Symbol Tables

Exercise: Identifying the Scoping Rules of a PL

Using the MeggyJava materials,

1. List how new names are introduced
2. List rules for visibility of names

Exercise: Identifying the Scoping Rules of a PL

Using the full MeggyJava grammar,

1. List how new names are introduced:

Class names
Method names
Formal parameter names
Class member variable names
Method local variable names

2. Where are existing names used/accessed:

Expressions:
New object creations
Method calls
Actual parameters
Other expressions

3. List rules for visibility of names (Scoping) rules: static scoping.

What features does PA4 add?

- Meggy.toneStart
- < operator
- User-defined methods
- Parameters (formals and actuals)
- Method calls

Notice: still no local or class variables, assignments, or arrays, or objects (though syntax is included for object creation)

What do you need to do to handle these new features?

1. Copy your working PA3 compiler to a separate folder.
2. Write test cases for new features of PA4, one feature per test case.
3. Add new PA4 grammar rules to the JavaCup file, and test the grammar additions incrementally.
4. Add the actions to build the corresponding AST parts for those rules and test tree building

Now, you have correct PA ASTs, Now what? Symbol Table Building!!

- Extend SymTable package from PA3 soPA4 compiler keeps track of:
  - For each method, INSERT: type signature, formal parameters and their types

  (MethodDecl nodes)

  -> At each call site, LOOK UP type information for methods, parameters, and expressions and storage location for parameters

  (CallExp and CallStmt nodes)

  -> At each access to actual parameter, LOOK UP type information and storage location (IdLiteral)

2-Team Exercise:

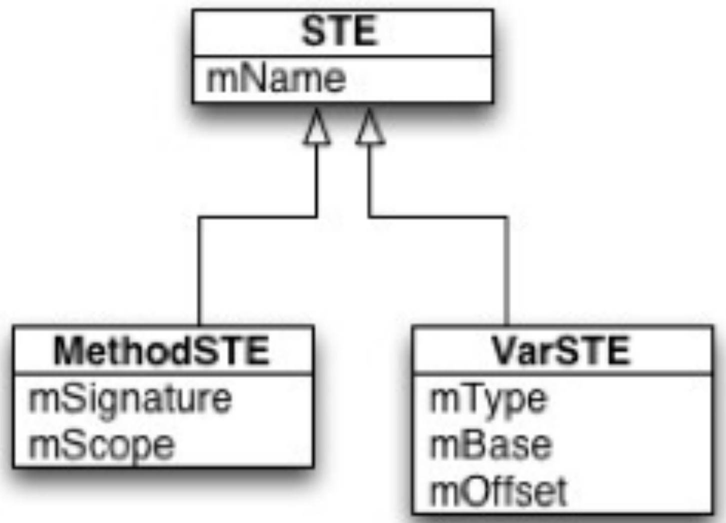
1. Using MeggyJava materials (PA4 assignment), identify what semantic checks need to be done by the compiler?
2. Using PA4raindrop.java and AST, identify where Insert and Lookup operations need to be done.
3. What are the type signatures for each of the methods in PA4raindrop.java?

A Symbol Table Entry

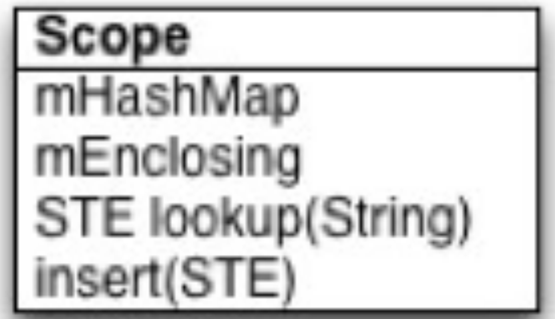
- For each kind of named object, what information is needed to
  - Perform semantic checks
  - Generate code efficiently (not have to traverse AST for the information in faraway places

PA4 Symbol Table Entries and Scopes

Symbol Table Entry Classes



Scope Class for a single scope, one for each Scope will be created and linked together to make a symbol table.



VarSTE, variable symbol table entry

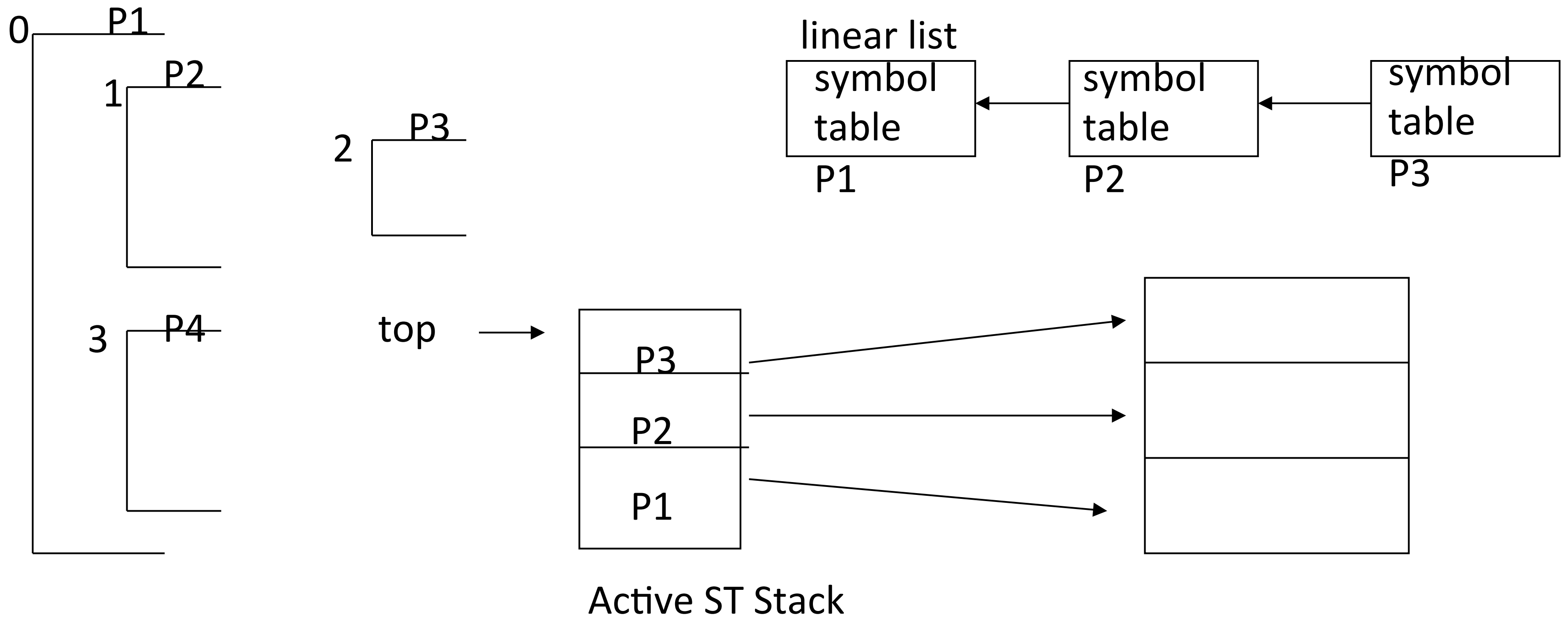
- type
- base, string for base register "Y", later will need another one
- offset, number or string for offset from base register

MethodSTE

The method symbol table entry contains a reference to signature information and to the method's scope.

Paired Team Exercise

1. Draw the Scopes and STE entries for PA4raindrop.java



During AST Visitor, we have an Active Symbol Table Stack and current ST pointer

SymTable Operations

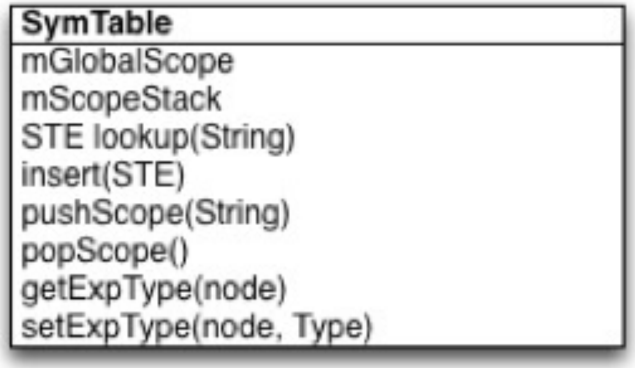
SymTable Class: A stack of scopes with current most deeply nested scope at top of stack
And a reference to the outermost (or global) scope.

STE lookup(String) - lookup in most nested

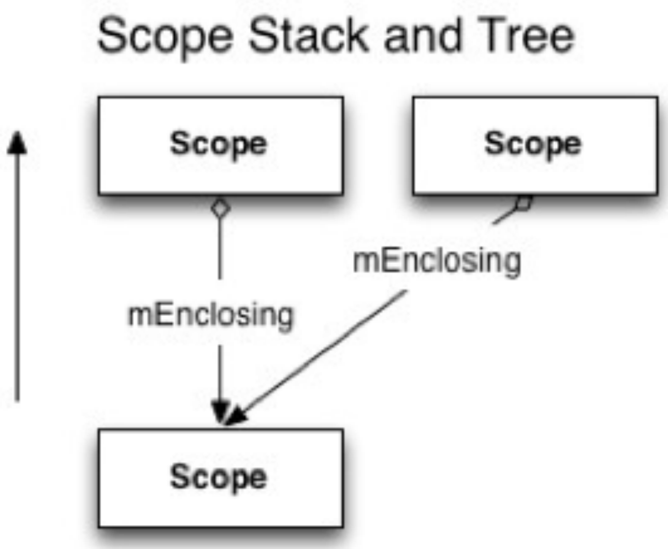
void insert(STE) - Insert STE into most deepl
Scope.

void pushScope(String)
- Look up a named scope like a method and then push its scope on stack.

void popScope()
- Pop top scope off stack.



SymTable
mGlobalScope
mScopeStack
STE lookup(String)
insert(STE)
pushScope(String)
popScope()
getExpType(node)
setExpType(node, Type)



Paired Team Exercise

1. Where would you perform the pushScope and popScope operations for PA4raindrop.java?
2. Draw the Complete SymTable for PA4raindrop.java by simulating an AST visitor to build the SymTable

What are the steps at inMethodDecl during BuildSymTable visitor?

1.
2.
3.
4.

What are the steps at inMethodDecl during BuildSymTable visitor?

inMethodDec:

(1) Look up method name in current symbol table to see if there are any duplicates. Generate an error if needed.

(2) create a function signature object of some kind

(3) create a MethodSTE

(4) insert the MethodSTE into the symbol table
with SymTable.insert

How about parameters?

Insert of formal parameter into SymTable:

inMethodDecl

(1) after creating MethodSTE and inserting it (see above), then call pushScope(methodname) on the symbol table being built

(2) set current offset in visitor to 1

How about parameters?

outFormal:

(1) check if var name has already been inserted in SymTable using st.lookup(name). Error if there is a duplicate.

(2) create VarSTE with current method offset and type of formal

(3) increment visitor maintained offset based on the type of the formal variable

(4) call st.insert

outMethodDecl:

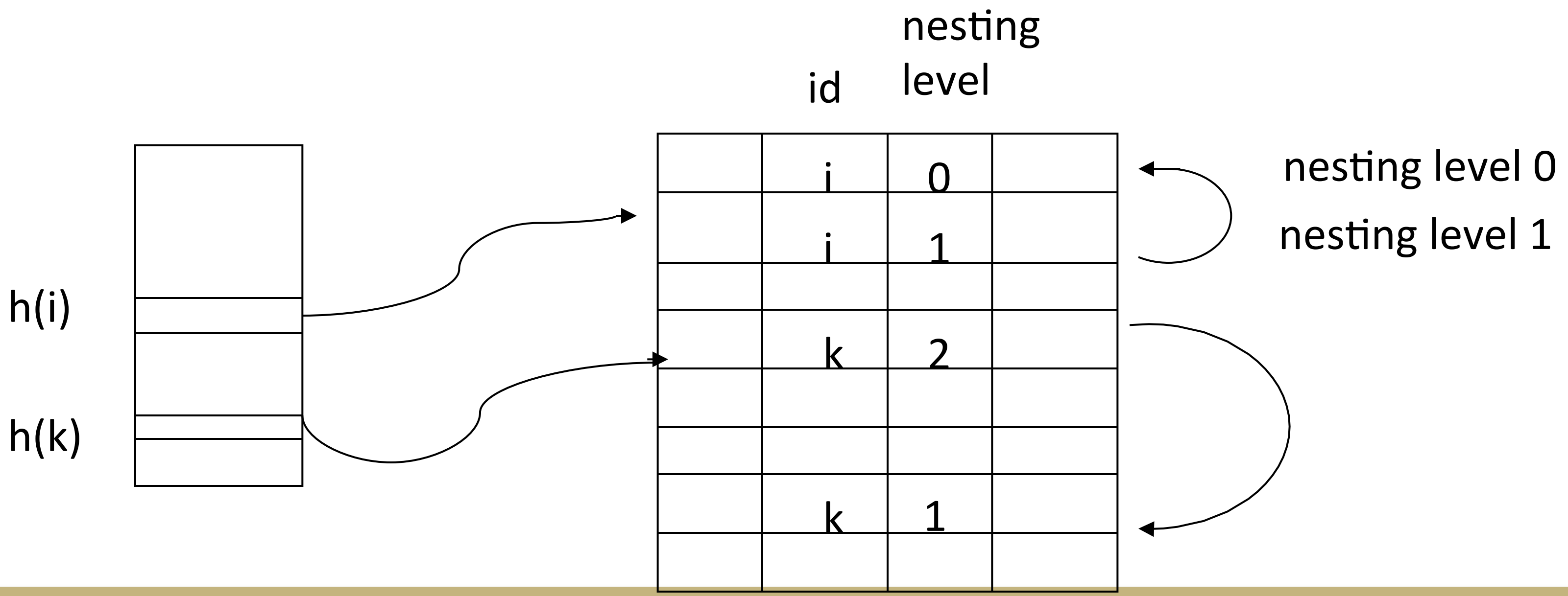
(1) Store the number of bytes needed for parameters as size of the method.

We will see more about code generation and stack frame allocation next time.

An Alternative: Single hash table for all symbol table entries

- Link together different entries for the same identifier

  and associate nesting level with each occurrence of same name.
- The first one is the latest occurrence of the name, i.e., highest nesting level

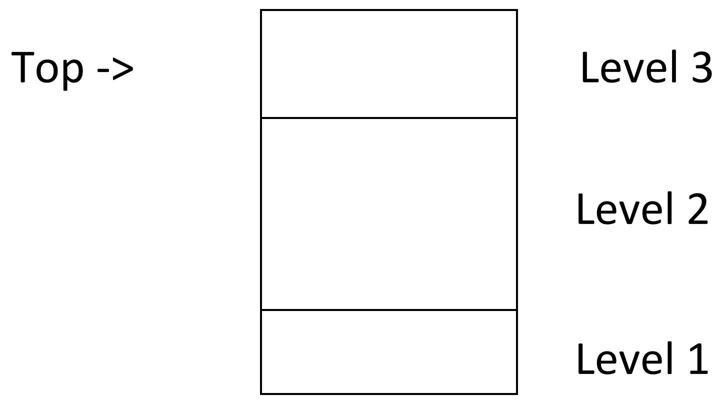


During exit from a procedure - delete all entries of the nesting level we are exiting

- must be able to do this - Remove links to most recent scope

Three ways to do this

1. Search for the correct items to remove – rehash -- expensive
2. Use extra pointer in each element to link all items of the same scope (scope chain)
3. Use an active ST stack that has entries for each scope



Pop entry and delete from hash table - keep stack entries around