Administration

- HW3 due Friday
- Prelim 1 in one week
  - covers topics up through this lecture
  - lexical, syntactic analysis
  - type checking and static semantics
  - syntax-directed translation and IR
- Prelim review Monday evening 7-9 PM
  - location TBA

Where we are

<table>
<thead>
<tr>
<th>source code</th>
<th>parsing</th>
</tr>
</thead>
<tbody>
<tr>
<td>abstract syntax tree (HIR)</td>
<td>syntax-directed translation</td>
</tr>
<tr>
<td>intermediate code (MIR)</td>
<td>syntax-directed translation</td>
</tr>
<tr>
<td>canonical intermediate code (MIR)</td>
<td>code generation</td>
</tr>
<tr>
<td>abstract assembly code (LIR)</td>
<td>register allocation</td>
</tr>
<tr>
<td>assembly code</td>
<td></td>
</tr>
</tbody>
</table>

What makes a good IR?

- Easy to translate from AST
- Easy to translate to assembly
- Narrow interface: small number of node types (instructions)
  - Easy to optimize
  - Easy to retarget

AST (40 node types) → IR (13 node types) → Pentium (>200 opcodes)

Canonical form

- Intermediate code has general tree form
  - easy to generate from AST, but...
- Hard to translate directly to assembly
  - assembly code is a sequence of statements
- Intermediate code (IR) has nodes corresponding to assembly statements deep in expression trees
- Canonical form: all statements brought up to top level of tree — generate assembly directly

One SEQ node

- In canonical form, only one SEQ node at the very top of tree
- Function is just a list of statements: SEQ(s_1, s_2, s_3, s_4, ..., s_n)
- Can translate to assembly by translating each s_i to assembly statement(s) and concatenating
**Canonical form**

- Idea: rewrite IR to get rid of constructs incompatible with assembly code
  - arbitrarily deep expression trees — deal with this later as part of instruction tiling
  - ESEQ & CALL nodes — rewrite tree so no ESEQ nodes, CALLs moved to top
  - C J U M P is two-way jump rather than fall-through — convert to one-way jumps

**Top-level CALL statements**

- CALL nodes have arbitrary side effects
- CALL node deep in expression tree will break translation to assembly
  
  Example: $x = f(g(x) + h(y))$
- Solution: move to top level
- Call that discards return result:
- Call that uses result:

  ```
  ... EXP ... CALL ...
  ...
  ...
  TEM P(t) CALL ...
  ```

**Canonical IR to assembly**

- **Canonical IR**
  - MOVE(dest, e)
  - MOVE(TEMP(t), CALL(...))
  - EXP(CALL(...))
  - JUMP(e)
  - CJUMP(e, l₁, l₂)
  - LABEL(l)

  $⇒$Straightforward translation to assembly
**ESEQ rewriting**

- Want to move ESEQ nodes up to top of tree where they can become SEQ nodes
- Idea: define syntax-directed rules that take an IR tree and move ESEQ nodes to top.
- **Goal:** avoid ripping apart expressions more than necessary -- leads to better code because expression patterns can be recognized and mapped to instruction set

**ESEQ rewrite rules**

- Example transformations:
  - \( \text{ESEQ}(s_1, \text{ESEQ}(s_2, e)) \rightarrow \text{ESEQ}(\text{SEQ}(s_1, s_2), e) \)
  - \( \text{MOVE}(\text{ESEQ}(s_1, e), \text{dest}) \rightarrow \text{SEQ}(s_1, \text{MOVE}(e, \text{dest})) \)
  - \( \text{OP}(\text{ESEQ}(s_1, e_1), e_2) \rightarrow \text{ESEQ}(s_1, \text{OP}(e_1, e_2)) \)
  - \( \text{OP}(e_1, \text{ESEQ}(s_1, e_2)) \rightarrow ? \)

**Rewriting expressions**

- \( \text{OP}(e_1, \text{ESEQ}(s_1, e_2)) \)

\[
\begin{align*}
\text{OP} & \quad ? \\
\text{ESEQ} & \quad \longrightarrow \quad s_1 \\
\text{OP} & \quad e_1 \\
\text{ESEQ} & \quad e_2
\end{align*}
\]

\[
\begin{align*}
e_1 + (a=0; e_2) & \quad \longrightarrow \quad (a=0; e_1 + e_2)
\end{align*}
\]

**Introducing temporaries**

- If \( e_1 \) does not *commute* with \( s_1 \)
  - i.e., \( (s_1; e_1; e_2) \neq (e_1; s_1; e_2) \)
  - must save value of \( e_1 \) in temporary

\[
\begin{align*}
\text{OP} & \quad \text{ESEQ} \\
\text{ESEQ} & \quad \longrightarrow \quad \text{SEQ} \\
\text{ESEQ} & \quad \text{OP} \\
\text{ESEQ} & \quad \text{MOVE} \\
\text{ESEQ} & \quad \text{TEMP(t)} \\
\end{align*}
\]

**Implementation options**

- Option 1: Walk over tree looking for places to apply rewrite rules (can use visitors)
  - "bad" nodes (ESEQ, CALL) percolate upward
  - Problem: need to restart tree traversal at every rewrite
- Option 2: Rewrite whole IR tree in one pass to build canonical IR tree
  - Syntax-directed translation!

**General case**

- When we move all ESEQ nodes to top, arbitrary expression node looks like:
  - ESEQ transformation takes arbitrary expression node, returns list of sub-statements \( s_i \) to be executed (all side effects of \( e \)) plus final expression \( e \) *free of side effects*.
  - Arbitrary statement node becomes a new SEQ node with no ESEQ nodes (or list of sub-statements \( s_i \).)
IR Simplification Interface

class CanonExpr {
    IRStmt[] pre_stmts;
    IRExpr expr;
}

abstract class IRExpr { CanonExpr simplify(); }
abstract class IRStmt { IRStmt[] simplify(); }

Simplification

Two translation functions:
- \( T \) \([\text{expr}] \) gives a list of canonical statements \( s_i \) and a canonical expression \( e' \) where executing the \( s_i \) and then evaluating \( e \) has same side effects and value as \( e' = \text{IRExpr.simplify} \)
- \( T \) \([\text{stmt}] \) gives a list of canonical statements \( s_i \) such that executing the \( s_i \) has same side effects as \( s = \text{IRStmt.simplify} \)

Simplifying a function body

- Last time: translate a function definition \( f(a_1, \ldots, a_n) = e \) as \( \text{SEQ}( \text{MOVE}(\text{TEMP}(\text{RV}), E), \text{LABEL}(\text{epilogue})) \)
- Canonical form: SEQ node with all of \( T \) \([\text{SEQ(MOVE(TEMP(RV), E))], \text{LABEL(\text{epilogue})}] \) as children.

Rules

- Simplify arbitrary expression \( e \):
  \( T \) \([\text{expr}] \) = (\( s_1, \ldots, s_n \), \( e' \))
- Goal: define \( T \) \([\text{expr}] \) and \( T \) \([\text{stmt}] \) for all 13 node types
- 3 trivial cases:
  - \( T \) \([\text{CONST(i)}] \) = (); \( \text{CONST(i)} \)
  - \( T \) \([\text{NAME(n)}] \) = (); \( \text{NAME(n)} \)
  - \( T \) \([\text{TEMP(t)}] \) = (); \( \text{TEMP(t)} \)
- These expressions are already free of side effects: already in canonical form

JUMP, CJUMP, MEM

- JUMP(e), CJUMP(e, l_1, l_2), MEM(e)
- Need to make sure \( e \) is canonical
- \( T \) \([\text{JUMP(e)}] \) = (\( s_1, \ldots, s_n \), JUMP(e'))
  \( T \) \([\text{JUMP(e)}] \) = (\( s_1, \ldots, s_n \), e')
- Similarly for CJUMP
- Can write as inference rule:
  \[
  T \llbracket e \rrbracket = (s_1, \ldots, s_n); e' \\
  T \llbracket \text{MEM(e)} \rrbracket = (s_1, \ldots, s_n); \text{MEM(e')}
  \]

ESEQ

- How to simplify an expression ESEQ(s, e) ?
  \( T \) \([\text{ESEQ(s, e)}] \) = (\( s, s_1, \ldots, s_n \), e')
- What is wrong with this rule?
Correct ESEQ rule

\[ T \llbracket e \rrbracket = (s_1, \ldots, s_n) ; e' \]
\[ T \llbracket s \rrbracket = (s'_1, \ldots, s'_m) \]
\[ T \llbracket \text{ESEQ}(s, e) \rrbracket = (s'_1, \ldots, s'_m, s_1, \ldots, s_n) ; e' \]

- Assuming \( T \llbracket e \rrbracket, T \llbracket s \rrbracket \) produce canonical statements \( s_i, s'_j \) and canonical expression \( e \), \( T \llbracket \text{ESEQ}(s, e) \rrbracket \) works properly.

Translating binary operators

\[ T \llbracket e_1 \rrbracket = (s_1, \ldots, s_m) ; e'_1 \]
\[ T \llbracket e_2 \rrbracket = (s'_1, \ldots, s'_n) ; e'_2 \]
\[ T \llbracket \text{OP}(e_1, e_2) \rrbracket = \]
\[ (s_1, \ldots, s_m, \text{MOVE}(\text{TEMP}(t), e'_1), s'_1, \ldots, s'_n) ; \text{OP}(\text{TEMP}(t), e'_2) \]

- Idea: save value of \( e_i \) in a temporary before executing all the side effects of \( e_2 \)

SEQ nodes

- How to get rid of SEQ nodes: concatenate canonical versions of all sub-statements

\[ T \llbracket s_1 \rrbracket = (s'_1, \ldots, s'_m) \]
\[ T \llbracket s_2 \rrbracket = (s''_1, \ldots, s''_n) \]
\[ T \llbracket \text{SEQ}(s_1, s_2) \rrbracket = (s'_1, \ldots, s'_m, s''_1, \ldots, s''_n) \]

CALL nodes

- CALL nodes call a function which may have side effects
- Overwrites return value register at least; can’t be operand at assembly-code level
- Therefore, CALL nodes must move to top

\[ T \llbracket e_f \rrbracket = (s_1, \ldots, s_m) ; e'_f \]
\[ T \llbracket e_i \rrbracket = (s'_1, \ldots, s'_n) ; e'_i \]
\[ T \llbracket \text{CALL}(e_f, e_i) \rrbracket = \]
\[ (s_1, \ldots, s_m, s'_1, \ldots, s'_n, \text{MOVE}(\text{TEMP}(t), \text{CALL}(e'_f, e'_i)); \text{TEMP}(t)) \]

EXP

- EXP(\( e \)) evaluates \( e \) for its side effects, discards value
- Simplified IR does same:

\[ T \llbracket e \rrbracket = (s_1, \ldots, s_n) ; e' \]
\[ T \llbracket \text{EXP}(e) \rrbracket = (s_1, \ldots, s_n) \]

Canonical intermediate code

- Syntax-directed translation function \( T \llbracket \cdot \rrbracket \) simplifies an IR tree into canonical form
- Yields recursive impl. of \( \text{IRStmt}.\text{Simplify}, \text{IRExpr}.\text{Simplify} \)
- Canonical form: IR is a sequence of simple IR statements ready for translation to assembly