Note: These lecture notes go with pages 337 to 357 of the textbook. I’ve altered the notation a bit, because I found the textbook notation a barrier to understanding — I don’t think it’s humanly possible to hold those rewrite rule number in one’s head, and I haven’t got the patience to keep flipping pages to decode them.

I may use these slides in class, but more likely I’ll just use the blackboard again. I think the textbook will make more sense after you look through these notes. Let me know.
Writing Code Generators Sucks

- Too much tedious code
- Too dependent on target machine
- Too easy to get wrong, and too hard to debug

Shouldn’t we be able to *generate* code generators? From what? How about a grammar to describe the target machine?

\[
\begin{align*}
\langle \text{reg} \rangle &::= \text{load } \langle \text{reg} \rangle, \text{mem} \\
\langle \text{reg} \rangle &::= \text{add } \langle \text{reg} \rangle, \langle \text{reg} \rangle 
\end{align*}
\]

etc.
How bad can it get? This bad:

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Instruction</th>
<th>64-Bit Mode</th>
<th>Compat/ Leg Mode</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>14 ib</td>
<td>ADC AL, imm8</td>
<td>Valid</td>
<td>Valid</td>
<td>Add with carry imm8 to AL.</td>
</tr>
<tr>
<td>15 iw</td>
<td>ADC AX, imm16</td>
<td>Valid</td>
<td>Valid</td>
<td>Add with carry imm16 to AX.</td>
</tr>
<tr>
<td>15 id</td>
<td>ADC EAX, imm32</td>
<td>Valid</td>
<td>Valid</td>
<td>Add with carry imm32 to EAX.</td>
</tr>
<tr>
<td>REX.W + 15 id</td>
<td>ADC RAX, imm32</td>
<td>Valid</td>
<td>N.E.</td>
<td>Add with carry imm32 sign extended to 64-bits to RAX.</td>
</tr>
<tr>
<td>80 /2 ib</td>
<td>ADC r/m8, imm8</td>
<td>Valid</td>
<td>Valid</td>
<td>Add with carry imm8 to r/m8.</td>
</tr>
<tr>
<td>REX + 80 /2 ib</td>
<td>ADC r/m8, imm8</td>
<td>Valid</td>
<td>N.E.</td>
<td>Add with carry imm8 to r/m8.</td>
</tr>
<tr>
<td>81 /2 iw</td>
<td>ADC r/m16, imm16</td>
<td>Valid</td>
<td>Valid</td>
<td>Add with carry imm16 to r/m16.</td>
</tr>
<tr>
<td>81 /2 id</td>
<td>ADC r/m32, imm32</td>
<td>Valid</td>
<td>Valid</td>
<td>Add with CF imm32 to r/m32.</td>
</tr>
<tr>
<td>REX.W + 81 /2 id</td>
<td>ADC r/m64, imm32</td>
<td>Valid</td>
<td>N.E.</td>
<td>Add with CF imm32 sign extended to 64-bits to r/m64.</td>
</tr>
<tr>
<td>83 /2 ib</td>
<td>ADC r/m16</td>
<td>Valid</td>
<td>Valid</td>
<td>Add with CF sign-extended</td>
</tr>
</tbody>
</table>

**NOTES:**
* In 64-bit mode, r/m8 cannot be encoded to access the following byte registers if an REX prefix is used: AH, BH, CH, DH.
Machine Grammar as Tree Patterns

<table>
<thead>
<tr>
<th>Name</th>
<th>Pattern</th>
<th>Emit Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>add:</td>
<td>reg₁ ::= (+ reg₁ reg₂)</td>
<td>{ ADD reg₁,reg₁,reg₂ }</td>
</tr>
<tr>
<td>load:</td>
<td>reg₁ ::= (mem₁)</td>
<td>{ LOAD reg₁,mem₁ }</td>
</tr>
<tr>
<td>addm:</td>
<td>reg₁ ::= (+ reg₁ mem₁)</td>
<td>{ ADDM reg₁,mem₁ }</td>
</tr>
<tr>
<td>addc:</td>
<td>reg₁ ::= (+ reg₁ const₁)</td>
<td>{ ADDC reg₁,const₁ }</td>
</tr>
<tr>
<td>loadc:</td>
<td>reg₁ ::= (const₁)</td>
<td>{ LOADC reg₁,const₁ }</td>
</tr>
</tbody>
</table>

Subtree templates:

**add:**
```
+  →  reg
\   /
reg  reg
```

**load:**
```
mem  →  reg
```

**addm:**
```
+  →  reg
\   /
reg  mem
```
## Patterns from Grune et al (Fig 4.60, pg 338)

### Code Interpretation

<table>
<thead>
<tr>
<th>Pattern</th>
<th>Code</th>
<th>Cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>$R_n ::= const$</td>
<td><code>{ Load_Const const, R_n }</code></td>
<td>1</td>
</tr>
<tr>
<td>$R_n ::= mem$</td>
<td><code>{ Load_Mem mem, R_n }</code></td>
<td>3</td>
</tr>
<tr>
<td>$R_n ::= ( + R_n \ mem)$</td>
<td><code>{ Add_Mem mem, R_n }</code></td>
<td>3</td>
</tr>
<tr>
<td>$R_n ::= ( + R_n \ R_1)$</td>
<td><code>{ Add_Reg R_1, R_n }</code></td>
<td>1</td>
</tr>
<tr>
<td>$R_n ::= (\ast R_n \ mem)$</td>
<td><code>{ Mult_Mem mem, R_n }</code></td>
<td>6</td>
</tr>
<tr>
<td>$R_n ::= (\ast R_n \ R_m)$</td>
<td><code>{ Mult_Reg R_m, R_n }</code></td>
<td>4</td>
</tr>
<tr>
<td>$R_n ::= ( + R_n (\ast \ const \ R_m))$</td>
<td><code>{ Add_Scaled_Reg const, R_m, R_n }</code></td>
<td>4</td>
</tr>
<tr>
<td>$R_n ::= (\ast R_n (\ast \ const \ R_m))$</td>
<td><code>{ Mult_Scaled_Reg const, R_m, R_n }</code></td>
<td>5</td>
</tr>
</tbody>
</table>

Interpretation: Subtree matching the *pattern* can be rewritten as *result* by generating *code* which has this *cost* in time or space.
Breaking down multi-level patterns

addS: (+ (* const reg₁) reg₂) { ADD_SCALED const, reg₁, reg₂ }

Decompose so we can work up from the leaves, matching the bottom subtree of the program first.
Multiple Matches

We could match a program subtree many ways ...

\[
\begin{align*}
&+ \\
&\quad a + \quad \text{reg: add} \\
&\quad \quad * \quad \text{addS.1} \\
&\quad \quad b \\
&\quad \quad 2 \\
&\quad \quad x
\end{align*}
\]

\[
\begin{align*}
&+ \\
&\quad a + \quad \text{reg: add} \\
&\quad \quad * \quad \text{reg: mult} \\
&\quad \quad b \\
&\quad \quad 2 \\
&\quad \quad x
\end{align*}
\]

= load
Multi-pass Strategy

- Pass 1: Match all, bottom up
- Pass 2: Select best, top down
- Pass 3: Emit code, bottom up

“Best” means least cost; add costs (or cost functions) to grammar rules.

Efficiency:
- \( O(\text{size of tree} \times \text{size of instruction set}) \)

Ouch! Can we get rid of the second factor?
Compiling a tree automaton

Our old friend the subset construction: Compute all possible transitions.
State: *set of matched rules (with costs)*
Transition: From *left operand state × right operand state × subtree root operator* to *subtree state*

- **Base case:** States for leaf nodes.
  Example: State for const has two entries, *const* (leave it alone) and *loadc* (put it in a register)
- **Closure:** For every triple (state, state, operator), if there are matching rules, add the new state with all the ways of matching
  Potentially explosive, but not too bad in practice.
Within a single state with multiple patterns:

- If two patterns yield the same type (e.g., “register”)
- and one is always cheaper than the other
- then remove the more expensive pattern; we’ll never choose it.

If costs are constant, then we can incorporate the cost calculation in the state transition table.
If costs are not constant (the usual case), we can still calculate them in a bottom up scan while matching. (10x slower.)
Perspective

- This is really only the instruction selection part of code generation; there are complex interactions with flow analysis, register allocation, etc.
- We’ve omitted control flow; it’s probably separate, but interacts.

Other approaches:

- Simpler code generation + aggressive peephole optimization
- Simpler instruction sets. RISC machines (MIPS, SPARC, PPC, ...) are load/store architectures, so usually we have just one choice at each AST node.

Priorities are shifting: Less emphasis on producing the fastest possible code, more on making the compiler fast and simple and detecting program bugs.