
SYMBOL TABLES
CMPT 379 Lecture 19a

Lecture Overview

- Symbols and their information
- Lexical scoping
- Dynamic scoping
- Interface to a symbol table
- Handling nested lexical scopes
- Object and struct tables

Section 5.5

- Run-time memory organization

Symbols

- Compilers encounter many types of symbols or names:

  Variables, defined constants, procedures, labels, filenames, object classes, filenames, etc.

- Each type of symbol will have some associated information that the compiler will need to collect.
- For example, variables need an associated data type, storage class, static nesting level, procedure name, etc. Procedures need the types of arguments and return values.

Symbol Information

- This information may be incorporated into the intermediate representation or it may be derived (and re-derived) when needed.
- For instance, one could store information about a variable in the node of the AST that corresponds to the declaration of the variable.
- Storing information in the AST is simple and reliable, but it can take a long time to navigate an AST to find a variable declaration, for instance.

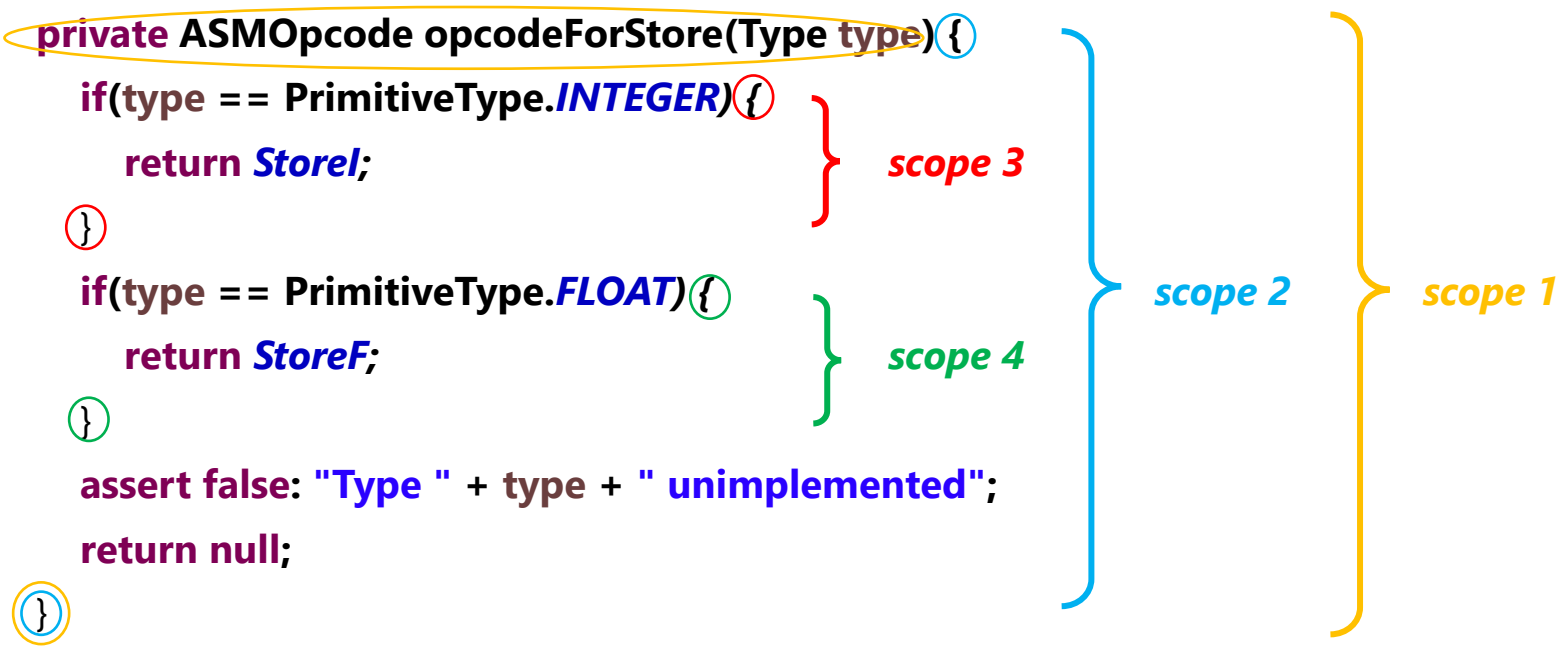
Symbol Tables

- An alternative is the symbol table: a table that collects associated information for one or more types of names.
- A symbol table may be centralized or distributed. A centralized symbol table is one table for the whole unit being compiled, and a distributed symbol table involves different tables being distributed around the AST, typically in nodes associated with scopes.
- Scopes are program areas, typically lexically defined, in which some variables are active and outside of which they are inactive.

Lexical (Static) Scoping

- Lexical scoping is the most common scoping discipline.
- The general principle is that in a given scope, each name refers to its lexically closest declaration.
- If *s* is used in the current scope, it refers to the *s* declared in the current scope, if one exists. If not, it refers to the declaration of s that occurs in the closest enclosing scope.

Lexical Scoping

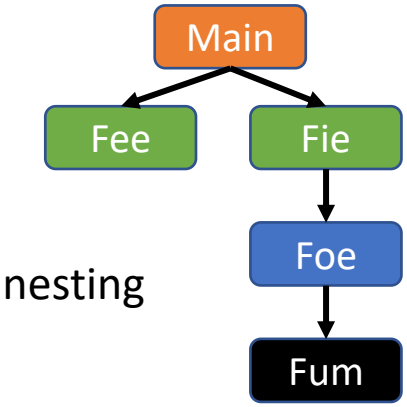


Lexical Scoping

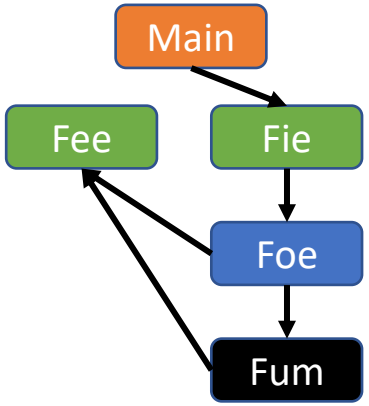
```
program Main0(input, output);
  var x1, y1, z1: integer;
  procedure Fee1;
    var x2: integer;
    begin { Fee1 }
      x2 := 1;
      y1 := x2 * 2 + 1
    end;

  procedure Fie1;
    var y2: real;
    procedure Foe2;
      var z3: real;
      procedure Fum3;
        var y4: real;
        begin { Fum3 }
          x1 := 1.25 * z3;
          Fee1;
          writeln('x = ', x1)
        end;
      begin { Foe2 }
        z3 := 1;
        Fee1;
        Fum3
      end;
    begin { Fie1 }
      Foe2;
      writeln('x = ', x1)
    end;
  begin { Main0 }
    x1 := 0;
    Fie1
  end.
```

scope	x	y	z
Main	(1, 0)	(1, 4)	(1, 8)
Fee	(2, 0)	(1, 4)	(1, 8)
Fie	(1, 0)	(2, 0)	(1, 8)
Foe	(1, 0)	(2, 0)	(3, 0)
Fum	(1, 0)	(4, 0)	(3, 0)

static coordinates



nesting



calling

Dynamic Scoping

- Dynamic scoping has scopes that start when a piece of code starts executing and ends when the code stops executing. It is hard to understand and has fallen into disfavour.

```
private void A() {
   int scale = 3;
   B();
   C();
}
private void B() {
   float alpha = 1.57;
   D();
}
private void C() {
   float alpha = 1.05;
   D();
}
private void D() {
   alpha = alpha / scale;
}
```

A starts
  B starts
    D starts
    D ends
  B ends
  C starts
    D starts
    D ends
  C ends
A ends

Symbol Table

- A centralized (or distributed) repository for information associated with symbols / names.
- Avoids searching the IR for declarations.
- Localizes information from potentially disparate portions of the program.
- Makes the information easily and efficiently available.
- Distributed tables are often referred to as the symbol table (as opposed to the symbol tables).
- Because programs may contain lots of symbols, easy and efficient expansion of the symbol table must be possible.

Hash Tables

- Most compilers use hash tables to implement their symbol tables. (CMPT 225)
- Hash tables use a hash function to map names to small numbers, which are used as an index in an array.
- Hash tables have expected constant-time lookup and install behaviour, and are efficient in practice.
- They can be quickly expanded should the need arise.

Building a Symbol Table

- The interface to the symbol table is often:
  - lookUp(name)
    returns the record associated with the *name*.
  - install(name, record)
    stores the record at the location for *name* in the table.

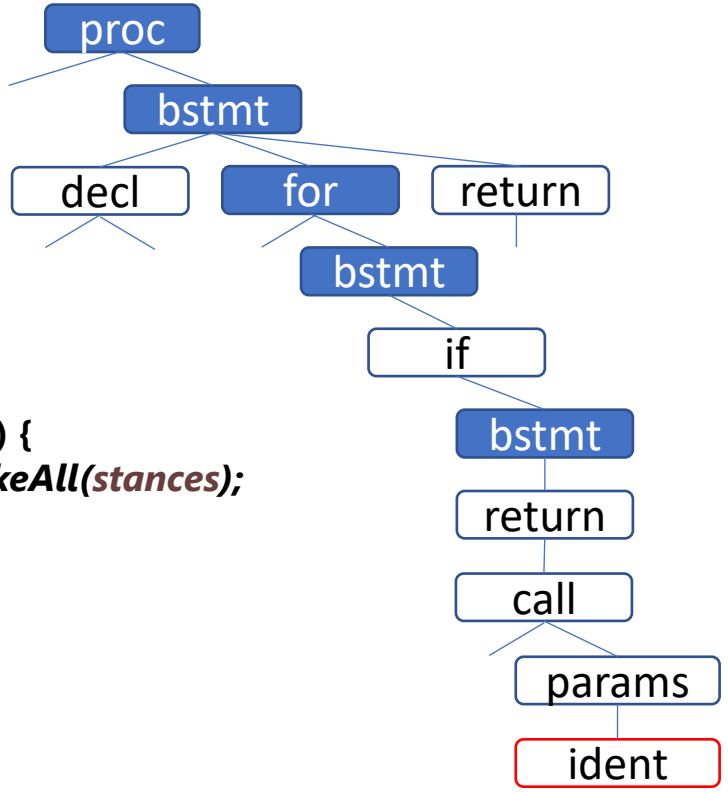
- When lookUp fails to find a record corresponding to name, it is often an error in the program being compiled, such as a "variable used before defined" error.
- In more complex systems where names can be declared after use (like functions in java), a partial record can be installed at first use and verified at declaration. Alternatively, multiple passes over the input are performed.

Handling Nested Lexical Scopes

- A new symbol table can be created for each scope as it is entered.
- This can be done in the parser or in the semantic analyzer. Conceptually, it is cleaner to have it in the semantic analyzer.
- A table may optionally have a link (pointer) to its parent scope's symbol table. This can save the compiler from having to walk the whole AST but comes at a cost of maintaining the pointers.
- Most AST nodes with children nodes have a scope associated with them, except for expression/operator nodes.
  - Operator nodes overwhelmingly occur near the node that causes a symbol-table lookup, so they must be navigated in either scheme.
  - Non-operator nodes tend to have their parent symbol table in their parent or grandparent node, so the link seems redundant.

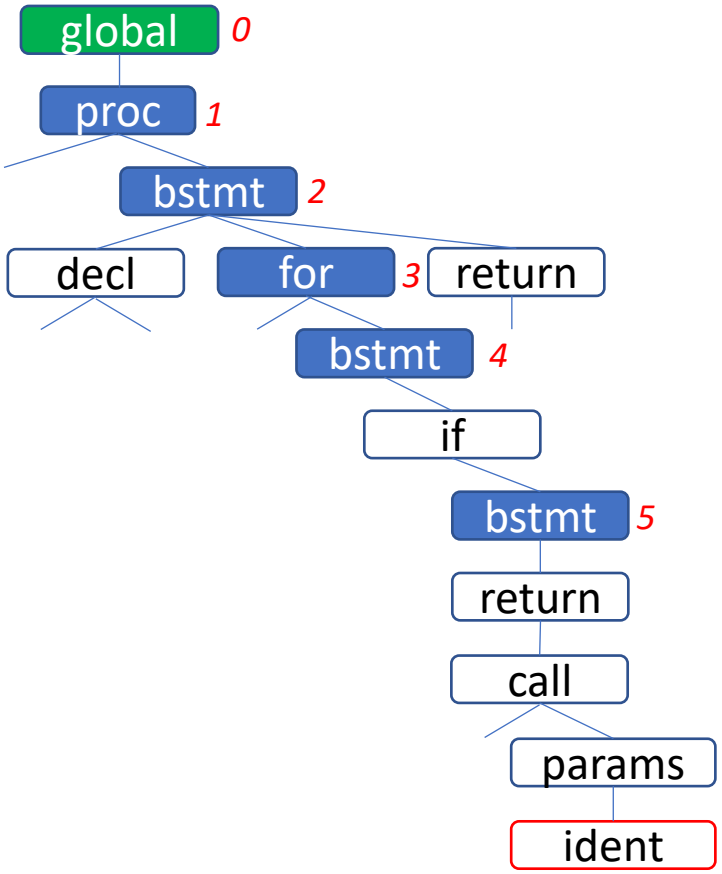
Handling Nested Lexical Scopes

```
public int motions(List<Stance> stances) {
   List<List<Motion>> allMotions = makeAll(stances);

   for(int i=0; i<= stances.size(); i++) {
      if(!allMotions.get(i).isEmpty()) {
         return allMotions.get(i);
      }
   }
   return allMotions.get(0);
}
```



Static Nesting Level

- The static nesting level of a symbol table is the number of ancestor scopes it has.
- The static coordinates of a variable are its scope's static nesting level and an offset in memory: (level, offset)

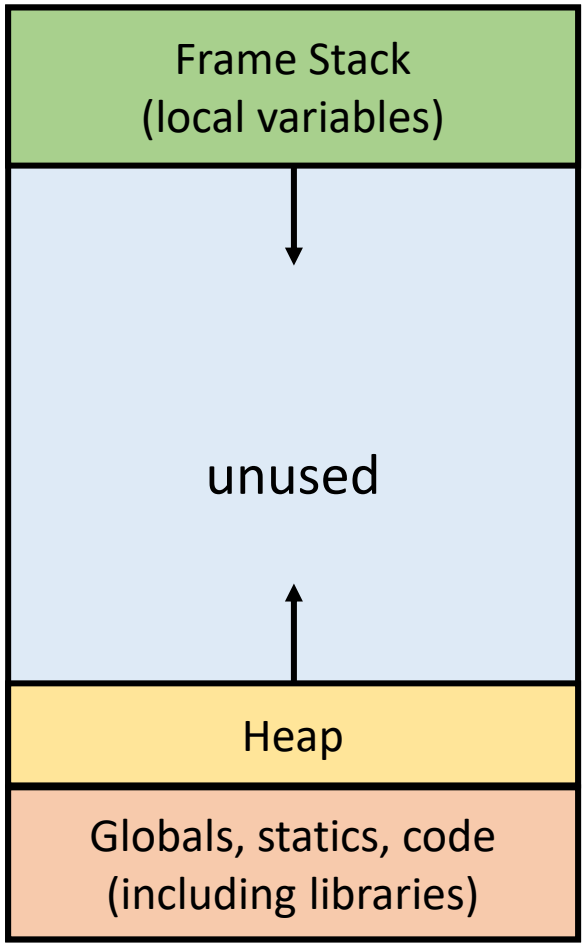


Struct/Object Tables

- For each struct or object, the compiler needs a separate symbol table of its fields and/or methods.
- Often the information that needs to be stored is very similar to a symbol table for a general scope.
- Some implementations cram it all into a central symbol table using qualified names, like Animals.Turtle.getAgeInMonths

```
Class Turtle {
    static int numTurtles = 0;
    int ageInMonths;
    float weightInKg;
    Facility location;
    Turtle(age, weight) { ... }
    void swim() { ... }
    int getAgeInMonths() {...}
    float getWeightInKg() {...}
    static int numTurtles() {...}
}
```

Typical Run-time Memory Organization



high memory

low memory
(addresses starting at 0)