Naming

Comp 412

Chapters 4, 5, 6 & 7 in EaC2e
Code Generation

In a modern, multi-IR compiler, code generation happens several times.

SDT differs from later code generation in that it must construct models of the source code’s name space. Later passes assume the models exist.

- The ST in the picture represents a collection of tables
  - Records names, values, constants, & locations, both explicit & implicit
  - Information must be namable & efficiently accessible
- The facts are derived from the source code, both syntax & values
- The derived knowledge is critical to the “meaning” of the source code

This lecture focuses on a compiler builds up that knowledge base.
Names, Types, Dimensions, and Scopes

Our examples, so far, have assumed a single type and known locations

• The compiler must handle the more general situation
  – Variables and values of multiple types
  – Multiple locations & classes for storage (local, static, global, heap, ...)

• For each value, the compiler needs its type, class, & location
  – This information comes from declarations, or from inference

```
Declaration → INT IntList
              | CHAR CharList
              | ... other types ...

IntList → IntList Spec
          | Spec

CharList → CharList Spec
          | Spec

Spec → NAME
```

Typical Declaration Syntax
If the language does not have a *declarations before executables* rule, then the compiler must derive its knowledge incrementally. It can (1) use a higher level of abstraction in the initial IR to avoid the need for specific addresses; or (2) make two passes over the code (e.g., build an AST and re-traverse it).

### Handling Type Specifications

<table>
<thead>
<tr>
<th>Production</th>
<th>SDT Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>Declaration → INT IntList</td>
<td>If needed, initialize symbol table</td>
</tr>
<tr>
<td></td>
<td>CHAR CharList</td>
</tr>
<tr>
<td></td>
<td>... other types ...</td>
</tr>
<tr>
<td>IntList → IntList Spec</td>
<td>Set Spec’s type to INT</td>
</tr>
<tr>
<td></td>
<td>Spec</td>
</tr>
<tr>
<td>CharList → CharList Spec</td>
<td>Set Spec’s type to CHAR</td>
</tr>
<tr>
<td></td>
<td>Spec</td>
</tr>
<tr>
<td>Spec → NAME</td>
<td>Create symbol table entry for NAME</td>
</tr>
</tbody>
</table>

**Grammar refactored to facilitate SDT** *(see lecture 20)*
Names, Types, Dimensions, and Scopes

Where does the compiler store the type & dimension information

The compiler creates a symbol table

- Entry for every name ("symbol")
- Associated attributes for each symbol
  - Lexeme, type, address, length, storage class (local, static, global, constant, ...)
- Compiler can use y’s symbol table index as a short name for y in the IR

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Addr</th>
<th>Len</th>
<th>Storage Class</th>
</tr>
</thead>
<tbody>
<tr>
<td>y</td>
<td>int</td>
<td>8</td>
<td>4</td>
<td>local</td>
</tr>
<tr>
<td>w</td>
<td>float</td>
<td>@_w</td>
<td>4</td>
<td>static</td>
</tr>
<tr>
<td>x</td>
<td>char</td>
<td>12</td>
<td>0</td>
<td>static</td>
</tr>
<tr>
<td>z</td>
<td>double</td>
<td>0</td>
<td>8</td>
<td>local</td>
</tr>
</tbody>
</table>

Conceptual image of a symbol table
Names, Types, Dimensions, and Scopes

Of course, neither the rules nor the table are that simple

PLs introduce rules to determine the scope associated with each name
• Rules arbitrate clashes of names and declarations (conflicts)
• For each name, determine which declaration defines its properties
• Large programs would be impractical if each name had to be unique

Common case: lexical scoping
• Each “scope” creates a new name space
• Well defined rules for inheriting & obscuring names from outer scopes
• Java, C, C++, and many others follow Algol’s scope rules
  – They differ in which scopes can be nested and which cannot
• The compiler must, of course, keep track of the details
  – And, figure them out during SDT

In lexical scoping, scopes are searched in the order in which they are encountered.
Lexical Scoping in an Algol-like Language

Each scope (procedure or block) creates its own name space

- Any name (almost) can be declared locally
- Local names obscure identical non-local names
- Local names cannot be seen outside the scope
- Scopes are searched in the order that they were encountered

Examples

- Algol and Pascal are the classic examples
  - Nested procedures, often with deep nesting
- Fortran had local, static, and named global scopes
- C has global, static, local, and block scopes (actually Fortran-like)
  - Blocks can be nested, procedures cannot
- Scheme has global, procedure-wide, and nested scopes (let)
  - Procedure scope (typically) contains formal parameters
Lexical Scoping in an Algol-like Language

Each scope (procedure or block) creates its own name space

• Any name (almost) can be declared locally
• Local names obscure identical non-local names
• Local names cannot be seen outside the scope
• Scopes are searched in the order that they were encountered

Python is a bit like C, except with different intuitions

• At any point in the code, it has several scopes: builtin, module, and local to a function — functions can be nested
• Assignment to undeclared name creates a symbol in the current scope
• Use before definition in procedure declares a name as global
• Global declaration is needed to assign a global from inside a procedure

Creates a strange set of context-sensitive implicit declarations that defy the intuitions used in programming languages since Algol 60

The intuitions from Python seem to confuse students when they are asked questions about lexical scoping. Remember that Python defies the rules, rather than defining them.
How Does The Compiler Model Scopes?

The compiler needs both compile-time & runtime mechanisms to manage scopes

The compile-time mechanism is a scoped symbol table

• Table tracks visibility and maps each name to a static coordinate
  – A static coordinate is a name that distinguishes $x$ across multiple scopes
  – A $<\text{level, offset}>$ pair where $\text{level}$ specifies the scope & $\text{offset}$ the location with storage for that scope
  – Code references variables by their static coordinates
• Table must allow cheap & easy addition and deletion of scopes
  – ... and a way to preserve the table for a scope (debugging)

The runtime mechanism maps a static coordinate to a runtime address

• Compiler must generate code (at compile time) to maintain the map
• Compiler must generate code that will (at runtime) use the map

Lots of important vocabulary on this slide.
Critical Point on Lexical Scopes

Lexical scopes nest properly
• At any point, each scope has at most one surrounding scope
• At red arrow, the code can see:
  – Variables from r
  – Variables from q
  – Variables from p
• It cannot see variables from s

A static coordinate specifies a unique scope
• <level,offset> maps to a data area and an offset inside it
• Assign lowest level to globals
  – May need label for global base address (special case)
How Does The Compiler Model Scopes?

At Compile Time:

The PL has some syntactic constructs that start a new scope
• New procedure, new block, class declaration, ...
• Compiler writer adjusts the grammar to have a reduction at the start and end of each block

On block entry:
• Create a new table for the scope & link it to surrounding scopes

On block exit
• Disconnect the table for the old scope & preserve or discard it

We will discuss the runtime support for scopes in a couple of weeks

Simple Interface to Table

insert(ST, name, level) : enters symbol into level table
lookup(ST, name) : finds name & returns its record & level
delete(ST, level) : makes the level table inactive
insert & lookup should be O(1)
Lexically-Scoped Symbol Tables

High-level idea
- Create a new table for each scope
- Chain them together for lookup

“Sheaf of tables” implementation
- `insert()` may need to create a new table. It always inserts at current level.
- `lookup()` walks chain of tables & returns first occurrence of name
- `delete()` throws away level \( p \) table if it is top table in the chain

If the compiler must preserve the table (for, say, the debugger), this picture is even more practical.

Individual tables are hash tables.
Arguably, \( O(1) \)

This high-level idea can be implemented as shown, or it can be implemented in more space-efficient (albeit complex) ways. (See EaC2e, § B.4)
Lexically-Scoped Symbol Tables

*In all discussions about the cost of support for lexical scopes, the costs will depend on the distribution of local versus non-local accesses.*

**“Sheaf of tables” implementation**

- *insert()* may need to create a new table. It always inserts at current level.
- *lookup()* walks chain of tables & returns first occurrence of name
- *delete()* throws away level $p$ table if it is top table in the chain

If the compiler must preserve the table *(for, say, the debugger)*, this picture is even more practical.

Individual tables are hash tables.

Arguably, $O(1)$

This high-level idea can be implemented as shown, or it can be implemented in more space-efficient (albeit complex) ways. (See EaC2e, § B.4)
What about Object-Oriented Languages?

**What is an OOL?**

- A language that supports “object-oriented programming”

**How does an OOL differ from an ALL?**

- **Data-centric** name scopes for values & functions
- **Dynamic resolution** of names to their implementations

**What information do we need to an OOL?**

- Need to define what we mean by an **OOL**
- Term is almost meaningless today —
  - Smalltalk to C++ to Java to Python
  - Huge differences in features & their support
- We will focus on name resolution & addressability in **OOLs**
  - Respectively, **compile-time** and **runtime** issues
- Differences from an **ALL** lie in naming and addressability

---

**OOL** \(\equiv\) **Object-Oriented Language**  
**ALL** \(\equiv\) **Algol-Like Language**
What Are The Issues?

**In an ALL, the compiler needs**

- Compile-time mechanism for name resolution
- Runtime mechanism to compute an address from a name

Compiler must emit code that builds & maintains the runtime structures for addressability

**In an OOL, the compiler needs**

- Compile-time mechanism for name resolution
- Runtime mechanism to compute an address from a name

Compiler must emit code that builds & maintains the runtime structures for addressability

*As stated earlier, this lecture focuses on the compile-time mechanisms. Runtime support for ALL & OOL name spaces will appear later.*
First, we need some common terminology.

**What is an Object?**

*An object is an abstract data type that encapsulates data, operations and internal state behind a simple, consistent interface.*

**The Concept:**

![Diagram](https://via.placeholder.com/150)

**Elaborating the concepts:**

- Each object has internal state
  - Data members are static (*lifetime of object*)
  - External access is through code members
- Each object has a set of associated procedures, or methods
  - Some methods are public, others are private
  - Locating a procedure by name is more complex than in an ALL
- Complex behavior arises from objects’ internal states

These ideas go back to Simula 67, & data-abstraction languages such as CLU & Alphard
OOL Name Spaces

What is the shape of a typical OOL’s name space?

• Local storage in objects (both public & private)
  • Storage defined in methods (they are procedures)
    – Local values inside a method
    – Static values with lifetimes beyond methods
• Methods shared among multiple objects
• Global name space for global objects and (some?) code

Classes

• Objects with the same state are grouped into a class
  – Same code, same data, same naming environment
  – Class members are static & shared among instances of the class
• Allows abstraction-oriented, or data-centric, naming
• Intended to foster code reuse in both source & implementation

The strength of a language’s object model varies wildly. For example, Python’s object model is a thin tissue over the underlying ALL.
The Java Name Space

Class Point {
    public int x, y;
    public void draw();
}

Class ColorPoint extends Point {
    Color c;
    public void draw() {...} // override (hide) Point’s draw
    public void test() { y = x; draw(); } // local code
}

Class C {
    int x, y;
    public void m() {
        Point cp = new ColorPoint(); // uses ColorPoint, and, by inheritance
        y = cp.x;
        cp.draw(); // the definitions from Point
    }
}

• Point is called a **base class** and ColorPoint a **derived class**
• Point is ColorPoint’s **superclass** and ColorPoint is a **subclass** of Point

The classic example for discussing inheritance. We will use & extend it.
The Java Name Space

Class Point {
    public int x, y;
    public void draw();
}

Class ColorPoint extends Point {
    Color c;
    public void draw() {...}
    public void test() { y = x; draw(); }
}

Class C { // uses ColorPoint
    int x, y;
    public void m()
    {
        Point cp = new ColorPoint();
        y = cp.x;
        cp.draw();
    }
}

In the example, Point cp is assigned a new ColorPoint, so it reference’s ColorPoint’s draw() code.

Point and ColorPoint have different runtime representations. The compiler needs:
• Names, types, and locations for members — Location is an offset from cp
• A mechanism to ensure that a call to draw() reaches the right instance
The Java Name Space

Class Point {
    public int x, y;
    public void draw();
}

Class ColorPoint extends Point {
    Color c;
    public void draw() {...}
    public void test() { y = x; draw(); }
}

Class C { // uses ColorPoint
    int x, y;
    public void m()
    {
        Point cp = new ColorPoint();
        y = cp.x;
        cp.draw();
    }
}

• Every object has an object record, including each class
• Items shared amongst a class are recorded in the class’ object record