;
    b: integer := a;
    c: integer := a;
```

All three are semantically identical and all three will be represented the same way in theAST. Namely, there will be a linked list of 6 **VarDecl** nodes.

[As a minor point, there will only be two **TypeName** nodes created in examples 1 and 2. Each of the six **VarDecl** nodes will point to one of these **TypeName** nodes. For example 3, there will be 6 **TypeName** nodes created, so actually there is a minor difference in the AST trees. In the first two examples, the **TypeName** nodes will have more than one parent and (technically) our AST will be a DAG, not a tree. However, you can safely ignore this point.]

Note that every variable must be initialized in PCAT. There is no way to pick up uninitialized data.

The **VarDecl** class is used to represent variable declarations:

```
static class VarDecl extends Node {
    String      id;
    TypeName    typeName;
    Expr        expr;
    VarDecl     next;
    int         lexLevel;                // Not used until proj 5
    ...
}
```

In the example we are looking at, 6 **VarDecl** objects will be needed, one for each variable. For example, it will take one **VarDecl** object to represent:

```
a: integer := 2;
```

Each **VarDecl** is for a single variable and the **id** points to the name of this variable, "a" in this case. Each variable has a type, which is given with a simple name. The name can be one of the predefined types:

```
integer
real
boolean
```

or it can be a user-defined type that was declared in a type declaration:

```
type MyArray is array of real;
...
var a: MyArray := ...;
```

In any case, the **typeName** field points to a **TypeName** object which contains the name of the type. Each variable must be initialized and the **expr** field points to an expression, which can be quite complex, as in:

```
var a: integer := x*y+z;
```

The linked list of **VarDecls** is chained together on the **next** field.

"Body"s can be nested in PCAT and each body has a "lexical level", which is sometimes called a "nesting depth". During final code generation, it will be important to know at which lexical level a variable was defined. During type checking, we stored that lexical depth in the **VarDecl** in the field **lexLevel**, but we won't need it again until project 11.

Type Declarations

In PCAT, the programmer can work with array and record types, but they must be declared in "type declarations". Here are the relevant grammar rules:

```
Declaration   → var VarDecl {VarDecl}
              → type TypeDecl {TypeDecl}
              → procedure ProcedureDecl {ProcedureDecl}
TypeDecl      → TypeName is CompoundType ';'
TypeName      → ID
CompoundType  → array of TypeName
              → record FieldDecl {FieldDecl} end
FieldDecl     → ID ':' TypeName ';'
```

Here is are four examples of the "FieldDecl" grammatical construct:

```
f2: real;
nums: MyArray;
age: integer;
married: boolean;
```

Here is are two examples of the "CompoundType" grammatical construct:

```
array of integer;
record age: integer; married: boolean; end;
```

Here are two examples of the "TypeDecl" grammatical construct:

```
MyArray is array of integer;
MyNewType is record age: integer; married: boolean; end;
```

Here is an example of the "Declaration" grammatical construct, involving only type declarations:

```
type MyArray is array of integer;
     MyRecord is record
                    f1: integer;
                    f2: real;
                    f3: boolean;
                  end;
     RecVec is array of MyRecord;
```

We say that "arrays" and "records" are "compound types", since they are made up of other types like "integer" , "real", or even other compound types. A type is either a compound type (like **MyArray**, **MyRecord**, and **RecVec**) or a basic type (like **integer**, **real**, or **boolean**).

Each compound type must be named in a type declaration and each type defined in a declaration must be a compound type. Therefore, each **TypeDecl** associates a name with either an array type or a record type.

Here is **TypeDecl**:

```
static class TypeDecl extends Node {
    String       id;
    CompoundType compoundType;
    TypeDecl     next;
}
```

The **id** gives the name (e.g., "MyArray") and the **compoundType** field points to either an **ArrayType** object or an **RecordType** object. Each **Body** points to a linked list (possibly empty) of **TypeDecl** nodes, linked on their **next** fields. To represent the declarations of **MyArray**, **MyRecord**, and **RecVec**, the AST will contain a linked list of 3 **TypeDecl** objects.

Note that a "Body" begins with "zero-or-more" declarations:

```
Body  → {Declaration} begin {Statement} end
```

Consequently, the PCAT programmer could also write the above example as shown below. Both would mean the same and would be represented in the AST identically. Namely, both would be represented with a linked list of 3 **TypeDecl** objects.

```
type MyArray is array of integer;
type MyRecord is record
                    f1: integer;
                    f2: real;
                    f3: boolean;
                  end;
type RecVec is array of MyRecord;
```

Mixing Up Declarations

The PCAT grammar allows a single body to contain lots of variable, type, and procedure declarations, all mixed together. Here is an example involving many variable declarations and type declarations:

```
var x, y, z: integer := 0;
type MyArray is array of integer;
     MyRecord is record
                    f1: integer;
                    f2: real;
                    f3: boolean;
                  end;
var w: real := 0.0;
var a, b, c: integer := 0;
    i, j, k: integer := 0;
type RecVec is array of MyRecord;
var q1, q2, q3: integer := 0;
type BoolArr is array of boolean;
type ArrTyp1 is array of integer;
     ArrTyp2 is array of ArrTyp1;
     ArrTyp3 is array of ArrTyp2;
```

The parser will sort all these out. This example declares 13 variables:

x, **y**, **z**, **w**, **a**, **b**, **c**, **i**, **j**, **k**, **q1**, **q2**, and **q3**

and 7 types:

MyArray, **MyRecord**, **RecVec**, **BoolArr**, **ArrTyp1**, **ArrTyp2**, and **ArrTyp3**

The AST will contain a linked list of 13 **VarDecl** nodes and 7 **TypeDecl** nodes. The exact grouping information is not semantically important and is not captured in the AST.

To put it another way, the list organization in the AST would be the same as the parser would produce for the following set of declarations:

```
type MyArray is array of integer;
type MyRecord is record
                    f1: integer;
                    f2: real;
                    f3: boolean;
                end;
type RecVec is array of MyRecord;
type BoolArr is array of boolean;
type ArrTyp1 is array of integer;
type ArrTyp2 is array of ArrTyp1;
type ArrTyp3 is array of ArrTyp2;
var x: integer := 0;
var y: integer := x;
var z: integer := x;
var w: real := 0.0;
var a: integer := 0;
var b: integer := a;
var c: integer := a;
var i: integer := 0;
var j: integer := i;
var k: integer := i;
var q1: integer := 0;
var q2: integer := q1;
var q3: integer := q1;
```

Likewise, if there are several procedure declarations mixed in among the other declarations, they too will be grouped together into a single linked list of **ProcDecl** nodes.

Array Types

Here is **ArrayType**:

```
static class ArrayType extends CompoundType {
    TypeName     elementType;
}
```

The only field, **elementType**, names the type of the array elements. For example, in

```
 array of MyRecord
```

each element in the array will be a record. (This presumes there is a type declaration for **MyRecord** elsewhere in the PCAT program.) In the **ArrayType** node representing

```
array of MyRecord
```

the **elementType** field will point to a **TypeName** node, whose **id** will name "MyRecord".

Record Types

PCAT supports record types. A record is like a "C" struct.

You can also think of a record as being similar to an "object", with the following differences: there are no methods associated with records and record types are not related to each other the way classes are related to other classes in the class-subclass hierarchy.

Here is an example record type from a PCAT program:

```
record
  f1: integer;
  f2: real;
  f3: boolean;
end;
```

The only place a record type can be used is in a type declaration, such as:

```
type MyRecord is record
                   f1: integer;
                   f2: real;
                   f3: boolean;
                 end;
```

A record type has a sequence of fields. Each field has a name and a type. In an AST, we use two kinds of nodes to represent a record type: **RecordType** and **FieldDecl**.

```
static class RecordType extends CompoundType {
    FieldDecl    fieldDecls;
    ...
}

static class FieldDecl extends Node {
    String       id;
    TypeName     typeName;
    FieldDecl    next;
    ...
}
```

[This may be a little confusing, since we are using "fields" in Java objects to represent "fields" in PCAT record types!!!]

The **RecordType** object contains a single field, called **fieldDecls**, which points to a linked list of **FieldDecl** nodes. There is one **FieldDecl** node for each field in the record. Our example has three fields (named **f1**, **f2**, and **f3**) so the linked list would have three nodes, each linked on **next**.

Each **FieldDecl** has the name of the field in **id**. For example, **id** might point to the string "f1". Each **FieldDecl** also has a **typeName**, which will point to a **TypeName** node. For example, the first **FieldDecl** in the list will point to a **TypeName** whose id is "integer".

Procedure Declarations

Here are the PCAT grammar rules for a procedure declaration:

```
ProcedureDecl   → ID FormalParams [':' TypeName] is Body ';'
FormalParams    → '(' FormalSection {';' FormalSection} ')'
                → '(' ')'
FormalSection   → ID {',' ID} ':' TypeName
```

Here is an example procedure declaration:

```
procedure foo (a,b: integer; i,j,k: real; x: integer;) : boolean is
    var t1, t2: integer;
    type MyArray is array of real;
         MyArray2 is array of boolean;
    procedure bar1 (...) ... is ... begin ... end;
    procedure bar2 (...) ... is ... begin ... end;
    begin
        t1 := a+b;
        t2 := i+j+k;
        if t1 > t2 then
            return false;
        end;
        return x<0;
    end;
```

Notice that a procedure may or may not return a value. In this example, **foo** happens to return a boolean value. If a procedure returns a value, it is called a "function". The " : return-type" part is optional. If it is not present, then it's called a "void procedure" and it doesn't return anything.

You can see that each procedure has an ID (e.g., **foo**), a list of zero or more formal parameters (e.g., **a**, **b**, **i**, **j**, **k**, and **x**), and optional return type (e.g., **boolean**). A procedure also has a body. In this example, the body is rather long. The body is shown below. It contains a bunch of stuff (including two nested procedures!) but it is all represented with an AST subtree whose root is a **Body** node.

```
var t1, t2: integer;
type MyArray is array of real;
     MyArray2 is array of boolean;
procedure bar1 (...) ... is ... begin ... end;
procedure bar2 (...) ... is ... begin ... end;
begin
    t1 := a+b;
    t2 := i+j+k;
    if t1 > t2 then
        return false;
    end;
    return x<0;
end;
```

To represent a procedure declaration, we use two kinds of node: **Ast.ProcDecl** and **Ast.Formal**:

```
static class ProcDecl extends Node {
    String      id;
    Formal      formals;
    TypeName    retType;
    Body        body;
    ProcDecl    next;
    int         lexLevel;               // Not used until proj 5
}
static class Formal extends Node {
    String      id;
    TypeName    typeName;
    Formal      next;
    int         lexLevel;               // Not used until proj 5
    ...
}
```

You can see that **ProcDecl** has a field for each of the components in the declaration. The **id** points to the name of the procedure (e.g., "foo"). The **formals** field points to a linked list of **Formal** nodes. There will be one **Formal** node for each formal parameter. In this example, the list would contain 6 **Formal** nodes, one each for **a**, **b**, **i**, **j**, **k**, and **x**. For other procedures, **formals** would be null if there were no parameters. If there is a return type, then **retType** will point to a **TypeName** (e.g., for "boolean" in this example). The **body** field will point to a **Body** node, which will represent all the rest of the procedure, namely the local declarations and the statement list.

ProcDecls are kept on a linked list, and the **next** field is used for the list of **ProcDecl** nodes. (By the way, each procedure declaration is contained within some "enclosing" body. Recall that each **Body** node contains a pointer, called **procDecls**, to a linked list of **ProcDecl** nodes.)

More precisely, procedures are contained in bodies and bodies are contained in procedures. As we go deeper and deeper, the nesting level gets greater and greater. The nesting depth could be kept in the **Body** nodes or it could kept in **ProcDecl** nodes; we have chosen to keep it in the **ProcDecl** nodes. This is the **lexLevel** field in **ProcDecl**.

Each **Formal** node contains the name of the parameter in the **id** field, (e.g., "a") and the type of the paratemer in **typeName**. They are linked on their **next** fields.

Constants in Expressions

Expressions can contain several kinds of constant values.

Expressions can contain integers, as in:

```
i + 123
```

Expressions can contain real values, as in:

```
(3.1415 * dia) < 1.0
```

Expressions can contain boolean constants **true** and **false**, as in:

```
(b1 == b2) and (b3 == false) and (b4 == b5) and (b6 == true)
```

Expressions can contain the **nil** constant, as in:

```
if (p == nil) then ...
```

String constants can be used as arguments in "write" statements, as in:

```
write ("The value is ", x);
```

To represent constants, we use the classes **IntegerConst**, **RealConst**, **BooleanConst**, **NilConst**, and **StringConst**. Here are their definitions:

```
static class IntegerConst extends Expr {
    int       iValue;
}
static class RealConst extends Expr {
    double    rValue;
    ...
}
static class BooleanConst extends Expr {
    int       iValue;
}
static class NilConst extends Expr {
}
static class StringConst extends Expr {
    String    sValue;
    ...
}
```

In **IntegerConst**, the field **iValue** will contain the value of the constant, e.g., 123.

In **RealConst**, the field **rValue** will contain the value of the constant, e.g., 3.1415.

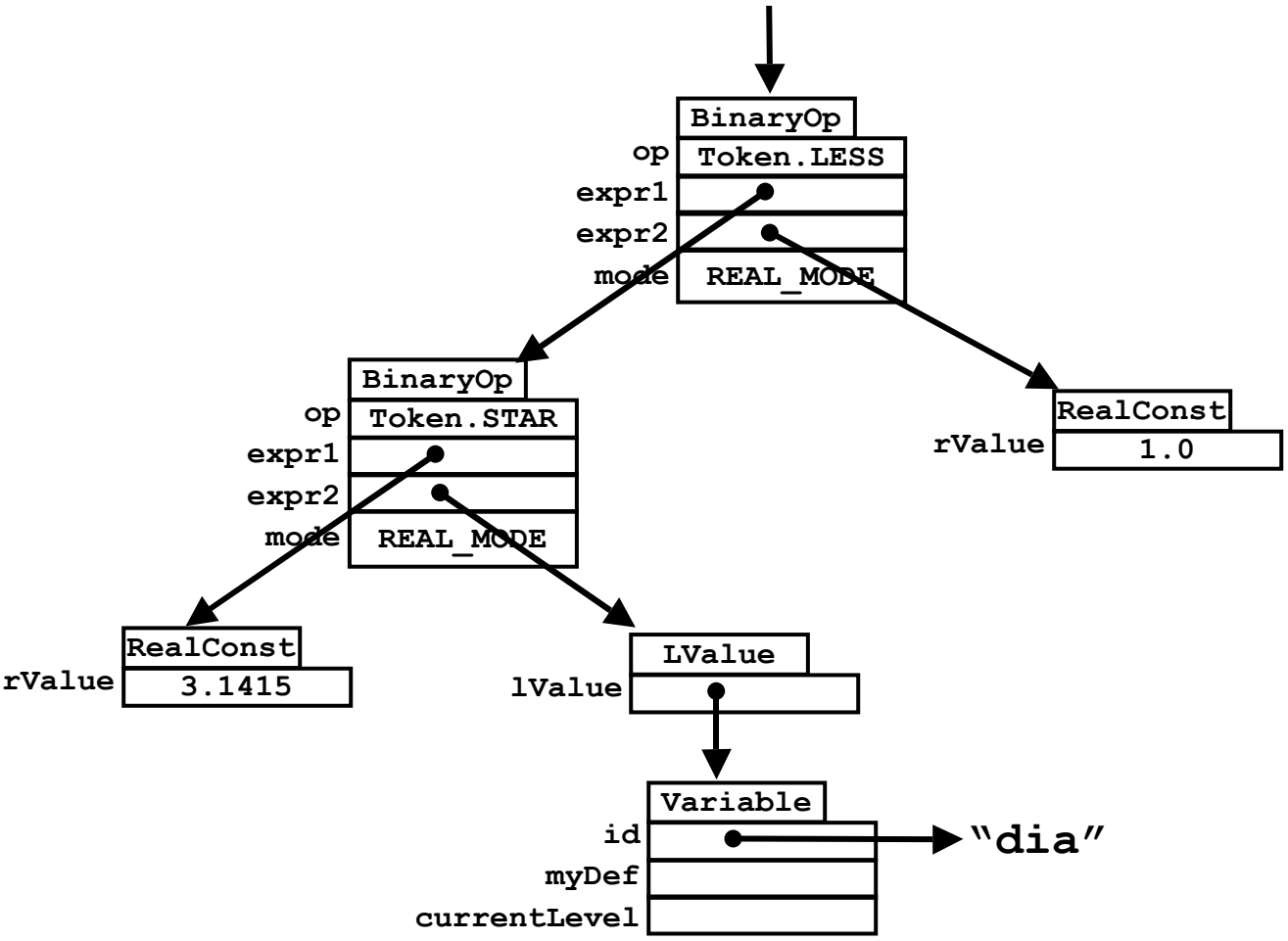
In **BooleanConst**, the field **iValue** will contain 1 for “true” or 0 for “false”.

In **StringConst**, the field **sValue** will contain the value of the constant, e.g., “The value is ”.

These values will appear in the AST in expression subtrees. For example, the expression

```
(3.1415 * dia) < 1.0
```

would be represented as:



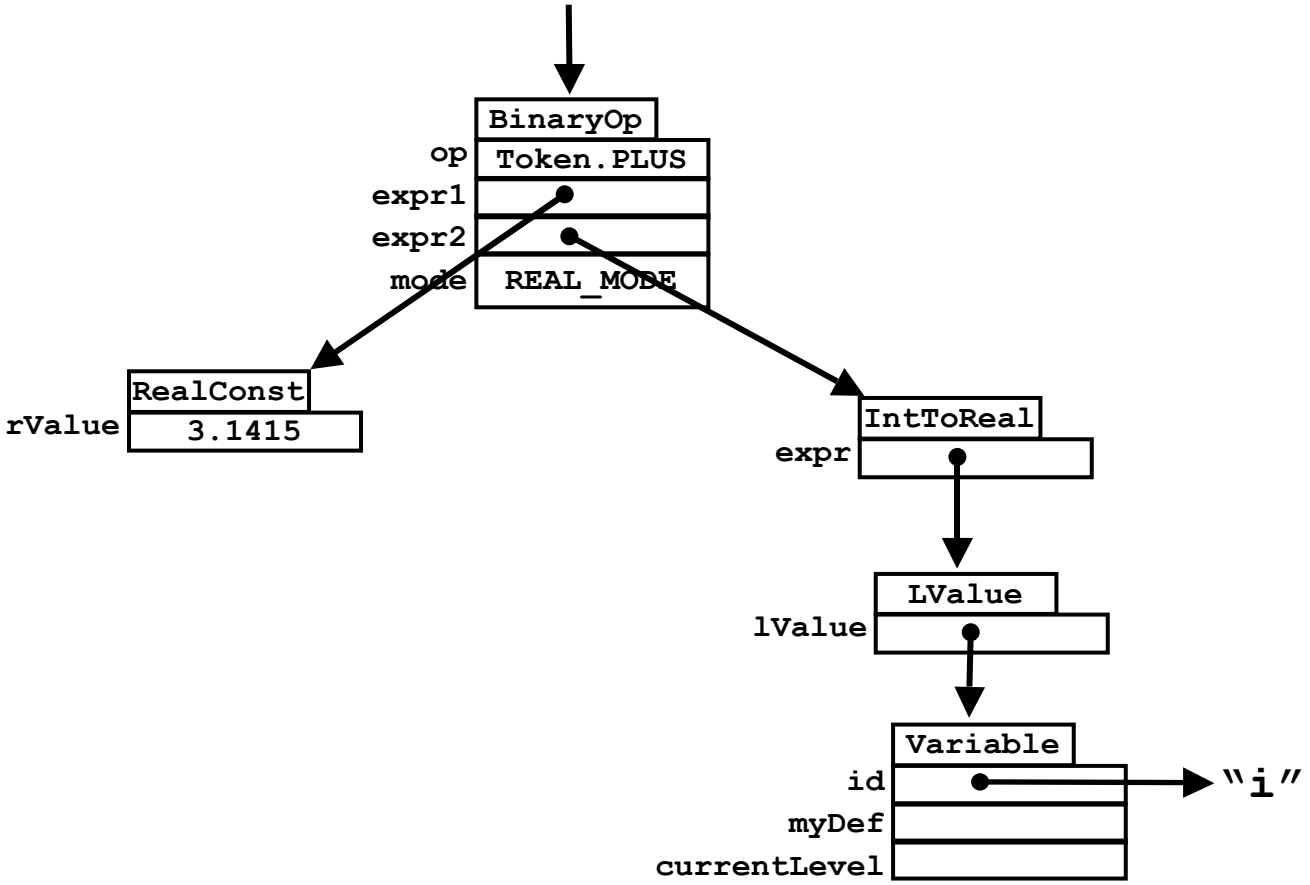
Integer-To-Real Conversions

According to the semantics of PCAT, an integer-to-real conversion must be inserted by the compiler in certain cases.

For example, in the following expression, assume that "i" has type **integer**. When integers are added to reals, the compiler must first insert a data-type conversion.

3.1415 + i

Fortunately, the need for a conversion was detected during type checking and, at that time, the AST was modified by the insertion of an extra node to represent the needed conversion. This expression would be represented like this:



When this expression is printed out by **PrettyPrint**, it will print as follows, to show the presence of the conversion in the AST.

```
(3.1415 + intToReal ( i ) )
```

PCAT only uses an integer-to-real conversion; no other data type conversions (like short-to-long, or boolean-to-integer) are necessary.

CallStmts versus FunctionCalls

A procedure may return a value or not. A "void" procedure does not return a value. The declaration of a void procedure will not have a return type; a non-void procedure will have return type. Here are two procedures:

```
procedure foo (a: integer) : integer is
    begin
        return a * 100;
    end;
procedure bar (x: integer) is
    begin
        if x<0 then
            write ("The value is negative");
        end;
    end;
```

The procedure **foo** returns a value. It must be invoked in a "function call", which can only occur in an expression. In other words, the PCAT programmer may only invoke **foo** in places that expressions may be used, which means that the returned value will always end up being used somewhere. Here are some legal invocations of **foo**:

```
i := foo (4);
if foo(j) < 34 then ...
write (foo(4*k));
```

The procedure **bar** does not return a value. It may only be invoked in a "call statement". Here are some legal invocations of **bar**:

```
i := 43;
bar(i*j);
bar(k-(4*i));
```

A non-void procedure like **foo** cannot be invoked in a call statement, like this:

```
foo (i);    (* error *)
```

Likewise, a void procedure like **bar** cannot be invoked within an expression. Both of these errors have been checked for in the front-end.

[Note that in "C" and Java it is legal to invoke a non-void procedure at the statement level. The returned value is simply discarded.]

When a void procedure is invoked at the statement level, it is done in a "call" statement, which is represented in the AST with a **CallStmt** node. Note that **CallStmt** is a kind of **Stmt**.

```
static class CallStmt extends Stmt {
    String      id;
    Argument    args;
    ProcDecl    myDef;   // Not used until proj 5
}
```

Here are the relevant grammar rules for a "call" statement:

```
Statement  → ...
           → ID Arguments ';'
           → ...
Arguments  → '(' Expression {',' Expression} ')'
           → '(' ')'
```

There can be any number of argument expressions. For example, in the following "call" statement:

```
myproc (4, j+k, x*y*z, n+1);
```

there are 4 expressions to be passed as arguments. The **args** field points to a linked list of **Argument** nodes. There will be one **Argument** node for each expression. The i-th **Argument** node contains a pointer to the i-th expression.

Here is the definition **Argument**:

```
static class Argument extends Node {
    Argument    next;
    Expr        expr;
    int         mode;                    // Not used until proj 6
}
```

The **mode** field is not used for **Arguments** of **CallStmts** and **FunctionCalls**. The **Argument** node is also used for **WriteStmt** nodes and, for these, the **mode** field is important.

Recall that **CallStmt** is a kind of **Stmt** node. Next, we show **FunctionCall**, which is a kind of **Expr** node.

```
static class FunctionCall extends Expr {
    String      id;
    Argument    args;
    ProcDecl    myDef;                   // Not used until proj 5
}
```

Here are the grammar rules relevant to function calls:

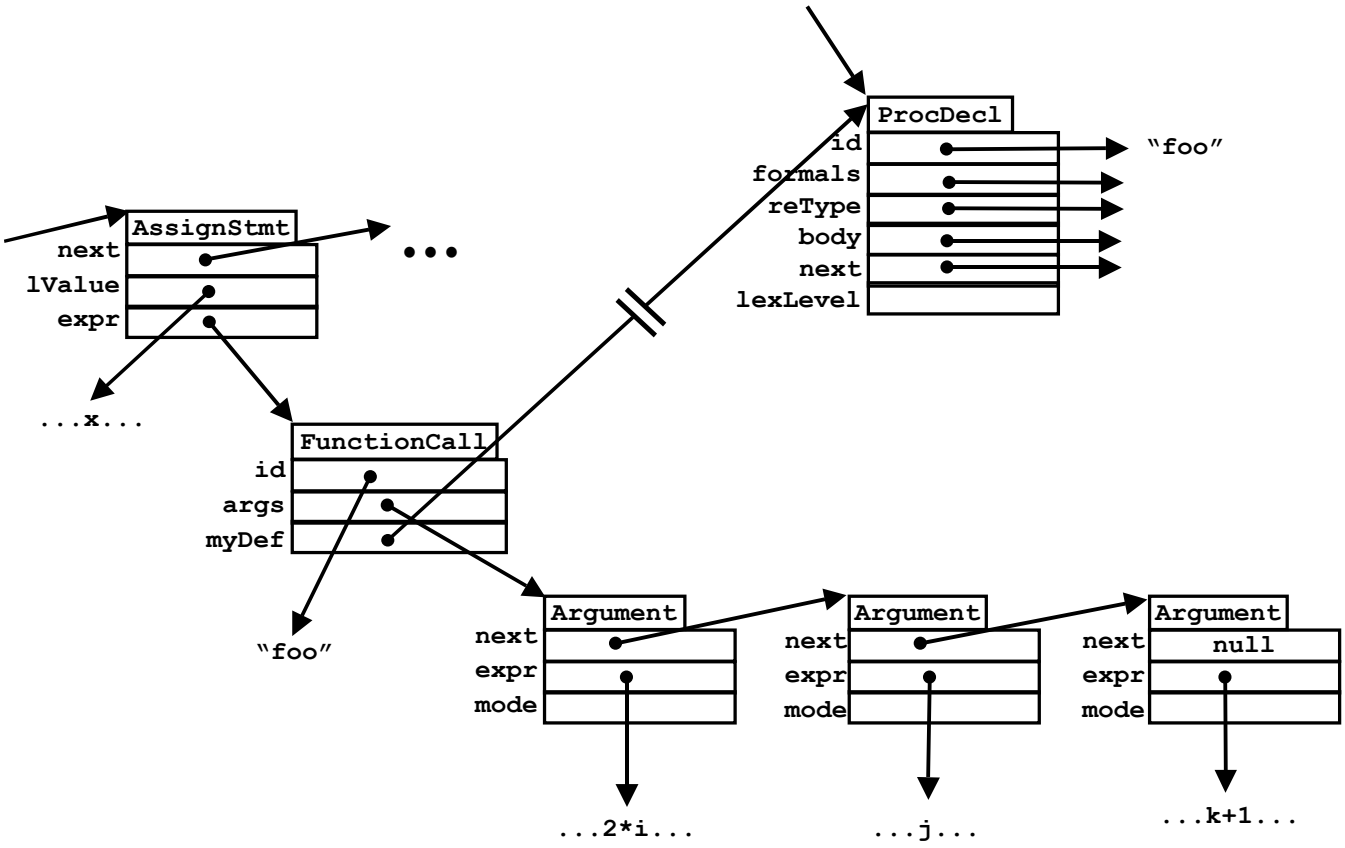
```
Expression   → ...
             → ID Arguments
             → ...
Arguments    → '(' Expression {',' Expression} ')'
             → '(' ')'
```

Here is an example function call. In this example, the function call expression is used as the right-hand side expression in an assignment statement:

```
x := myfun (2*i, j, k+1);
```

Like **CallStmt**, a **FunctionCall** contains a pointer, also named **args**, to a linked list of **Argument** nodes. There will be one **Argument** node for each expression. In this example, **args** would point to a linked list of 3 **Argument** nodes, each pointing to an expression.

Below is the AST for the above assignment statement, showing the **FunctionCall** node and the linked list of **Argument** nodes.



Both the **CallStmt** and **FunctionCall** nodes contain a semantic field called **myDef**, which points to the **ProcDecl** in which the procedure was declared. This diagram shown the **myDef** field pointing up to a **ProcDecl** node, somewhere higher up in the AST.

A PCAT program may contain several procedure declarations with the same name. For example, a program might legally contain several procedures named **foo**. Any one invocation of "**foo**" must refer to exactly one of the procedure declarations for **foo**. During type-checking, the compiler figured out which invocation applied to which version of **foo**. This information was saved in the **myDef** field at that time, so there is a pointer from the invocation node back to the correct **ProcDecl** node.

Array Constructors

The PCAT language includes a grammatical construct—the "array constructor"—to create an array. Here are the relevant syntax rules:

```
Expression   → ...
             → ID ArrayValues
             → ...
ArrayValues  → '{{' ArrayValue { ',' ArrayValue} '}}'
ArrayValue   → [ Expression of ] Expression
```

Here is an example which contains an array constructor on the right-hand side of each assignment statement:

```
type MyArray is array of real;
var a, b, c: MyArray := nil;
...
a := MyArray {{ 1.1, 2.2, 3.3, 4.4 }};
b := MyArray {{ 1000 of –9.999 }};
c := MyArray {{ 4.4, 100 of 5.5, 6.6, 7.7, 100 of 8.8, 9.9 }};
```

This version of PCAT uses {{ and }} instead of { and } for array constructors to make the parsing easier and to avoid grammatical confusion with record constructors, which use { and }.

An array constructor is an expression that, when executed, will "create" and array and initialize it. In our implementation, arrays will be placed on the heap, not in the activation record. So by "create an array", we mean that a large block of memory will be allocated on the heap when this expression is executed.

The sequence of ArrayValues is used to provide initial values of the array elements and to indicate the size of the array. In the first array constructor, the array will contain four elements, and will be initialized to the four real values listed. Once created, an array's size will not change although we can update the elements with statements like this:

```
a[2] := a[3] * 123.456;
```

In order to accommodate large arrays, each initial value may be given a count. In the second array constructor, the array will have size 1000 and each element will be initialized to –9.999. The expression to the left of the "**of**" keyword will be an integer expression. In this example, the expression is a simple integer, but it can be something more complex that must be computed at runtime, as in:

```
b := MyArray {{ i*foo(j*2) of –9.999 }};
```

The last array constructor in the example shows that there can be a mix of the above two forms. Some initial value expressions have counts and some do not. The size of this array will be:

1 + 100 + 1 + 1 + 100 + 1 = 204

Array constructors are represented with **ArrayConstructor** and **ArrayValue** nodes.

```
static class ArrayConstructor extends Expr {
    String        id;
    ArrayValue    values;
    TypeDecl      myDef;                 // Not used until proj 5
}
static class ArrayValue extends Node {
    ArrayValue    next;
    Expr          countExpr;
    Expr          valueExpr;
    ...
}
```

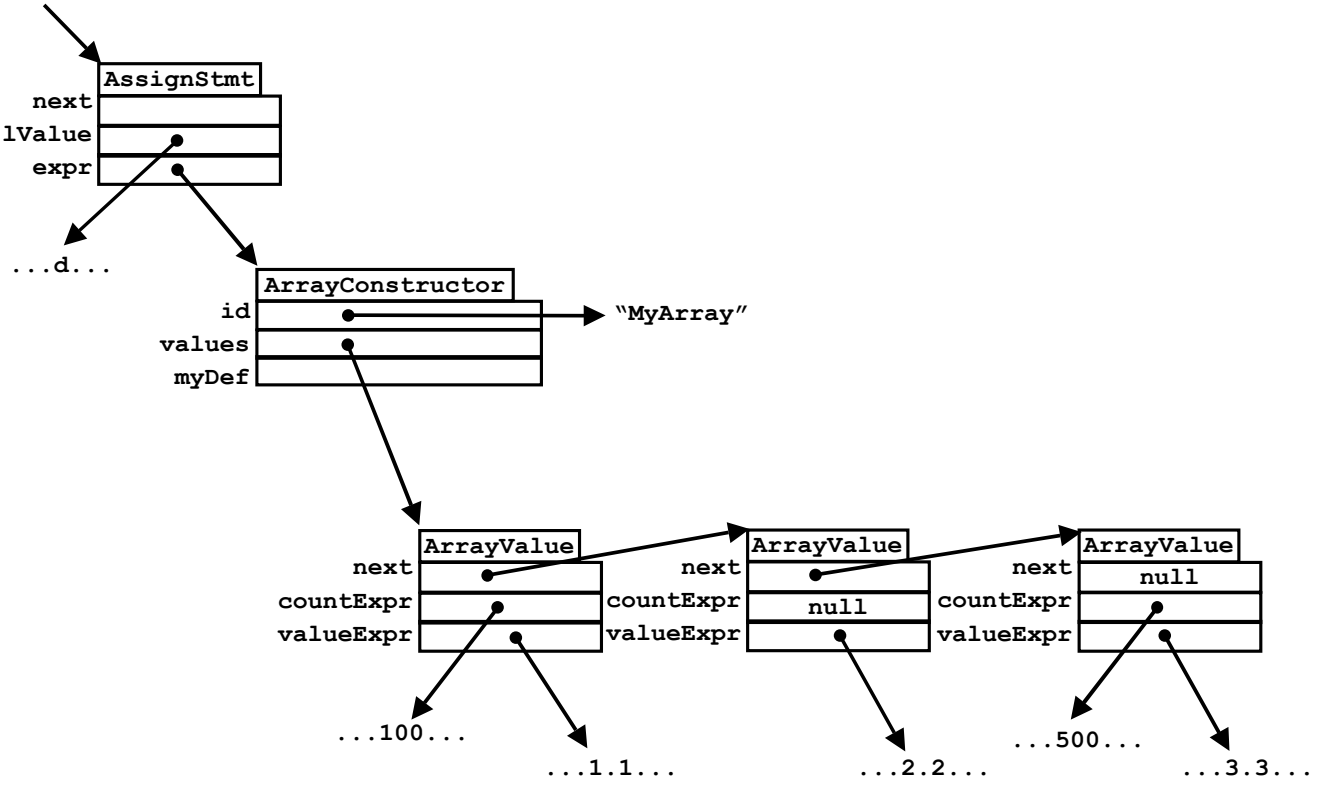
The **id** field in an **ArrayConstructor** node names the type, e.g., "MyArray". The **myDecl** field points to the **TypeDecl** in which this type was defined. The **values** field points to a linked list of **ArrayValue** nodes.

There will be one **ArrayValue** node for each initial value expression in this list. If there is a "count expression" then **countExpr** will point to it; if there is no count, then **countExpr** will be null.

For example, the following assignment statement:

```
d := MyArray {{ 100 of 1.1, 2.2, 500 of 3.3 }};
```

would be represented as follows:



Record Constructors

The PCAT language includes a grammatical construct—the "record constructor"—to create a record. Here are the relevant syntax rules:

```
Expression   → ...
             → ID FieldInits
             → ...
FieldInits   → '{' ID ':=' Expression { ';' ID ':=' Expression} '}'
```

Here is an example which contains a record constructor on the right-hand side of the assignment statement:

```
type MyRecord is record
                   f1: integer;
                   f2: real;
                   f3: boolean;
                 end;
var r: MyRecord := nil;
...
r := MyRecord {f1:=123; f2:=3.1415; f3:=false};
```

When a record constructor expression is executed at runtime, it will create the record and initialize all the fields in it. Our implementation will allocate records on the heap, not in the activation stack frame. A variable like **r** will actually be a pointer to the record, much the way Java works.

The record constructor must have the same fields as the corresponding type, but they may be listed in a different order. For example, the above assignment could be rewritten as follows, with no change in meaning:

```
r := MyRecord {f2:=3.1415; f3:=false; f1:=123};
```

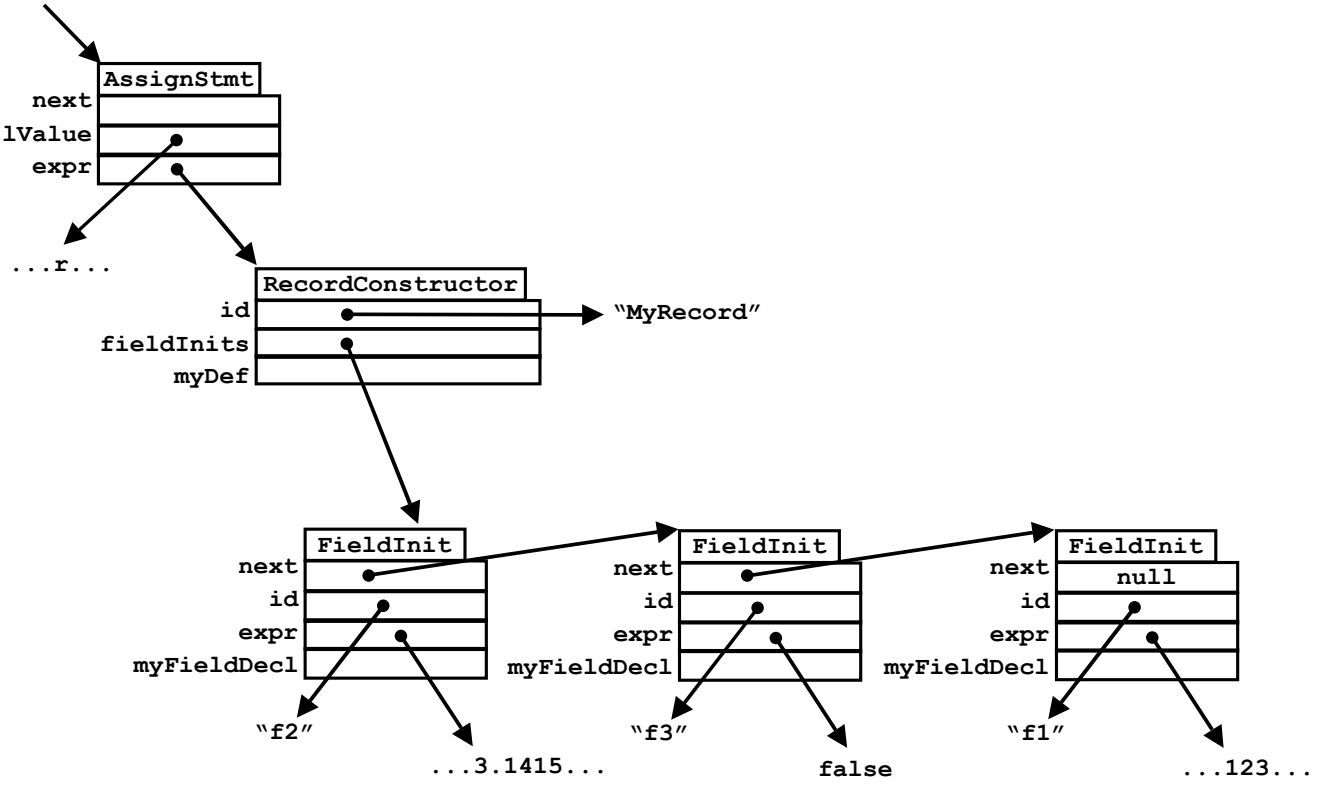
Each record constructor expression will be represented with a single **RecordConstructor** node and a linked list of **FieldInit** nodes.

```
static class RecordConstructor extends Expr {
    String             id;
    FieldInit          fieldInits;
    TypeDecl           myDef;            // Not used until proj 5
}
static class FieldInit extends Node {
    FieldInit           next;
    String              id;
    Expr                expr;
    FieldDecl           myFieldDecl;     // Not used until proj 6
}
```

For example, this assignment statement:

```
r := MyRecord {f2:=3.1415; f3:=false; f1:=123};
```

would be represented as:



Array Dereferencing

Array elements are accessed with bracket notation like this:

```
a[i] := a[j+1];
```

An array dereference, such as **a[i]**, is an L-Value, not an expression. Here are the relevant grammar rules:

```
LValue   → ID
         → LValue '[' Expression ']'
         → ...
```

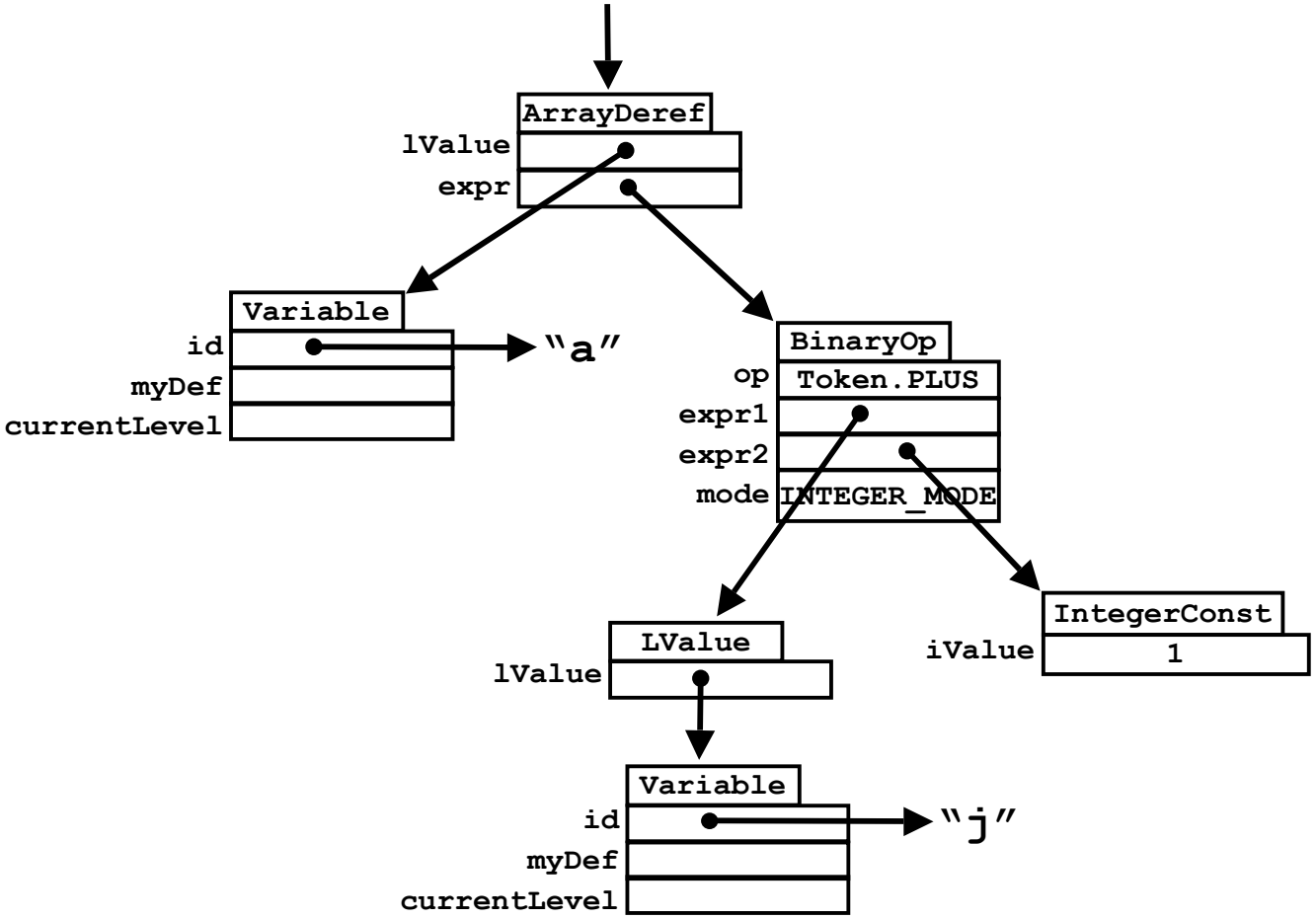
An array dereference is represented with an **ArrayDeref** node:

```
static class ArrayDeref extends LValue {
    LValue      lValue;
    Expr        expr;
}
```

As an example

```
a[j+1]
```

would be represented like this:



Notice that the grammar rules for LValue are left recursive, and allow such things as:

```
b[5][j+1]
```

This is perfectly legal, assuming that **b** is an array of arrays. Although parentheses cannot be used within L-Values in PCAT, we can use them here to see how this L-Value is parsed:

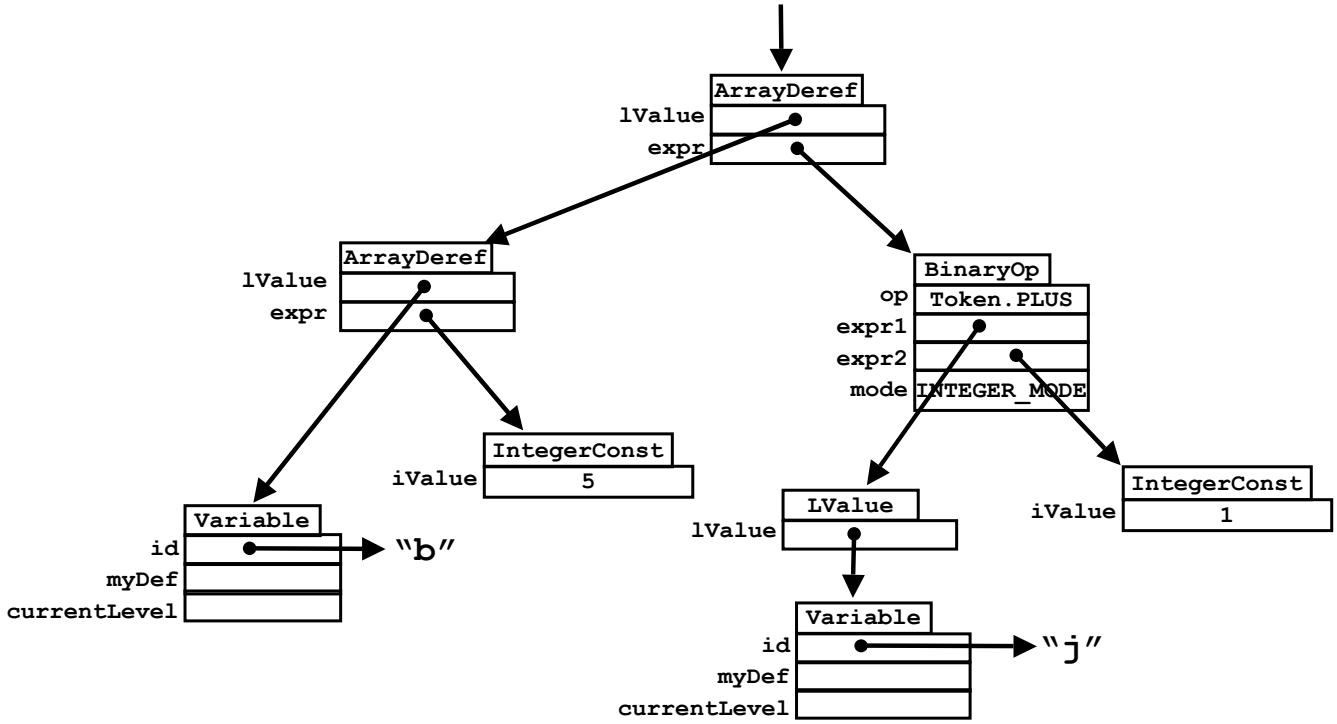
```
( b[5] ) [j+1]
```

Here is an appropriate declaration of **b**.

```
type MyIntArr is array of real;
     ArrOfArrys is array of MyIntArr;
var b: ArrOfArrys := ...;
```

In representing something like **b[5][j+1]**, notice that **ArrayDeref** counts a field called **lValue** and notice that **ArrayDeref** is itself a kind of **LValue**.

The L-Value **b[5][j+1]** would be represented as:



Record Dereferencing

The fields in records are accessed with "dot" notation like this:

```
r1.age := r2.age;
```

A record dereference, such as **r1.age**, is an L-Value, not an expression. Here are the relevant grammar rules:

```
LValue   → ID
         → LValue '.' ID
         → ...
```

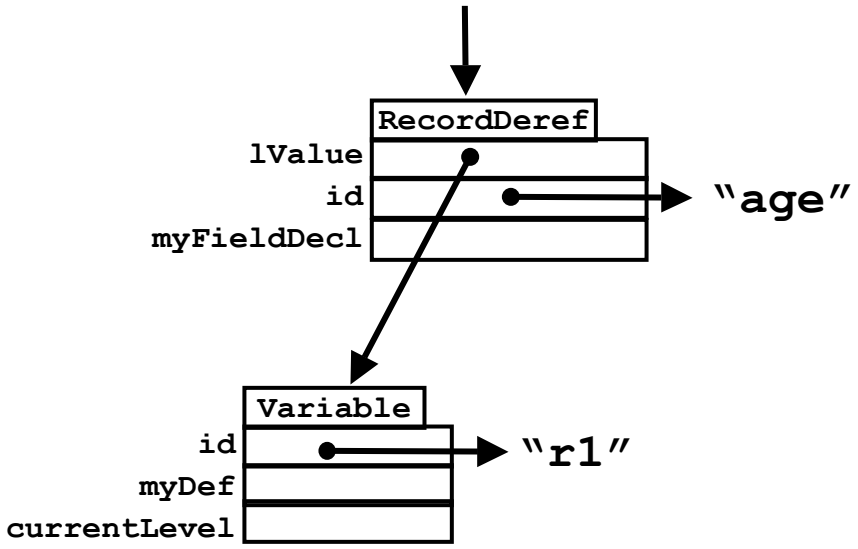
A record dereference is represented with a **RecordDeref** node:

```
static class ArrayDeref extends LValue {
    LValue      lValue;
    String      id;
    FieldDecl   myFieldDecl;     // Not used until proj 6
}
```

As an example

```
r1.age
```

would be represented as:



Notice that the **lValue** field of **RecordDeref** can point to any L-Value. For example, an **ArrayDeref** is an **LValue**, so the following is legal PCAT, assuming that **a** is an array of records:

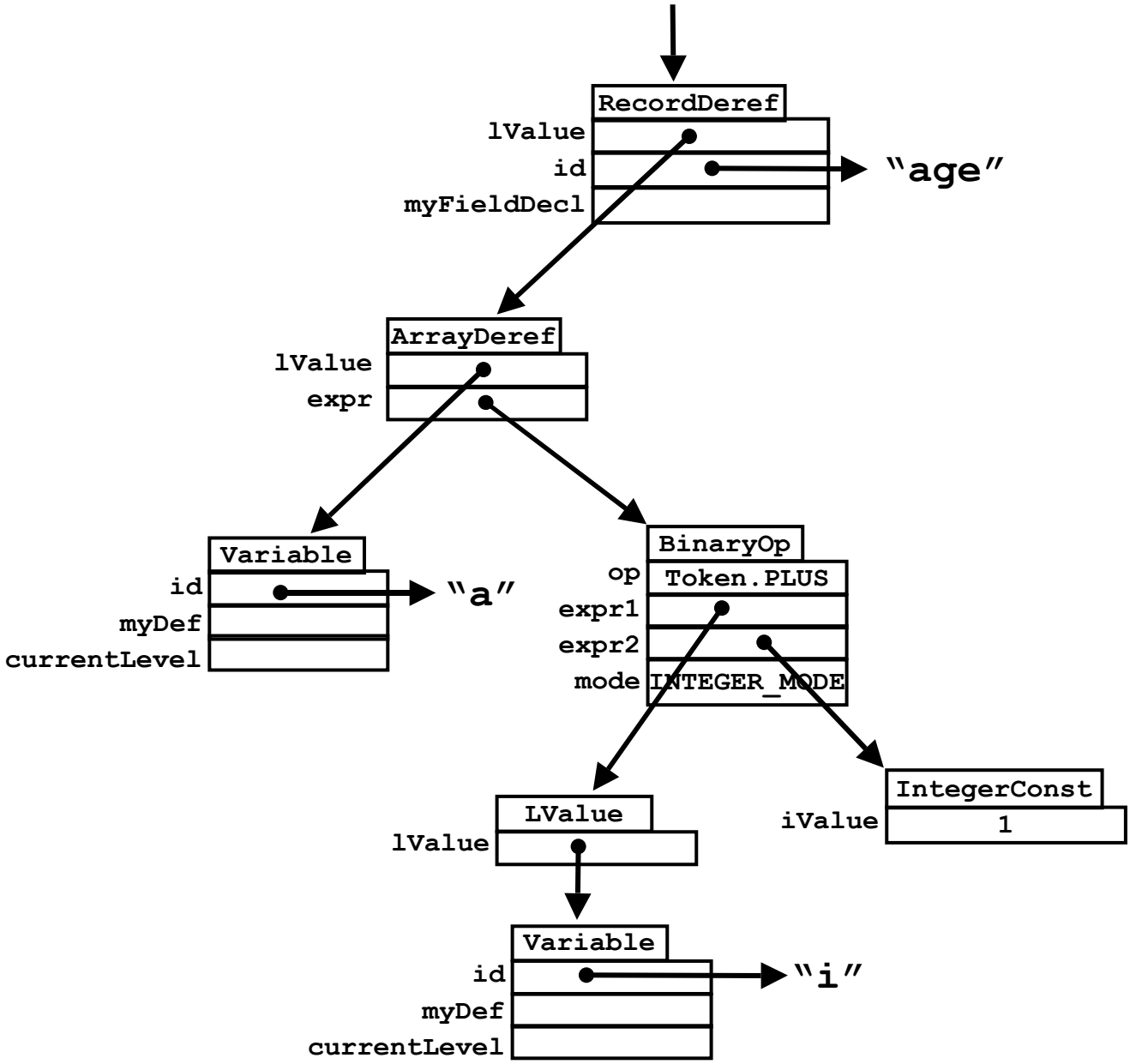
```
a[i+1].age
```

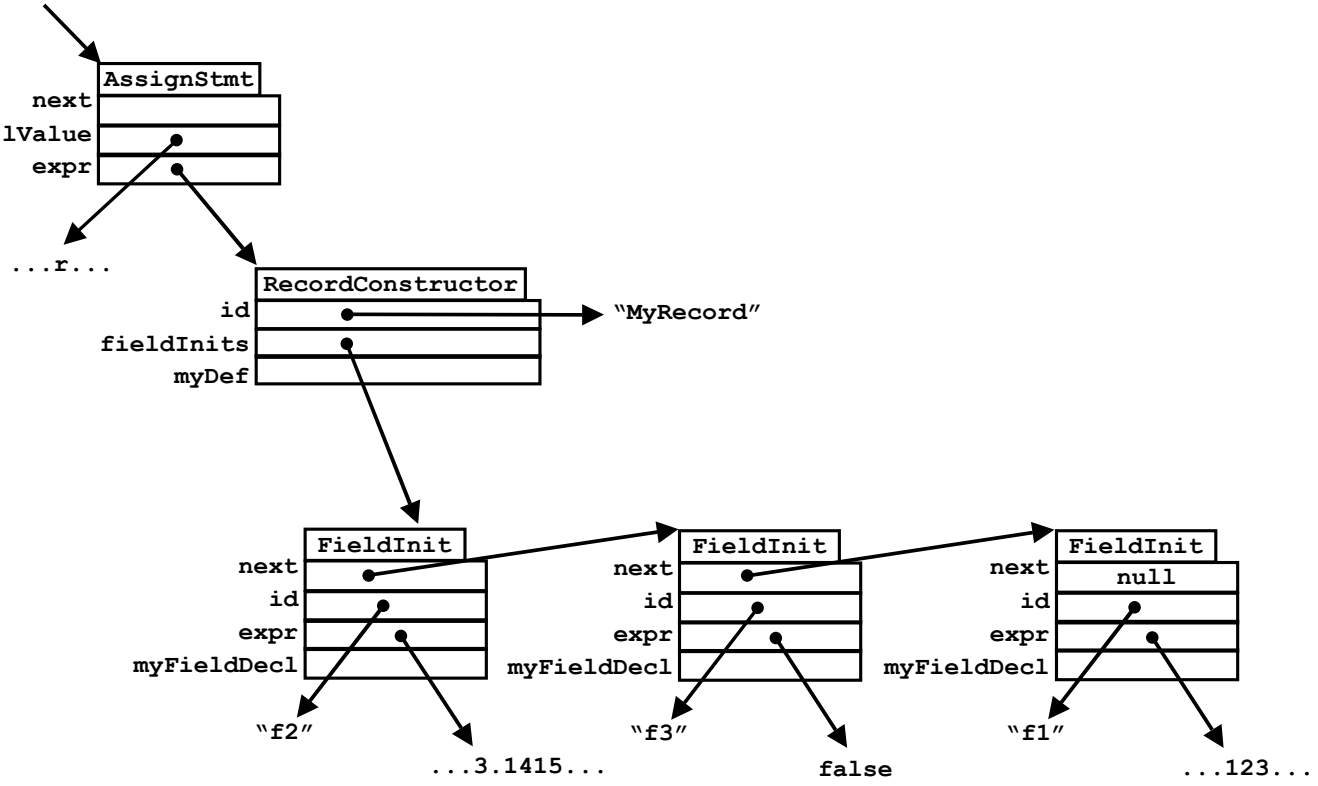
Here are some appropriate declarations:

```
type MyRec is record
                ss: integer;
                age: integer;
                married: boolean;
              end;
type MyArr is array of MyRec;
var a: MyArr := ...;
```

Below is the AST representation of

```
a[i+1].age
```





Read Statements

Here is the grammar rule for the "read" statement:

```
Statement   → ...
            → read '(' LValue {',' LValue} ')' ';'
            → ...
```

The "read" statement takes a list of variables, not expressions. In other words, the statement uses L-Values, since we will need to store an input value in memory at runtime.

A "read" statement is represented using a **ReadStmt** nodes a linked list of **ReadArg** nodes.

```
static class ReadStmt extends Stmt {
    ReadArg     readArgs;
}
static class ReadArg extends Node {
    ReadArg     next;
    LValue      lValue;
    int         mode;         // Not used until proj 6
}
```

In PCAT, the program can only read in values of type integer and real. Here is an example:

```
var x: real: = 0.0;
    i,j,k: integer: = 0;
...
read (i, j, x, k);
```

This statement will represented using a linked list of 4 **ReadArg** nodes. The **mode** field in a **ReadArg** node will be either **INTEGER_MODE** or **REAL_MODE**, to indicate whether we'll need to look for a real or integer value at runtime.

Write Statements

Here are the grammar rules for "write" statements:

```
Statement   → ...
            → write WriteArgs ';'
            → ...
WriteArgs   → '(' WriteExpr {',' WriteExpr} ')'
            → '(' ')'
WriteExpr   → STRING
            → Expression
```

A "write" statement takes a list of expressions and strings. Here is an example:

```
var x: real: = 0.0;
    i: integer: = 0;
    b: boolean: = 0;
...
write ("i = ", i, "x = ", x, "b = ", b);
```

A "write" statement is represented using a **WriteStmt** node, which will point to a linked list of **Argument** nodes.

```
static class WriteStmt extends Stmt {
    Argument  args;
}
static class Argument extends Node {
    Argument  next;
    Expr      expr;
    int       mode;       // Not used until proj 6
}
```

When generating code, we'll need to know how to print each value out. The **node** field in the **Argument** nodes will tell us. For **WriteStmts**, the **mode** field will be either **INTEGER_MODE**, **REAL_MODE**, **BOOLEAN_MODE**, or **STRING_MODE**.