Chapter 1: Overview of Compilation

A Presentation by Gregory Breard
Introduction

- Today we will be discussing compilers. This will be a rather high-level introduction to compiler design, and most of the material covered should be familiar to you.

- **Compiler** - a computer program that translates other computer programs to prepare them for execution
Compilers translate software written in one language into another language.

To perform this translation, the compiler must:
- Understand the form of the language (or syntax)
- Understand the meaning of the language (or semantics)
- And have a scheme for mapping content from the source language to the target language

Compilers typically have a front end for dealing with the source language, and a back end for dealing with the target language.
• In general, compilers translate programming languages into machine instructions for a specific processor (or target machine)
• Viewed as a black box:
• Typical source languages are: C++, Java, etc.
• The target language is usually the instruction set of the target machine
Overview (continued)

- **Instruction set** - the set of operations supported by a processor; the overall design of an instruction set is often called an *instruction set architecture* (or ISA).
- Some compilers target programming languages instead of an instruction set, these are referred to as *source-to-source translators*.
- There are many other systems that qualify as compilers (i.e. typesetting programs).
A program that reads source code and produces results (instead of translating to a target language) is called an interpreter.

Some languages' translation schemes include both compilation and interpretation, one example being Java.
Overview (continued)

- Java is compiled into bytecode, which is then executed by a bytecode interpreter, the Java Virtual Machine (JVM)
- **Virtual machine** - A virtual machine is a simulator for some processor. It is an interpreter for that machine's instruction set.

- Compilers and interpreters are similar and perform many of the same tasks. However, the outputs of these programs are significantly different.
The Fundamental Principles of Compilation

- There are two fundamental principles of compilation that are essential to compiler design:

  1. *The compiler must preserve the meaning of the program being compiled.*

  2. *The compiler must improve the input program in some discernible way.*
A compiler must both understand the source program and map its functionality to the target machine. These two distinct tasks are separated into the *front end* and *back end* of the compiler.
The front end focuses on understanding the source language program.

The back end focuses on mapping programs to the target language.

Between these tasks, the compiler uses an intermediate representation (IR) to store information about the program.

IR - A compiler uses some set of data structures to represent the code that it processes. That form is called an intermediate representation.
The IR is a definitive representation of the code it is translating.
Compilers may even use several different IRs depending on the task it is performing.
The front end ensures the source code is well formed, and maps it to the IR.
The back end therefore only processes the IR, and can assume the IR contains no syntactic or semantic errors.
This two-phase approach to compiling also simplifies the process of retargeting.

Retargeting - the task of changing the compiler to generate code for a new processor is often called retargeting the compiler.

The compiler can be made to read a different source program by changing out the front end. Similarly, the compiler can be made to translate to a different target program by changing out the back end.
A compiler can also have a third phase added between the front end and back end, an **optimizer**.

**Optimizer** - analyzes and transforms the IR to improve it.
● The optimizer is an IR-to-IR transformer
● It can make one or more passes over the IR, analyzing and rewriting it.
● The optimizer may have a variety of objectives, i.e. a faster target program or a smaller target program
● It should be noted that although the term optimization is used, the problems of optimization are so complex and interrelated that they cannot, in practice, be solved optimally.
Overview of Translation

● In translating from a programming language to machine executable code, a compiler runs through many steps.

● Following, we will discuss the steps taken by:
  ○ The Front End
  ○ The Optimizer
  ○ The Back End
The Front End

- Before translating the code, the compiler must understand the syntax and semantics of the source program.
- If the syntax and semantics are valid, the front end produces an intermediate representation for the source program.
- If the syntax or semantics are invalid, a diagnostic error message is returned to the user and compilation is halted.
The Front End: Checking Syntax

- To check the syntax of a program, the compiler must compare the program's structure to a definition of the language.
- The source language is defined by a finite set of rules, called a grammar.
- Programming language grammars refer to words by their parts of speech, or syntactic categories.
For example, an English sentence may have the definition:

\[ \text{Sentence} \rightarrow \text{Subject} \text{ verb Object endmark} \]

Here, \text{verb} and \text{endmark} are parts of speech and \text{Subject} and \text{Object} are syntactic variables.

\text{Sentence} represents any string with the form described by the rule.

The \rightarrow symbol is read "derives" and means the instance on the right can be abstracted to the syntactic variable on the left.
The Front End: Checking Syntax (continued)

- Two separate passes in the front end (called the **scanner** and the **parser**) determine if the input program is valid.
- **Scanner** - the compiler converts a string of characters into a stream of classified words.
- **i.e.** "Compilers are engineered objects." would be converted to the (part of speech, spelling) pairs:
  
  (noun, "Compilers"), (verb, "are"),
  (adjective, engineered"), (noun, "objects"), (endmark, ".")
The Front End: Checking Syntax (continued)

● Example grammar:

Sentence → Subject verb Object endmark
Subject → noun
Subject → Modifier noun
Object → noun
Object → Modifier noun
Modifier → adjective
• **Parser** - performs a series of automatic derivations in order to determine if the input stream is a sentence in the language definition.

• Derivation for our example:

```
Sentence
Subject verb Object endmark
noun verb Object endmark
noun verb Modifier noun endmark
noun verb adjective noun endmark
```
However, a grammatically correct sentence may be meaningless i.e. "Rocks are green vegetables."

Semantic analysis is used to determine if a sentence's "meaning" is valid.

One example of semantic analysis is checking for type consistency i.e. to make sure an int is not assigned a string value.

Type Checking - the compiler pass that checks for type-consistent uses of names in the input program.
The Front End: Intermediate Representation

- The front end is also responsible for generating the IR.
- Compilers use a variety of different types of IRs, depending on the specific needs of the compiler.
- However, for every source-language construct the compiler needs a strategy for how it will implement the construct in the IR.
The Optimizer

- The optimizer analyzes the IR to discover facts about how the code will behave at runtime.
- It then uses this information to rewrite the code so that it produces the same answer in a more efficient way.
- Efficiency can have many meanings in this context, i.e. reduced running time, reduced compiled code size, reduced processor energy consumption, etc.
The Optimizer: Analysis

- The first step of optimization is to analyze the code to determine where the compiler can safely and profitably apply transformations.
- Compilers use several kinds of analysis.
- **Data-flow analysis** - a form of compile time reasoning about the runtime flow of values.
- **Dependence analysis** - uses number-theoretic tests to reason about the values that can be assumed by subscript expressions.
The Optimizer: Transformation

- After analyzing the code, the compiler must use the results to rewrite the code in a more efficient form.
- A multitude of transformations have been invented to do just that.
- One example is to move loop-invariant computations outside of loops to improve running time of the program.
- Transformations vary in their effect, the scope over which they operate, and the analysis required to support them.
The Back End

- The back end reads the IR and generates code for the target machine.
- It selects target machine operations to perform the operations represented in the IR and chooses an order in which these operations will execute efficiently.
- It also decides which values will reside in registers and which will reside in memory, and generates the code that will enforce these decisions.
The first step in code generation is *instruction selection*, in which each IR operation is rewritten as one or more target machine operations.

**Example:**  
\[ a \leftarrow a \times 2 \times b \times c \]  
IR for the expression:

\[
\begin{align*}
t_0 & \leftarrow a \times 2 \\
t_1 & \leftarrow t_0 \times b \\
t_2 & \leftarrow t_1 \times c \\
a & \leftarrow t_2
\end{align*}
\]
Rewritten for the ILOC virtual machine:

```plaintext
loadAI r_{arp}, @a ⇒ r_a // load 'a'
loadI 2 ⇒ r_2 // constant 2 into r_2
loadAI r_{arp}, @b ⇒ r_b // load 'b'
loadAI r_{arp}, @c ⇒ r_c // load 'c'
mult r_a, r_2 ⇒ r_a // r_a = a * 2
mult r_a, r_b ⇒ r_a // r_a = (a * 2) * b
mult r_a, r_c ⇒ r_a // r_a = (a * 2 * b) * c
storeAI r_a ⇒ r_{arp}, @a // write r_a back to // 'a'
```
In the code in the previous slide, a straightforward approach has been used to rewrite the IR. The values are loaded into registers, the multiplication operations are performed, and the result is stored in the memory location for a. The compiler assumes there is an unlimited supply of registers, which it names symbolically. Implicitly, the instruction selector relies on the register allocator to map these *virtual registers* to the actual registers of the target machine.
The Back End: Register Allocation

● The instruction selector deliberately ignores the fact that the target machine has a limited set of registers.

● In practice, the earlier stages of compilation may create more demand for registers than the hardware can support.

● It is the job of the register allocator to map the virtual registers to actual registers on the target machine.

● On the following slide is our previous example, rewritten to minimize register use.
The Back End: Register Allocation (continued)

- Rewritten for the ILOC virtual machine:

\[
\begin{align*}
\text{loadAI } & \ r_{\text{arp}}, @a \Rightarrow r_1 \quad // \text{load 'a'} \\
\text{add } & \ r_1, r_1 \Rightarrow r_1 \\
\text{loadAI } & \ r_{\text{arp}}, @b \Rightarrow r_2 \quad // \text{load 'b'} \\
\text{mult } & \ r_1, r_2 \Rightarrow r_1 \quad // \ r_1 = (a \ast 2) \ast b \\
\text{loadAI } & \ r_{\text{arp}}, @c \Rightarrow r_2 \quad // \text{load 'c'} \\
\text{mult } & \ r_1, r_2 \Rightarrow r_1 \quad // \ r_1 = (a \ast 2 \ast b) \ast c \\
\text{storeAI } & \ r_1 \Rightarrow r_{\text{arp}}, @a \\
& \quad // \text{write } r_1 \text{ back to 'a'}
\end{align*}
\]

- This sequence uses 3 registers instead of 6.
The Back End: Instruction Scheduling

- To increase performance the operations may be reordered to reflect the performance constraints of the target machine.
- i.e. memory access operations may take hundreds of cycles, while arithmetic operations may take only several.
- For example, assume `loadA` and `storeA` take 3 cycles, and `mul` takes 2 cycles to complete.
- Following is a demonstration of how reordering operations improves performance.
## The Back End: Instruction Scheduling

<table>
<thead>
<tr>
<th>Start</th>
<th>End</th>
<th>Instruction</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>3</td>
<td><code>loadAI r_{arp}, @a \Rightarrow r_1</code> // load 'a'</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>4</td>
<td><code>add r_1, r_1 \Rightarrow r_1</code> // $r_1 = a \times 2$</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>7</td>
<td><code>loadAI r_{arp}, @b \Rightarrow r_2</code> // load 'b'</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>9</td>
<td><code>mult r_1, r_2 \Rightarrow r_1</code> // $r_1 = (a \times 2) \times b$</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>12</td>
<td><code>loadAI r_{arp}, @c \Rightarrow r_2</code> // load 'c'</td>
<td></td>
</tr>
<tr>
<td>13</td>
<td>14</td>
<td><code>mult r_1, r_2 \Rightarrow r_1</code> // $r_1 = (a \times 2 \times b) \times c$</td>
<td></td>
</tr>
<tr>
<td>15</td>
<td>17</td>
<td><code>storeAI r_1 \Rightarrow r_{arp},@a</code> // write $r_1$ back to 'a'</td>
<td></td>
</tr>
</tbody>
</table>

- These 8 operations take 17 cycles to complete
The Back End: Instruction Scheduling

<table>
<thead>
<tr>
<th>Start</th>
<th>End</th>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>3</td>
<td>loadAI r_{arp}, @a ⇒ r_1 // load 'a'</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>4</td>
<td>loadAI r_{arp}, @b ⇒ r_2 // load 'b'</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>5</td>
<td>loadAI r_{arp}, @c ⇒ r_3 // load 'c'</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>4</td>
<td>add r_1, r_1 ⇒ r_1 // r_1 = a * 2</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>6</td>
<td>mult r_1, r_2 ⇒ r_1 // r_1 = (a * 2) * b</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>8</td>
<td>mult r_1, r_2 ⇒ r_1 // r_1 = (a * 2 * b) * c</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>11</td>
<td>storeAI r_1 ⇒ r_{arp},@a // write r_1 back to 'a'</td>
<td></td>
</tr>
</tbody>
</table>

- These 8 operations take 11 cycles to complete
The Back End:  
Interactions Among Code-Generation Components

- Code generation is complicated further by the interaction of complex problems.
- For example, instruction scheduling moves load operations away from the arithmetic operations that depend on them.
- This increases the amount of time that these registers hold values, and therefore may increase the number of registers needed.
- Also, a false dependency can be created between operations when specific registers are used.
Summary

- Compiler design is a complicated task.
- Compilers use many methods to address a variety of complex problems.
- Many of these problems are too hard to solve optimally, so compilers use approximations and heuristics.
- This often results in interactions that may produce surprising results - which may be good or bad.
Sources

All material included in these slides is from:

*Engineering A Compiler, 2nd Edition*
by Keith Cooper and Linda Torczan, pgs 1 - 21

FIN