COMPILERS
Symbol Tables

hussein suleman
uct csc3003s 2007

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

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.

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.

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.

Symbol Table Implementation

- Implemented as a collection of dictionaries in which each symbol is placed.
- Two operations:
  - insert adds a binding to a table and
  - lookup locates the binding for a name.
- 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.

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;
```