Project 5: Checking Symbol Usage

Old Files:

- Main.java
- PrintAst.java

} Slight modifications

- Ast.java
- FatalError.java
- LogicError.java
- StringTable.java
- Token.java
- Lexer.class
- go, run, runAll, makefile
- tst/

File You Must Create:

- Checker.java

Project 6: Modify this file

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...

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

Returns null, if not found

All are static methods...

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

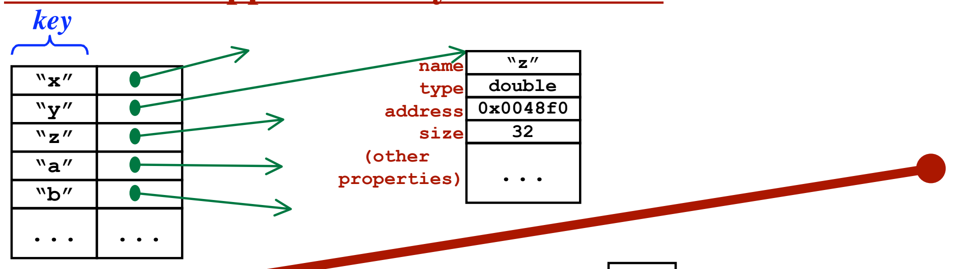
These methods are augmented with "print" statements [for testing]

What is a "definition"?

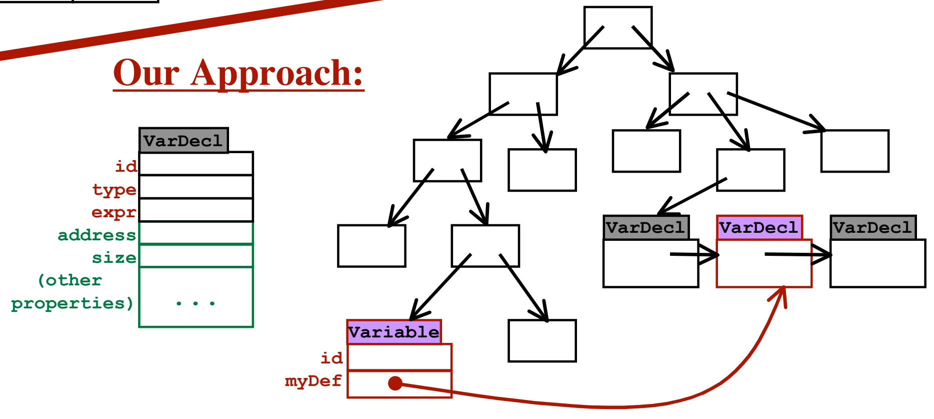
```
Ast.VarDecl
Ast.TypeDecl
Ast.ProcDecl
Ast.Formal
```

The places where a new ID may be defined

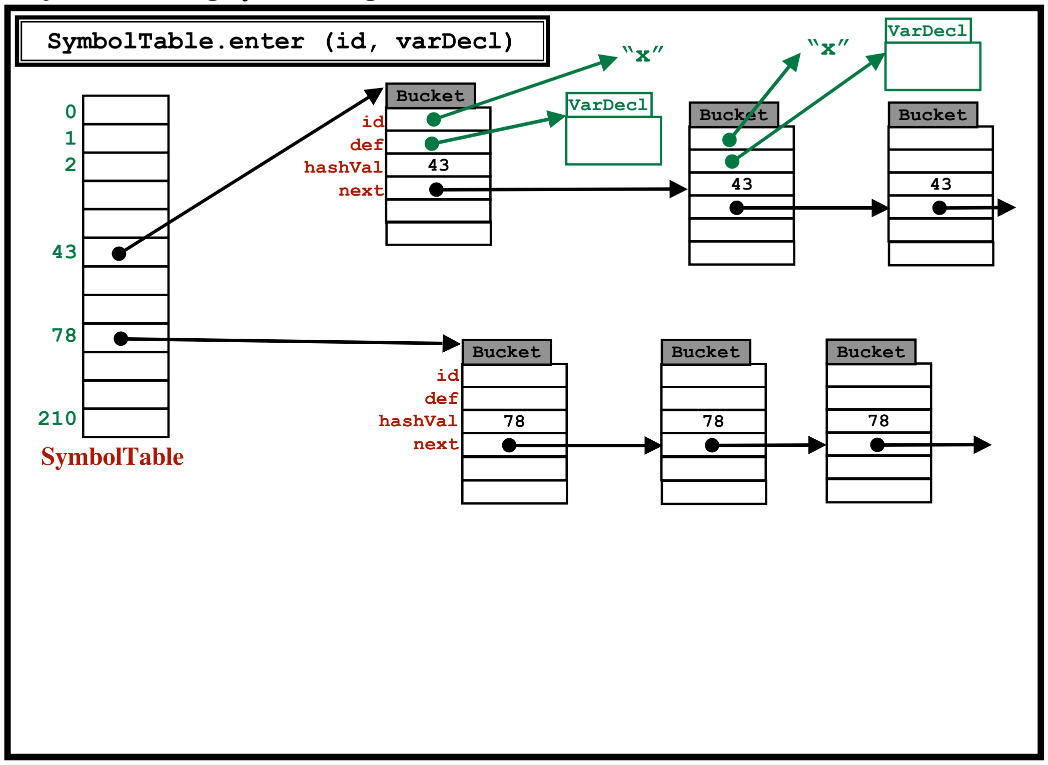
Traditional Approach to Symbol Tables:



Our Approach:



Project 5: Checking Symbol Usage



Project 5: Checking Symbol Usage

IDs in PCAT

- **Variable Names (VarDecl):**

```
var x: integer := 123;
```

Usage: in a "Variable" node (in an L-Value)

```
x := ... (x + 5) ...;
```

- **Parameter Names (Formal):**

```
procedure foo (..., p: integer, ...) is ...
```

Usage: in a "Variable" node (in an L-Value)

```
p := ... (p + 5) ...;
```

- **Type Names (TypeDecl):**

```
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 ...
```

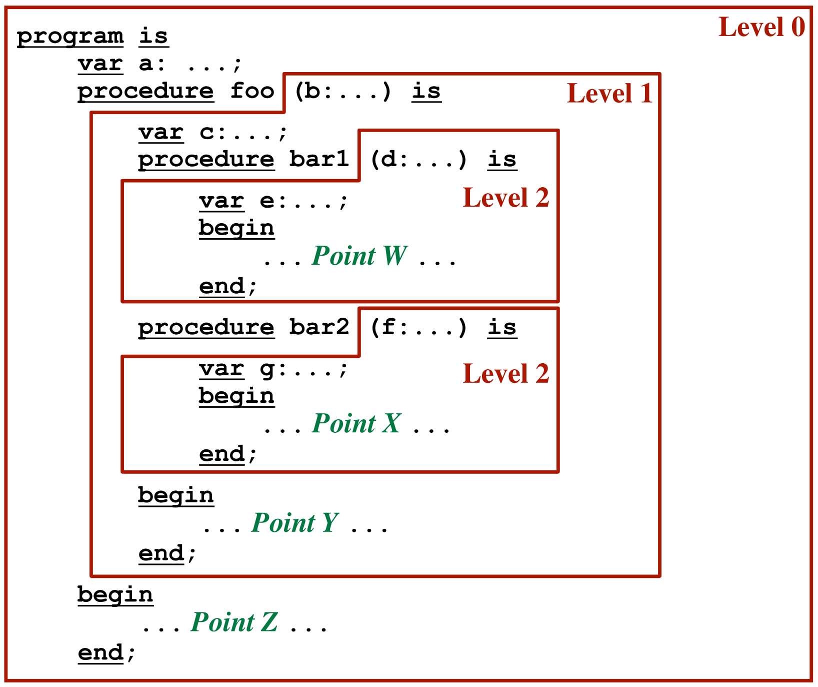
Usage: Call Statements

```
x := 4;
foo (a, b, c);
y := 5;
```

Usage: Function Calls (within expressions)

```
x := (4 + foo (a, b, c)) * y;
```

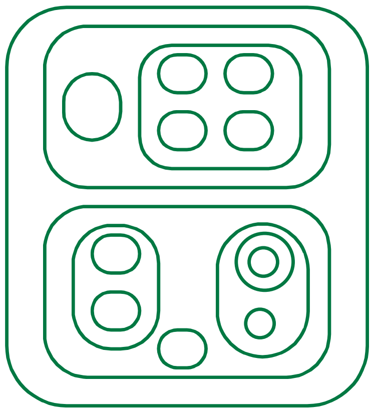
Lexical Level ("Scope Level")

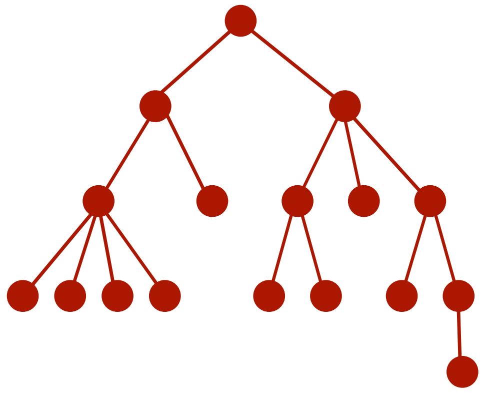


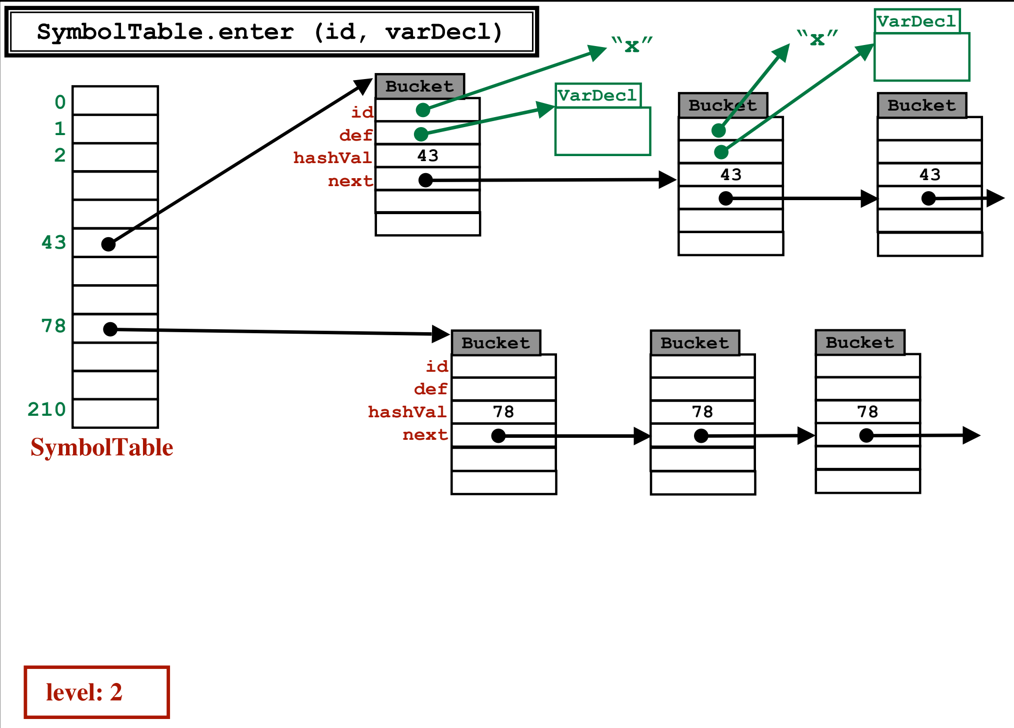
```
program is
    var a: ...;
    procedure foo (b:...) is
        var c:...;
        procedure bar1 (d:...) is
            var e:...;
            begin
                ... Point W ...
            end;
        procedure bar2 (f:...) is
            var g:...;
            begin
                ... Point X ...
            end;
        begin
            ... Point Y ...
        end;
    begin
        ... Point Z ...
    end;
```

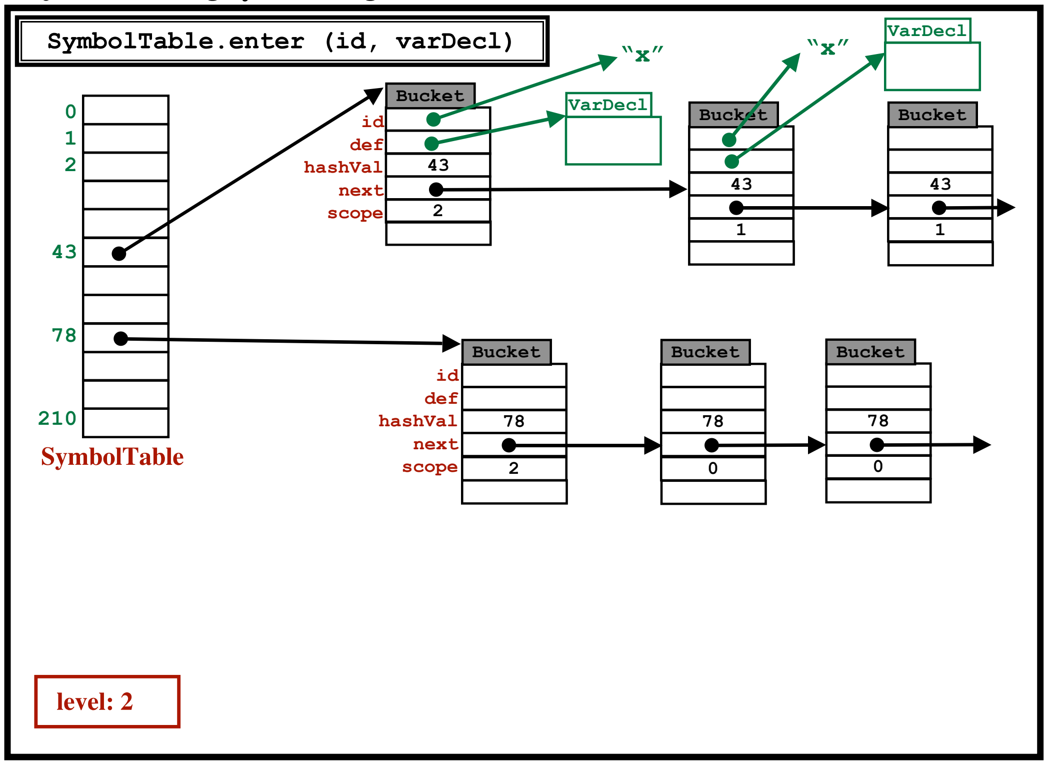
Equivalent (“Isomorphic”) Structures

```
{{{{} {} {} {}} {}} {{{} {}} {} {{} {{}}}}}}
((()  ()  ()  ())  ()) ((()  ()) () (() (())))))
```

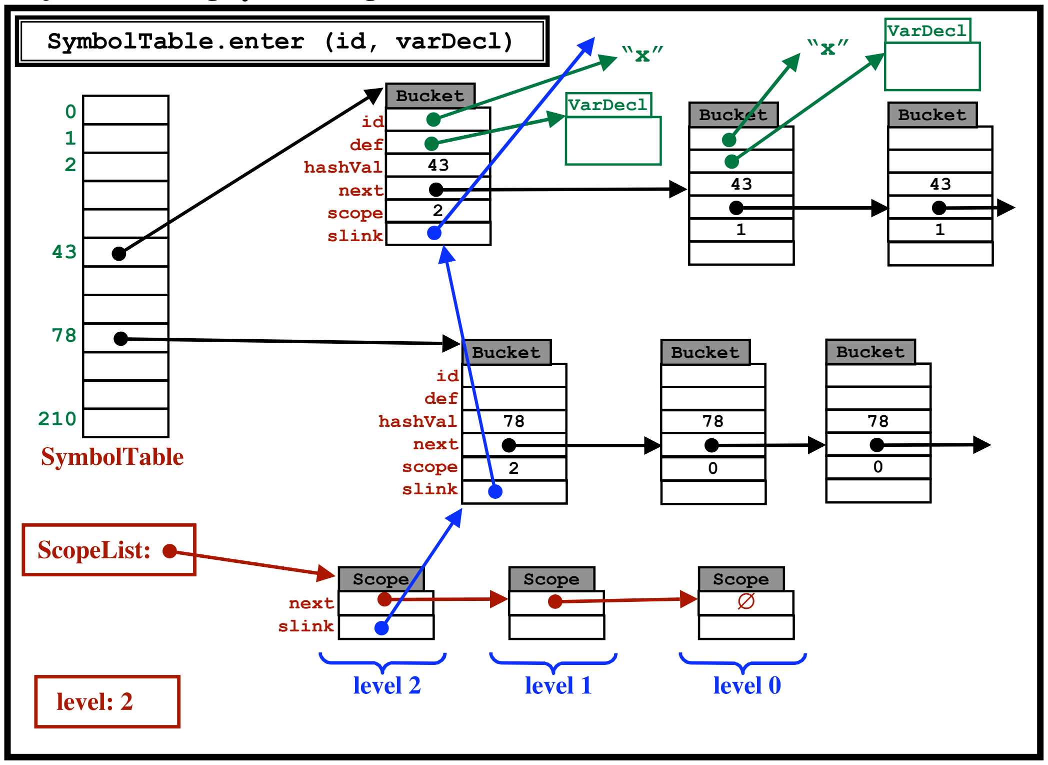




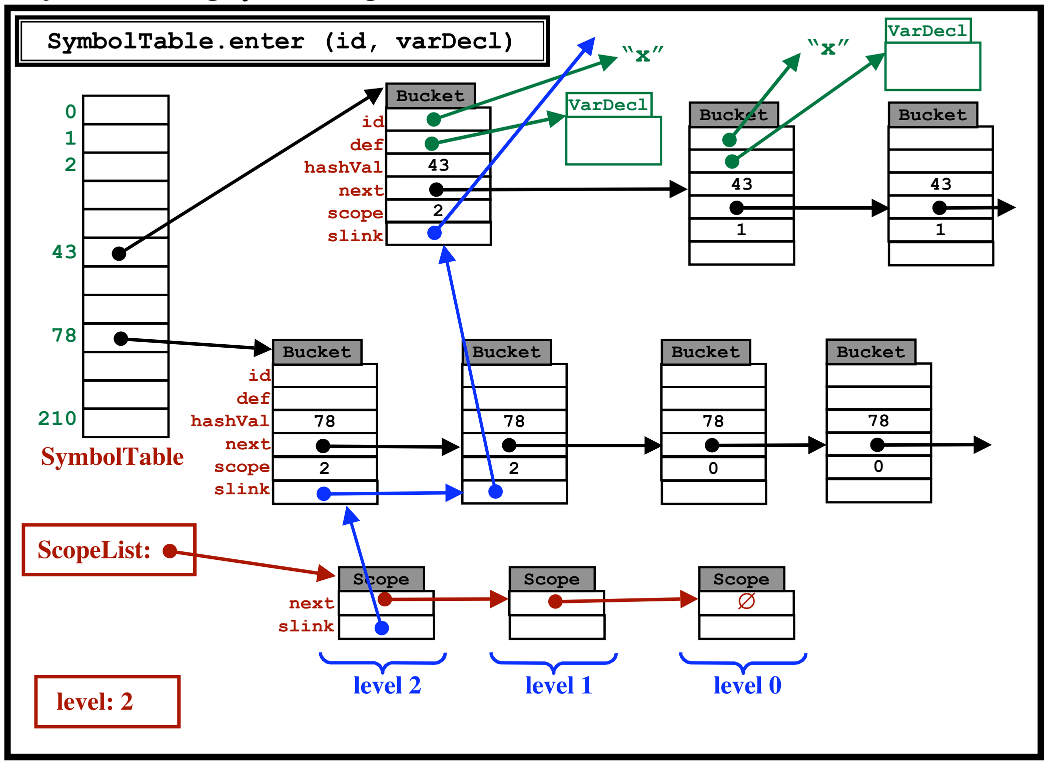




SymbolTable.enter (id, varDecl)
0
1
2
43
78
210
SymbolTable
Bucket
id
def
hashVal
next
scope
43
2
"x"
VarDecl
43
1
78
0
level: 2



SymbolTable.enter (id, varDecl)
0
1
2
43
78
210
SymbolTable
Bucket
id
def
hashVal
next
scope
slink
43
2
"x"
VarDecl
43
1
78
0
ScopeList:
Scope
next
slink
∅
level 2
level 1
level 0
level: 2



Fields in Ast.java Relevant to this Project

New field: myDef in...

`Variable` } Points to a `VarDecl` or `Formal` node

`CallStmt`
`FunctionCall` } Points to a `ProcDecl` node

`ArrayConstructor`
`RecordConstructor` } Points to a `TypeDecl` node

`TypeName` } Points to an `ArrayType` or `RecordType` node

New field: lexLevel in...

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

New field: currentLevel in...

`Variable` } The lexical level at the point the ID is used

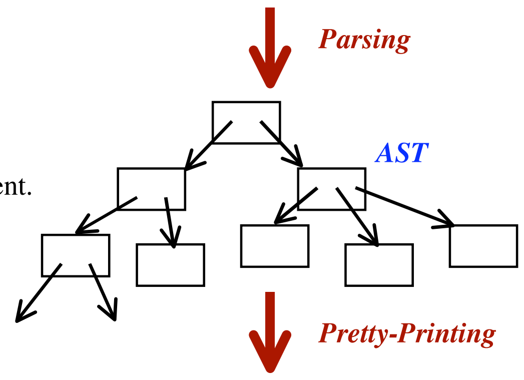
You must fill in these fields' values!

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.

```
program is begin
      (* Comment *)
              x := a + b+c; end;
```



```
PROGRAM IS
  BEGIN
    x := ((a + b) + c);
  END;
```

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...)

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...)
  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
  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:

```
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
```

```
var x: T := 123;

y = 8 * z;

w = foo (1,2,3);
```

```
type MyRecType is record
                    f:integer;
                    g:real;
                    f:boolean;
                  end;
```

```
Multiple assignment to field in RECORD constructor
```

```
x := MyRecType { g:=3.14; f:=123; g:=5.5 };
```

Errors To Identify (continued)

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

variable name

```
var myName1: real := 123;


type myName2 is record
                    myName3: integer;
                    myName4: real;
                    myName5: boolean;
                end;


procedure myName6 (myName7: real) : boolean is ...
```

type name

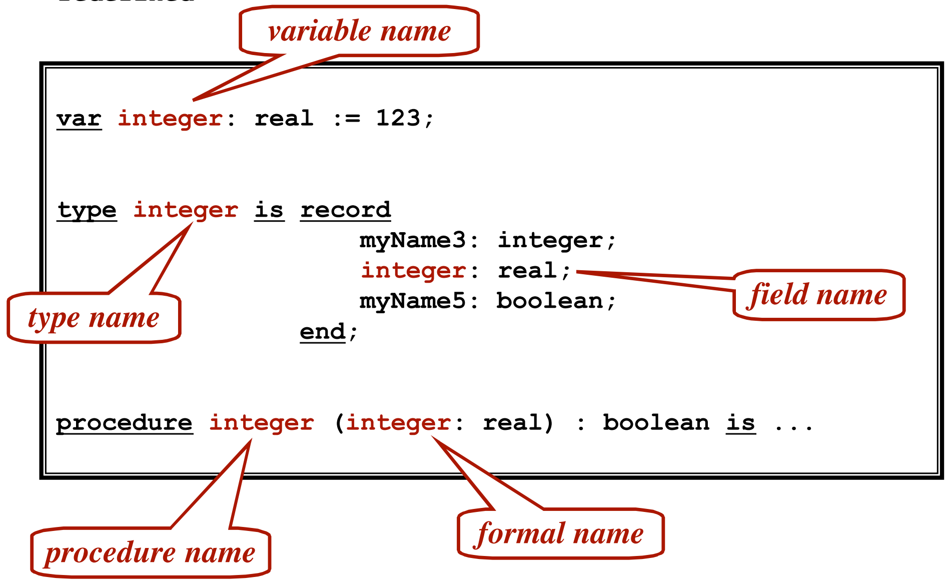
field name

procedure name

formal name

Errors To Identify (continued)

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



```
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.

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

Checking for Repeated Field Names

Idea: Use the Symbol Table!

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

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

Example:

Duplicates: Error!

These are variable names, not field names. No Error!

```
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!

Handling Defined Types

Each TypeDecl associates a name
with a CompoundType
(ArrayType or RecordType)

$$\underbrace{\texttt{\underline{type} MyRec } \overbrace{\texttt{\underline{is} \underline{record} ... \underline{end}}}^{\text{CompoundType}}\texttt{;}}_{\text{TypeDecl}}$$

$$\texttt{x := } \underbrace{\texttt{MyRec}}_{\text{TypeName}} \texttt{ \{ ... \};}$$

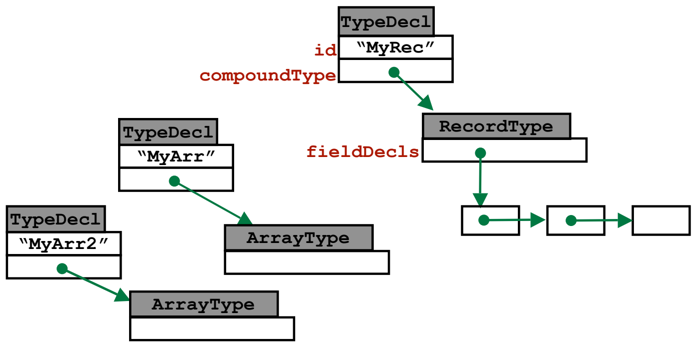
Elsewhere in the program...
The name alone is used
TypeName

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

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

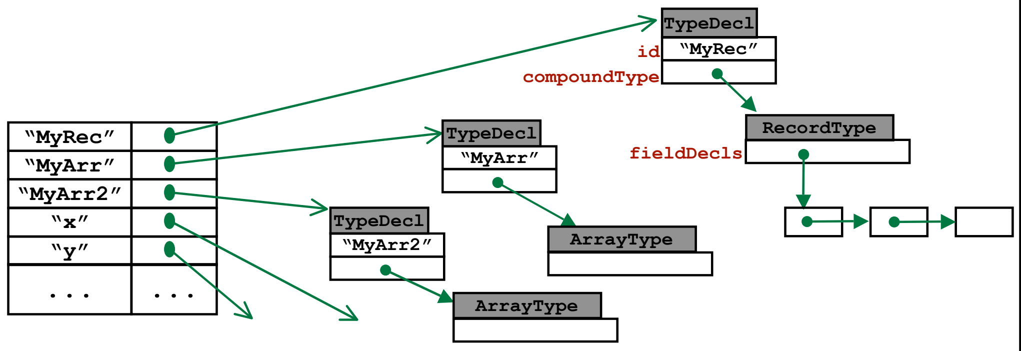
Handling Defined Types

$$\underbrace{\texttt{\underline{type} MyRec } \overbrace{\texttt{\underline{is} \underline{record} ... \underline{end}}}^{\text{CompoundType}}\texttt{;}}_{\text{TypeDecl}}$$



Handling Defined Types

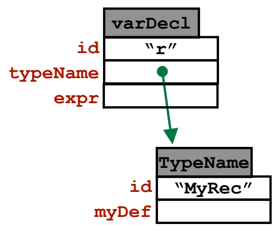
$$\underbrace{\texttt{type MyRec } \overbrace{\underline{\texttt{is}}\ \underline{\texttt{record}}\ \texttt{...}\ \underline{\texttt{end}}}^{\text{CompoundType}}\texttt{;}}_{\text{TypeDecl}}$$

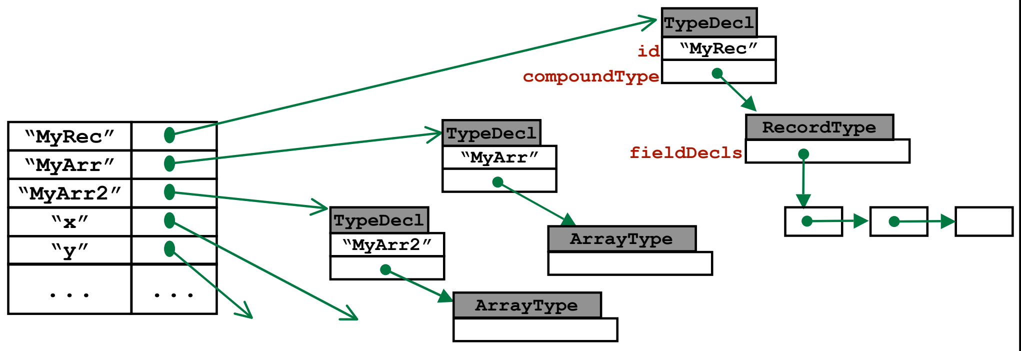


Handling Defined Types

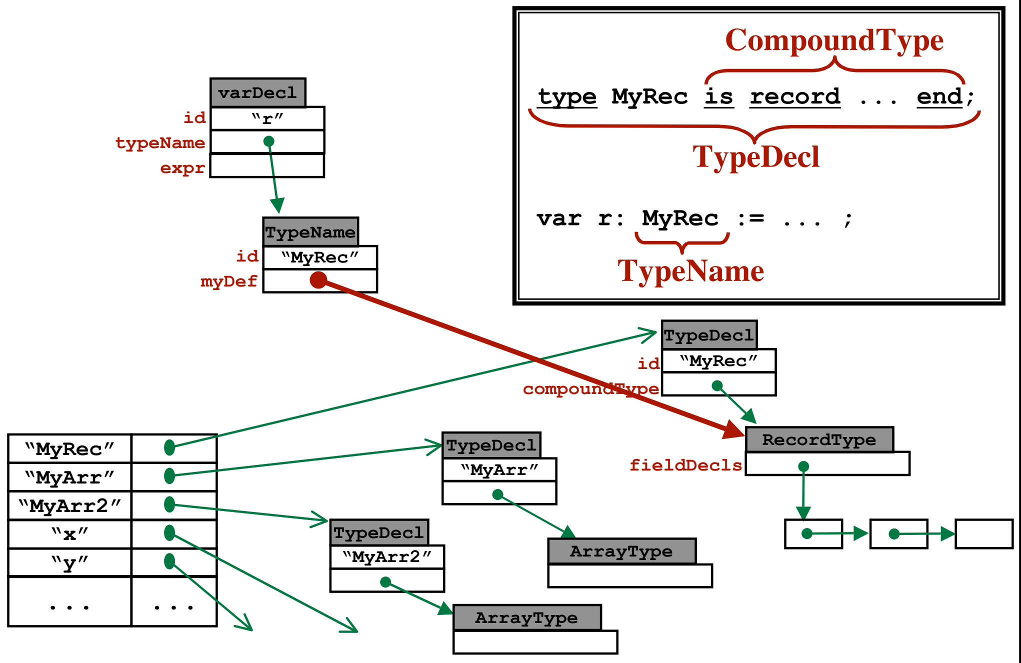
$$\underbrace{\texttt{type MyRec } \overbrace{\underline{\texttt{is}}\ \underline{\texttt{record}}\ \texttt{...}\ \underline{\texttt{end}}}^{\text{CompoundType}}\texttt{;}}_{\text{TypeDecl}}$$

$$\texttt{var r: }\underbrace{\texttt{MyRec}}_{\text{TypeName}}\texttt{ := ... ;}$$



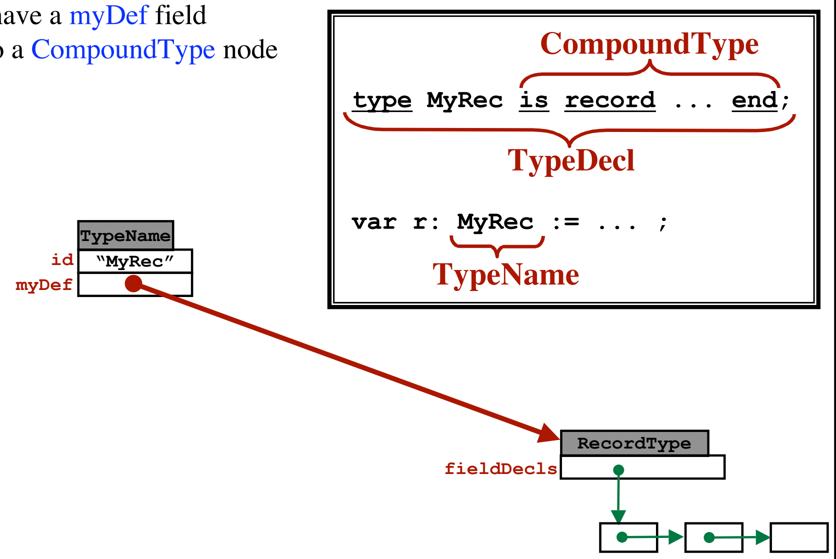


Handling Defined Types



Handling the Basic Types

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

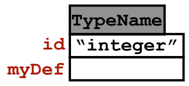


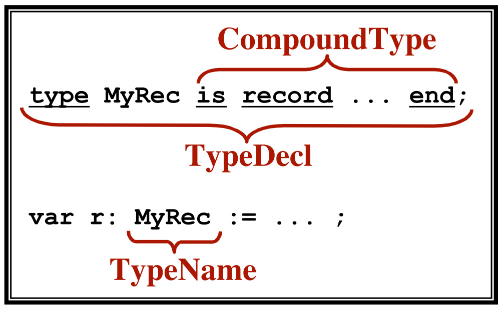
Handling the Basic Types

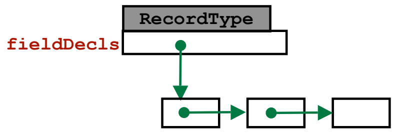
TypeNames have a myDef field
Set to point to a CompoundType node

What about

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





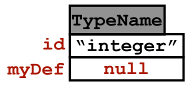


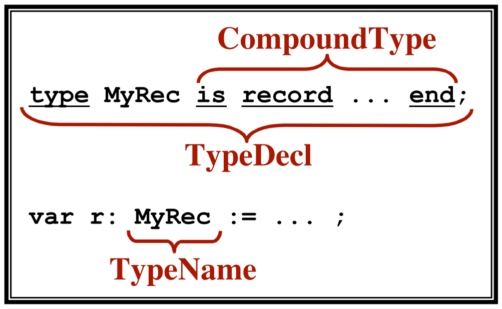
Handling the Basic Types

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

What about

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

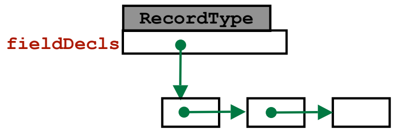




No entry for `"integer"` in the SymbolTable
... But this is not an error

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:

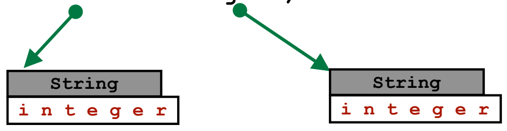
```
if (typeName.id == "integer") ...
```

Ideas for Handling Common Strings

You will often need to check for particular strings

Example: in checkTypeName:

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



Problem:

In Java, equal strings may not be ==

Really ought to use

```
if (typeName.id.equals ("integer")) ...
```

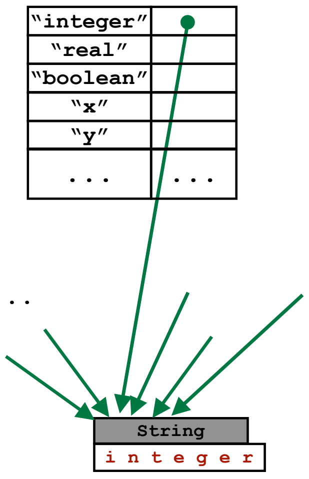
But this is slow!

Ideas for Handling Common Strings

You will often need to check for particular strings

Example: in checkTypeName:

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



Problem:

In Java, equal strings may not be ==

Really ought to use

```
if (typeName.id.equals ("integer")) ...
```

But this is slow!

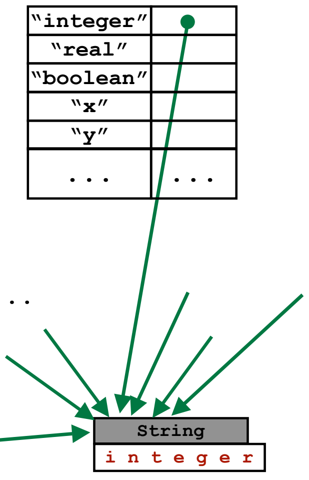
Solution: Use the StringTable to Share Strings!

Ideas for Handling Common Strings

You will often need to check for particular strings

Example: in checkTypeName:

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



Problem:

In Java, equal strings may not be ==

Really ought to use

```
if (typeName.id.equals ("integer")) ...
```

But this is slow!

Solution: Use the StringTable to Share Strings!

During initialization:

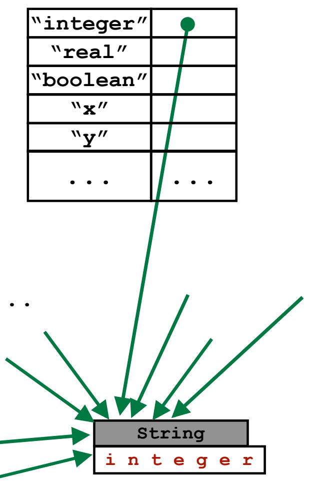
```
String integerString = ...;
```

Ideas for Handling Common Strings

You will often need to check for particular strings

Example: in checkTypeName:

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



Problem:

In Java, equal strings may not be ==

Really ought to use

```
        if (typeName.id.equals ("integer")) ...
```

But this is slow!

Solution: Use the StringTable to Share Strings!

During initialization:

```
        String integerString = ...;
```

Within checkTypeName:

```
        if (typeName.id == integerString) ...
```

Ideas for Handling Common Strings

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...

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

uniqueString (String str) → String

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

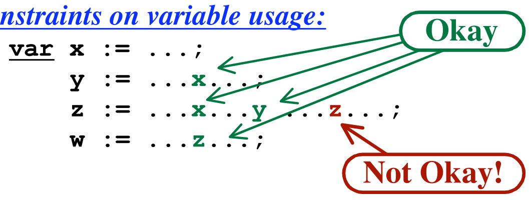
Order of Processing a Body

Constraints on variable usage:

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

Order of Processing a Body

Constraints on variable usage:



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

Order of Processing a Body

Constraints on variable usage:

Okay

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

Not Okay!

Constraints on type usage:

```
var a: T1 := ...;
procedure foo(...p:T1...) is ...;
type T2 is record
             ...
             f: T1;
             ...
           end;
type T1 is record
             ...
             g: T2;
             ...
           end;
```

Order of Processing a Body

Constraints on variable usage:

Okay

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

Not Okay!

Constraints on type usage:

```
var a: T1 := ...;
procedure foo(...p:T1...) is ...;
type T2 is record
             ...
             f: T1;
             ...
           end;
type T1 is record
             ...
             g: T2;
             ...
           end;
```

All are okay

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

Constraints on Procedure usage:

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

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: ...;
   type T1 is ...;
```

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: ...;
   type T1 is ...;
```

Must add all procedures *before* checking the first procedure!
Must add all vars and types *before* checking the the procedures!

From PrettyPrint...

```
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...

From PrettyPrint...

```
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...

```
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.