# Programming Project #5: Symbol Table

**Due Date:** Tuesday, November 15, 2005, Noon

## Overview

In this project, you will create a file called **Checker.java** which will traverse (i.e., "walk") the abstract syntax tree, processing symbol table information and catching several errors related to symbol usage.

## Files

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

```
~harry/public_html/compilers/p5
```

**Main.java**
This file has been altered to create a **Checker** object and invoke the method **checkAst()**.

**Checker.java**
This is the file you will create and turn in.

**CheckerStarter.java**
You may use this file to get started.

**SymbolTable.java**
This file is new and includes methods to create and access the symbol table.

**Ast.java**
This file is unchanged, but you will use some fields that were not used in the last project.

**PrintAst.java**
This file has been modified to include printing of the new fields.

**PrettyPrintAst.java**
This file is new. It prints out the AST in "pretty" format, looking like source code.

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

**Parser.class**
This is a compiled version of my parser. You may use it if your **Parser.java** is incomplete, just as you may use my version of the lexical analyzer, **Lexer.class**. Comment out the appropriate lines in the **makefile**.

**makefile**
**tst**
**go**
**run**
**runAll**
Same as before, but altered for this project. (There is only one **tst** directory this time.)

**Main.jar**
This is the "black box" code, which was used to produce the output files in **tst**. You may run this program on various PCAT source files to determine what output your program should produce.

You may temporarily modify other **.java** or **tst** files in the course of debugging (for example, you might modify **Main.java** to comment out the call to **printAst**, to reduce the size of the output), but your final program (**Checker.java**) should work with the versions of the files I provide. Your program will be tested using my version of **Lexer.class** and **Parser.class**.

## Overall Organization

You will write a method called **checkAst()** to walk the abstract syntax tree. The **main()** method will call **parseProgram()** to produce an AST. It will then call your **checkAst()** method to traverse the entire tree. Finally, it will call **printAst()** to print out the AST. It also calls **prettyPrintAst()** to produce a formatted version of the AST.

Here is the **main** driver I have provided (more-or-less):

```
parser = new Parser (args);
ast = parser.parseProgram ();
checker = new Checker ();
checker.checkAst (ast);
PrintAst.printAst (ast);
PrettyPrint.prettyPrintAst (ast);
```

The code in **checkAst()** and the routines it calls should check for and report certain semantic errors (discussed later) and your code should also initialize the new fields (**myDef**, **lexLevel**, and **currentLevel**) in several AST nodes.

You should create a single file called **Checker.java**. This file should contain a class called **Checker**. This class should have a method called **checkAst()** as well as related methods as needed.

You may use the **Lexer.java** and **Parser.java** files you wrote or you may use the **Lexer.class** and **Parser.class** files I have provided. If you use my lexer and/or parser you will need to modify the **makefile**.

## CheckerStarter.java

A file called **CheckerStarter.java** has been included and you may use it to get started.

## PrettyPrint.java

A file called **PrettyPrint.java** has been included and you may find it useful in debugging. The **main()** method contains a call to the method **prettyPrintAst()**.

The class **PrettyPrint** contains a collection of methods which walk over the AST, printing it out as it goes. You may wish to study this code to see one approach to walking the abstract syntax tree. It is not necessarily the best way or the way you will want to use, but it may provide a starting point.

The tree is printed as a more-or-less legal PCAT program, which makes it much easier to read than the output from **printAst()**.

For example, the following source:

```
(* A very simple program *)

program
  is
var x,y  : integer := 0;
     begin
               if x   then

               x := y+3;
            end;
     end;
```

will be printed as follows. Note that comments are lost and the indentation is standardized.

```
PROGRAM IS
    VAR
        x: integer := 0;
        y: integer := 0;
    BEGIN
        IF x THEN
            x := (y + 3);
        END;
    END;
```

The **prettyPrint** methods may be modified, if you desire, to print out additional information, such as the values of certain fields in the AST. This may be helpful in debugging, but the **tst** files all make use of the more thorough **printAst** method. For further details on printing out additional fields, see the comment at the beginning of **PrettyPrint.java**.

For complex compilers, such "pretty-print" code is *invaluable* in knowing what your data structures look like at various points in compiler development. In my experience, the pretty-print code is developed in parallel with the compiler itself. By enabling and disabling various print statements in the pretty-print methods, we can see whatever information we are interested in at the moment.

## PrintAst.java

The methods in **PrintAst.java** have been modified to print out the new fields. The output of **printAst()** is often so long as to be unreadable; however it does a fairly comprehensive job of dumping the entire AST and should catch any differences between your AST and the ones in the **.bak** files.

## New Fields in Ast.java

In this project, we will be concerned with several fields that were previously ignored. You will need to fill in the following fields.

```
VarDecl
      int              lexLevel;

ProcDecl
      int              lexLevel;

Formal
      int              lexLevel;

TypeName
      Ast.CompoundType myDef;

CallStmt
      Ast.ProcDecl     myDef;

FunctionCall
      Ast.ProcDecl     myDef;

ArrayConstructor
      Ast.TypeDecl     myDef;

RecordConstructor
      Ast.TypeDecl     myDef;

Variable
      Ast.Node         myDef;
      int              currentLevel;
```

## The Symbol Table Routines

The file **SymbolTable.java** contains code to implement the Symbol Table. Since there will only be one symbol table, all methods are static (class) methods and all fields are static. There will never be any instances of this class.

Here are the methods you may call to manipulate the symbol table:

```
static void enter (String str, Ast.Node def)
static Ast.Node find (String str)
static boolean alreadyDefined (String str)
static void openScope ()
static void closeScope ()
static void printTable ()
```

There is a static variable in this class called **level** which will keep track of the current lexical "depth." The level of the outer main program block is zero, so **level** is initialized to zero. The lexical level of procedures defined in the main program body is 1; and nested procedures have higher levels.

As your compiler enters a procedure, you should call **openScope**(); as you finish processing a procedure, you should call **closeScope()**. **OpenScope()** and **closeScope()** will increment and decrement **level**. They will also open a new scope in the symbol table into which variables may be entered. When a scope is closed, all the symbol table entries for that level are forgotten.

The following kinds of symbols will be entered into the symbol table: variables, type names (from a TypeDecl), procedure names, and formal parameter names.

Upon encountering the declaration of a symbol (i.e, a variable, type, formal, or procedure name), you should enter it in the symbol table by calling **enter()**. Since every symbol may only be defined once in each scope, you should call **alreadyDefined**() before calling **enter()**. If **alreadyDefined()** returns TRUE, then an "Identifier is already defined" error is needed. (Of course, there is no need to call **enter()** since we apparently have a definition for the symbol already.)

To look up a symbol, you should use **find()**. It will search all scope levels, starting with the highest (i.e., the top of the symbol table stack; the most deeply nested level) and continue to the scope of the main program body. **Find()** will return the definition that was provided when the symbol was entered, which will be a pointer to an **Ast.Node**, or null if the symbol was not found.

The symbol table routines have been augmented with "print" statements to produce a trace of their usage in the hope that this will make testing easier.

There is also a method called **printTable()** which may be used to get a dump of the entire symbol table. In the final version of your code, this method will not be called, but I included it since it may help you in debugging. One reasonable place to call **printTable()** is right before the statements in a **Body** are processed; this will be just after a bunch of variables, types and procedures have been added.

## Details of the Symbol Table Implementation

The symbol table is maintained as a hash table array called **symbolTable[]**. Each entry points to a linked list of objects, of class **Bucket**. (Each of these lists is often called a "hash table bucket list.") There will be one **Bucket** per defined symbol. If a given identifier is declared several times in several nested scopes, there will be one **Bucket** for each entry. The **Bucket** objects in any given bucket list will be linked on their **next** field and will be in decreasing (more precisely non-increasing) scope level order. When **find()** searches a bucket list, it will stop at the first **Bucket** whose **id** matches the name being sought, thus returning the declaration from the greatest (innermost) scope level in which a definition exists.

```
static final int HASH_TABLE_SIZE = 211;
static Bucket [] symbolTable = new Bucket [HASH_TABLE_SIZE];
static int level = 0;

static class Bucket {
    String   id;          // Key: The symbol
    Ast.Node def;         // Value: The definition of this symbol
    int      hashVal;     // What this id hashed to
    int      scope;       // Scope level at which this symbol entered
    Bucket   next;        // Ptr to next Bucket in this bucket list
    Bucket   slink;       // Ptr to next Bucket in this scope list
}
```

Each **Bucket** contains a **def** field which points to the definition associated with that symbol and a **scope** field that gives the lexical level at which this symbol was defined.

In order to implement **closeScope()**, we could just go through all the entries in the hash table and go through every list. For every bucket list, we would need to scan the list, removing the **Buckets** at the front of it, until we hit an entry with a lower scope.

Often a program will have many blocks, each defining only one or two symbols. The above approach to closing a scope must go through the entire hash table. In production systems, we like to make hash tables very large to improve performance but this makes closing a scope time consuming. Consequently, we will use a more complex data structure to make closing a scope execute in time proportional to the number of symbols being removed from the table.

There will be one **Scope** object allocated every time we call **openScope()**. It will be added to the front of a linked list. Every time we call **closeScope()** we will remove the **Scope** record at the front of the list. The **Scope** records are linked on their **next** field.

```
// There is one "Scope" object for each active scope level; they are
// linked together in decreasing scope-level order by field "next".
// All symbols at a given level are linked into a list (the "scope
// list"), which is pointed to by the "slink" field.
//
static class Scope {
    Bucket slink;         // Ptr to latest Bucket for this scope
    Scope next;           // Ptr to next Scope record
}

static Scope scopeList = new Scope ();
```

(Notice that the initialization of **scopeList** to point to a list of one **Scope** object relies on the initialization in Java of pointers to zero / null.)

Each **Scope** object will contain a pointer (called **slink**) to a linked list of all the **Buckets** that have been added at that scope level. Thus, each **Bucket** object will be on two lists: one is the bucket list, using the field **next**; the other is the linked list we are discussing here, using the **slink** field.

To close a scope, all we need to do is run through the **slink** list for the topmost scope and remove each **Bucket** from its bucket list. To make this go even faster, we save the hash value of the symbol so we don't have to recompute it each time.

Finally, there is a routine called **printTable()** which will print out the symbol table. The **printTable()** method is never called by the black box code, so it is not strictly necessary in doing this project, but you may find it helpful while debugging.

(By the way, you are free to modify your copy of **Main.java** to call **printTable()**, to comment out calls to **printAst()**, or to do whatever else you want. Of course the grader will use the "standard" copies of files like **Main.java** when he or she tests your **Checker.java** file.)

## Symbol Definitions

The symbol table will contain an entry for each variable ID, type ID, procedure ID, and formal ID.

We could define a new class for symbol definition information, and depending on the details of the language, this may be necessary or desirable. With PCAT, however, we can do something kind of tricky, but which will reduce the amount of coding we need to do.

We will use **VarDecl**, **TypeDecl**, **ProcDecl**, and **Formal** nodes as the definitions of the symbols.

For example, when the compiler parsed the following source:

```
var x: integer := 0;
```

a **VarDecl** structure was created. During the checking phase, we will enter "x" into the symbol table. The definition of "x" will simply be a pointer to this **VarDecl** node.

Likewise, we will enter each procedure ID into the symbol table with a pointer to its **ProcDecl** node. We will enter each type ID with a pointer to its **TypeDecl** node and we will enter each formal parameter with a pointer to its **Formal** node.

Later, when we process a use of "x", as in:

```
x := ...;
```

we will be looking at a **Variable** node. At this point, we consult the symbol table to obtain a pointer to the correct node to which this use of "x" refers. We will then store this pointer directly into the **Variable** node.

A new field called **myDef** is in the following AST classes. These correspond to places where the PCAT programmer may use an ID. (The ID should be defined elsewhere, for example in a **VarDecl** or **TypeDecl**).

**Variable**
**CallStmt**
**FunctionCall**
**ArrayConstuctor**
**RecordConstructor**
**TypeName**

For all **Variable** nodes, your code must set this field to point to the corresponding **VarDecl** or **Formal** node. For **CallStmt** and **FunctionCall** nodes, you must set **myDef** to point to the corresponding **ProcDecl**node. For **ArrayConstructor** and **RecordConstructor** nodes, you will set **myDef** to point to a **TypeDecl** node. In each case, you can simply look up the ID in the symbol table to find the definition.

In the case of the **TypeName** node, you must set the **myDef** field to point to a **CompoundType** (i.e., to either an **ArrayType** or a **RecordType**), so you'll have to first look up the ID to find a **TypeDecl** node. Then, you'll have to go into that **TypeDecl** node to get the definition, which will point to a **CompoundType**.

You'll also need to check for errors. Consider the following code:

```
var x: ...;
type t is array of x;   (* Error: x is not a type *)
...
if (t > 5) ...          (* Error: t is not a variable or formal *)
```

Be sure to check that the right kind of symbol is being used, as well as simply making sure the symbol is found in the symbol table.

## SemanticError()

The starter file, **CheckerStarter.java**, includes a method you can call to print out errors:

```
void semanticError (Ast.Node t, String msg)
```

It is passed a pointer to a node from which it will extract a line number. Depending on the class of the node, this method extract some additional information, such as an id or keyword. A call such as:

```
semanticError (p, "Identifier is already defined");
```

will print a message such as:

```
Error on line 4 near foobar: Identifier is already defined
```

The **semanticError()** routine will then return to its caller—unlike **syntaxError()**, which aborted—so the compiler can keep walking the AST looking for more errors.

## Errors to Identify

In this project, you should look for and report the following errors:

`Identifier is already defined`

Whenever we encounter a **TypeDecl**, **VarDecl**, **ProcDecl**, or **Formal** we need to enter a new symbol into the symbol table. If there is already an entry at the current scope level, it is an error. Each symbol may only be declared once in each scope.

`Identifier is not defined`

Whenever we encounter an ID, we need to look it up in the symbol table to make sure it is defined in this scope or an enclosing scope. This includes (1) in a **CallStmt**, (2) in a **FunctionCall**, (3) in a **NamedType**, (4) in a **Variable**, (5) in a **RecordConstructor**, and (6) in an **ArrayConstructor**.

`Expecting a local or formal name`
`Expecting a type name`
`Expecting a procedure name`

It may be that the programmer has supplied a name that is in the symbol table but that name is not the right kind of thing. For example:

```
type t is ...;
x := 4 * t;
```

Here, **t** is not undefined; the problem is that its definition points to a **TypeDecl** and not a **VarDecl** or **Formal**. We also expect to see a procedure name in a **CallStmt** and a **FunctionCall**, and we expect to see a type name in a **NamedType**, an **ArrayConstructor**, and a **RecordConstructor**.

`INTEGER, REAL, BOOLEAN, TRUE, FALSE, and NIL may not be redefined`

These IDs all have predefined meanings and should be appear in **var**, **type**, or **procedure** declarations.

`This field is already defined in this RECORD`

In a record type declaration, each field must have a different name.

`Multiple assignment to field in RECORD constructor`

The following is in error since f1 is repeated.

```
x := R { f1:=4; f1:=5};
```

## Predefined Identifiers

The following identifiers have predefined meanings in PCAT:

```
nil
true
false
integer
real
boolean
```

There are restrictions on their use that must be checked during this phase. For example, it is invalid to attempt to redefine any of these names in:

- procedure definitions
- type definitions
- variable definitions
- fields within record types
- formal parameters within procedure headers

An error message should be issued in each of these cases and, of course, there is no need to enter the bad name into the symbol table.

(By the way, “nil” is just the same thing as “null” in other languages: a zero pointer.)

## Checking for repeated field names

We need to make sure that no field name is repeated in **RecordConstructors** and in **RecordTypes**.

Here is an idea about how to approach this. It is kind of a hack, but we can "borrow" the symbol table to make this checking easier. For example, upon encountering a **RecordConstructor**, call **openScope()**. This is not really a new scope, but keep reading.

Next, go through the list of **FieldInits**. For each **id**, check to see if it is **alreadyDefined()**. If so, call **semanticError()**. If not, **enter()** it into the symbol table, supplying a dummy NULL definition. Finally, after processing the entire list of **FieldInits**, call **closeScope()**. The symbol table will be the same as you found it before. A similar thing can be done for **RecordType**s.

Don't forget that we also need to check the expressions in the **RecordConstructor**. For example:

```
VAR f1: INTEGER := ...;
...
      ... R { f1:=2; f2:=(4*f1); f3:=2} ...
```

When we hit **f1** in the expression **(4*f1)** we must identify it with the variable **f1** and not the field **f1**. To use the hack just described, we will need to make two passes over the list of **FieldInits**. The first pass will check for repeated assignments to the same field. It will begin with **openScope** and will end with **closeScope**, leaving the symbol table unchanged.

The second pass over the list of **FieldInits** will check each of the expressions. By the time we hit the **f1** in (4***f1**), all the fields (**f1**, **f2**, **f3**) will have been removed from symbol table and we'll find the previous (correct) definition of **f1**.

A similar concern applies to processing **RecordType** nodes: we must first traverse the list of **FieldDecls** to check for repeated definitions of the same field name and then traverse the list a second time to check the **TypeNames** in the **FieldDecl** list.

## Handling Defined Types

A type declaration associates a type name with a **CompoundType**. There are two kinds of **CompoundTypes**: **ArrayType** and **RecordType**.

```
type MyArrType is array of real;
type MyRecType is record
                     f1: integer;
                     f2: real;
                     next: MyRecType;
                  end;
```

The TypeDecl node has fields named **id** and **compoundType**.

Elsewhere in the program the programmer can use these defined types. For example:

```
var a: MyArrType := ...;
    r: MyRecType := ...;
```

These "uses" are represented with **TypeName** nodes, which have fields named **id** and **myDef**. (By the way, I probably should have called the field "compoundType" instead of "myDef", but I do not want to change this.)

Our ultimate goal is to set the **TypeName.myDef** field to point to the definition, i.e., to the **ArrayType** or **RecordType** node. Then, later in project 6, when we need that information during type checking, it will be right there.

To achieve this, first, whenever we encounter a new type declaration, we will make an entry into the symbol table. Using the name being defined (e.g., "MyArrType") we will associate this name with a pointer to its declaration.

```
SymbolTable.enter (...ID..., ...ptr to TypeDecl...);
```

Then later, when we encounter a **TypeName**, as in

```
var a: MyArrType := ...;
```

we'll need to (1) lookup the ID in the symbol table, (2) make sure it is a **TypeDecl**, (3) locate its definition, a **CompoundType**, and (4) copy that pointer into **TypeName.myDef**.

## Dealing with Basic Types

Most programming languages have several predefined types. Sometimes these are called "basic types" or "predefined types" or "built-in types". For example, C/C++ includes these basic types:

```
int
short
long
single
double
char
```

In PCAT, there are 3 predefined, basic type names:

```
integer
real
boolean
```

Their definitions are built in to the language; these names can be used anywhere a **TypeName** can be used but the PCAT program should never contain a **TypeDecl** for these types.

For **TypeName** nodes that refer to these built-in types, we will simply leave their **myDef** fields null.

In project 6 we'll also need two additional built-in basic types, but you can ignore them for now. In particular, we have two types of values that may appear in PCAT programs. Consider the types of these expressions:

```
"hello"
nil
```

These value do not have type **integer**, **real**, or **boolean** and, in project 6, we'll create two additional built-in basic types to handle them:

```
_string
_nilType
```

By placing an underscore in the type names, we can prevent the PCAT programmer from using these types directly. After all "string" and "nilType" may be used as legal names in PCAT; they do not have any built-in meaning. However, "_string" and "_nilType" are not legal identifiers and would cause lexical errors if the programmer tried to use them.

## Redefining Built-in Types

Some languages allow the built-in basic types to be redefined. You might see stuff like this:

```
type integer is array of real;      // NOT LEGAL PCAT!
var x: integer;                     // NOT LEGAL PCAT!
...                                 // NOT LEGAL PCAT!
... x[7] ...                        // NOT LEGAL PCAT!
```

My opinion is that this is a horrible idea in language design since it permits program code that is needlessly misleading.

Note that there is another approach to implementing built-in types, which is used in some compilers. The idea is to make an entry in the symbol table for each predefined type. Then, when you encounter the use of a type, for example,

```
var x: MyTypeName;
```

you look it up in the symbol table to see what its definition is. For the built-in types, like "integer" and "real", you'll need to have some definition. For each built-in type, we would create a special dummy object to represent the built-in type.

This approach has its benefits and it allows you to compile languages that allow the predefined type names to be redefined.

## Type Equality in PCAT

When are two types considered equal? For PCAT, the answer will be slightly different, depending on whether they are built-in basic types or not.

If the two types are built-in basic types, then they are equal if and only if they have exactly the same name. In other words, we'll have to compare the **TypeName.id** strings.

If the two types are defined type names, then we'll have to look at their definitions. If they are both defined in the same **TypeDecl**, then they are equal, but it's possible for two types to have the same name, yet be defined in different ways.

For example, consider this PCAT code:

```
type T1 is array of integer;
var x: T1;
...
procedure foo (...) is
    type T1 is record ... end;
    var y: T1;
    ...
    x := y;
```

The assignment of **y** to **x** is legal if both variables have the same type, that is, it is legal if their types are equal. Both have type **T1**, but these were defined in different places and were given different definitions. Later, in project 6, we will detect a semantic error in the above code.

Next, consider this example:

```
type T2 is array of integer;
var a: T2;
...
procedure foo (...) is
    type T2 is array of integer;
    var b: T2;
    ...
    a := b;
    ...
```

According to the semantics definition of PCAT, this code also contains a type error!!! Even though **a** and **b** have the type **array of integer**, these array types are defined in different places. This is the question of "name equality" versus "structural equality", which will be discussed in more detail later.

## Comparing Strings in Java

In Java, a string is represented using an object. The object stores the actual bytes of the string. According to the Java definition, there may be several objects that all contain identical sequences of characters. For example, there may be 361 objects floating around that all have the characters

```
"integer"
```

Consider the following code for comparing string objects:

```
if (str1 == str2) ...
```

The variables **str1** and **str2** may point to different objects with the very same characters. This test is fast, since it just compares pointers, but may not do what you want.

The following test is almost certainly wrong:

```
if (str1 == "integer") ...
```

Instead, you want to use the **equals** method in class **String**:

```
if (str1.equals("integer")) ...
```

However, this is slow since (1) it involves a method invocation and (2) it has to look at each character in turn.

In this project, we'll need to compare strings a lot and we want to do it efficiently using ==.

The whole point of using the **StringTable** in the lexer was to make sure that there was only one object around for each unique sequence of characters. In the lexer, every time we found a new identifier, we looked it up in the **StringTable** and, if found, used the previous String object instead of a new String object. This way, even if the string "integer" appears in the program 361 times, there will only be one String object with the character "i n t e g e r" used during type checking.

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

In this project, we need to compare the **TypeName.id** to see if it is one of the built-in basic type names. I recommend creating some global static variables, one for each of the following strings:

```
nil
true
false
integer
real
boolean
_string
_nilType
```

(The last two will be useful in project 6.)

So, in class **Checker** consider adding fields like this:

```
String nilString;
String trueString;
...
```

You'll need to initialize these variables and the logical place is in **checkAst**, where the code will be executed exactly once at the beginning of the type-checking phase.

For each string, you'll need to look it up in the **StringTable**. If it is not there, you'll need to add it. In either case, you'll set the variable to point to the one shared object.

A subroutine might be appropriate here to simplify our code. Let's call it **uniqueString**. It is passed a pointer to a String and it returns a pointer to the shared canonical version of the String. Here is some pseudo-code:

```
String uniqueString (String str) {
    i = StringTable.lookupToken (str)
    if i == -1
        StringTable.insert (str, Token.ID)
    return StringTable.lookupString (str)
}
```

Then, in **checkAst**, you can initialize pointers to the canonical versions easily:

```
nilString = uniqueString ("nil");
trueString = uniqueString ("true");
...
```

## Processing Declarations

Type, variable, and procedure declarations may refer to a type declared elsewhere. In the following legal code, type **t** is used before its definition is encountered:

```
var x: t := ...
procedure f (... :t ...) is ... t ... end;
type s is record
            f: t;
            ...
          end;
type t is ...
```

Therefore, the type declarations of a **Body** must be entered into the symbol table before the types, variables, and procedures can be checked. We must begin processing a **Body** by first running through all the **TypeDecls** and **enter**()-ing them into the symbol table so they will be available when we need them.

After we have entered all the types in the body into the symbol table at the current scope, we can then run through the **TypeDecl** list and check those same type definitions. Making two passes over this list allows us to handle recursive types and forward references, as in the following legal code:

```
type t1 is record
            f1: t1;
            f2: t2;
           end;
type t2 is ...;
```

When we recursively check procedures, we may encounter forward references and recursive references to types, procedures, and variables defined in the body. This code is legal:

```
procedure f1 () is
  ... t ... x ... f1 ... f2
end;
var x: integer := ...;
type t is ...;
procedure f2 () is ...;
```

Thus, we can't check **ProcDecls** until after we have entered all types, variables, and procedures into the symbol table.

With **VarDecls**, we do not allow forward references. The following code is illegal:

```
var x: integer := y;
var y: integer := ...;
```

We will execute the initializing expressions in exactly the order given and don't want to use what would be an uninitialized variable. Thus, we must perform the checking of the variables in parallel with **enter**()-ing them into the symbol table. To accomplish this, we will walk the **VarDecl** list just

once. For each **VarDecl**, we will check its type expression (if any) and check its initializing expression. Then, after checking it, we will **enter()** that variable into the symbol table.

After all this, we are free to check the statement list. Here is an order of processing that satisfies these constraints:

```
void checkBody (Ast.Body body) {
  enterTypeDecls (body.typeDecls);
  checkTypeDecls (body.typeDecls);
  enterProcDecls (body.procDecls);
  enterAndCheckVarDecls (body.varDecls);
  checkProcDecls (body.procDecls);
  checkStmts ( body.stmts);
}
```

Note that the call to **checkTypeDecls()** could be moved around a little. If, for example, you called it a little later, this program:

```
program is
  var x := 1;
  type t is x;
begin end;
```

would change the error on line 3 from "X is undefined" to "X is not a type." The order shown above is what I used in the black box, so this program would produce the "X is undefined" error message when run through my code.

## Scopes in Procedures

Consider the following code:

```
var x ...;
procedure foo (...a...) is
  var b ...;
  begin
     ...c...
  end;
...
```

Here **foo** and **x** are different names, but if they had been the same, it is an "already defined" error. Thus, procedure names must be **enter()**-ed into the symbol table in the scope of the enclosing block—before **openScope()** is called—and not within the inner scope of the procedure.

If **x** and **a** are the same name it is okay. The declaration of a formal will override declarations from the surrounding scope since we are now within a new scope. A use at point **c** will refer to the formal declared at point **a** and not to something with the same name from the surrounding scope.

If variable **b** has the same name as **x**, it is okay since **b** is in the inner scope and **x** is in the outer scope. But if **b** has the same name as formal **a**, it is an "already defined" error. There is only one new scope for each procedure and both formals and local variables are both **enter()**-ed at the same scope level. A reference at point **c** will refer either to the formal **a** or to the local **b**, since they must have different names.

## Setting 'lexLevel' and 'currentLevel'

In PCAT, variables are introduced in **var** declarations and when used as formal parameters in procedure definitions. In the following example, variables **a** and **b** are introduced on lines 2 and 3.

```
1.  program is
2.    var a: ... ;
3.      procedure foo (b: real) is
4.        begin
5.          ... XXX ...
6.        end;
7.    begin
8.       ... YYY ...
9.    end;
```

Corresponding to the declaration of each variable, there will be a **VarDecl** or a **Formal** node in the AST. These nodes have a field, called **lexLevel**, which should be set to be the current lexical level at the point the variable is declared.

In addition to being declared, variables will also be "used." For example, **a** may be used at points **XXX** and **YYY**. The variable **b** may be used at point **XXX**, but use of **b** at point **YYY** would be illegal. The lexical level at the point a variable is used may be equal or greater than the lexical level at which the variable was defined. In this example, the **foo** procedure creates a new scope and therefore increase the lexical level by one. The variable **a** is declared at lexical level 0 and **b** is declared at level 1. Any use at point **XXX** would be at level 1. Any use at point YYY would be at level 0.

At every place a variable is used (as opposed to declared), we will have a **Variable** node. The **Variable** node includes a field called **currentLevel**, which must be set to the lexical level at the time the variable is used.

In class, we'll discuss in detail why we need this information. In summary, each variable will be stored in an "activation record" (also called a "stack frame"). The variable may be stored in the frame at the top of the stack (i.e., in the activation record of the currently executing routine), or it may be stored in a frame buried in the stack (i.e., in the activation record of a routine that is currently suspended during a call to another routine). Knowing both the **currentLevel** and lexical level at which the variable was defined will allow us (during code generation, next term) to generate code to locate the correct activation record and hence to find the memory location of the variable.

## Big Hint

In this project, you must write a collection of routines that walk the Abstract Syntax Tree. Note that the routines in **PrettyPrint.java** do exactly this. One approach would be to start with the **PrettyPrint.java** file and modify it as follows:

0. Read over **PrettyPrint.java** to be sure you understand how it works.
1. Create a file **Checker.java** from the starter file. Then copy and paste **PrettyPrint.java** into **Checker.java**.
2. Change the names of all methods from "ppXXX" to "checkXXX." For example, **ppIfStmt()** becomes **checkIfStmt()**.
3. Remove all print statements and everything relating to printing and indentation. (Don't just comment it out; get rid of it!!!)

4. Change all comments to reflect the fact that the methods perform type checking, not printing.
5. Insert code to perform the required operations to check symbol usage and set the new fields.

## Details...

It is considered cheating to decompile or look inside any **.class** or **.jar** file I provide. If you have questions about what these files do, please ask me!

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 5 - 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.