CSE 504: Compiler Design

Intermediate Representations
Symbol Table

Pradipta De
pradipta.de@sunykorea.ac.kr
Current Topic

- Intermediate Representations
  - Graphical IRs
  - Linear IRs
  - Symbol Table
Information in a Program

• Compiler manages different forms of information, occurring as names in a program:
  – Variables: data type, storage class, name, lexical scope, base address and offset
  – Array: number of dimensions, upper and lower bounds
  – Structures: list of fields
  – Functions or procedures: number of parameters and their type, types of return values, what variables a procedure can reference (in OOP)
How to manage the information

• Record the information in the IR
  – Common approach for compiler design
  – Record information as attributes of nodes
    • Variables in an AST are annotated
  – Disadvantage: navigate AST to find the information

• Derive the information every time it is required
  – If IR is on disk, then recompute information may be faster than disk access
Symbol Table

• A central repository of facts with efficient access mechanisms to it
  – Localizes information derived from distant parts of the code
  – Simplifies design and implementation of code that accesses the data during compilation
  – Avoids the expense of searching a tree
Implementation: Symbol Table

• Hash table based implementation
  – Names, along with any information derived by the compiler, are mapped into the hash table
  – Constant time expected case lookup

• Open Hashing
  – On collision, use a linear list of records

• Open Addressing
  – Avoid collision by recomputing a new slot if the slot is not empty
Symbol Table Interface

- **Lookup**(name)
  - return the record stored in the table at h(name)
  - No record found ➔ error
- **Insert**(name, record)
  - Stores the information in record in the table at h(name)

- Suitable for Adhoc Syntax Directed Translation
  - While processing declaration, build up a set of attributes for each variable
  - On a production that declares a variable, insert name and attributes into the symbol table
  - When variable name occurs outside declaration syntax, use lookup for the name
    - Lookup fails on undeclared names

- For languages with a single name space this is implementation is sufficient
  - Typically languages have nested scopes
Nested Scopes

• Programming languages allow names to be declared at multiple levels
• Each level determines a scope
  – Scope is region in program’s text where the name is valid
• When scopes are nested, then a name must be associated to the correct scope
  – Compiler uses a lexically scoped symbol table
  – Map each variable name to its specific declaration (called name resolution)
Example: Nested Scope

```c
static int w;   /* level 0 */
int x;

void example(int a, int b) {
    int c;   /* level 1 */
    {
        int b, z;   /* level 2a */
        ...
    }
    {
        int a, x;   /* level 2b */
        ...
        {
            int c, x;   /* level 3 */
            b = a + b + c + w;
        }
    }
}
```

<table>
<thead>
<tr>
<th>Level</th>
<th>Names</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>w, x, example</td>
</tr>
<tr>
<td>1</td>
<td>a, b, c</td>
</tr>
<tr>
<td>2a</td>
<td>b, z</td>
</tr>
<tr>
<td>2b</td>
<td>a, x</td>
</tr>
<tr>
<td>3</td>
<td>c, x</td>
</tr>
</tbody>
</table>
Managing nested scopes

• Create a new symbol table on entering a new lexical scope
  – Creates a sheaf of tables linked together in the lexical nesting order
  – Insert operates on the current symbol table
  – On lookup, start by checking the current symbol table, and move up the nesting order

Two additional interfaces for symbol table
  • InitializeScope()
    • increment current level
    • create new symbol table and link to enclosing levels table
  • FinalizeScope()
    • Change current level pointer, and decrement current level
Level 2 is entered twice

\[ b = a + b + c + w \]
Nested Symbol Table: Implementation Choices

Lexical Scoping with Open Hashing

\[
\begin{align*}
&\uparrow{w,0} \langle x,0 \rangle \langle \text{example},0 \rangle \uparrow \langle a,1 \rangle \langle b,1 \rangle \langle c,1 \rangle \\
&\uparrow \langle b,2 \rangle \langle z,2 \rangle \downarrow \uparrow \langle a,2 \rangle \langle x,2 \rangle \uparrow \langle c,3 \rangle, \langle x,3 \rangle \downarrow \downarrow \downarrow
\end{align*}
\]

\[
\begin{align*}
&\text{static int } w; \quad \text{/**< level 0 */}
&\text{int } x;
&\text{void example(int a, int b) [}
&\text{\quad \text{int } c; \quad \text{/**< level 1 */}
&\text{\quad \{}
&\text{\quad \text{int } b, z; \quad \text{/**< level 2a */}
&\text{\quad \quad ...
&\text{\quad \}
&\text{\quad \{}
&\text{\quad \text{int } a, x; \quad \text{/**< level 2b */}
&\text{\quad \quad ...
&\text{\quad \}\}
&\text{\quad \text{int } c, x; \quad \text{/**< level 3 */}
&\text{\quad \quad b = a + b + c + w;}
&\text{\quad \}}
&\text{\}}
&\text{]}
\end{align*}
\]
Nested Symbol Table: Implementation Choices

- A two dimensional hash table implementation
  - Uses two separate hash tables
    - One table hashes the symbol name
    - Another is the hash table for field names or symbol attributes (like type, offset, length)
Scoping in Object Oriented Languages

- Scoping is guided by the structure of the data, in addition to structure of code
  - Record inheritance hierarchy
  - Java: external classes that are referenced in the code

- Simple implementation
  - Attach a symbol table to each class with two nesting hierarchies
    - One for lexical scoping within individual methods
    - Other for inheritance hierarchy for each class

- To resolve a name “fee” in method \( m \) in Class \( C \),
  - Check lexically scoped symbol table for \( m \)
  - If not found, then search scope for various classes in the inheritance hierarchy, starting with \( C \)
  - If this fails, then check global symbol table for a class