Programming Project #6: Type Checking

Due Date: Tuesday, November 29, 2005, Noon

Overview

In this project, you will modify your **Checker.java** file to catch all remaining semantic errors and modify the abstract syntax tree as necessary for code generation.

Files

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

```
~harry/public_html/compilers/p6
```

Checker.java
You created this file in project 5; you will modify it in this project.

SymbolTable.java
This file is has been changed to comment-out the print statements, which were used in testing the previous project.

Ast.java
This file is unchanged, but we will use a class and some fields that were previously ignored.

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

Main.java
Lexer.class
Parser.class
Token.java
StringTable.java
FatalError.java
LogicError.java
PrettyPrint.java
makefile
go
run
These files are unchanged.

runErr
This file is new. It is like **run**, but it displays only the **stderr** output (the error messages). It does not display the AST. It is used for tests that have semantic errors. When there are semantic errors, we will ignore the AST altogether. We care only that you print the correct message.

tst
runAll
Same as before, but altered for this project.

Main.jar
The new "black box" code, which was used to produce the output files in **tst**.

Overview

This project has two aspects: (1) to fill in some additional information in the abstract syntax tree data structure, and (2) to detect all remaining semantic errors in the source program. The two tasks are related and best done in parallel.

For example, one check is to determine whether RETURN statements have a value for non-void procedures and no value for void procedures. We also need to link the **ReturnStmt** node to the correct **ProcDecl** node. These two actions are best programmed together at the same time.

In this project, you will modify your **Checker.java** file.

The driver class **Main.java** is unchanged, but the code to print the AST has been modified to print the new fields in the AST that are being filled in during this project.

Changes to "Ast.java"

A class called **IntToReal** has been added to deal with integer-to-real conversions within expressions.

A new field called **myLoop** has been added to **ExitStmt** nodes.

A new field called **myProc** has been added to **ReturnStmt** nodes.

A new field called **myFieldDecl** has been added to **FieldInit** and **RecordDeref** nodes.

A new field field called **mode** has been added to the several nodes. The **mode** will be an integer and we'll use the following different values:

```
1 = INTEGER_MODE        This data will be an integer
2 = REAL_MODE           This data will be a floating number
3 = STRING_MODE         This data will be a (pointer to a) string
4 = BOOLEAN_MODE        This data will be a Boolean value
```

You can use these symbolic names (always a good practice!) by adding the following lines to your **Checker** class. (Recall that "static final" just means that this variable is a constant.)

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

The **mode** field will be set to indicate what kind of data is being manipulated. The field **mode** has been added to these classes:

```
ReadArg
BinaryOp
UnaryOp
Argument
```

The "mode" Field

In the compiled assembly code, we will only be manipulating integers, reals, and pointers. Arrays and records will not be manipulated directly; instead we will manipulate them with sequences of instructions that manipulate integers, reals, and pointers. At runtime, our type system will be very simple: just that of the target machine. However, we will need to keep information about the "machine type" of the data. This is what the **mode** field tells us.

During code generation, we will only need to know the **mode**; other type information will be completely ignored.

The **mode** field reflects the sort of data that is being manipulated by the various constructs:

```
ReadArg        the size/type of data to be read in (Int or Real, only)
BinaryOp       the size/type of the result (Int or Real, only)
UnaryOp        the size/type of the result (Int or Real, only)
Argument       the size/type of the argument in a WRITE statement
```

In our implementation for the SPARC we will represent and manipulate TRUE exactly the same way we represent the integer 1. FALSE will be treated as the integer 0.

Therefore, logical operations like AND and OR will use mode=INTEGER_MODE. BOOLEAN_MODE and STRING_MODE are only used for the arguments of **WriteStmts**.

It is mandatory that the **mode** field be filled in correctly for programs without errors. It is much less important for programs with errors, since the **mode** field is not needed until code generation. When there are semantic errors in the program, the compiler will terminate before the **mode** field is ever looked at.

Testing PCAT Programs With Semantic Errors

Whenever our compiler detects any errors before code generation, we will abort and no attempt at code generation will be made. Therefore, we don't really care what you do to the AST.

For tests on PCAT programs that contain semantic errors, we will not even look at the AST. You are required only to print the correct error messages. To facilitate this, there is a new shell script called **runErr**. It is just like **run**, except that it performs the **diff** on only the **stderr** output. The **stdout** output (which is where we print the AST) is ignored.

The **runAll** script calls **run** or **runErr**, as appropriate for each of the tests.

Changes to ExitStmt and ReturnStmt Nodes

There is a new field called **myLoop** in **ExitStmt** nodes. This field should be set to point to the innermost **ForStmt**, **LoopStmt**, or **WhileStmt** that encloses this EXIT. During code generation, labels will be associated with the looping statements and the names of these labels will be stored in the **ForStmt**, **LoopStmt**, or **WhileStmt** node. When generating code for an EXIT, we can follow the **myLoop** pointer to find the name of the label to which to branch.

Likewise, the field **myProc** in **ReturnStmts** should be set to point to the **ProcDecl** for the procedure containing the RETURN. RETURN statements must not occur in the main procedure body.

To set the **myLoop** and **myProc** field, you may wish to consider adding a new parameter to the "check" functions that are related to statements. For example, you may have:

```
checkStmts  (..., Ast.Stmt currentLoop, Ast.ProcDecl currentProc)
checkIfStmt (..., Ast.Stmt currentLoop, Ast.ProcDecl currentProc)
...etc...
```

The idea is to pass these parameters down through the various "check" invocations until we reach a **checkExitStmt()** or **checkReturnStmt()**, where they will be used. If there is no surrounding loop, the **currentLoop** argument will be NULL. If there is no surrounding procedure, the **currentProc** argument will be NULL.

Within the **checkExitStmt()** method, we can then test **currentLoop** against NULL and produce the appropriate error message.

If we are in the main program body, then **currentProc** will be NULL. Within the **checkReturnStmt()** method, we will see this NULL and print an error message.

Note that not all "check" routines will need these parameters. For example, **checkReturnStmt()** doesn't need the **currentLoop** argument, so it is pointless to pass it in. Likewise, the **checkWhileStmt()** doesn't need the **currentLoop** argument since it will be supplying itself as the **currentLoop** to the "check" routines it calls. Other, non-statement "check" routines (like **checkBinaryOp()**) don't need these parameters since they can not contain embedded EXIT or RETURN statements.

Coercing Integers to Reals

In certain places, a coercion from integer to real must be inserted. For example:

```
2.5 + (3 * 4)
```

should be changed by inserting a call to a built-in routine named **intToReal()**:

```
2.5 + intToReal (3 * 4)
```

To represent a call to this built-in routine, we will use a new class called **Ast.IntToReal**. This class contains a single field (**expr**), which will point to the sub-expression that needs to be converted.

[A second design option would have been to use a UnaryOp node, and to define a new kind of operation, giving PLUS, MINUS, NOT, and INT_TO_REAL. A third design option would have been to insert a **FunctionCall** node, but this would get tricky since we would need a corresponding **ProcDecl**.]

In the above example, we will first allocate a new **IntToReal** node. Then we will set the **expr** field of the **IntToReal** to point to the sub-expression 3*4 (i.e, to the value of the **expr2** field in the " + "

BinaryOp node). Next, we will modify the **expr2** field of the "+" **BinaryOp** to point to the new **IntToReal** node.

Note that this is one case where we are actually changing—rather than simply adding to—the AST. Normally, I would really rather not change information in the AST, since it risks introducing bugs in previously tested code, but in this case, I think this approach is best.

Basic, Built-in Types

During type-checking, we'll need to work with 5 basic types:

```
integer
real
boolean
_string
_niltype
```

Be sure to create just one copy of each string (by entering them into the **StringTable**), since we'll be doing == comparisons in **typeEquals**. You'll need to save references to these strings (using variables with names like "integerString" and "nilTypeString") and use those later.

The last two types begin with an underscore, which makes them impossible for the PCAT programmer to use directly. Something like this:

```
var i: _string := ...;
```

would be rejected by the Lexer / Parser since identifiers cannot contain the underscore character.

In PCAT, strings can appear in only one place: as arguments in the WRITE statement, e.g.,

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

We have string values and a string type, but no variables of type string and no operators on string values.

The type of **nil** is a little tricky. We can use nil as a legal value of any record type or any array type. The following is legal:

```
type MyRec is record ... end;
     MyArray is array of integer;
var r: MyRec;
    a: MyArray;
...
r := nil;
a := nil;
```

The value "nil" is compatible with every RECORD and ARRAY, but we can't give it any single **RecordType** or **ArrayType**, since that would make it incompatible with all other **RecordType**s and **ArrayType**s.

In order to type-check expressions and assignments, we need to give **nil** a special type, which we will call "_nilType". This is the type of the **nil** value. Our type checking rules will handle this type specially.

This new type needs to be handled a little differently from other types, since it is "assignment compatible" with any **RecordType** or **ArrayType**, but type-equal to no other type except itself.

Hints on Type-Checking

Type-checking in PCAT is relatively straightforward, since the types of expressions can be synthesized from the bottom, upward. To this end, you should modify all of the "check" routines concerned with expressions (e.g., **checkExpr**, **checkBinaryOp**, etc.) to return the type of the expression.

Whenever you check an expression for semantic correctness, the check method will end by returning a pointer to an **Ast.TypeName** node representing the type of that expression.

Each L-Value also has a type and you'll need to take a look at it. (For example, in an assignment statement, you'll need to see if the type of the RHS expression matches the type of the LHS L-Value.) So you'll also need to return an **Ast.TypeName** from all methods that check L-values (i.e., **checkLValue**, **checkArrayDeref**, and **checkRecordDeref**).

If the expression or L-Value contains errors, the check methods can simply return NULL. Subsequently, when the type is used, you'll need to watch for the possibility of a NULL. If a type is NULL, it indicates that there was some error detected earlier, so the best thing to do ignore the NULL and keep going without printing any additional error message.

I recommend that you create the following support routines, which will come in handy during the type-checking. For example, **typeEquals(t1,t2)** will return TRUE iff its two arguments are the same type.

```
boolean typeEquals (Ast.TypeName t1, Ast.TypeName t2)
```

This method is passed two types. It returns TRUE iff the two types are the same. Either argument could be null, indicating a previous error of some sort. If so, we just return true, in an attempt to reduce further errors. If either type has a definition, then we must compare definitions. If neither type has a definition then return true iff their names are equal. (If a type has no definition, then either it is a basic type or there was an "undefined name" error earlier; in either case comparing names is the thing to do.)

```
boolean assignOK (Ast.TypeName to, Ast.TypeName from)
```

This routine is passed two types. It returns TRUE iff it is legal to assign a value from type FROM to type TO.

```
Ast.CompoundType getCompoundType (Ast.TypeName t)
```

This routine is passed a typename. If it has a definition, then return a pointer to the ArrayType or RecordType. Else return null.

```
boolean needCoercion (Ast.TypeName to, Ast.TypeName from)
```

This routine is passed two types. It returns true if from="integer" and to="real".

```
Ast.Expr insertCoercion (Ast.Expr p)
```

This routine allocates an IntToReal node, makes it point to p, and returns a pointer to the new IntToReal node.

It is often the case that you need to check whether it is okay to assign from one type to another. For example, here is the code I used to check assignment statements. Similar code occurs elsewhere.

```
void checkAssignStmt (Ast.AssignStmt t)
    throws FatalError
{
    Ast.Type toType = checkLValue (t.lValue);
    Ast.Type fromType = checkExpr (t.expr);
    if (assignOK (toType, fromType)) {
        if (needCoercion (toType, fromType)) {
            t.expr = insertCoercion (t.expr);
        }
    } else {
        semanticError (t.expr, "In assignment, type of LHS
                              is not compatible with type of RHS");
    }
}
```

The Difference Between the RecordType and RecordConstructor Nodes

Consider the following program:

```
1  program is
2    type t is record
3                f1: integer;
4                f2: t;
5              end;
6    var r: t := nil;
7    begin
8      r := t { f1:=123; f2:=nil };
9    end;
```

The type on lines 2-5 will be represented using **RecordType** and **FieldDecl** nodes. The constructor on line 8 will be represented using **RecordConstructor** and **FieldInit** nodes.

The first is a type. The second is a sort of literal expression that will evaluate to a record value. More precisely, a record constructor will allocate a chunk of memory on the heap and will result in a pointer, which will then be copied into the variable **r**. Record variables (such as **r**) and array variables will always be pointers and will always be 32-bits wide in our implementation, although the record or array will often be much larger.

The "myFieldDecl" Field

A new field called **myFieldDecl** has been added to **FieldInit** and **RecordDeref** nodes. You must be set this field to point back to the **FieldDecl** node that is being referenced.

Consider this program:

```
1   program is
2     type MyRec is record
3                     f: integer;
4                   end;
5     var r: MyRec := nil;
6   begin
7     r := MyRec { f := 123 };
8     r.f := 456;
9   end;
```

In the record type “MyRec”, there is a single field “f”. The type “MyRec” will be represented by a **RecordType** node, which will point to a linked list containing a single **FieldDecl** node, which will represent the field “f” (from line 3). In line 7, we see code to construct a new record and initialize field “f”. This “f” is the same “f” as on line 3, so we will need a pointer back to the node created when this field was first introduced. The record constructor in line 7 will be represented by a **RecordConstructor** node, which will point to a linked list containing a single **FieldInit** node, which will represent the assignment to field “f”. We have added a field (called **myFieldDecl**) to **FieldInit** nodes; you must set it to point back to the **FieldDecl** node created during the type declaration on line 3.

Likewise, in line 8 we are setting a field called “f”, which is the same field as mentioned on line 3, so we will need a pointer back to the **FieldDecl** node from line 3. The l-value “r.f” is represented with a **RecordDeref** node; a new field (**myFieldDecl**) has been added to **RecordDeref** . It must be set to point back to the **FieldDecl** node from line 3.

Later, in CS-322, we will assign an offset to each field within a record type. We will store these offsets in the **FieldDecl** nodes associated with that **RecordType**. Then, when we go to generate code for line 8, we will need to generate a “move” instruction. At the time we generate that instruction, we will need to know the offset of field “f”. We will obtain it by following the **myFieldDecl** pointer back to the correct **FieldDecl** node.

The Order of Processing in “checkBody()”

Here is the order that things are done in **checkBody**()...

```
boolean checkBody (Ast.Body body, Ast.ProcDecl currentProc)
    throws FatalError
{
    enterTypeDecls (body.typeDecls);
    checkTypeDecls (body.typeDecls);
    enterProcDecls (body.procDecls);
    enterAndCheckVarDecls (body.varDecls);
    checkProcDecls (body.procDecls);
    return checkStmts (body.stmts, null, currentProc);
}
```

The routine **checkTypeDecls()** fills in the **myDef** fields in **TypeName** nodes. For example, when we check the following:

```
type t1 is array of t2;
```

the **checkTypeDecls()** routine will look up **t2** and link the **TypeName** node to point to **t2**’s definition. This must be done before we check the variable declarations, since we may very well need this information. For example, the same program might also contain:

```
        var x: t1 := t1 {{ A, B, C }};
```

In order to check whether this declaration is legal, we'll need to verify that **A**, **B**, and **C** have type **t2**. In order to check this, the **myDef** fields of **TypeNames t1** and **t2** must already be filled in. This is why we must check the type declarations before we check the variable and procedure declarations.

The Types of Basic Values

As we said earlier, whenever you check an expression, you'll need to return the type of the expression. What happens when you check an integer constant? You'll need to return a pointer to a **TypeName** node whose **id** is "integer" and whose **myDef** is NULL.

Since this will happen a lot, it makes sense to create one such **TypeName** node during initialization and save a pointer to it. Then, whenever an **IntegerConst** is encountered, you can just return this pointer.

It would work fine to create a new **TypeName** node each time you need one, but it would be less efficient.

Can this approach be used to deal with **RealConst**, **StringConst**, **BooleanConst**, and **NilConst** nodes, too?

Optional Types in VAR Declarations

Consider the following program:

```
    program is
      var x : integer := 100;
      var y           := 200;
      ...
```

According to the syntax, the type is optional. If it is supplied (as it is for **x**), then the type of the initializing expression must be "assignment compatible" with the type supplied; otherwise it is an error.

In the case of variable **y**, there is no type provided. But later, when we are checking expressions that use variable **y**, we will need to know its type, so we can type-check those expressions. (To get **y**'s type when checking expressions, we will follow the **Variable**'s **myDef** field to the **VarDecl** and then we will look at the **typeName** field. This field must not be NULL at that time.) Therefore, we will have to fill in the **typeName** field with an appropriate node when we check the **VarDecl**.

So, if the type is not supplied in the **VarDecl**, the **typeName** field must be filled in. Here's how. First, call **checkExpr()** to check the initializing expression. This will return its type. (In this example, we have "200" so **checkExpr()** will return a pointer to a **TypeName** node for "integer".) Then we must fill in the **typeName** field of the **VarDecl** accordingly.

Finally, if the initializing expression is "nil", the type is required. You'll have to issue an error if the type is missing. Also, we must check that the type is either a **RecordType** or an **ArrayType**.

```
program is
  var a           := nil;   (* Error *)
  var b : integer := nil;   (* Error *)
  var c : MyRec   := nil;   (* OK *)
  var d : MyArray := nil;   (* OK *)
  ...
```

Testing

I understand that some students are using only the **tst/** files that I am providing. Consider writing your own test files, especially as you begin this project. You do not necessarily need to create a new file; you can type

java Main

and then type small PCAT programs directly in through standard input.

I have designed the test files to test working programs; it may be easier if you create your own files for debugging. (Keep in mind that "testing" and "debugging" are different.) This may be especially helpful in the beginning, when there will be lots of differences between your program and the black box output.

For example, just about the last thing I did when I created **Checker.java** was set the **mode** fields. (Setting the **mode** field is almost trivial after all the other type-checking and coercions are in place.) Since **printAst()** prints the **mode** all over the place, it will tend to flag every line of your output as being wrong until you begin setting the **mode** field. If you are not careful, you might spend a lot of unnecessary effort manually comparing output files.

Another difficulty is that some test files are quite lengthy. For example, **binaryOK.pcat** and **binaryErr.pcat** attempt to be exhaustive in testing all cases. Every time they are run they produce a lot of output and that may become a nuisance.

When you test your program on someone else's test data, there is some danger that you are modifying your code to satisfy the test data instead of creating code that is intrinsically correct. There is a natural tendency to focus on the differences between your output and the black box output, and to modify your code in any expedient way to get the outputs to agree.

Instead, you should debug your program on you own test files first and then, after it has been debugged, you should try it on the black box files. Then, if it works correctly on the black box files on the first try, you can be almost positive that your program is correct. If it fails, it is probably because of a conceptual misunderstanding, which you can focus on.

The output from the black box tends to be very lengthy and hard for a human to read. During debugging, you may wish to make use of **prettyPrintAst()** to print out selected portions of your AST. Feel free to modify **PrettyPrint.java** however you like. For example, you may add **print** statements to print out certain values at different points in your tree. This may be easier than wading through the output from the black box, which is designed more to evaluate the correctness of your code.

Of course, if you modify **PrettyPrint.java**, be sure to re-copy the "standard" version of it and use it for your final testing, since that is what will be used when the grader tests your program.

“Last Executable Stmt In This PROCEDURE Is Not A RETURN”

In detecting this error, this is the sort of code we wish to disallow. **foo** is supposed to return an **integer** but it exits without a RETURN; what value gets returned?

```
procedure foo (...) : integer is
  begin
    write ("What happens next?");
  end;
...
i := foo (...);
```

There are three possible approaches. (The Black Box program implements approach (3).)

(1) Don’t catch the error at all. Concentrate on more important errors. If the programmer happens to leave out the RETURN, unpredictable results will occur at run-time (perhaps a core dump). Very likely, what will be executed after the WRITE will depend on what random thing happens to follow in RAM. It might be the beginning of another procedure or it might even be some data, such as the string characters, which are not even executable instructions.

(2) Require the last statement in every procedure to be a RETURN. This will prevent any unpredictable results from occurring, but may force the programmer to occasionally include a RETURN statement that will never be executed. Consider the following examples:

```
procedure max (x,y: integer) : integer is
  begin
    if x<y then
      return y
    else
      return x
    endif;
  end;

procedure foo2 (...) : integer is
  begin
    loop
      write ("Harry Porter is wonderful.");
    end;
  end;
```

In the case of “max”, the last statement is an IF. Under option (2), the programmer would be forced to insert a RETURN after the IF statement, even though it is clear that it can never be executed. In the case of “foo2”, any statement after the (infinite) LOOP will never be executed, yet the programmer would be forced to insert a RETURN that would never be executed.

(3) Complain at exactly the correct places, saying that the RETURN is missing only when execution could reach the end of the procedure without returning. This is tricky, though. Consider this procedure:

```
procedure foo3 (...) : integer is
  begin
    loop
      ...
      if ... then exit; end;
      ...
    end;
  end;
```

Unlike "foo2", a RETURN is required and we need to print an error. But it is still more complex. Consider this code:

```
procedure foo4 (...) : integer is
  begin
    loop
      ...
      while ... do
        ...
        if ... then exit; end;
        ...
      end;
    end;
  end;
```

In this code, a RETURN is not required, since the EXIT is not associated with the outermost loop.

Option (3) is the trickiest to implement, but it is what a compiler really ought to do.

There are a few cases when we can see that execution will never reach a certain point, but this depends on "meta-reasoning." As an example, consider this code:

```
procedure foo5 (...) : integer is
  begin
    i := 0;
    loop
      if i<0 then exit; end;
      i := i + 1;
    end;
    (*here*)
  end;
```

In "foo5" our knowing that (*here*) will never be reached depends on our ability to reason about runtime behavior and about the values of variables at runtime. In general, a compiler can not do this; recall the theory of Turing machines and the halting problem. All production compilers will handle programs like "foo5" by saying "RETURN required." (However, some experimental program verification systems are able to reason about programs in ways humans reason about them and can identify some special cases.)

Detection of “Dead Code”

“Dead code” is a statement in a PCAT program that can never be executed, regardless of which branches get taken in IF and WHILE statements. Dead code is sometimes called “unreachable code.” The Black Box program detects dead code and reports it as a semantic error, with these two messages:

```
Dead code - execution can never reach this statement
Dead code - execution can never reach the bottom of this loop
```

Here is a simple example of dead code. The first “dead code” error message would be printed for line 5.

```
1  procedure foo (...) is
2    begin
3      ...
4      return;
5      write ("Execution can never get here!!!");
6    end;
```

The second dead-code check asks whether execution can reach the bottom of a LOOP, WHILE, or FOR statement. Something is obviously the matter with the following code, since the loop body can never be repeated. The problem is that the EXIT statement will always be executed whenever the body of the loop is executed. The bottom of the loop (i.e., the point right before the END—that is, right before the branch back to the top) can never be reached.

```
while (i<10) do
  i := i + 1
  write ("Hello");
  exit;
end;
```

The following files test dead-code detection.

```
deadErr.pcat
deadOK.pcat
```

The detection of dead-code is not as important as detecting other errors. After all, a program containing dead-code will still run without problems or run-time errors. In many compilers, dead-code is either not detected at all or a warning is issued (instead of an error). However, the presence of dead-code almost always indicates a problem in the program and I feel it should elicit an error message.

Please leave the detection of dead-code until you have caught the other errors, since it is rather tricky.

Checking Flow of Control

To summarize, there are two errors are associated with “unreachable” or “dead” code:

```
(1)  Dead code - execution can never reach this statement
(2)  Dead code - execution can never reach the bottom of this loop
```

and two errors associated with flow-of-control and RETURN statements:

```
(3)  RETURN not allowed in the main program body
(4)  Last executable stmt in this PROCEDURE is not a RETURN
```

The main body must not contain a RETURN statement and it is fairly easy to check for this. You will be passing around **currentLoop** and **currentProc** parameters to many of the "check" routines. The **currentLoop** will point to the surrounding **LoopStmt**, **WhileStmt**, or **ForStmt**, or will be NULL if we are not nested within some kind of a looping statement. The **currentProc** will point to the surrounding **ProcDecl**, when we are checking statements within a procedure and NULL if we are in the main **Body**.

These will be used to link EXIT statements to their looping statements, and to link RETURN statements to their **ProcDecl**s. If, when checking a RETURN statement, you find that **currentProc** is NULL, then you can conclude there is no surrounding procedure. This occurs exactly when the statements occur in the main body. This takes care of message (3).

Every procedure must contain a RETURN statement. Or more precisely, the execution of every procedure must end with the execution of a RETURN statement. Message (4) concerns the "flow-of-control" within a statement list, and is best handled in conjunction with the detection of unreachable code.

The key to detecting this error is to focus on the concept of execution "falling through." Consider some statement in the middle of a sequence of statements. Ask what happens after the statement is executed. Will the next sequential statement following it be executed? In other words, will execution "fall through" to the next statement? Or will execution jump to someplace else.

We can also talk about a sequence of statements "falling through" (instead of a single statement falling through). A sequence of statements will fall through if execution falls through for the last statement in the sequence.

In the case of RETURN statements and EXIT statements, execution will jump to someplace else. In the case of ASSIGNMENT, CALL, READ, and WRITE statements, execution will fall through to the next sequential statement. In the case of an IF statement, it is more complex. Whether execution can fall through after the IF statement depends on the THEN statement sequence and the ELSE statement sequence.

For example:

```
if (condition) then
  statement;
  statement;
  ...
  statement;
  RETURN;
else
  statement;
  statement;
  ...
  statement;
  RETURN;
end;
```

Execution will obviously not fall through after this IF. But in the following:

```
if (condition) then
  statement;
  statement;
  ...
  statement;
  WRITE ("Hello");
else
  statement;
  statement;
  ...
  statement;
  RETURN;
end;
```

execution does fall through. The rule is: execution of the IF statement will fall through if execution could fall through in either of the THEN and ELSE statement sequences. Execution will not fall through only if execution does not fall through in both statement sequences.

This attribute of **fallsThrough** can be computed as a synthesized attribute, with the following meaning:

```
TRUE = Execution might fall through
FALSE = Execution will definitely not fall through
```

Note that we are only concerned with whether execution "might" fall through. The compiler cannot know exactly. Consider this code:

```
if (foo()) then
  WRITE ("Hello");
else
  RETURN;
end;
```

We can only know for sure whether execution falls through after this IF if we know what value "foo()" returns. But the boolean condition can involve arbitrary computation and we must, in general, execute it to know what it returns. The compiler cannot do this. So the compiler must conclude **fallsThrough**=TRUE, since execution might fall through sometimes. (It is possible that "foo()" always returns FALSE, so the ELSE statements will always be executed and execution will always jump and never reach the statement after this IF, but the compiler can never know this.)

What about the looping statements? The FOR statement can certainly fall through, since the index will eventually get incremented past the stopping value and execution will fall through to the statement after the FOR.

```
for i := 1 to 100 do
  ...
end;
```

In the case of a WHILE, the compiler cannot make any assumptions about the boolean condition. Therefore, we must assume that the WHILE will eventually fall through.

```
while (boolean-condition) do
  ...
end;
```

In the case of a LOOP statement, whether it falls through depends on whether it contains an EXIT statement. If the LOOP does not contain an EXIT, then there is no way execution can continue after the LOOP

```
loop
  i := i + 1;
  write ("i = ", i);
end;
(* execution never gets here *)
```

If the LOOP contains an EXIT, then it is possible that the LOOP will fall through, although we cannot be certain since we don't know if the EXIT will actually be executed.

```
loop
  ...
  if (arbitrary-condition) then
    exit;
  end;
  ...
end;
```

Note that what we look for is whether the LOOP contains an EXIT and that EXIT is bound to that LOOP. In the following code, you might think the LOOP contains an EXIT, but the EXIT really belongs to the WHILE, so it doesn't count. Execution cannot fall through after the LOOP (at least because of this EXIT; there might be others that are not shown).

```
loop
  ...
  while
    ...
    exit;
    ...
  end;
  ...
end;
```

What want to compute and return the **fallsThrough** synthesized attribute for all statements and statement lists. So we can make all of the routines

```
checkStmts ()
checkAssignmentStmt()
checkWriteStmt()
checkIfStmt()
...
```

return a boolean value. My code for **checkIfStmt()** looks like this:

```
boolean checkIfStmt (t...)
...
    boolean thenCanFallThru = true;
    boolean elseCanFallThru = true;
    ...
    thenCanFallThru = checkStmts (t.thenStmts, currentLoop,
                                                    currentProc);
    elseCanFallThru = checkStmts (t.elseStmts, currentLoop,
                                                    currentProc);
    ...
    return (thenCanFallThru || elseCanFallThru);
}
```

I deal with LOOP statements as follows. First, I have a global variable called **anyExitsSeen**, which is a boolean. The **checkExit()** routine simply sets **anyExitsSeen** to TRUE whenever it is called. The difficulty occurs with nested WHILE, FOR, and LOOP statements. The idea is that for any of these looping statements, you must save the old value of **anyExitsSeen** while you look at the inner looping statement. After saving the old value, you can reset it to FALSE, just before calling **checkStmts()** to process the loop-body statement sequence. After processing the statement sequence, you must remember to restore the old value of **anyExitsSeen** before returning. In the case of the LOOP statement, if **anyExitsSeen** got set to true when processing the loop-body statement sequence, then the LOOP can fall through. Otherwise, it cannot. (Perhaps it is an infinite loop or perhaps it contains a RETURN statement.)

Now, given that our routines compute and return the **fallsThrough** attribute, we can detect errors (1), (2), and (4) easily.

To catch the error

```
(1)  Dead code - execution can never reach this statement
```

we need to modify **checkStmts()**. This routine goes through each statement in a linked list. If one statement does not fall through and it is not the last statement, then this error applies to the next statement. If the last statement in the sequence falls through, then so does the entire statement sequence.

To catch the error

```
(2)  Dead code - execution can never reach the bottom of this loop
```

We need only add code within **checkLoop()**, **checkWhile()**, and **checkFor()**. This code will call **checkStmts()** on the loop-body statement sequence. If it returns **fallsThrough**=FALSE, then we have an error.

To catch the error

```
(4)  Last executable stmt in this PROCEDURE is not a RETURN
```

all we need to do is look at the statement sequence in the BODY of a procedure. If it falls through, we have an error. Before doing this test, we will need to know if we are working on the main body or the body of some procedure. This error doesn't apply to the main body's statement sequence.

How to Get Started

Above, I discussed several routines (**typeEquals**, **assignOK**, etc.). You'll want to get these working as soon as possible.

I suggest that you do things in more-or-less the following order:

(1) Add code so that **checkExpr** returns the correct type for **IntegerConst**, **RealConst**, **BooleanConst**, **StringConst**, **NilConst**. These will be pointers to **TypeName** nodes.

(2) Add code so that **checkExpr** returns the correct type for **Variable** nodes. You will already have a call to **find** to set **myDef**; use this value to find the type and return that type.

(3) Write the code for **typeEquals**, and **assignOK**. Write dummy code for **needCoercion** (i.e., just return FALSE) and **insertCoercion** (just return the parameter).

(4) Type in the code for **checkAssignStmts**, which is given above.

(5) Get **checkAssignStmt** working and test **typeEquals** and **assignOK** on PCAT programs that you create.

(6) Write and debug **needCoercion** and **insertCoercion**.

(7) Add the **myProc** and **MyLoop** parameters to the "check" functions relating to statements.

(8) Perform the testing associated with RETURN and EXIT statements.

(9) Move on to type checking expressions. (There is a lot of code involved in this step.)

(10) Work on the flow-of-control tests.

On Your Output Exactly Matching My Output

It may be quite difficult to make your error reporting match mine exactly. Be careful not to waste all of your time trying to get one error message perfect, and not have enough time for the other messages.

When there are semantic errors in a program, there is no need to worry about making the AST match the test data exactly. In grading this project, we will use test files with no semantic errors and test files with semantic errors.

For test files with no errors, the criterion of correctness is that both **stdout** and **stderr** must match the test data exactly. For tests with errors, the criterion of correctness is that **stderr** must match the test data, but **stdout** will be ignored.

When a PCAT program has errors, the grader should check that your code reports an error. If there is a minor difference - for example, you are catching the error on a slightly different token - that is MUCH LESS important than the fact that you are catching the error at all. When there are errors, you must catch them and when the program contains no errors, you must not report any errors.

<u>Standard Boilerplate...</u>

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 6 - John Doe
```

<u>DO NOT EMAIL YOUR PROGRAM TO THE CLASS MAILING LIST!!!</u>

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) *<u>Please ask</u>* 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.