Symbol Tables

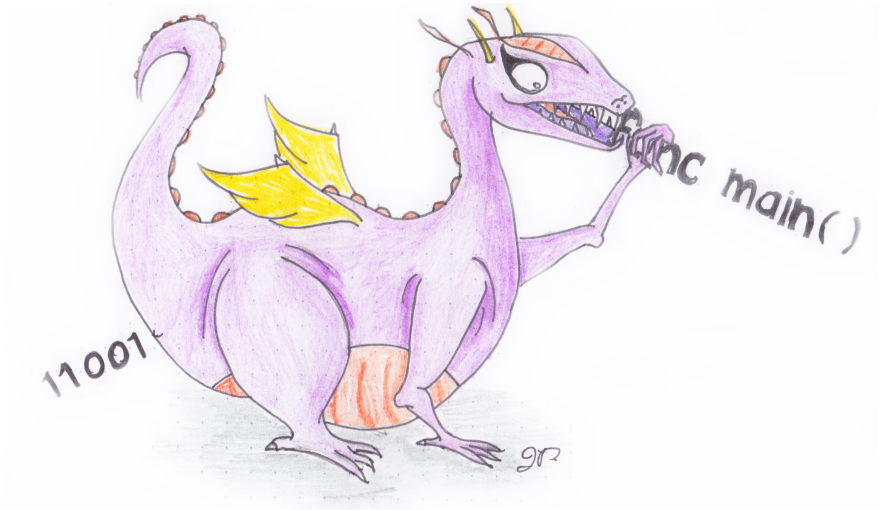
COMP 520: Compiler Design (4 credits)

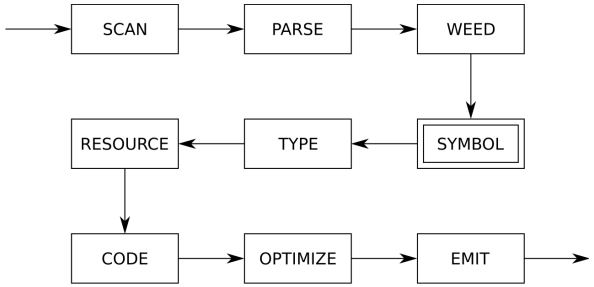
Alexander Krolik

alexander.krolik@mail.mcgill.ca

MWF 8:30-9:30, TR 1080

http://www.cs.mcgill.ca/~cs520/2019/





Semantic Analysis

Until now we were concerned with the lexical and syntactic properties of source programs (i.e. structure), ignoring their intended function. The following program is therefore syntactically valid

```
var a : string;
var b : int;
var c : boolean;

b = a;         // assigns string to int
c = (b < a);   // compares string and int
```

Definition

Semantic analysis is a collection of compiler passes which analyze the *meaning* of a program, and is largely divided into two sections

1. Symbol table: analyzing variable definitions and uses
2. Type checking: analyzing expression types and uses

The above program *may or may not* produce a semantic error depending on the exact rules of the source language.

Symbol Tables

Symbol tables are used to describe and analyze definitions and uses of identifiers.

Grammars are too weak to express these concepts; the language below is not context-free.

$$\{w\alpha w | w \in \Sigma^*\}$$

To solve this problem, we use a symbol table - an extra data structure that maps identifiers to meanings

`i`	local	int
`done`	local	boolean
`insert`	method	...
`List`	class	...
`x`	formal	`List`
⋮	⋮	⋮

To handle scoping, ordering, and re-declaring, we must construct a symbol table for every program point.

Symbol Tables

In general, symbol tables allow us to perform two important functions

- Collect and analyze symbol declarations; and
- Relate symbol uses with their respective declarations.

We can use these relationships to enforce certain properties that were impossible in earlier phases

- Variables must be declared before use;
- Identifiers may not be redeclared (in all circumstances);
- . . .

These are important features of modern programming languages

Symbol Tables in JOOS

JOOS uses symbol tables to perform numerous type and declaration functions

- Class hierarchies
  - Which classes are defined;
  - What is the inheritance hierarchy; and
  - Is the hierarchy well-formed.
- Class members
  - Which fields are defined;
  - Which methods are defined; and
  - What are the signatures of methods.
- Identifier use
  - Are identifiers defined twice;
  - Are identifiers defined when used; and
  - Are identifiers used properly?

Static, Nested Scoping

A scope is a range over which a variable is live, and is used to associate uses with declarations.

In static scoping, symbol references (identifiers) are resolved using properties of the source code (it is also called *lexical scoping* for this reason)

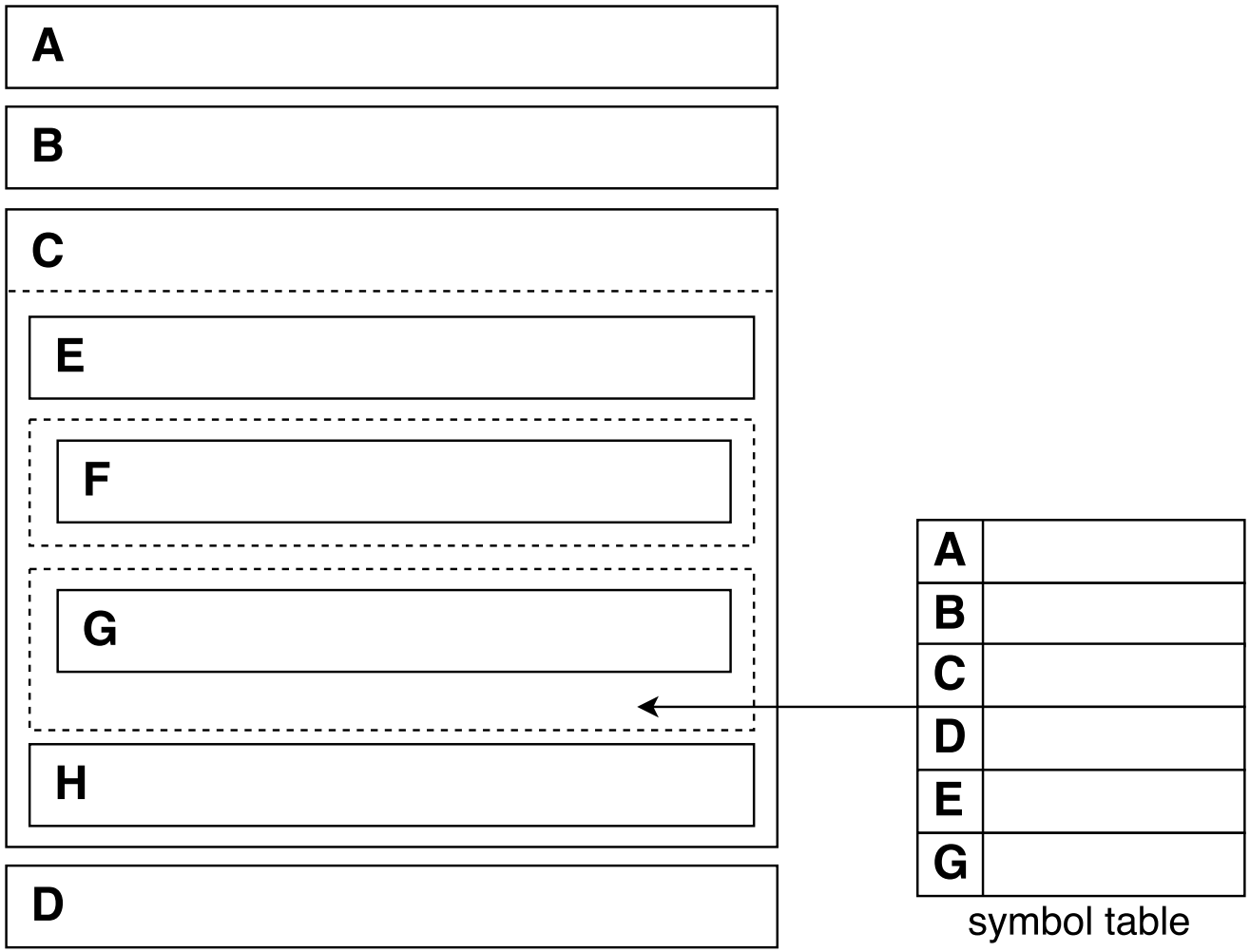
- Scopes may be nested;
- Blocks may (or may not) define new scopes – block scoping; and
- There is typically a *global scope* at the top level.

Scoping rules

- Statements/expressions may reference symbols of
  - The current scope;
  - Any parent scope (may depend on lexical ordering); or
  - The global scope; but
- Typically we cannot reference symbols defined in sibling scopes.

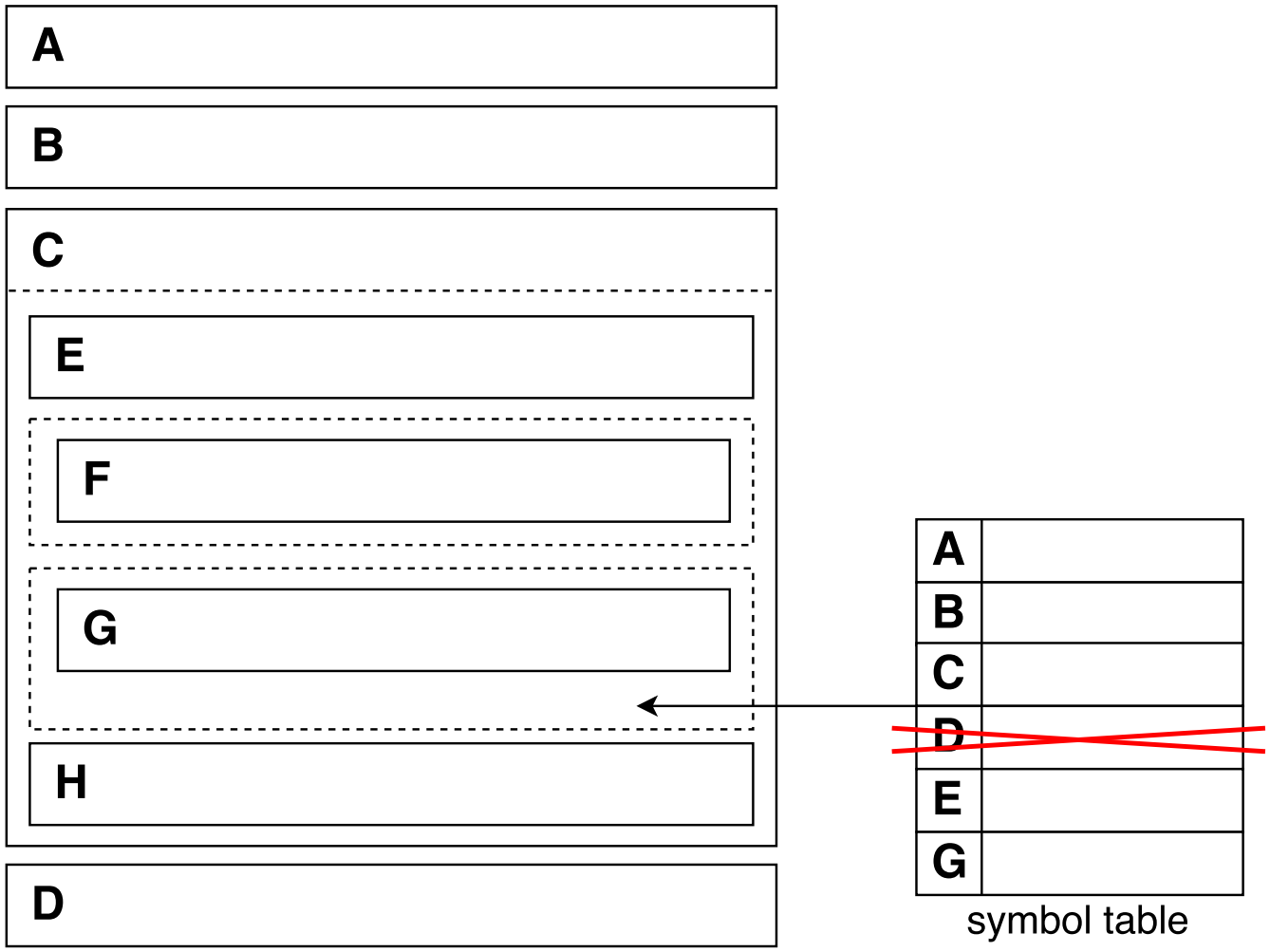
This is the standard scoping used in modern programming languages.

Static, Nested Scoping



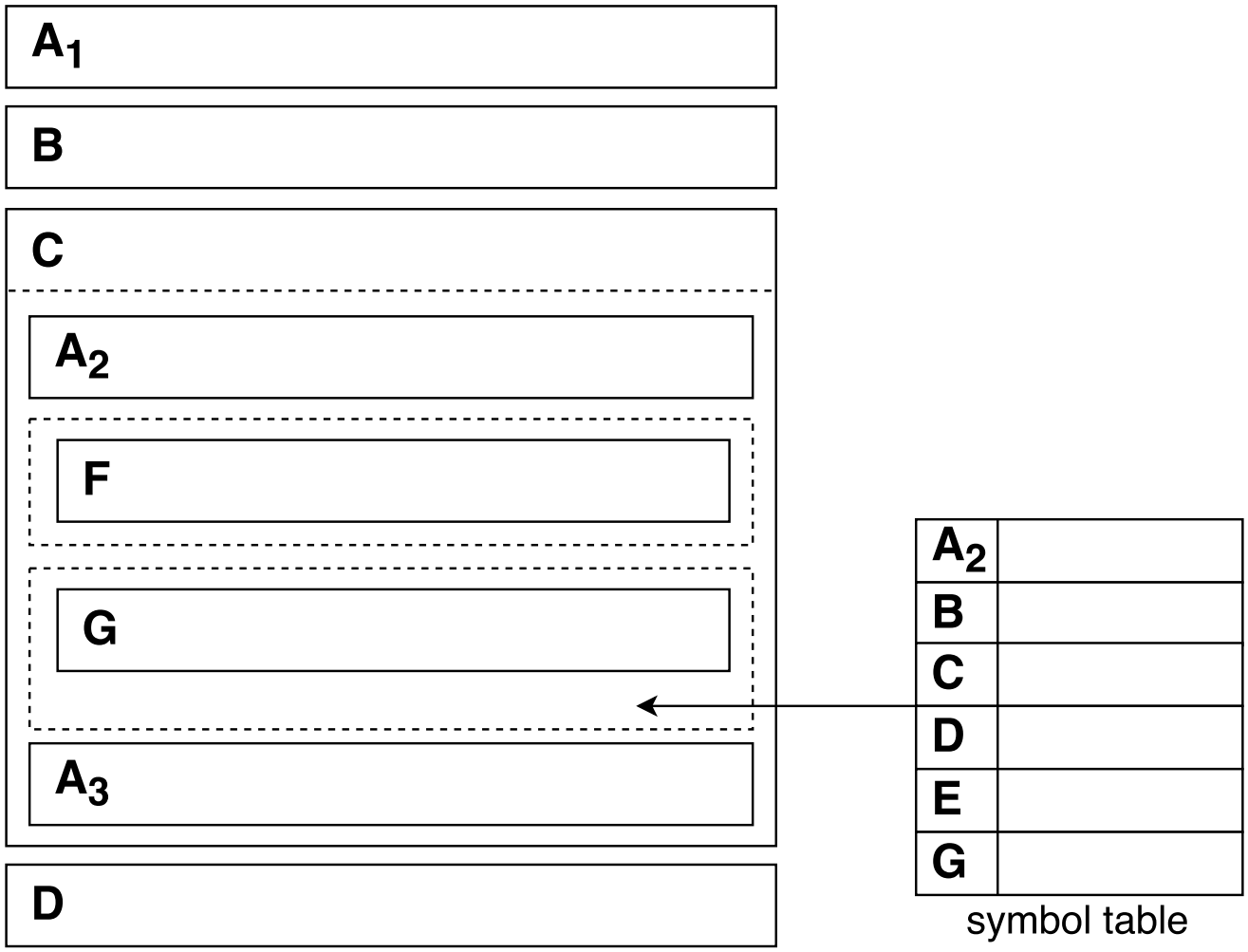
One-Pass Technology

Historically, only a single pass was performed during compilation. Elements in the global scope (or any order scopes) were thus not visible until they were defined.

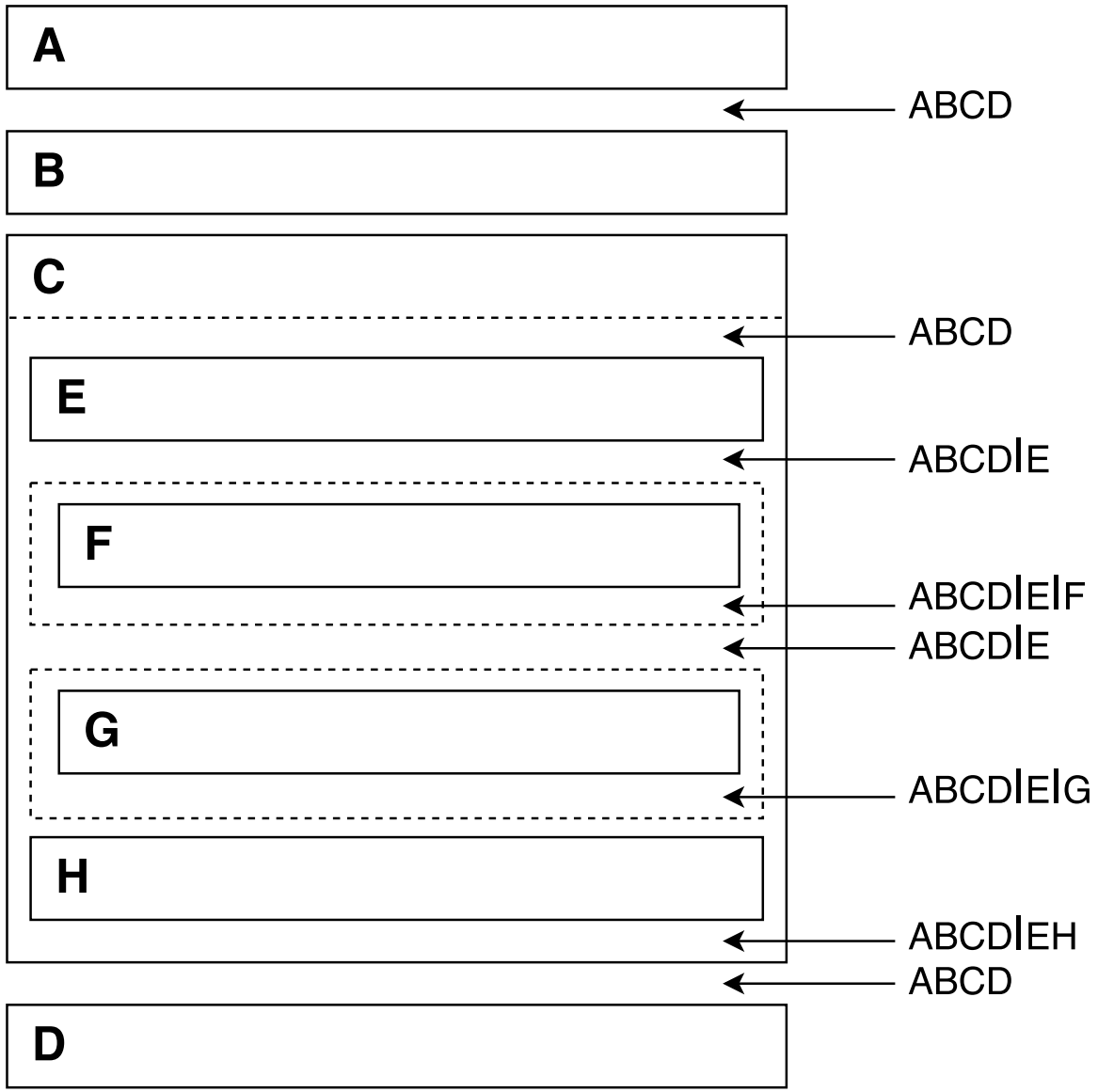


Redefinitions

If multiple definitions for the same symbol exist, use the closest definition. Identifiers in the same scope must be unique.



Scope Stack



Dynamic Scoping

Dynamic scoping is much less common, and significantly less easy to reason about. It uses the *program state* to resolve symbols, traversing the call hierarchy until it encounters a definition.

Static scoping

```
int b = 10;

func foo() {
    return b;
}

func main() {
    print foo(); // prints 10
    int b = 5;
    print foo(); // prints 10
}
```

Dynamic scoping

```
int b = 10;

func foo() {
    return b;
}

func main() {
    print foo(); // prints 10
    int b = 5;
    print foo(); // prints 5
}
```

Symbol Table Hierarchies

A flat hierarchy symbol table can be implemented as a simple map from identifiers to “information”

- `putSymbol(SymbolTable *t, char *name, ...)`
- `getSymbol(SymbolTable *t, char *name)`

But how do we handle a hierarchy of scopes?

Cactus stack

A cactus stack has multiple branches, where each element has a pointer to its parent (it’s also called a parent pointer tree)

- `scopeSymbolTable(SymbolTable *t)`
- `unscopeSymbolTable(SymbolTable *t)`
- `putSymbol(SymbolTable *t, char *name, ...)`
- `getSymbol(SymbolTable *t, char *name)`

Finding elements is linear search up the hierarchy.

Symbol Table Hierarchies

Symbol tables are implemented as a cactus stack of *hash tables*

Scope rules

- Each hash table contains the identifiers for a scope, mapped to information;
- When entering a new scope, we push an empty hash table; and
- When exiting a scope, we pop the top-most hash table.

Declarations

- For each declaration, the identifier is entered in the top-most hash table;
- It is an error if it is already there (redeclaration);
- A use of an identifier is looked up in the hash tables from top to bottom; and
- It is an error if it is not found (undefined).

Hash Functions

What is a good hash function on identifiers?

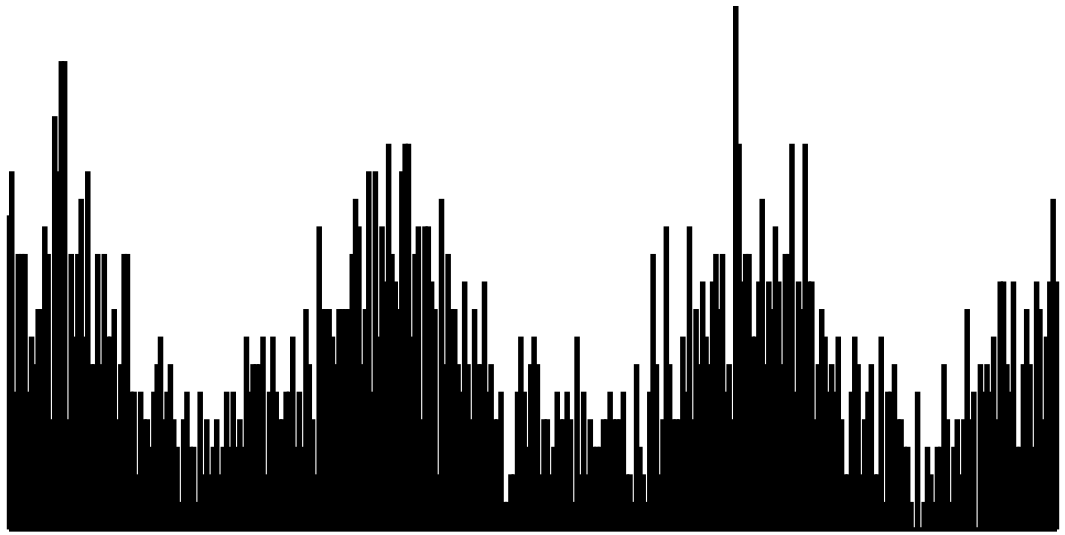
1. Use the initial letter
   - `codePROGRAM`, `codeMETHOD`, `codeEXP`, `...`
2. Use the sum of the letters
   - Doesn't distinguish letter order
3. Use the shifted sum of the letters

```
  "j" = 106 = 0000000001101010
shift       0000000011010100
+ "o" = 111 = 0000000001101111
=           0000000101000011
shift       0000001010000110
+ "o" = 111 = 0000000001101111
=           0000001011110101
shift       0000010111101010
+ "s" = 115 = 0000000001110011
=           0000011001011101 = 1629
```

Hash Function - Option 1 (JOOS source code)

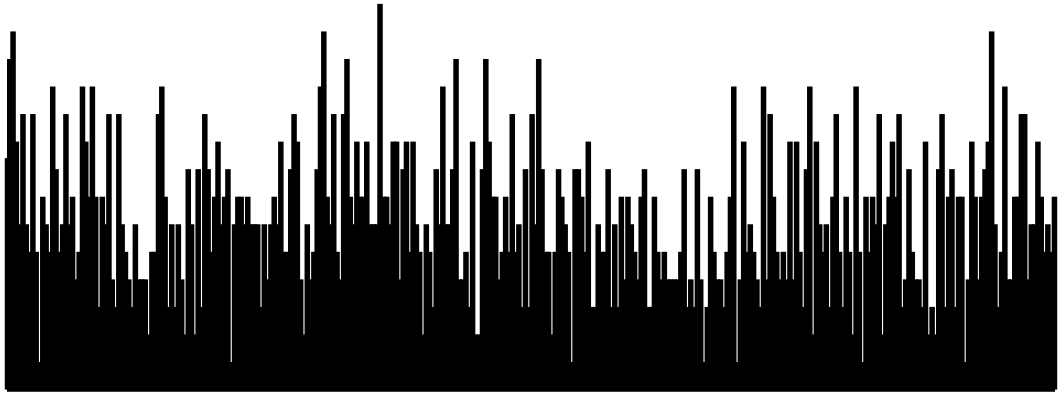
```
hash = *str;
```

Hash Function - Option 2 (JOOS source code)



```
while (*str) hash = hash + *str++;
```

Hash Function - Option 3 (JOOS source code)



```
while (*str) hash = (hash << 1) + *str++;
```

Implementing a Symbol Table - Structure

We begin by defining the structure of the symbol table and the hash functions.

```
#define HashSize 317

typedef struct SYMBOL {
   char *name;
   SymbolKind kind;
   union {
     struct CLASS *classS;
     struct FIELD *fieldS;
     struct METHOD *methodS;
     struct FORMAL *formalS;
     struct LOCAL *localS;
   } val;
   struct SYMBOL *next;
} SYMBOL;

typedef struct SymbolTable {
   SYMBOL *table[HashSize];
   struct SymbolTable *parent;
} SymbolTable;
```

The symbol table contains both the table of symbols, as well as a pointer to the parent scope.

Implementing a Symbol Table - Scoping

For each new scope we construct a blank symbol table

```
SymbolTable *initSymbolTable() {
   SymbolTable *t = malloc(sizeof(SymbolTable));

   for (int i = 0; i < HashSize; i++) {
      t->table[i] = NULL;
   }

   t->parent = NULL;
   return t;
}
```

When opening a new scope, we first construct a new symbol table and then set its parent to the current scope. Note that by construction, the global (top-level) scope has no parent.

```
SymbolTable *scopeSymbolTable(SymbolTable *s) {
   SymbolTable *t = initSymbolTable();
   t->parent = s;
   return t;
}
```

Implementing a Symbol Table - Symbols

To enter symbols, we must first decide on the appropriate hash function. Using the previous analysis, a shifted sum provides a more even distribution of values in the table (important for lookup speed).

```
int Hash(char *str) {
   unsigned int hash = 0;
   while (*str) hash = (hash << 1) + *str++;
   return hash % HashSize;
}
```

Adding a new symbol consists of inserting an entry into the hash table (note the check for redefinitions)

```
SYMBOL *putSymbol(SymbolTable *t, char *name, SymbolKind kind) {
   int i = Hash(name);
   for (SYMBOL *s = t->table[i]; s; s = s->next) {
      if (strcmp(s->name, name) == 0) // throw an error
   }
   SYMBOL *s = malloc(sizeof(SYMBOL));
   s->name = name;
   s->kind = kind;
   s->next = t->table[i];
   t->table[i] = s;
   return s;
}
```

Implementing a Symbol Table - Symbols

Lastly, to fetch a symbol from the table we recursively check each scope from top (current scope) to bottom (global scope)

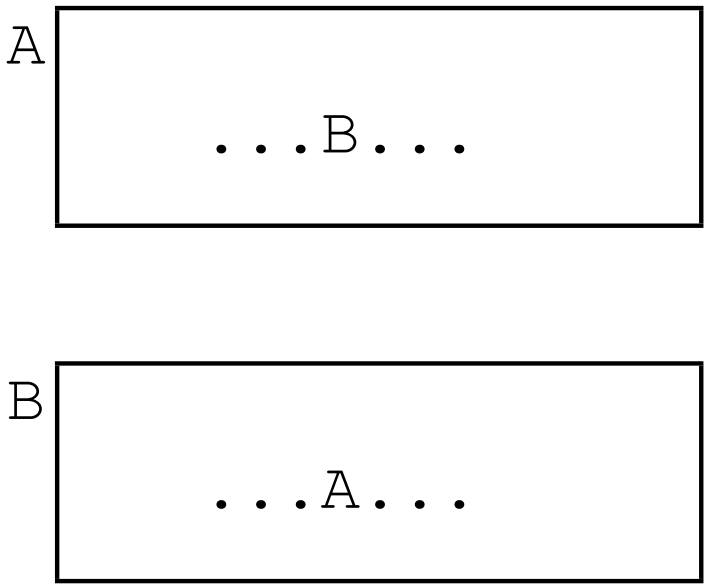
```
SYMBOL *getSymbol(SymbolTable *t, char *name) {
   int i = Hash(name);

   // Check the current scope
   for (SYMBOL *s = t->table[i]; s; s = s->next) {
      if (strcmp(s->name, name) == 0) return s;
   }

   // Check for existence of a parent scope
   if (t->parent == NULL)
      return NULL;

   // Check the parent scopes
   return getSymbol(t->parent, name);
}
```

Mutual Recursion

A typical symbol table implementation does a single traversal of the program. Upon identifier

- **Declaration**: add a new mapping to the symbol table; and
- **Use**: link the identifier use to the nearest declaration.

This naturally assumes that declarations come before use. This works well for statement sequences, but it fails for mutual recursion and members of classes.



A single traversal of the abstract syntax tree is not enough.

Mutual Recursion

Solution: Make two traversals

- The first traversal collects definitions of identifiers; and
- The second traversal analyzes uses of identifiers.

For cases like recursive types, the definition is not completed until the second traversal.

JOOS

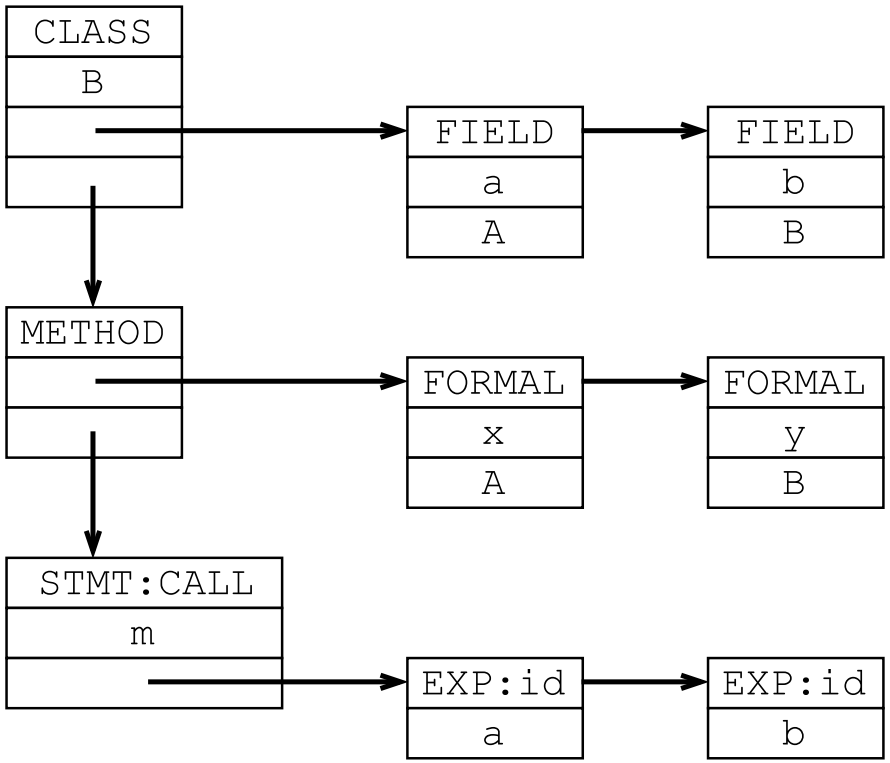
Like in Java, JOOS supports mutual recursion and allows functions to reference any member of the class *regardless* of lexical order.

1. `symInterface*`: Collect definitions of identifiers
2. `symImplementation*`: Analyze use of identifiers

Weaving Symbol Tables with the AST

Consider the following example program with its corresponding AST

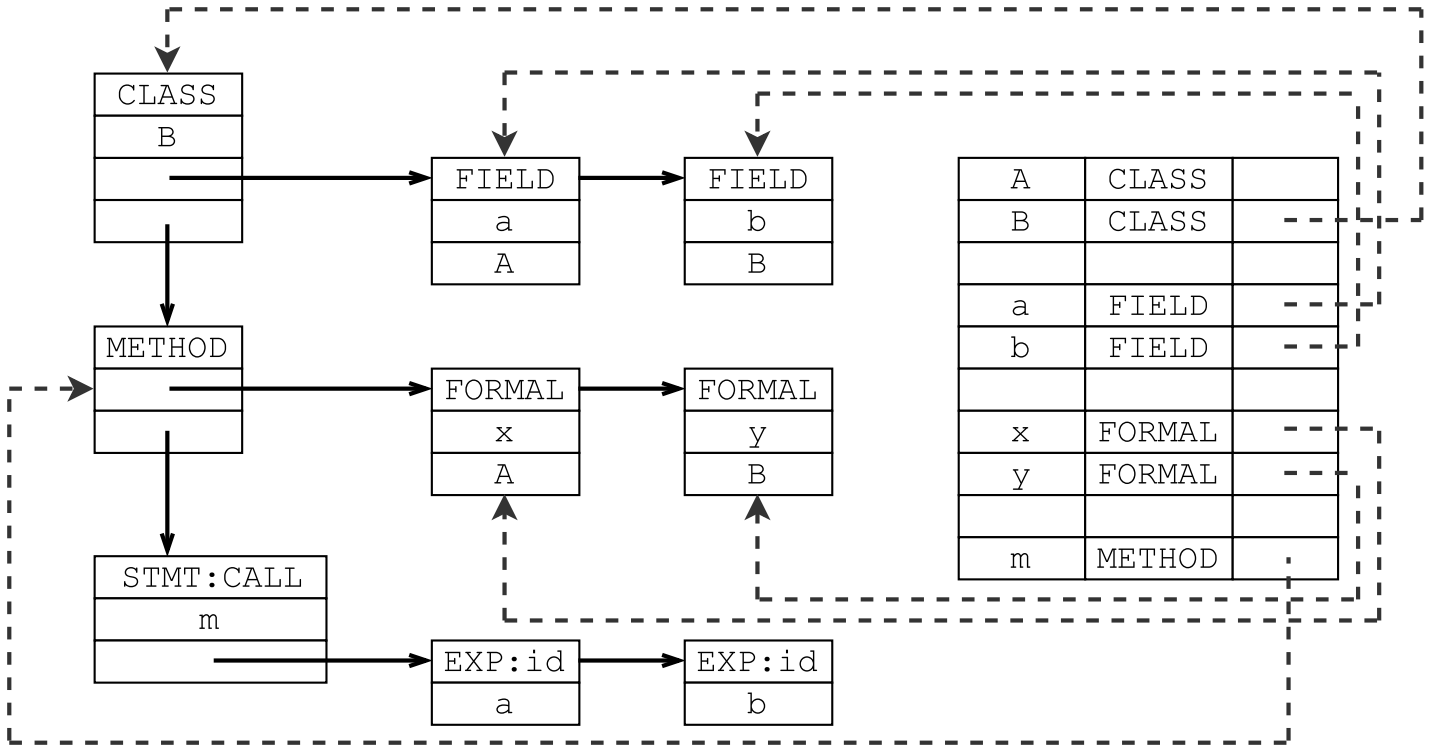
```
public class B extends A {
   protected A a;
   protected B b;
   public void m(A x, B y) {
      this.m(a,b);
   }
}
```



Weaving Symbol Tables with the AST

The symbol table contains all declarations, their kind, and a pointer to the corresponding AST node

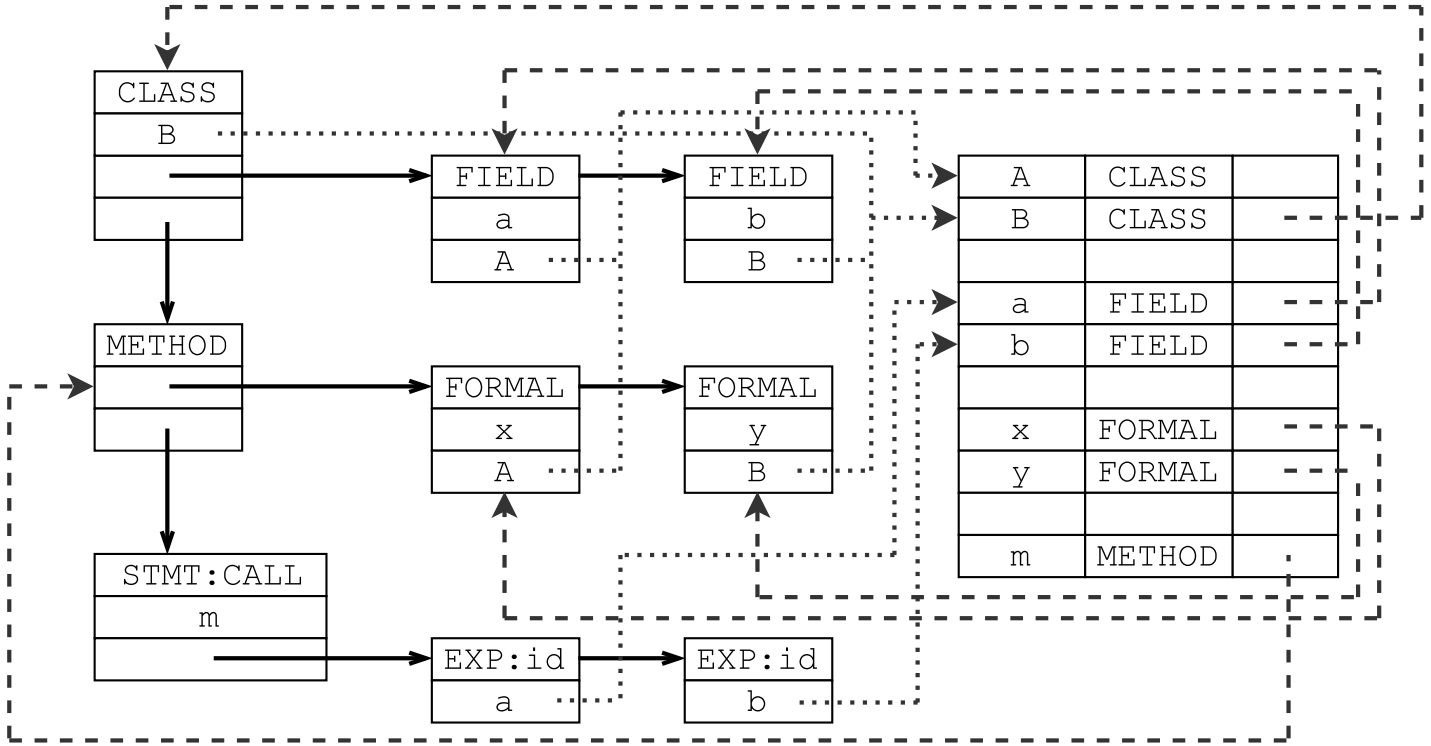
```
public class B extends A {
   protected A a;
   protected B b;
   public void m(A x, B y) {
      this.m(a,b);
   }
}
```



Weaving Symbol Tables with the AST

Lastly, we associate all uses of identifiers with their respective symbols (and declarations)

```
public class B extends A {
   protected A a;
   protected B b;
   public void m(A x, B y) {
      this.m(a,b);
   }
}
```



Constructing the Symbol Table - JOOS

Like Java, JOOS supports complex mutual recursion and out-of-order definitions. To construct the symbol table, 3 passes are used which incrementally go into further detail.

```
void symPROGRAM(PROGRAM *p) {
   classlib = initSymbolTable();
   symInterfacePROGRAM(p,classlib);
   symInterfaceTypesPROGRAM(p,classlib);
   symImplementationPROGRAM(p);
}
```

Symbol passes

- `symInterfacePROGRAM`: define classes and their interfaces;
- `symInterfaceTypesPROGRAM`: build hierarchy and analyze interface types; and
- `symImplementationPROGRAM`: define locals and analyze method bodies.

Constructing the Symbol Table - JOOS Classes Interface

The first pass to construct the symbol table collects classes and their respective members.

```
void symInterfaceCLASS(CLASS *c, SymbolTable *symbolTable) {
   SYMBOL *s = putSymbol(symbolTable, c->name, classSym);
   s->val.classS = c;
   c->localsym = initSymbolTable();
   symInterfaceFIELD(c->fields, c->localsym);
   symInterfaceCONSTRUCTOR(c->constructors, c->name, c->localsym);
   symInterfaceMETHOD(c->methods, c->localsym);
}
```

Note that no types are resolved at this time since a class is a type (and we might have recursive types)! Resolving types is thus performed in the second pass.

Constructing the Symbol Table - JOOS Methods Interface

In the first symbol table pass, we collect all method names

```
void symInterfaceMETHOD(METHOD *m, SymbolTable *symbolTable) {
   if (m == NULL) {
      return;
   }
   symInterfaceMETHOD(m->next, symbolTable);
   SYMBOL *s = putSymbol(symbolTable, m->name, methodSym);
   s->val.methodS = m;
}
```

Signature

In the second pass, we can now resolve both the return types and formal (parameter) types since the class discovery is complete.

```
void symInterfaceTypesMETHOD(METHOD *m, SymbolTable *symbolTable) {
   if (m == NULL) {
      return;
   }
   symInterfaceTypesMETHOD(m->next, symbolTable);
   symTYPE(m->returntype, symbolTable);
   symInterfaceTypesFORMAL(m->formals, symbolTable);
}
```

Constructing the Symbol Table - JOOS Implementation

The implementation pass verifies variable definitions (locals, formals, etc) are unique, and associates uses with their respective symbols (declarations).

Classes

```
void symImplementationCLASS(CLASS *c) {
   SymbolTable *symbolTable = scopeSymbolTable(classlib);
   symImplementationFIELD(c->fields, symbolTable);
   symImplementationCONSTRUCTOR(c->constructors, c, symbolTable);
   symImplementationMETHOD(c->methods, c, symbolTable);
}
```

Methods

```
void symImplementationMETHOD(METHOD *m, CLASS *this, SymbolTable *symbolTable) {
   if (m == NULL) {
      return;
   }
   symImplementationMETHOD(m->next, this, symbolTable);
   SymbolTable *m_symbolTable = scopeSymbolTable(symbolTable);
   symImplementationFORMAL(m->formals, m_symbolTable);
   symImplementationSTATEMENT(m->statements, this, m_symbolTable,
         m->modifier == staticMod);
}
```

Constructing the Symbol Table - JOOS Implementation

Statements

```
void symImplementationSTATEMENT(STATEMENT *s, CLASS *this,
          SymbolTable *symbolTable, int isStatic) {
   if (s == NULL) {
      return;
   }
   switch (s->kind) {
      case localK:
         symImplementationLOCAL(s->val.localS, symbolTable);
         break;
      case blockK:
         SymbolTable *l_symbolTable = scopeSymbolTable(symbolTable);
         symImplementationSTATEMENT(s->val.blockS.body, this, l_symbolTable,
               isStatic);
         break;

      [...]
   }
}
```

Note that block statements open a new scope for the body which allows variable scoping.

Constructing the Symbol Table - JOOS Implementation

Local declarations

Locally declared variable names are added to the symbol table and associated with their declaration

```
void symImplementationLOCAL(LOCAL *l, SymbolTable *symbolTable) {
   if (l == NULL) {
      return;
   }
   symImplementationLOCAL(l->next, symbolTable);
   symTYPE(l->type,sym);
   SYMBOL *s = putSymbol(symbolTable, l->name, localSym);
   s->val.localS = l;
}
```

Using the Symbol Table - JOOS Expressions

Recursively traverse the expression, resolving identifiers to their respective symbols

```
void symImplementationEXP(EXP *e, CLASS *this, SymbolTable *symbolTable,
          int isStatic) {
   switch (e->kind) {
      case idK:
         e->val.idE.idsym = symVar(
            e->val.idE.name, symbolTable,
            this, e->lineno, isStatic
         );
         break;
      case assignK:
         e->val.assignE.leftsym = symVar(
            e->val.assignE.left, symbolTable,
            this, e->lineno, isStatic
         );
         symImplementationEXP(
            e->val.assignE.right,
            this, symbolTable, isStatic
         );
         break;
      [...]
   }
}
```

Using the SymbolTable - Resolving Identifiers

Check first the current scope (formals, locals), then traverse the class hierarchy to check for fields.

```
SYMBOL *symVar(char *name, SymbolTable *symbolTable,
          CLASS *this, int lineno, int isStatic) {
   SYMBOL *s = getSymbol(symbolTable, name);
   if (s == NULL) {
      s = lookupHierarchy(name, this);
      if (s == NULL) {
         // throw an error: undefined
      } else if (s->kind != fieldSym)
         // throw an error: wrong kind
      }
   } else if (s->kind != fieldSym && s->kind != formalSym && s->kind != localSym)
      // throw an error: wrong kind
   }
   if (s->kind == fieldSym && isStatic)
      // throw an error: illegal static reference
      return s;
}
```

Using the SymbolTable - Lookup Hierarchy

Looking up an identifier recursively checks, scope-by-scope, for the nearest nested declaration.

```
SYMBOL *lookupHierarchy(char *name, CLASS *start) {
   if (start == NULL) return NULL;

   // Check the current class
   SYMBOL *s = getSymbol(start->localsym, name);
   if (s != NULL) return s;

   // Check the parent class
   if (start->parent == NULL) return NULL;
   return lookupHierarchy(name, start->parent);
}
```

Symbol Table Testing

A typical testing strategy for symbol tables involves an extension of the pretty printer.

- A textual representation of the symbol table is printed once for every scope area;
- These tables are then compared to a corresponding manual construction for a sufficient collection of programs;
- And every error message should be provoked by some test program.

In Java, use `toString()` to print the `HashMap`