COMPILERS
Semantic Analysis

hussein suleman
uct csc3003s 2008

Semantic Analysis

- The compilation process is driven by the syntactic structure of the program as discovered by the parser.
- Semantic routines:
  - interpret meaning of the program based on its syntactic structure
  - it has two functions:
    - finish analysis by deriving context-sensitive information
    - begin synthesis by generating the IR or target code
- Associated with individual productions of a context free grammar or subtrees of a syntax tree.

Context Sensitive Analysis - Why

- What context-sensitive issues can be determined?
  - Is X declared before it is used?
  - Are any names declared but not used?
  - Which declaration of X does this reference?
  - Is an expression type-consistent?
  - Do the dimensions of a reference match the declaration?
  - Where can X be stored? (heap, stack, ,,,)
  - Does *p reference the result of a malloc()?
  - Is X defined before it is used?
  - Is an array reference in bounds?
  - Does function foo(…) produce a constant value?

Context Sensitive Analysis - How

- How to check symbols and their semantics at various points in the program?
  - Process program linearly (roughly, in-order tree traversal).
  - Maintain a list of currently defined symbols and what they mean as the program is processed – this is called a Symbol Table.

Symbol Tables

- Associate lexical names (symbols) with their attributes.
- Can contain:
  - variable names
  - defined constants
  - procedure/function/method names
  - literal constants and strings
  - source text labels
  - compiler-generated temporaries
  - subtables for structure layouts (types) (field offsets and lengths)

Symbol Table Attributes

- The following attributes would be kept in a symbol table:
  - textual name
  - data type
  - dimension information (for aggregates)
  - declaring procedure
  - lexical level of declaration
  - storage class (base address)
  - offset in storage
  - if record, pointer to structure table
  - if parameter, by-reference or by-value?
  - can it be aliased? to what other names?
  - number and type of arguments to functions/methods

Binding

- As the declarations of types, variables, and functions are processed, identifiers are bound to “meanings” in the symbol table.
- A symbol table is a set of bindings.
- … But this binding is not static – it changes over the course of the program.

Scope

- An identifier has scope when it is visible and can be referenced.
- An out-of-scope identifier cannot be referenced.
- Identifiers in open scopes may override older/outer scopes temporarily.
- 2 Types of scope:
  - Static scope is when visibility is due to the lexical nesting of subprograms/blocks.
  - Dynamic scope is when visibility is due to the call sequence of subprograms.

Basic Static Scope

- Usually, a name begins life where it is declared and ends at the end of its block.

```
void foo()  {
        int k;
 …….
}
```

Why Scope?

- Scope is not necessary.
  - Languages such as assembler have exactly one scope: the whole program.
- Modern programming languages have more than one scope.
  - Information hiding and modularity.
- Goal of any language is to make the programmer’s job simpler.
  - One way: keep things isolated.
  - Make each thing only affect a limited area.
  - Make it hard to break something far away.

Changing Scope

- Identifiers come into scope at the beginning of a subprogram/block and go out of scope at the end.
- Example (in C++):

```
void testfunc ()
{
    int a; // a enters scope;
    for ( int b=1; b<10; b++ ) // b in scope for for
    {
        int c; // c enters scope
        …
    } // b,c leave scope
    …
} // a leaves scope
```

Static Scope

- Consider the Pascal program (which uses static scoping):

```
program test;
var a : integer;

   procedure proc1;
   var b : integer;
   begin
   end;

   procedure proc2;
   var a, c : integer;
   begin
      proc1;
   end;

begin
   proc2;
end.
```

in scope: b (from proc1), a (from test)

in scope: a, c (from proc2)

in scope: a (from test)

Dynamic Scope

- Consider the Pascal-like code (assume dynamic scoping):

```
program test;
var a : integer;

   procedure proc1;
   var b : integer;
   begin
   end;            <-- in scope: b (from proc1) a, c (from proc2)

   procedure proc2;
   var a, c : integer;
   begin
      proc1;       <-- in scope: a, c (from proc2)
   end;

begin
   proc2;          <-- in scope: a (from test)
end.
```

Static vs. Dynamic Scope

- Dynamic scope makes it easier to access variables with lifetime, but it is difficult to understand the semantics of code outside the context of execution.
- Static scope is more restrictive – therefore easier to read – but may force the use of more subprogram parameters or global identifiers to enable visibility when required.

Scope in a symbol table

- Most modern programming languages have nested static scope.
  - The symbol table must reflect this.
- What additional information can reflect nested scope?
  - A name query must access the most recent declaration, from the current scope or some enclosing scope.
  - Innermost scope overrides declarations from outer scopes.

Scope and Symbol Table Operations

- What symbol table operations do we need?
  - void put (Symbol key, Object value)
    - binds key to value
  - Object get(Symbol key)
    - returns value bound to key
  - void beginScope()
    - remembers current state of table
  - void endScope()
    - restores table to state at most recent scope that has not been ended

Attribute Information

- Attributes are internal representations of declarations.
- Symbol table associates names with attributes.
- Names may have different attributes depending on their meaning:
  - variables: type, procedure level, frame offset
  - types: type descriptor, data size/alignment
  - constants: type, value
  - methods: formals (names/types), result type, block information (local decls.), frame size

Symbol Table Implementation

- Implemented as a collection of dictionaries in which each symbol is placed.

- Many different possible data structures:
  - linked list
  - hash table
  - binary tree

Symbol Table Lookup

- Basic operation is to find the entry for a given symbol.
- Each symbol table may have a pointer to its parent scope.
- Lookup: if symbol in current table, return it, otherwise look in parent.
- Hash tables and binary trees can be used more efficiently.

Types of Implementation

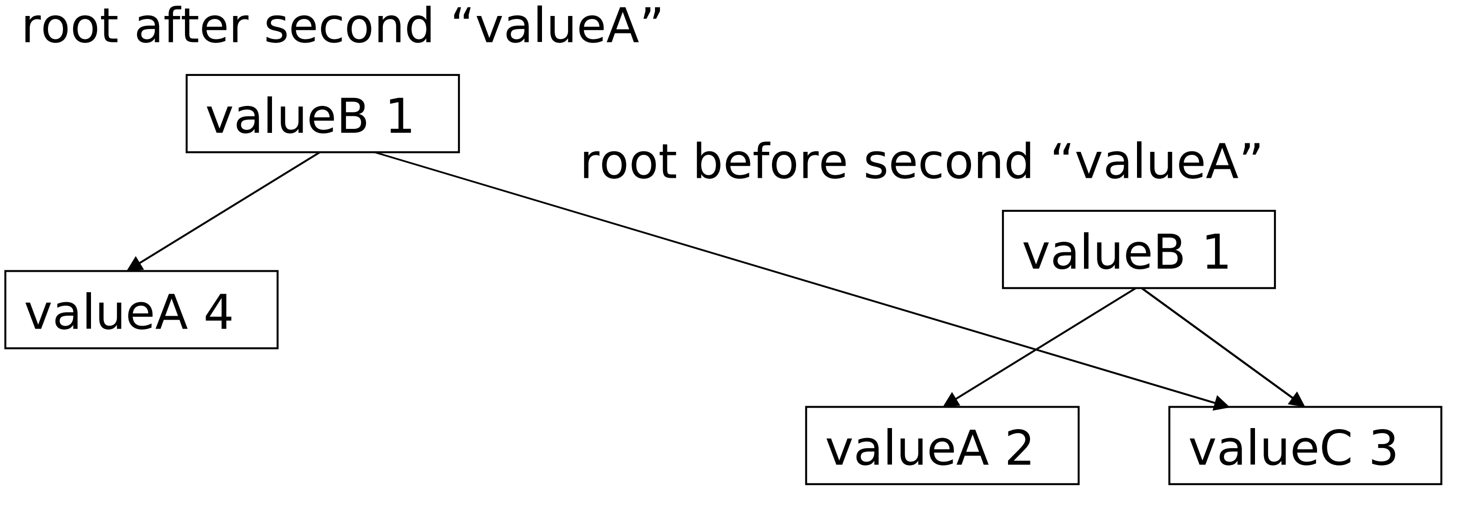
- Imperative
  - Auxiliary data structures are modified as the analysis progresses, always reflecting only the current state.
- Functional
  - Auxiliary data structures are maintained intact as the analysis progresses, with new versions created when needed – thus previous and current states are all available at any time.

Hash Table

- beginScope/put
  - Imperative - Chain new entries to beginning of table, thus overriding older entries.
  - Functional - Create copy of hash table array.
- endScope
  - Imperative – Remove entries from head of each linked list.
    - Each entry can point to the next one that should be removed.
  - Functional – Dispose of array.

Binary Tree

- put
  - Functional – Insert new entries into a new subtree, duplicating nodes up to the root.
- endScope
  - Functional – Delete all nodes in new subtree.



Symbols vs. Names

- Names are the textual entities found in the source code.
- Symbols are entities assigned to each name for more efficient processing during compilation.
- Example:
  - Name: valueA
    - Symbol: V001
  - Name: valueB
    - Symbol: V002
- Remember perfect hash functions?

Type Checking

- Static semantics should be checked after/as the symbol table is populated.
  - Is every name defined before it is used?
  - Does the type of each subexpression conform to what is expected?
  - Are the types on either side of an assignment compatible?
- The tree can be walked/visited to perform these checks.
  - May need multiple passes – so retain symbol table across passes.

Type Equivalence

- Two approaches:
  - Name equivalence: each type name is a distinct type.
  - Structural equivalence: two types are equivalent iff. they have the same structure (after substituting type expressions for type names).
- Example (structural):

```
typedef int bignumber;
int c;
bignumber b =c;
```

Error Handling

- If errors are detected, correct program representation and continue analysis to detect other errors.
- Example:

```
int a, b;
String c;
c = a;
b = a;
```