High-Level Compiler Structure

Optimizing Compilers

Theory  Practice
Lexing and Parsing

**Lexer**  Forms tokens from character stream

**Parser**  Syntax-directed translation to parse tree

<table>
<thead>
<tr>
<th>Production</th>
<th>Rule</th>
</tr>
</thead>
<tbody>
<tr>
<td>E → E + T</td>
<td>E.ptr := mnode('+', E.ptr, T.ptr)</td>
</tr>
<tr>
<td>E → E - T</td>
<td>E.ptr := mnode('-', E.ptr, T.ptr)</td>
</tr>
<tr>
<td>E → T</td>
<td>E.ptr := T.ptr</td>
</tr>
<tr>
<td>T → id</td>
<td>T.ptr := mleaf(id, identry)</td>
</tr>
<tr>
<td>T → num</td>
<td>T.ptr := mleaf(num, num.val)</td>
</tr>
</tbody>
</table>

LR Parsing

<table>
<thead>
<tr>
<th>Grammar</th>
<th>Stream</th>
<th>Stack</th>
<th>Rule Used</th>
</tr>
</thead>
<tbody>
<tr>
<td>E → E + T</td>
<td>1</td>
<td>7</td>
<td></td>
</tr>
<tr>
<td>E → E - T</td>
<td>2</td>
<td>7</td>
<td></td>
</tr>
<tr>
<td>E → T</td>
<td>3</td>
<td>7</td>
<td></td>
</tr>
<tr>
<td>T → id</td>
<td>4</td>
<td>7</td>
<td></td>
</tr>
<tr>
<td>T → num</td>
<td>5</td>
<td>7</td>
<td></td>
</tr>
<tr>
<td>E → E + T</td>
<td>1</td>
<td>x</td>
<td></td>
</tr>
<tr>
<td>E → E - T</td>
<td>2</td>
<td>x</td>
<td></td>
</tr>
<tr>
<td>E → T</td>
<td>3</td>
<td>x</td>
<td></td>
</tr>
<tr>
<td>T → id</td>
<td>4</td>
<td>x</td>
<td></td>
</tr>
<tr>
<td>T → num</td>
<td>5</td>
<td>x</td>
<td></td>
</tr>
<tr>
<td>E → E + T</td>
<td>1</td>
<td>x</td>
<td></td>
</tr>
<tr>
<td>E → E - T</td>
<td>2</td>
<td>x</td>
<td></td>
</tr>
<tr>
<td>E → T</td>
<td>3</td>
<td>x</td>
<td></td>
</tr>
<tr>
<td>T → id</td>
<td>4</td>
<td>x</td>
<td></td>
</tr>
<tr>
<td>T → num</td>
<td>5</td>
<td>x</td>
<td></td>
</tr>
<tr>
<td>E → E + T</td>
<td>1</td>
<td>E</td>
<td></td>
</tr>
<tr>
<td>E → E - T</td>
<td>2</td>
<td>E</td>
<td></td>
</tr>
<tr>
<td>E → T</td>
<td>3</td>
<td>E</td>
<td></td>
</tr>
<tr>
<td>T → id</td>
<td>4</td>
<td>E</td>
<td></td>
</tr>
<tr>
<td>T → num</td>
<td>5</td>
<td>E</td>
<td></td>
</tr>
<tr>
<td>E → E + T</td>
<td>1</td>
<td>E</td>
<td></td>
</tr>
<tr>
<td>E → E - T</td>
<td>2</td>
<td>E</td>
<td></td>
</tr>
<tr>
<td>E → T</td>
<td>3</td>
<td>E</td>
<td></td>
</tr>
<tr>
<td>T → id</td>
<td>4</td>
<td>E</td>
<td></td>
</tr>
<tr>
<td>T → num</td>
<td>5</td>
<td>E</td>
<td></td>
</tr>
<tr>
<td>E → E + T</td>
<td>1</td>
<td>E</td>
<td></td>
</tr>
<tr>
<td>E → E - T</td>
<td>2</td>
<td>E</td>
<td></td>
</tr>
<tr>
<td>E → T</td>
<td>3</td>
<td>E</td>
<td></td>
</tr>
<tr>
<td>T → id</td>
<td>4</td>
<td>E</td>
<td></td>
</tr>
<tr>
<td>T → num</td>
<td>5</td>
<td>E</td>
<td></td>
</tr>
</tbody>
</table>
Why is Optimization needed?

\[
X := Y + 3 \\
\text{temp1 := inttoreal(3)} \\
\text{load(temp2, Y)} \\
\text{temp3 := temp2 + temp1} \\
\text{store(X, temp3)} \\
\]

\[
Z := X + 5 \\
\text{temp4 := inttoreal(5)} \\
\text{load(temp5, X)} \\
\text{temp6 := temp5 + temp4} \\
\text{store(Z, temp6)} \\
\]

X already in temp3, so needn’t reload!

Not because of dumb code!

Different operations (loads, addressing) exposed at different levels

Why Optimization not solved

Machines evolve

New Features
Ex. EPIC, Multithreading, ...
New ways of using machines
Ex. Grids, Peer-to-peer, ...
Changing Costs
Ex. L1 cache misses less costly

Languages evolve

New
Ex. Java
Higher level, special purpose
Ex. Matlab

Applications evolve

Ex. adaptive, web-based,...
Why Optimization not solved
Finding "optimal" undecidable

Most interesting properties of languages are
Undecidable

Ex. Determining whether variable is used assigned a value

Ex. Determining if program terminates

Ex. Determining if 2 programs have same Input/Output behavior

Halting Problem for T.M.

No general algorithm
All optimization techniques are approximate
Good news: better optimization always possible

When is Optimization needed?

High-level               Close to source

  Loop structure evident, e.g., CFG
  Ex. Loop Fusion, Distribution

Medium-level

  Language- and machine independent
  Registers, temporaries exposed
  Ex. Constant propagation, value numbering

Low-level               Close to machine

  May be actual machine instructions
  Ex. Register allocation, Scheduling

Link-time

  Library routines available
  Ex. Interprocedural Register allocation

Run-time

  Runtime constants available
  Ex. Selection of code variants using run-time info
What about optimization order?

**Theory**
- Transformations interact

**Practice**
- Pick "right" order & hope for the best
- Architectural complexity of machines
- Take interactions into account

---

**Theory: Transformations Interact**

Constant Propagation *enables* Dead Code Elimination

**Ex.** $A = 4$

```plaintext
if A < 17 then ...
```

Can eliminate test!
Theory: Transformations Interact

Loop Distribution

Do $i = 1, 100$

\[ a(i) = b(i) + c(i) \]

\[ d(i) = e(i) \times 2 \]

Enddo

Loop Fusion

Enddo

Do $i = 1, 100$

\[ a(i) = b(i) + c(i) \]

Enddo

\[ d(i) = e(i) \times 2 \]

Enddo

---

Theory: Transformations Interact

Instruction scheduling \textit{interferes with} Register allocation

Ex. \[ A = A \]

\[ B = B \]

Instruction level parallelism

Reuse of registers

Integration: [Norris, Pollock], [Bradlee, Eggers, Henry]
Practice: Transformation Order

Pick order $T_1 \rightarrow T_2 \rightarrow \ldots \rightarrow T_k$

Ex.

Enabling:

Constant Propagation

Dead Code Elimination

Theory: Transformation Order

Partial Order: [Whitfield, Soffa]

Constant Propagation

Dead Code Elimination

Code Motion

Loop

Loop Fusion

Interchange

General case: Guidance needed!
**Guidance in Practice**
Separate architectural features optimized separately using independent cost functions

**Ex.**

- Parallelism
- Locality

- Locality
- Parallelism

SUIF [Lam et al] [McKinley et al]

---

**Guidance in Practice**
Single compiler for multiple architectures needs architecture-based guidance

**Ex.**

- Coarse-grain Parallelism:
  - Use Loop Fusion

- Vector processor:
  - Use Loop Distribution
Compiler Challenges
Interaction with other tools, runtime
Higher level, special purpose languages
Optimization tradeoffs, interactions
New platforms
New applications of techniques