Intermediate Representation

- Intermediate representation = internal representation
  - Is language-independent and machine-independent
- High IR: captures high-level language constructs
- Low IR: captures low-level machine features

```
C
Fortran
HIR   LIR
Java bytecode
Pentium
```

High-level IR

- Tree node structure very similar to the AST
- Contains high-level constructs common to many languages
  - Expression nodes
  - Statement nodes
- Expression nodes for:
  - Integers and program variables
  - Binary operations: \( e_1 \text{ OP } e_2 \)
    - Arithmetic operations
    - Logic operations
    - Comparisons
  - Unary operations: \( e \text{ OP } \)
  - Array accesses: \( e_1[e_2] \)

```

High-level IR

- Statement nodes:
  - Block statements (statement sequences): \( (b_1, ..., b_N) \)
  - Variable assignments: \( v = e \)
  - Array assignments: \( e_1[e_2] = e_3 \)
  - If-then-else statements: if \( c \) then \( s_1 \) else \( s_2 \)
  - If-then statements: if \( c \) then
  - While loops: while \( (c) \) do
  - Function calls: \( f(e_1, ... , e_N) \)
  - Return statements: return or return

- May also contain:
  - For loop statements: \( for (v = e_1 to e_2) \{ \)
  - Break and continue statements
  - Switch statements: \( \text{switch}(e) \{ v_1: s_1, ..., v_N: s_N \} \)

```

Low-level IR

- Represents a set of instructions which emulates an abstract machine
- Arithmetic and logic instructions:
  - Binary: \( a \text{ OP } b \)
  - Arithmetic operations
  - Logic operations
  - Comparisons
  - Unary operations: \( a \text{ OP } b \)
- Data movement instructions:
  - Copy: \( a = b \)
  - Load: \( a = [b] \) (load into the value at address \( b \))
  - Store: \( [a] = b \) (store at address \( a \) the value \( b \))

There is a high IR node for each of the above.
- All AST nodes are translated into the above IR nodes
Low-level IR

- Function call instructions:
  - Call instruction: call f1, f2, ..., fn
  - Call assignment: i = ci (f1, ..., fn)
  - Return instruction: return
  - Value return: return

- Branch instructions:
  - Unconditional jump: jump L
  - Conditional jump: jump L

- These instructions are also called quadruples or three-address instructions

Translating High IR to Low IR

- We need to translate each high-level IR node to a low-level IR sequence of instructions
  - Expressions nodes: arithmetic, logic, comparison, unary, etc.
  - Statements nodes: blocks, if-then-else, if-then, while, function calls, etc.
  - Expression statements nodes: if-then-else, calls, etc.

Notation

- Use the following notation:
  - [e] = the low-level IR representation of high-level IR construct e
  - [e] is a sequence of Low-Level IR instructions
  - If e is an expression (or a statement expression), it represents a value
  - Denote by t = [e] the low-level IR representation of e, whose result value is stored in t
  - For variable v: t = [v] is the copy instruction t = v

Translating Expressions

- Binary operations: t = [ e1 OP e2 ]
  (arithmetic operations and comparisons)
  \[ t1 = [ e1 ] \]
  \[ t2 = [ e2 ] \]
  \[ t = t1 \text{ OP } t2 \]

- Unary operations: t = [ OP e ]
  \[ t1 = [ e ] \]
  \[ t = \text{ OP } t1 \]

Translating Boolean Expressions

- t = [ e1 OR e2 ]
  \[ t1 = [ e1 ] \]
  \[ t2 = [ e2 ] \]
  \[ t = t1 \text{ OR } t2 \]

- ... how about short-circuit OR?
- Should compute e2 only if e1 evaluates to false

Translating Short-Circuit OR

- Short-circuit OR: t = [ e1 SC-OR e2 ]
  \[ t = [ e1 ] \]
  \[ \text{c} \text{jump t Lend } \]
  \[ t = [ e2 ] \]
  \[ \text{label Lend} \]

- ... how about short-circuit AND?
Translating Short-Circuit AND

- Short-circuit AND: \( t = [ \text{e1 SC-AND e2} ] \)
  
  \[
  \begin{align*}
  t &= [ \text{e1} ] \\
  \text{cjump t Lnext} & \quad \text{SC-AND} \\
  \text{jump Lend} & \\
  \text{label Lnext} & \quad \text{e1 e2} \\
  t &= [ \text{e2} ] \\
  \text{label Lend}
  \end{align*}
  \]

Another Translation

- Short-circuit AND: \( t = [ \text{e1 SC-AND e2} ] \)
  
  \[
  \begin{align*}
  t1 &= [ \text{e1} ] \\
  t2 &= \text{not} t1 & \quad \text{SC-AND} \\
  \text{cjump t2 Lend} & \\
  t &= [ \text{e2} ] \\
  \text{label Lend} & \quad \text{e1 e2}
  \end{align*}
  \]

Yet Another Translation

- Use another low-level IR: abstract machine with two kinds of conditional jumps
  - \( \text{tjump c L} \) : jump to L if c is true
  - \( \text{fjump c L} \) : jump to L if c is false
- Short-circuit AND: \( t = [ \text{e1 SC-AND e2} ] \)
  
  \[
  \begin{align*}
  t &= [ \text{e1} ] \\
  \text{fjump t2 Lend} & \quad \text{SC-AND} \\
  \text{t} &= [ \text{e2} ] \\
  \text{label Lend} & \quad \text{e1 e2}
  \end{align*}
  \]

Translating Array Accesses

- Array access: \( t = [\text{v[e]}] \)
  (type of e is array[T] and S = size of T)
  
  \[
  \begin{align*}
  t1 &= \text{addr v} & \quad \text{array} \\
  t2 &= [\text{e}] & \\
  t3 &= t2 * S & \\
  t4 &= t1 + t3 \\
  t &= [t4]
  \end{align*}
  \]

Translating Statements

- Statement sequence: \([ s1; s2; ...; sN ] \)
  
  \[
  \begin{align*}
  [ s1 ] \\
  [ s2 ] \\
  \vdots \\
  [ sN ] & \quad \text{seq} \\
  s1 & \quad s2 \quad ... \quad sN
  \end{align*}
  \]

- IR instructions of a statement sequence = concatenation of IR instructions of statements

Assignment Statements

- Variable assignment: \([ \text{v = e} ] \)
  \( v = [\text{e}] \)

- Array assignment: \([\text{v[e]} = \text{e2} ] \)
  \( t1 = \text{addr v} \)
  \( t2 = [\text{e1}] \)
  \( t3 = t2 * S \)
  \( t4 = t1 + t3 \)
  \( t5 = [\text{e2}] \)
  \( [t4] = t5 \)
Translating If-Then-Else

- \([If\ (e)\ then\ s1\ else\ s2]\)

  \[
  t1 = [\ e ]
  \]

  fjump t1 Lfalse

  \[
  [\ s1 ]
  \]

  jump Lfalse

  label Lfalse

  \[
  [\ s2 ]
  \]

  label Lend

Translating If-Then

- \([If\ (e)\ then\ s]\)

  \[
  t1 = [\ e ]
  \]

  fjump t1 Lend

  \[
  [\ s ]
  \]

  label Lend

While Statements

- \([While\ (e)\ \{\ s\}]\)

  \[
  label \text{ltest}
  \]

  \[
  t1 = [\ e ]
  \]

  fjump t1 Lend

  \[
  [\ s ]
  \]

  jump \text{ltest}

  label Lend

Switch Statements

- \([\ Switch\ (e)\ \{\ case\ \ v1: \ s1, ...;\ case\ vN: \ sN;\}]\)

  \[
  t = [\ e ]
  \]

  c = \text{tl} \leftarrow \text{v1}

  jump c L2

  \[
  [\ s1 ]
  \]

  jump Lend

  labeled L2

  c = \text{tl} \leftarrow \text{v2}

  jump c L3

  \[
  [\ s2 ]
  \]

  jump Lend

  labeled LN

  c = \text{tl} \leftarrow \text{vN}

  jump c Lend

  \[
  [\ sN ]
  \]

  labeled LN

  \[
  [\ sN ]
  \]

Call and Return Statements

- \([\ Call\ f(e1, e2, ..., eN)]\)

  \[
  t1 = [\ e1 ]
  \]

  \[
  t2 = [\ e2 ]
  \]

  ... 

  \[
  tN = [\ eN ]
  \]

  call (t1, t2, ..., tN)

- \([Return\ e]\)

  \[
  t = [\ e ]
  \]

  return t

Statement Expressions

- So far: statements which do not return values
- Easy extensions for statement expressions:
  - Block statements
  - If-then-else
  - Assignment statements
- \([t = [s]]\) is the sequence of low IR code for statement \(s\), whose result is stored in \(t\).
<table>
<thead>
<tr>
<th>Statement Expressions</th>
</tr>
</thead>
<tbody>
<tr>
<td>• ( t = [ \text{if}(e) \text{then} s_1 \text{else} s_2 ]</td>
</tr>
<tr>
<td>( t_1 = [ e ] )</td>
</tr>
<tr>
<td>( \text{assign} )</td>
</tr>
<tr>
<td>( \text{jump t}_1 \text{Ltrue} )</td>
</tr>
<tr>
<td>( t = [ s_2 ] )</td>
</tr>
<tr>
<td>( \text{if-then-else} )</td>
</tr>
<tr>
<td>( \text{jump Lend} )</td>
</tr>
<tr>
<td>( t = [ s_1 ] )</td>
</tr>
<tr>
<td>( \text{label Lend} )</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Block Statements</th>
</tr>
</thead>
<tbody>
<tr>
<td>• ( t = [ s_1; s_2; \ldots; s_N ] )</td>
</tr>
<tr>
<td>( [ s_1 ] )</td>
</tr>
<tr>
<td>( \text{seq} )</td>
</tr>
<tr>
<td>( [ s_2 ] )</td>
</tr>
<tr>
<td>( \ldots )</td>
</tr>
<tr>
<td>( t = [ s_N ] )</td>
</tr>
<tr>
<td>( \text{Result value of a block statement = value of last statement in the sequence} )</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Assignment Statements</th>
</tr>
</thead>
<tbody>
<tr>
<td>• ( t = [ v = e ] )</td>
</tr>
<tr>
<td>( v = [ e ] )</td>
</tr>
<tr>
<td>( \text{assign} )</td>
</tr>
<tr>
<td>( t = v )</td>
</tr>
<tr>
<td>( v )</td>
</tr>
<tr>
<td>( e )</td>
</tr>
<tr>
<td>( \text{Result value of an assignment statement = value of the assigned expression} )</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Nested Expressions</th>
</tr>
</thead>
<tbody>
<tr>
<td>• In these translations, expressions may be nested;</td>
</tr>
<tr>
<td>• Translation recursively on the expression structure</td>
</tr>
<tr>
<td>Example: ( t = [(a - b) * (c + d)] )</td>
</tr>
<tr>
<td>( t_1 = a )</td>
</tr>
<tr>
<td>( t_2 = b )</td>
</tr>
<tr>
<td>( t_3 = t_1 - t_2 )</td>
</tr>
<tr>
<td>( t_4 = b )</td>
</tr>
<tr>
<td>( t_5 = c )</td>
</tr>
<tr>
<td>( t_6 = t_4 + t_5 )</td>
</tr>
<tr>
<td>( t = t_3 * t_5 )</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Nested Statements</th>
</tr>
</thead>
<tbody>
<tr>
<td>• Same for statements: recursive translation</td>
</tr>
<tr>
<td>Example: ( [\text{if } c \text{ then if } d \text{ then } a = b ] )</td>
</tr>
<tr>
<td>( t_1 = c )</td>
</tr>
<tr>
<td>( \text{jump t}_1 \text{Lend} )</td>
</tr>
<tr>
<td>( t_2 = d )</td>
</tr>
<tr>
<td>( \text{jump t}_2 \text{Lend} )</td>
</tr>
<tr>
<td>( t_3 = b )</td>
</tr>
<tr>
<td>( a = t_3 )</td>
</tr>
<tr>
<td>( \text{label Lend} )</td>
</tr>
<tr>
<td>( \text{label Lend} )</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Issues</th>
</tr>
</thead>
<tbody>
<tr>
<td>• These translations are straightforward</td>
</tr>
<tr>
<td>• and inefficient:</td>
</tr>
<tr>
<td>• May generate many temporary variables</td>
</tr>
<tr>
<td>• May generate many labels</td>
</tr>
<tr>
<td>• Can optimize translation process:</td>
</tr>
<tr>
<td>• Don't create temporary variables</td>
</tr>
<tr>
<td>• Reuse temporary variables</td>
</tr>
<tr>
<td>• Merge adjacent labels</td>
</tr>
</tbody>
</table>
Optimize Translation

- Example: \( t = [(a - b) \times (c + d)] \)

- \( t_1 = a \)
- \( t_2 = b \)
- \( t_3 = t_1 - t_2 \)
- \( t_4 = b \)
- \( t_5 = c \)
- \( t_5 = t_4 + t_5 \)
- \( t = t_3 \times t_5 \)