Naming

Comp 412

Copyright 2017, Keith D. Cooper & Linda Torczon, all rights reserved.

Students enrolled in Comp 412 at Rice University have explicit permission to make copies of these materials for their personal use.

Faculty from other educational institutions may use these materials for nonprofit educational purposes, provided this copyright notice is preserved.

Chapters 4, 5, 6 & 7 in EaC2e
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

\[
\begin{align*}
\text{Declaration} & \rightarrow \text{TypeSpecifier} \ NameList \\
\text{TypeSpecifier} & \rightarrow \text{int} \mid \text{char} \mid \text{float} \mid \text{....} \\
\text{NameList} & \rightarrow \text{NameList} \ NameSpec \\
& \mid \ NameSpec \\
\text{NameSpec} & \rightarrow \text{name} \\
& \mid \text{name} \ DimensionSpec
\end{align*}
\]

Typical Declaration Syntax
Names, Types, Dimensions, and Scopes

The compiler must gather information about names during SDT

Handling Type Specifications

Production

- Declaration → TypeSpecifier NameList
- TypeSpecifier → int | char | float | ....
- NameList → NameList NameSpec | NameSpec
- NameSpec → name

 name DimensionSpec

SDT Action

- Set current declaration’s type on reduction (maybe in a global?)
- Set name’s type to type of current declaration
- Set name’s dimension info. from info. derived in “DimensionSpec”

Typical Declaration Syntax
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

**Conceptual image of a symbol table**

<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>

h(y)
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 less intuitive
- At any point in the code, it has several scopes: built-in, 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 <level, offset> pair where level specifies the scope & 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 1 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

• With offset, a unique location
• Assign lowest level to globals
  – May need label for global base address (special case)

```c
procedure p {
    int a, b, c
}

procedure q {
    int v, b, x, w
    procedure r {
        int x, y, z
        ....
    }
    procedure s {
        int x, a, v
        ...
    }
    .... r ... s
}
... q ...
```
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)
In all discussions about the cost of support for lexical scopes, the costs will depend on the distribution of local versus non-local access.

“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 ≡ Object-Oriented Language
ALL ≡ 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.
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:

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

In some OOLs, everything is an object.
In others, variables co-exist with objects.

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 fundamental question

Name Resolution in an OOL

What names can an executing code member access?

• Names defined by the code member
  – *And its surrounding lexical context*
• The receiving object’s data members
  – Smalltalk terminology: *instance variables*
• The code & data members of the class that defines it
  – *And its context from inheritance*
  – Smalltalk terminology: *class variables and methods*
• Any object defined in the global name space

The method might need the address for any or all of these objects

An OOL resembles an ALL, with a different name space

• Scoping is relative to hierarchy *in the code* of an ALL
• Scoping is relative to *hierarchy in both the code & the data* of an OLL
Concrete Example: The Java Name Space

**Code within a method** \( M \) **for object** \( O \) **of class** \( C \) **can see:**

- Local variables declared within \( M \)  
  *lexical scoping*
- All instance variables of \( O \) & class variables of \( C \)
- All public and protected variables of any **superclass** of \( C \)
- Classes defined in the same package as \( C \) or in any explicitly imported package
  - public class variables and public instance variables of imported classes
  - package class and instance variables in the package containing \( C \)
- Class declarations can be nested!
  - Member declarations hide outer class declarations of the same name
  - Accessibility options: public, private, protected, package

Both lexical nesting & class hierarchy at play

Superclass is an ancestor in the inheritance hierarchy

COMP 412, Fall 2016
To compile method M of object O in class C, the compiler needs:

1. Lexically scoped symbol table for the current block and its surrounding scopes
   - Just like ALL — inner declarations hide outer declarations

2. Chain of symbol tables for inheritance
   - Class C and all of its superclasses
   - Need to find methods and instance variables in any superclass

3. Symbol tables for all global classes (package scope)
   - Entries for all members with visibility
   - Need to construct symbol tables for imported packages and link them into the structure in appropriate places

Three sets of tables are needed for name resolution. In an ALL, we can combine 1 & 3 for a single unified set of tables. In Java, we need to split them so that the compiler can check the inheritance hierarchy between the lexical hierarchy & the global name space.
Compile-time Structures for OOLs — Java

**Conceptually**

- **Lexical Hierarchy**
- **Class Hierarchy**
- **Global Scope**

**Search Order:** lexical, class, global

Again, the “sheaf of tables” implementation simplifies the conceptual picture. Static coordinate needs a hierarchy field, as well.
Compile-time Structures for OOLs — Java

To find the address for a reference to $x$ in method $M$ for an object $O$ of class $C$, the compiler must:

• For an unqualified use (i.e., $x$):
  – Search the symbol table for the method’s lexical hierarchy
  – Search the symbol tables for the receiver’s class hierarchy
  – Search global symbol table (current package and imported)
  – For each hit, check visibility attribute of $x$

• For a qualified use (i.e.: $Q.x$):
  – Find $Q$ by the method above
  – Search from $Q$ for $x$
    → Must be a class or instance variable of $Q$ or some class it extends
  – Check visibility attribute of $x$

Compile-time cost increases by a small constant factor.
What About Storage Layout?

Where does all of this stuff go? And how does it get there?

The compiler must classify all code & data so that it can assign storage

- **Scopes**: local, global, subject to some set of lexical scoping constraints
- **Lifetimes**: entire execution, execution of a procedure, or indeterminate

Given these classifications & the state of the naming model, the compiler can assign data to specific data areas

- Each procedure has a local data area
  - Local data from a scope smaller than a procedure usually goes here
- Declarations, procedures, files, & modules may have a static data area
  - Depends on the specific declarations in the code
- A program may have one or more global data areas & constant pools
- Each object has an object record
  - A class, being an object, has an object record of class “class”
When Does the Compiler Assign Storage?

The compiler must assign storage before using addresses in the IR

- Code contains implicit information about storage
- Compiler must make its decisions after parsing declarations
  - Can envision batch schemes or incremental schemes
  - Either way, the compiler assigns addresses

Typical Grammatical Organization for Procedures

```
Procedure     →   Header { DeclList StmtList }
DeclList      →   DeclList Declaration
                |   Declaration
StmtList      →   StmtList Statement
                |   Statement
```

Compiler wants to perform storage assignment at this point in the parse
When Does the Compiler Assign Storage?

The compiler must assign storage before using addresses in the IR

- Code contains implicit information about storage
- Compiler must make its decisions after parsing declarations
  - Can envision batch schemes or incremental schemes
  - Either way, the compiler assigns addresses

```
Procedure → Header { Decls StmtList }
Decls → DeclList
DeclList → DeclList Declaration
         | Declaration
StmtList → StmtList Statement
         | Statement
```

Modify the grammar to create a reduction to assign storage.

Typical Grammatical Organization for Procedures

Final points:
1. When does this happen? At compile time
2. Where does stuff go? Next lecture