The View from 35,000 Feet
High-level View of a Compiler

Traditional Compiler

Source code

Machine code

Errors
Responsibilities

- Front end produces intermediate representation (IR)
- Back end produces machine code
The Front End

Responsibilities
- Recognize legal (and illegal) programs
- Produces IR
The Front End

**Scanner**
- Maps character stream into words
  - the basic unit of syntax
- Produces pairs — a word & its part of speech
The Front End

Parser
- Recognizes syntax (context-free) and reports errors
- Builds IR for source program
The Front End

Context-free syntax is specified with a grammar

\[
\text{SheepNoise} \rightarrow \text{baa SheepNoise} \\
| \quad \text{baa}
\]

This grammar defines the set of noises that a sheep makes under normal circumstances.

It is written in a variant of Backus-Naur Form (BNF).
Formally, a grammar $G = (S, N, T, P)$

- $S$ is the start symbol
- $N$ is a set of non-terminal symbols
- $T$ is a set of terminal symbols or words
- $P$ is a set of productions or rewrite rules
The Front End

Context-free syntax can be put to better use

- This grammar defines simple expressions with addition & subtraction over "number" and "id"

1. $goal \rightarrow expr$
2. $expr \rightarrow expr \ op \ term$
3. $\mid term$
4. $term \rightarrow number$
5. $\mid id$
6. $op \rightarrow +$
7. $\mid -$

$S = goal$
$T = \{ \text{number, id, +, -} \}$
$N = \{ \text{goal, expr, term, op} \}$
$P = \{ 1, 2, 3, 4, 5, 6, 7 \}$
The Front End

Given a CFG, we can **derive** sentences by repeated substitution.

<table>
<thead>
<tr>
<th>Production</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>goal</code></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td><code>expr</code></td>
</tr>
<tr>
<td>2</td>
<td><code>expr op term</code></td>
</tr>
<tr>
<td>5</td>
<td><code>expr op y</code></td>
</tr>
<tr>
<td>7</td>
<td><code>expr - y</code></td>
</tr>
<tr>
<td>2</td>
<td><code>expr op term - y</code></td>
</tr>
<tr>
<td>4</td>
<td><code>expr op 2 - y</code></td>
</tr>
<tr>
<td>6</td>
<td><code>expr + 2 - y</code></td>
</tr>
<tr>
<td>3</td>
<td><code>term + 2 - y</code></td>
</tr>
<tr>
<td>5</td>
<td><code>x + 2 - y</code></td>
</tr>
</tbody>
</table>

To recognize a valid sentence in some CFG, we reverse this process and build up a **parse**.
The Front End

A parse can be represented by a tree (parse tree or syntax tree)

\[ x + 2 - y \]

This contains a lot of unneeded information.
The Front End

Compilers often use an *abstract syntax tree*

An AST is just one of several *intermediate representations (IR)* that can be used in a compiler.

This is much more concise.
The Back End

Responsibilities

• Translate IR into target machine code
• Choose instructions to implement each IR operation
• Decide which values to keep in registers

Automation has been less successful in the back end
Instruction Selection

- Produce fast, compact code
- Take advantage of target machine features
- Usually viewed as a pattern matching problem
  → ad hoc methods, pattern matching, dynamic programming
The Back End

Register Allocation

- Allocating variables (i.e., values) into registers
- Manage a limited set of registers
  - Often more variables than registers available
- Optimal allocation is NP-Complete
The Back End

Instruction Scheduling
- Tries to find a better ordering of the assembly instructions
- Architecture dependent
- Finding optimal ordering (schedule) is NP-complete
Traditional Three-pass Compiler

**Code Improvement (or **Optimization**)**

- Analyzes IR and rewrites (or **transforms**) IR
- Primary goal is to reduce running time of the compiled code
  → May also improve space, power consumption, ...
- Must preserve “meaning” of the code
  → Measured by values of named variables
Modern optimizers are structured as a series of passes

Typical Transformations

• Discover and propagate some constant value
• Move a computation to a less frequently executed place
Next Week

- Introduction to Scanning (aka Lexical Analysis)
  - Material is in Chapter 2

- Phase 2 available next Monday (2/12)