Project 5: Checking Symbol Usage

Old Files:
Main.java
PrintAst.java
Ast.java
FatalError.java
LogicError.java
StringTable.java
Token.java
Lexer.class
go, run, runAll, makefile
tst/

File You Must Create:
Checker.java

Slight modifications

New Files I Provide:
CheckerStarter.java
SymbolTable.java
PrettyPrint.java
Parser.class

Goal: Check Symbol Usage

program is
  var x: integer := 123;
  begin
    ...
    y := (3 * x);
    ...
  end;

Definition (or “Declaration”)
“Identifier is already defined”

Use:
“Identifier is not defined”

Additional Actions:
• Make an entry in Symbol Table.
• Link each use to the correct entry.
SymbolTable.java

Methods I am providing...

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

All are static methods...

```
SymbolTable.enter (id, myNewDef);
```

These methods are augmented with “print” statements [for testing]

What is a “definition”?  
```
Ast.VarDecl  
Ast.TypeDecl  
Ast.ProcDecl  
Ast.Formal
```

The places where a new ID may be defined

Traditional Approach to Symbol Tables:

```
key  
“x”  
“y”  
“z”  
“a”  
“b”  
...  ...
```

Our Approach:
IDs in PCAT

• **Variable Names (VarDecl):**

```plaintext
var x: integer := 123;
Usage: in a “Variable” node (in an L-Value)
x := ... (x + 5) ...;
```

• **Parameter Names (Formal):**

```plaintext
procedure foo (...; p: integer, ...) is ... Usage: in a “Variable” node (in an L-Value)
p := ... (p + 5) ...;
```

• **Type Names (TypeDecl):**

```plaintext
type T1 is array of boolean;
Usage: TypeName
procedure foo (...; p: T1, ...) is ... Anywhere a type can occur.
Usage: Array Constructors
a := T1 {{ true, false, false, true }};
Usage: Record Constructors
r := T2 { name := n; age := 29; ss := 123456789 };
```
IDs in PCAT

- **Procedure Names (ProcDecl):**
  ```
  procedure foo (..., p: integer, ...) is ...
  ```
  Usage: Call Statements
  ```
  x := 4;
  foo (a, b, c);
  y := 5;
  ```
  Usage: Function Calls (within expressions)
  ```
  x := (4 + foo (a, b, c)) * y;
  ```

Lexical Level ("Scope Level")

```
program is
    var a: ...;
    procedure foo (b:...) is Level 1
        var c:...;
        procedure bar1 (d:...) is Level 2
            var e;...
            begin
            ... Point W ...
            end;
        begin
            ... Point X ...
        end;
    begin
        ... Point Y ...
    end;
begin
    ... Point Z ...
end;
```
**Equivalent ("Isomorphic") Structures**

\[
((((())) (((()))) (((())) (((())) (((()))) (((())) (((())) (((())) (((()))))))))
\]

---

**Project 5: Checking Symbol Usage**

---

**SymbolTable.enter (id, varDecl)**

- SymbolTable
  - level: 2
  - SymbolTable.enter (id, varDecl)
  - id
defhashValnext
  - Bucket
  - VarDecl
  - id
defhashValnext
  - Bucket
  - Bucket
  - Bucket

---

© Harry H. Porter, 2005
Project 5: Checking Symbol Usage

SymbolTable.enter (id, varDecl)

SymbolTable

ScopeList:

level: 2

© Harry H. Porter, 2005
Project 5: Checking Symbol Usage

Fields in Ast.java Relevant to this Project

New field: myDef in...

<table>
<thead>
<tr>
<th>Variable</th>
<th>Points to a VarDecl or Formal node</th>
</tr>
</thead>
<tbody>
<tr>
<td>CallStmt</td>
<td>Points to a ProcDecl node</td>
</tr>
<tr>
<td>FunctionCall</td>
<td>Points to a TypeDecl node</td>
</tr>
<tr>
<td>ArrayConstructor</td>
<td>Points to an ArrayType or RecordType node</td>
</tr>
<tr>
<td>RecordConstructor</td>
<td></td>
</tr>
<tr>
<td>TypeName</td>
<td></td>
</tr>
</tbody>
</table>

New field: lexLevel in...

| VarDecl     | The lexical level at the point the ID is declared |
| Formal      |                                                   |
| ProcDecl    |                                                   |

New field: currentLevel in...

<table>
<thead>
<tr>
<th>Variable</th>
<th>The lexical level at the point the ID is used</th>
</tr>
</thead>
</table>

You must fill in these fields’ values!
Pretty Printing

Input:
AST

Output:
Printed version of the program
Close to "source code format"
Comments are lost
Indentation is standardized
Parentheses maybe added
Invaluable in compiler development.

Main Method:

```java
Main Method:
t = parseProgram ();
checker = new Checker ();
checker.checkAst (t);
printAst (t);
prettyPrintAst (t);
```

Ideas:

- Comment out "printAst" to reduce output
- Augment PrettyPrinter with code to print fields of interest
  (see next slide)
- Checker must walk the AST
  (PrettyPrint walks the AST...)

© Harry H. Porter, 2005
**Main Method:**

```java
t = parseProgram ();
checker = new Checker ();
checker.checkAst (t);
printAst (t);
prettyPrintAst (t);
```

**Ideas:**

- Comment out “printAst” to reduce output
- Augment PrettyPrinter with code to print fields of interest (see next slide)
- Checker must walk the AST (PrettyPrint walks the AST...)
  1. Read and understand PrettyPrint.java
  2. Merge CheckerStarter.java and a copy of PrettyPrint.java
  3. Change method names
     - `ppExpr` → `checkExpr`
     - `ppIfStmt` → `checkIfStmt`
  4. Remove all printing stuff
  5. Modify comments!!!

---

**PROGRAM IS**

```java
VAR
  [#1:] x [lexLevel=0] := 123;
  [#2:] y [lexLevel=0] := 456;
BEGIN
  ... (x [myDef=#1][currentLevel=0] + 5) ...
  ... foo [myDef=null] (3,5)...
END;
```

**Within PrettyPrint:**

```java
void ppVariable (Ast.Variable p) {
    System.out.print (p.id);
    // printMyDef (p.myDef);
    // System.out.print ("[currentLevel=
    //     p.currentLevel + "]");
}
```
Errors To Identify

Identifier is already defined
Identifier is not defined
Expecting a type name
Expecting a local or formal name
Expecting a procedure name

This field is already defined in this RECORD

Multiple assignment to field in RECORD constructor

Errors To Identify (continued)

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

procedure name
field name
variable name
type name
formal name
Errors To Identify (continued)

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

```plaintext
var integer: real := 123;

type integer is record
  myName3: integer;
  integer: real;
  myName5: boolean;
end;

procedure integer (integer: real) : boolean is ...
```

semanticError (p, "Identifier is already defined")

In CheckerStarter.java

Prints:

```
Error on line 13 near 'foo': Identifier is already defined
```

Uses node from AST to get this info.

SemanticError() does not abort!
- Unlike "syntaxError", it returns
- Why? Catch more errors.
Checking for Repeated Field Names

Idea: Use the Symbol Table!

Create a new scope and put the field IDs into the table.

```plaintext
openScope()
for each field ID
    if already defined at this scope level
        semanticError
    else
        enter (fieldID, null)
    end
end
closeScope()
```

Example:

```plaintext
record f: integer;
g: real;
f: boolean;
end
x := MyRecType { g:=3.14; g:=5.5; f:=5+f*g };
```

Must make two passes over RecordConstructors!

Duplicates: Error!

These are variable names, not field names. No Error!
**Handling Defined Types**

Each `TypeDef` associates a name with a **CompoundType** (ArrayType or RecordType)

Elsewhere in the program...
The name alone is used `TypeName`

**Goal:**
Initialize a pointer
...from the `TypeName` at the point of usage
...to the corresponding ArrayType or RecordType
`TypeName.myDef`

**Approach:**
For each `TypeDef`...
Add an entry into the SymbolTable
For each `TypeName`...
Look the id up in the SymbolTable

```plaintext
type MyRec is record ... end;

x := MyRec { ... };
```

© Harry H. Porter, 2005
Handling Defined Types

```
type MyRec is record ...
end;
```

```
var r: MyRec := ...
```

```
```

© Harry H. Porter, 2005
Handling Defined Types

```
varDecl
   id "r"
   typeName "MyRec"

varDecl
   id "r"
   typeName "MyRec"
```

```
type MyRec is record ...
   ...
end;

```

Handling the Basic Types

```
CompoundType
   type MyRec is record ...
   ...
end;

```

```
var r: MyRec := ... ;
```

TypeNames have a myDef field
Set to point to a CompoundType node
TypeNames have a myDef field
Set to point to a CompoundType node

What about

```plaintext
var x: integer := ...;
y: real := ...;
z: boolean := ...;
```

No entry for "integer" in the SymbolTable
... But this is not an error

Within checkTypeName...
Must compare id field to "integer"

If you have a basic type, then...
Do not check SymbolTable
Leave myDef set to null
Ideas for Handling Common Strings

You will often need to check for particular strings
Example: in checkTypeName:

```java
if (typeName.id == "integer") ...
```

Problem:

In Java, equal strings may not be ==
Really ought to use

```java
if (typeName.id.equals ("integer")) ...
```
But this is slow!
Ideas for Handling Common Strings

You will often need to check for particular strings
Example: in checkTypeName:
   
   ```java
   if (typeName.id == "integer") ...
   ```

Problem:
In Java, equal strings may not be ==
Really ought to use
   
   ```java
   if (typeName.id.equals ("integer")) ...
   ```
   
   But this is slow!

Solution: Use the StringTable to Share Strings!

During initialization:
   ```java
   String integerString = ...;
   ```

© Harry H. Porter, 2005
Ideas for Handling Common Strings

You will often need to check for particular strings

Example: in `checkTypeName`:

```java
if (typeName.id == "integer") ...
```

| "integer" | "real" | "boolean" |
| "x"       | "y"    |           |
|           |        | ...       |

Problem:
In Java, equal strings may not be `==`
Really ought to use
```java
if (typeName.id.equals ("integer")) ...
```
But this is slow!

Solution: Use the StringTable to Share Strings!

During initialization:
```java
String integerString = ...;
```

Within `checkTypeName`:
```java
if (typeName.id == integerString) ...
```

Global Data (i.e., fields in class Checker)

- `String nilString;` “nil”
- `String trueString;` “true”
- `String falseString;` “false”
- `String integerString;` “integer”
- `String realString;` “real”
- `String booleanString;` “boolean”

In `CheckAst`...
```java
nilString = uniqueString ("nil");
trueString = uniqueString ("true");
... etc...
```

uniqueString (String str) → String
```java
i = StringTable.lookupToken (str);
if (i == -1) {
    StringTable.insert (str, Token.ID);
}
return StringTable.lookupString (str);
```
Project 5: Checking Symbol Usage

Order of Processing a Body

Constraints on variable usage:

```plaintext
var x := ...;
y := ...x...;
z := ...x... y ...z...;
w := ...z...;
```

Okay

Not Okay!
Order of Processing a Body

Constraints on variable usage:

\[
\begin{align*}
\text{var} & \quad \text{x := ...;} \\
& \quad \text{y := ...x...;} \\
& \quad \text{z := ...x...y...z...;} \\
& \quad \text{w := ...z...;}
\end{align*}
\]

Constraints on type usage:

\[
\begin{align*}
\text{var} & \quad \text{a: T1 := ...;} \\
\text{procedure} & \quad \text{foo(...p:T1...) is ...;} \\
\text{type} & \quad \text{T2 is record} \\
& \quad \text{...} \\
& \quad \text{f: T1;} \\
& \quad \text{...} \\
& \quad \text{end;} \\
\text{type} & \quad \text{T1 is record} \\
& \quad \text{...} \\
& \quad \text{g: T2;} \\
& \quad \text{...} \\
& \quad \text{end;}
\end{align*}
\]

Must add all types before checking vars, procedures, types!

Okay

Not Okay!

All are okay
Constraints on Procedure usage:

procedure foo(...) is
...
  bar(...) ... x ... T1 ...
...
procedure bar(...) is
...
  foo(...) 
...

Must add all procedures before checking the first procedure!
Constraints on Procedure usage:

procedure foo(...) is
...
    bar(...) ... x ... T1 ...
...
procedure bar(...) is
...
    foo(...) ...
...
var x: ....;

Must add all procedures before checking the first procedure!

Must add all vars and types before checking the procedures!
From PrettyPrint...

```c
void ppBody (Ast.Body p) {
    ppTypeDecls (p.typeDecls);
    ppProcDecls (p.procDecls);
    ppVarsDecls (p.varDecls);
    print ("BEGIN");
    ppStmts (p.stmts);
    print ("END;");
}
```

What you’ll need to do...

```c
void checkBody (Ast.Body p) {
    enterTypes (p.typeDecls);
    checkTypes (p.typeDecls);
    enterProcDecls (p.procDecls);
    enterAndCheckVarsDecls (p.varDecls);
    checkProcDecls (p.procDecls);
    checkStmts (p.stmts);
}
```

At this point it may seem that checkTypes can be called later. However, in project 6 we’ll add additional processing in checkTypes. So, call checkTypes here.