Scope & Symbol Table

- Most modern programming languages have some notion of scope.
- Scope defines the “lifetime” of a program symbol.
- If a symbol is no longer accessible then we say that it is “out of scope.”
- The simplest scope is the “block scope.”
- With scope we need a notion of variable declaration which allows us to assert in which scope the variable is visible or accessible.
Simple2

- We extend our Simple1 language with variable declarations of the form

  declare x = 10;

- Declares the variable x in the current scope and initializes it to the value 10
- If the current scope is the global (outermost) scope then we call x a “global” variable.
We can now write properly scoped programs

Consider:

```c
declare x;
get x;
If ( 0 <= x) {
    declare i = x;
    while (i) {
        put i;
        i = i - 1;
    }
} else {
    declare j = x;
    while (j) {
        put j;
        j = j + 1;
    }
}
```
Variable Shadowing

- An issue with scoped declarations is that inner declarations can “overshadow” outer declarations.
- Consider:

```plaintext
declare x = 2;
{
    declare x = 3;
    {
        declare y = x + 2;
        put y;
    }
}
```

What is the output of the program once it is run?
Variable update

- A variable update can be outside of our current scope.
- Consider

```plaintext
declare x = 2;
{
    declare y = 3;
    x = y + x;
    put x;
}
put x;
```
Symbol Tables

- To deal with programs like that we need something more sophisticated for variable lookup than a hashmap.
- A hashmap stack

- This stack needs to be able to support the following functionality
  - Declare a variable (insertion)
  - Lookup a variable
  - Update a variable value
Semantic Rules for Variable Declarations

- Here are the rules which we informally used in the previous examples:
  - The ‘declare’ statement inserts a variable declaration into the current scope
  - A variable lookup returns a variable value from the current scope or the surrounding scopes
  - Every variable needs to be declared before use
  - No variable can be declared more than once in the current scope.
Symbol Tables

- **Design:**
  - we have a class `SymbolTable` that holds a stack of scopes
  - The class `SymbolTableScope` implements hashmaps at each scope level.
  - By default, the SymbolTable is initialized with a single `SymbolTableScope` object – *the global scope*. 
Symbol Tables

```
declare x = 2;
{
    declare y = 3;
    x = y + x;
    put x;
}
put x;
```
Symbol Tables

```
declare x;
get x;
If (0 <= x)
{
    declare i = x;
    put i;
}
else
{
    declare j = -1 * x;
    put j;
}
put x;
```
Symbol Tables

Symbol Table

Global Scope

Local Scope

Current Scope Pointer

Local Scope

```plaintext
declare x = 2;
{
    declare x = 3;
    {
        declare y = x + 2;
        put y;
    }
}
```
Symbol Tables

Global Scope

Local Scope

Current Scope Pointer

Local Scope

```
declare x;
get x; // assume positive value
declare i = 1;
while (i <= x)
{
    if (i <= x/2)
    {
        declare q = -1;
        put q;
    }
    else
    {
        declare p = 1;
        put p;
    }
    i = i + 1;
}
```
public class SymbolTable {

    private SymbolTableScope globalScope = new SymbolTableScope(null);
    private SymbolTableScope currentScope = globalScope;

    public SymbolTableScope pushScope() {
        currentScope = new SymbolTableScope(currentScope);
        return currentScope;
    }

    public SymbolTableScope popScope() {
        // go up one entry
        currentScope = currentScope.getParentScope();
        return currentScope;
    }

    public void declareSymbol(String symbol, Integer value) {
        // check that the current symbol was not already declared in the
        // current scope, if so then we have an error
        if (currentScope.lookupSymbol(symbol) != null) {
            System.err.println("Error: redclaring symbol "+symbol+".");
            System.exit(1);
        }
        // all clear...enter the symbol into the scope
        currentScope.enterSymbol(symbol,value);
    }

    ...
}
public Integer lookupSymbol(String symbol) {
    // lookup the symbol in the current scope
    SymbolTableScope lookupScope = currentScope;
    Integer value = lookupScope.lookupSymbol(symbol);

    // if not in current scope search up the stack
    while (value == null) {
        lookupScope = lookupScope.getParentScope();
        if (lookupScope == null) {
            // no parent scope, symbol not found
            System.err.println("Error (lookup): symbol "+symbol+" not declared.");
            // could do some more intelligent recovery here.
            System.exit(1);
            return null;
        }
        value = lookupScope.lookupSymbol(symbol);
    }
    // all done, return the value, guaranteed to be here
    // by the nature of our search procedure
    return value;
}
public void updateSymbol(String symbol, Integer initValue) {
    // find the scope where the symbol was declared
    SymbolTableScope lookupScope = currentScope;
    Integer value = lookupScope.lookupSymbol(symbol);

    // if not in current scope search up the stack
    while (value == null) {
        lookupScope = lookupScope.getParentScope();
        if (lookupScope == null) {
            // no parent scope, symbol not found
            System.err.println("Error (update): symbol " + symbol + " not declared.");
            // could do some more intelligent recovery here.
            System.exit(1);
        }
        value = lookupScope.lookupSymbol(symbol);
    }

    // we found a scope where symbol is defined, update it
    lookupScope.enterSymbol(symbol, initValue);
}
public class SymbolTableScope {
    // scope stack is built as a linked list
    private SymbolTableScope parentScope = null;

    // symbols are kept in a hashmap indexed by their name
    private HashMap<String,Integer> symbols = new HashMap<String,Integer>();

    public SymbolTableScope(SymbolTableScope parentScope) {
        this.parentScope = parentScope;
    }

    public SymbolTableScope getParentScope() {
        return parentScope;
    }

    public void enterSymbol(String name, Integer value) {
        symbols.put(name,value);
    }

    public Integer lookupSymbol(String name) {
        return symbols.get(name);
    }
}

Note: linked list is linked in “reverse” to access parents.
// the dispatcher for the interpreter visitor
public Integer dispatch(AST ast) {
    if (ast.getClass() == AssignStmt.class) return interp((AssignStmt)ast);
    else if (ast.getClass() == BlockStmt.class) return interp((BlockStmt)ast);
    else if (ast.getClass() == DeclStmt.class) return interp((DeclStmt)ast);
    else if (ast.getClass() == GetStmt.class) return interp((GetStmt)ast);
    else if (ast.getClass() == IfStmt.class) return interp((IfStmt)ast);
    else if (ast.getClass() == PutStmt.class) return interp((PutStmt)ast);
    else if (ast.getClass() == WhileStmt.class) return interp((WhileStmt)ast);
    else if (ast.getClass() == StmtList.class) return interp((StmtList)ast);
    else if (ast.getClass() == BinopExpr.class) return interp((BinopExpr)ast);
    else if (ast.getClass() == NumExpr.class) return interp((NumExpr)ast);
    else if (ast.getClass() == ParenExpr.class) return interp((ParenExpr)ast);
    else if (ast.getClass() == VarExpr.class) return interp((VarExpr)ast);
    else {
        System.out.println("PrettyPrintVisitor: unknown class type");
        System.exit(1);
        return null;
    }
}

Note: Same as Simple1 interpreter except for the addition of the declaration statement and additional functionality in block statements and variable expressions.
// declaration statements
private Integer interp(DeclStmt ast) {
    // evaluate the init expression
    Integer value = this.dispatch(ast.getAST(0));
    // declare the variable with its init value
    Interpret.symbolTable.declareSymbol(ast.Var(), value);
    // statements do not have return values -- null
    return null;
}

// block statements
private Integer interp(BlockStmt ast) {
    // set up the scope for this block
    Interpret.symbolTable.pushScope();
    // interpret each of the statements in the block
    for (int i = 0; i < ast.size(); i++) {
        this.dispatch(ast.getAST(i));
    }
    // leaving this scope -- set current scope to parent scope
    Interpret.symbolTable.popScope();
    // statements do not have return values -- null
    return null;
}

// rhs variable expressions
private Integer interp(VarExpr ast) {
    // fetch the variable value from symbol table
    return Interpret.symbolTable.lookupSymbol(ast.rhsVar());
}
Syntactic vs Semantics Errors

- Grammars allow us to construct parsers that recognize the syntactic structure of languages.
- Any program that does not conform to the structure prescribed by the grammar is rejected by the parser.
- We call those errors “syntactic errors.”
Syntactic vs Semantics Errors

- Semantic errors are errors in the behavior of the program and cannot be detected by the parser.
- Programs with semantic errors are usually syntactically correct.
- A certain class of these semantic errors can be caught by the interpreter/compiler. Consider:
  ```
  declare x = 10;
  put x + 1;
  declare x = 20;
  put x + 2;
  ```
  Here we are redeclaring the variable ‘x’ which is not legal in many programming languages.
- Many other semantic errors cannot be detected by the interpreter/compiler and show up as “bugs” in the program.
Symbol Tables

Global Scope

Current Scope Pointer

declare x = 10;
put x + 1;
declare x = 20;
put x + 2;
Symbol Tables

Symbol Table

Global Scope

Current Scope Pointer

x = x + 1;
put x;
Code

- SIMPLE2INTERPRETER.zip