Copyright 2018, 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
Midterm Exam

The Midterm will cover
• Class content through last class
• Chapters 1, 2, 3, and 5 in the book
• Local register allocation material from the Chapter 13 excerpt

The format will be
• Closed book, closed notes, closed devices
• Designed as a two-hour exam

Review Session
• Tuesday, October 16, 7PM
• Location tba
• Bring your questions

A number of you need alternate arrangements for the exam.
We will sort those out over the weekend and send you email.
If you have not heard from us by class time on Monday, send an email Monday afternoon to both Zoran and Keith.
Code Generation

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

• Generate an IR straight out of the parser
  – Might be an AST, might be some high-level (abstract) linear form
  – Almost always accompanied by a “symbol table” of some form

• Code passes through one or more “lowering” pass
  – Takes the code and decreases (“lowers”) the level of abstraction
  – Expand complex operations (e.g., call or mvcl), make control-flow explicit

The problems are, essentially, the same
The mechanisms are, likely, quite different
Mechanism versus Content

In the lectures on SDT, we focused on how to generate code.

In the lectures on code shape, we will focus on what code to generate.

Different instruction sequences for the same source code produce significantly different performance:

- When two sequences produce the same result, choose the fastest one.
- The compiler makes fundamental choices that are hard to change.

Tradeoffs in performance often involve tradeoffs in generality:

- Code that handles the general case is robust and, often, slower.
- Code tailored to specific cases is more brittle and, often, faster.
  - Code optimization is, to a large extent, the art of capitalizing on context.
  - The compiler can specialize, with the constraint the code is correct in its actual context (its “provable” context).
Performance versus Generality

Consider copying characters from src to dst

```c
char src[256], dst[256];
int len, i;
if (len < 256) {
    i = 0;
    while(i < len) {
        dst[i] = src[i];
        i++;
    }
}
```

```c
char src[256], dst[256];
int64 *p, * q;
int len, i;
if (len < 256 && len % 8 == 0) {
    i = 0;
    p = (int64 *) & src[0];
    q = (int64 *) & dst[0];
    while(i < len) {
        *p++ = *q++;
        i += 8;
    }
}
```

**Generic Version**
len loads & stores

**Aligned Strings of 8x bytes**
(len / 8) loads & stores

**Context:**
Declarations ensure that src and dst enforce the necessary storage alignment.
Separate declarations mean that src and dst do not overlap.

Now, imagine the version that uses 8 byte loads as long as it can, then gets that last (<7) bytes with 4, 2, & 1 byte loads.

(as long as the strings do not overlap and start on an aligned boundary)
Code Shape

(Chapter 7)

Definition
• All those nebulous properties of the code that affect performance
• Includes code, approach for implementing different constructs, cost, storage requirements & mapping, and choice of operations
• Code shape is the end product of many decisions (*big & small*)

Impact
• Code shape influences algorithm choice & results
• Code shape can encode important facts, or hide them

Rule of thumb: expose as much derived information as possible
• Example: explicit branch targets in ILOC simplify analysis
• Example: hierarchy of memory operations in ILOC (*EaC2e, p 251*)

See Bob Morgan’s book for more ILOC examples [268 in EaC2e Bibliography]
My favorite code shape example

- What if $x$ is 2 and $z$ is 3?
- What if $y+z$ is evaluated earlier?

The “best” shape for $x+y+z$ depends on contextual knowledge
  - There may be several conflicting options

Addition is commutative & associative for integers.
See also Floyd’s paper, on the Lectures page.
Another example — implementing the case statement

• Implement it as cascaded if-then-else statements
  – Cost depends on where your case actually occurs
  – $\mathcal{O}(\text{number of cases})$

• Implement it as a binary search
  – Need a dense set of conditions to search
  – Uniform ($\log n$) cost

• Implement it as a jump table
  – Need a compact & computable set of case labels
  – Lookup address in a table & jump to it
  – Uniform ($\text{constant}$) cost

Compiler must choose best implementation strategy
No amount of massaging or transforming in the optimizer will convert one strategy into another

Performance depends on order of cases in the final code ⇒ compiler should reorder based on frequency!
Why worry about code shape? Can’t we just trust the optimizer and the back end?

- Optimizer and back end approximate the answers to many hard problems
- The compiler’s individual passes must run quickly
- It often pays to encode useful information into the IR
  - Shape of an expression or a control structure
  - A value kept in a register rather than in memory
- Deriving such information may be expensive, when possible
- Recording it explicitly in the IR is often easier and cheaper

A good optimizer can tune the engine, but it cannot make a Volkswagen Beetle into a Porsche
The “best” code shape depends on context

- Context in the code being compiled
  - What are the common subexpressions? the known constants?
  - Do case labels form a compact set?
  - What values in registers are live after a call?

- Context in the compiler itself
  - In SDT and optimization, the compiler should shape the code so that it optimizes well
  - In back end, the compiler should shape the code so that it runs quickly

And, again, code shape is a content issue not a mechanism issue

*Neither case statement nor string copy implementation change between SDT, a tree-walk lowering pass, and a pattern-matching instruction selector*

*Code shape is one of the places where compilation resembles art.*
Consider \( a + b + c + d + e + f + g + h \)

Left-recursive expression grammar (*e.g.*, LR parser) would produce a left-associative tree

Right recursion and right associative trees have a symmetric problem
Consider “a + b + c + d + e + f + g + h”

Balanced tree

- No rational parser will produce this tree.
- It results from an explicit reordering pass.
- Best approach might be a post-optimization, pre-scheduling pass.

Sequential dependences are much more relaxed

COMP 412, Fall 2018

See draft section posted with lecture
### Evaluation Order

**Code Shape & Instruction Scheduling**

#### Balanced Tree

<table>
<thead>
<tr>
<th>Unit 0</th>
<th>Unit 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>$t_1 \leftarrow a + b$</td>
</tr>
<tr>
<td>2</td>
<td>$t_3 \leftarrow e + f$</td>
</tr>
<tr>
<td>3</td>
<td>$t_5 \leftarrow t_1 + t_2$</td>
</tr>
<tr>
<td>4</td>
<td>$t_7 \leftarrow t_5 + t_6$</td>
</tr>
</tbody>
</table>

#### Left-Associative Tree

<table>
<thead>
<tr>
<th>Unit 0</th>
<th>Unit 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>$t_1 \leftarrow a + b$</td>
</tr>
<tr>
<td>2</td>
<td>$t_2 \leftarrow t_1 + c$</td>
</tr>
<tr>
<td>3</td>
<td>$t_3 \leftarrow t_2 + d$</td>
</tr>
<tr>
<td>4</td>
<td>$t_4 \leftarrow t_3 + e$</td>
</tr>
<tr>
<td>5</td>
<td>$t_5 \leftarrow t_4 + f$</td>
</tr>
<tr>
<td>6</td>
<td>$t_6 \leftarrow t_5 + g$</td>
</tr>
<tr>
<td>7</td>
<td>$t_7 \leftarrow t_6 + h$</td>
</tr>
</tbody>
</table>

**Schedules for Balanced Tree versus Left-Associative Tree**

Assumes two functional units

**Free for other work**

---

Multi-unit parallelism is called **instruction-level parallelism or ILP**
However, deep trees *may* need fewer registers than broad trees.

- Deep trees may lead to less spilling
- Broad trees may lead to more instruction-level parallelism

**As with most code shape decisions, there is not a simple answer**

COMP 412, Fall 2018
Evaluation Order

Order of evaluation for expression trees
• The compiler can use commutativity & associativity to improve code
• This problem is truly hard

Commuting operands at a single operation is much easier
• 1\textsuperscript{st} operand must be preserved while 2\textsuperscript{nd} is evaluated
• Takes an extra register for 2\textsuperscript{nd} operand
⇒ Evaluate the more demanding subtree first

(Ershov in the 1950’s, Sethi in the 1970’s)

Taken to its logical conclusion, this creates Sethi-Ullman scheme for register allocation

See Sethi & Ullman, [311] in EaC2e
See Floyd’s 1961 CACM paper, [150] in EaC2e, also on the website
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

\[
\text{Declaration} \rightarrow \begin{array}{c}
\text{INT } \text{IntList} \\
| \\
\text{CHAR } \text{CharList} \\
| \\
\ldots \text{ other types } \ldots
\end{array}
\]

\[
\text{IntList} \rightarrow \begin{array}{c}
\text{IntList } \ldots \text{Spec} \\
| \\
\text{Spec}
\end{array}
\]

\[
\text{CharList} \rightarrow \begin{array}{c}
\text{CharList } \ldots \text{Spec} \\
| \\
\text{Spec}
\end{array}
\]

\[
\text{Spec} \rightarrow \text{NAME}
\]

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 \((\text{conflicts})\)
- 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 $<\text{level}, \text{offset}>$ 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 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)

```
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
      ...
    }
    ...
  }
  ...
}
```

COMP 412, Fall 2018
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

COMP 412, Fall 2018
Lexically-Scoped Symbol Tables

High-level idea

• Create a new table for each scope
• Chain them together for lookup

“Sheaf of tables” implementation

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

If the compiler must preserve the table (\textit{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 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

I tend to use Smalltalk 80 terminology ...

OOL ≡ Object-Oriented Language
ALL ≡ Algol-Like Language
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:

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

The object used to locate the code member is termed the “receiver”.

Inheritance adds some twists.
Concrete Example: The Java Name Space

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

- Local variables declared within M \( (\text{lexical scoping}) \)
- 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*
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

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

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