Individual Exercises - Lecture 10

1. (optional - but recommended) Follow the studio associated with Crafting a Compiler Chapter 8 http://www.cs.wustl.edu/~cytron/cacweb/Chapters/8/studio.shtml
   For this a virtual machine will be provided with the necessary tools pre installed.

2. Do Fischer et al exercise 14 and 16 on pages 336-341 (exercise 15 and 17 on pages 370-373 in GE)
   14. Consider the following C program:

```
int func() {
   int x, y;
   x = 10;
   {
       int x;
       x = 20;
       y = x;
   }
   return(x * y);
}
```

   The identifier x is declared in the method's outer scope. Another declaration occurs within the nested scope that assigns y. In C, the nested declaration of x could be regarded as a declaration of some other name, provided that the inner scope's reference to is appropriately renamed x. Design a set of AST visitor methods that

   (a) Renames and moves nested variable declarations to the method's outermost scope

   You could assign a unique name to every scope.
   Then, for every variable declaration node you visit, prefix the variable name with the scope name. For example `int x = 1;` inside scope4 becomes `int scope4_x = 1;`
   Save this alias in a dictionary structure. Then move the declaration node to the outermost method scope.
   When your visitor exits out of a scope, remove all aliases that were created inside the scope.
   For every id node you visit, replace it by the most recent alias for that variable name.
   Example :

```
Int func() { // scope1
      int x, y;  // becomes scope1_x and scope1_y;
      x = 10; // scope1_x;
      { // scope 2
            x *= 2; // scope1_x
            int x; // scope2_x;
            x = 20; // scope2_x
            y = x; // scope1_y and scope2_x
      }
      return(x * y); // scope1_x, scope1_y
```

The following is a very extensive pseudo code version of the solution that was just proposed. It is okay if your solution is just an outline of a design.

In this approach we extend the symbol table to keep track of scope names. Each scope has its own dictionary to keep track of renamed variables.When we enter a new scope, it inherits all previous renamings. If a symbol is added that already exists in the dictionary, it is just overwritten with the new name:

```
public class SymbolTable {
  Stack<Scope> scopeStack = ...;
  private class Scope(String name, Dict<String, String> renamings){...}

  // Push new scope to stack passing on all renamings from current scope
  // Generate a new unique name for the new scope
  void enterScope() {...}
  // Pop scope from stack
  void exitScope() {...}

  // Add renaming to dictionary of current scope
  void addSymbol(String variableName) {
    String newName = scopeStack.peek().name + variableName;
    scopeStack.peek().renamings.put(variableName, newName);
  }
  // Look up variableName in current scope renaming dictionary
  String getSymbol(variableName){...}
}
```

We can now create a visitor for moving and renaming declarations according to the scope they are declared in:

```
public class ScopeLabelVisitor extends ASTVisitor {
  private SymbolTable symbolTable = new SymbolTable();
  private Node currentMethodScope;

  public void visit(MethodNode funcNode){
      currentMethodScope = funcNode;
      symbolTable.enterScope();
      // ... visit children
      // (Here we may visit a scopeNode)
      symbolTable.exitScope();
  };


// Continues on the next page!
```

```
  public void visit(ScopeNode scopeNode){
      scopeTracker.enterScope();
      for (Node child : scopeNode.getChildren()) {
        child.accept(this); // Visit child
        // (The child may be of the type VarDeclNode)

        // Move renamed declaration to method node
        if (child instanceof VarDeclNode) {
            scopeNode.remove(child);
            currentMethodScope.insertChild(child);
        }
      }
      scopeTracker.exitScope();
  };

  public void visit(VarDeclNode decl){
      symbolTable.addSymbol(decl.name);
      ...
  };
  ...
}
```

(b) Appropriately renames symbol references to preserve the meaning of the program

We can add the following method to the previous visitor:

```
   public void visit(VariableIdNode idNode){
       idNode.name = scopeTracker.getRename(idNode.name);
   }
```

16. As mentioned in Section 8.4.2, C allows the same identifier to appear as a struct name, a label, and an ordinary variable name. Thus, the following is a valid C program:

```
main() {
    struct xxx {
        int a,b;
    } c;
    int xxx;

    xxx:
        c.a = 1;
}
```

In C, the structure name never appears without the terminal struct preceding it. The label can only be used as the target of a goto statement. Explain how to use the symbol table interface given in Section 8.1.2 to allow all three varieties of xxx to coexist in the same scope.

Section 8.1.2 Interface:

- **openScope()** opens a new scope in the symbol table. New symbols are en- tered in the resulting scope.

- **closeScope()** closes the most recently opened scope in the symbol table. Symbol references subsequently revert to outer scopes.

- **enterSymbol(name,type)** enters name in the symbol table's current scope. The parameter type conveys the data type and access attributes of name's declaration.

- **retrieveSymbol(name)** returns the symbol table's currently valid declaration for name. If no declaration for name is currently in effect, then a null pointer is returned.

- **declaredLocally(name)** tests whether name is present in the symbol table's current (innermost) scope. If it is, true is returned. If name is in an outer scope, or is not in the symbol table at all, false is returned.

If we want to preserve the current symbol table interface without changes, we could prefix every symbol name with its corresponding type. So to add the struct xxx to the symbol_table we would call enterSymbol("struct_xxx", "struct"). When retrieving the struct we would then call retrieveSymbol("struct_xxx). This way we can differentiate between symbols that have the same name, but different types.

Alternatively we can modify the interface in one of the following ways:
1. Since the struct and labels are semantically completely different to variable declarations, it would make sense to extend the symbol table with the following methods:

retrieveStruct(name), which returns the struct for the given name or null
retrieveLabel(name), which returns the label if it present, else null

Determining which of the two methods to call can be done based on the type of node the visitor finds. If the node is of type Struct, use retrieveStruct etc. The symbol table

would then have to store all three types in a way that it can distinguish them from each other.

2. Another possible way to allow for the three different types in the symbol table is to simply add an extra column in the symbol table that holds the type of a declaration. In addition to this, the symbol table method for retrieving a symbol should then be modified to the following:

retrieveSymbol(name, type)

This method should return a valid declaration with the given name and type, if one exists.

E.g the struct declaration xxx would have a type parameter of struct, integer declaration of type integer and label declaration of type label. Thus when accessing the identifer xxx as a certain type, all the xxx identifiers will be distinguishable.

Group Exercises - Lecture 10

3. Discuss the outcome of the individual exercises
   Did you all agree on the answers?

4. Do Fischer et al exercise 1, 2, 15 on pages 336-341 (exercise 3, 4, 14 on pages 370-373 in GE)

   1. The two data structures most commonly used to implement symbol tables in production compilers are binary search trees and hash tables. What are the advantages and disadvantages of using each of these data structures for symbol tables?

   Binary search tree:

   - Advantages:
     - Compact
     - Fast search times(O(log n))
   - Disadvantages:
     - Constant need to rebalance the tree

   Hash table:

   - Advantages:
     - Constant time lookup
   - Disadvantages:
     - Can be less memory efficient
     - Overkill for very small tables

2. Consider a program in which the variable is declared as a method's parameter and as one of the method's local variables. A programming language includes parameter hiding if the local variable's declaration can mask the parameter's declaration. Otherwise, the situation described in this exercise results in a multiply defined symbol. With regard to the symbol table interface presented in Section 8.1.2, explain the implicit scope-changing actions that must be taken if the language calls for

(a) Parameter hiding
(b) No parameter hiding

Section 8.1.2 Interface:

- **openScope()** opens a new scope in the symbol table. New symbols are en- tered in the resulting scope.

- **closeScope()** closes the most recently opened scope in the symbol table. Symbol references subsequently revert to outer scopes.

- **enterSymbol(name,type)** enters name in the symbol table's current scope. The parameter type conveys the data type and access attributes of name's declaration.

- **retrieveSymbol(name)** returns the symbol table's currently valid declaration for name. If no declaration for name is currently in effect, then a null pointer is returned.

- **declaredLocally(name)** tests whether name is present in the symbol table's current (innermost) scope. If it is, true is returned. If name is in an outer scope, or is not in the symbol table at all, false is returned.

a) Parameter hiding

We create two scopes for the method. The first scope is for the method parameters. All parameters symbols will go inside this scope. Inside this scope we open a new scope for the method body. The locally declared variable within the method body will be entered into this scope. When looking up the variable in the scope, we first check our immediate scope, and then the parent scope. If the variable is declared inside the method body scope as well as in the parameter scope, we will find the one in the method body scope first and hence shadow the one from the parameter list.

b) No parameter hiding

We create one common scope for both the parameter symbols and the symbols in the method body. If a parameter and a variable declared in the method has the same name, the symbol table would identify this as an error.

15. Using the symbol table interface given in Section 8.1.2, describe how to implement structures (structs in C) under each of the following assumptions:

• All structures and fields are entered in a single symbol table.

• A structure is represented by its own symbol table, whose contents are the structure's subfields.

Section 8.1.2 Interface:

- **openScope()** opens a new scope in the symbol table. New symbols are en- tered in the resulting scope.

- **closeScope()** closes the most recently opened scope in the symbol table. Symbol references subsequently revert to outer scopes.

- **enterSymbol(name,type)** enters name in the symbol table's current scope. The parameter type conveys the data type and access attributes of name's declaration.

- **retrieveSymbol(name)** returns the symbol table's currently valid declaration for name. If no declaration for name is currently in effect, then a null pointer is returned.

- **declaredLocally(name)** tests whether name is present in the symbol table's current (innermost) scope. If it is, true is returned. If name is in an outer scope, or is not in the symbol table at all, false is returned.

We will use the same solutions used in Section 8.2.2 on page 285 to handle block-structured symbol tables. If all structures are to share the same symbol table, we assign each structure a unique *scope number*. Scope numbers should be distinguishable from the numbers used for nested scopes. Very large integers or perhaps negative values might be used. The scope number assigned is stored with the type information for a structure. When a reference like a.b is processed, a is first looked up like an ordinary identifier.

We verify that a is a structure, with scope number *s*. When field b is looked up, *s* is used to restrict lookup to the fields of the appropriate structure. When the scope of a structure is closed, its fields *are not* purged from the symbol table since later qualified lookups of its fields are possible.

If fields are given their own symbol table, lookup is easier still. When processing a structure, a new symbol table is created and populated with all the fields declared

within the structure. A pointer to this table is stored with the type information for the structure. When a reference like a.b is processed, a is first looked up like an ordinary identifier. We verify that a is a structure, with its fields contained in symbol table *st*. Then field b is looked up in *st*.

5. (optional - but recommended) Follow the Lab associated with Crafting a Compiler Chapter 8 http://www.cs.wustl.edu/~cytron/cacweb/Chapters/8/lab.shtml
   For this a virtual machine will be provided with the necessary tools pre installed.

6. Do exercise 3 in Chapter 5 of Sebesta on page 252.

   3. Write a simple assignment statement with one arithmetic operator in some language you know. For each component of the statement, list the various bindings that are required to determine the semantics when the statement is executed. For each binding, indicate the binding time used for the language.

   You may consider the following Java code:

```java
public static int increment(int x) {
    return x + 1;
}
```

   What is the binding time of
   – Value of argument x?
   – Set of values of argument x?
   – Type of argument x?
   – Set of types of argument x?
   – Properties of operator +?

   Value of argument x: Run Time

   Set of values of argument x: Language design time (Note that for languages like c/c++ the size of primitive values can vary depending on the machine you are running it on, so in that case it would be bound at language implementation time)

   Type of argument x: Compile Time

   Set of types of argument x: Language Design Time. We can for example design the language to accept float values into int parameters by implicitly casting them.

   Properties of operator +: Compile Time (Not possible to dynamically overload operators in Java)

7. What is the scope of the various x's in the following C program (with static scoping)?

```c
// Static scoping example
#include <stdio.h>
int x;
void Proc1(){
   int x;
   x = 3;
   printf("In Proc1: x = %d\n", x);
}

void Proc2(){
   x = 9;
   printf("In Proc2, x = %d\n", x);
}

int main(){
   x = 1;
   printf("Before Proc1, x = %d\n", x);
   Proc1();
   printf("After Proc1, x = %d\n", x);
   Proc2();
   printf("After Proc2, x = %d\n", x);
   return 0;
}
```

Output:
Before Proc1, x = 1
In Proc1, x = 3
After Proc1, x = 1
In Proc2, x = 9
After Proc2, x = 9

"In lexical scoping (and if you're the interpreter), you search in the local function (the function which is running now), then you search in the function (or scope) in which that function was defined, then you search in the function (scope) in which that function was defined, and so forth. "Lexical" here refers to text, in that you can find out what variable is being referred to by looking at the nesting of scopes in the program text."

See this link for a comparison between static and dynamic scoping:
https://wiki.c2.com/?DynamicScoping

8. What is the scope of the various x's in the following C Program (with dynamic scoping)?

```c
// Dynamic Scoping Example
#include <stdio.h>
int x;

void Proc1(){
   x = 1;
}
void Proc2(){
   int x;
   x = 2;
   printf("In Proc2 before Proc1, x = %d\n", x);
   Proc1();
   printf("In Proc2 after Proc1, x = %d\n", x);
}
int main(){
   x = 3;
   printf("Before Proc2, x = %d\n", x);
   Proc2();
   printf("After Proc2, x = %d\n", x);
   Proc1();
   printf("After Proc1, x = %d\n", x);
   return 0;
}
```

Result:
Before Proc2, x = 3
In Proc2 before Proc1, x = 2
In Proc2 after Proc1 x = 1
After proc 2, x = 3
After proc 1, x = 1

"In *dynamic* scoping you search in the local function first, then you search in the function that *called* the local function, then you search in the function that called *that* function, and so on, up the call stack. "Dynamic" refers to *change,* in that the call stack can be different every time a given function is called, and so the function might hit different variables depending on where it is called from."
See this link for a more thorough explanation:
https://wiki.c2.com/?DynamicScoping

Note: In this exercise you are supposed to imagine c has dynamic scoping. If this program is run, it will yield a different result.