**Intermediate Code**

- IR = Intermediate Representation
- Allows language-independent, machine-independent optimizations and transformations

```
AST → IR
```

**Multiple IRs**

- Usually two IRs:
  - High-level IR: Language-independent (but closer to language)
  - Low-level IR: Machine independent (but closer to machine)

```
C
Fortran → HIR → LIR → Java bytecode → Alpha
```

**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: \( e_1 \ OP \ e_2 \)
    - Arithmetic operations
    - Logic operations
    - Comparisons
  - Unary operations: \( e \ OP \)
  - Array accesses: \( e_1[e_2] \)

**High-level IR**

- Statements may be expressions
- Statement expression nodes:
  - Block statements: \( (s_1, ... s_N) \)
  - 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-else statements: \( if \ c \ then \ s \)
  - While loops: \( while (c) \ s \)
  - Function call statements: \( f(e_1, ..., e_N) \)
  - Return statements: \( return \ or \ return \ e \)

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

- Low-level representation is essentially an instruction set for an abstract machine
- Alternatives for low-level IR:
  - Three-address code or quadruples (Dragon Book):
    \[ a = b \text{ OP } c \]
  - Tree representation (Tiger Book)
  - Stack machine (like Java bytecode)

Three-Address Code

- In this class: three-address code
  \[ a = b \text{ OP } c \]
- Has at most three addresses (may have fewer)
- Also named quadruples because can be represented as:
  \( (a, b, c, \text{ OP}) \)
- Example:
  \[ a = (b+c)^*(-e); \quad t1 = b + c \]
  \[ t2 = -e \]
  \[ a = t1 * t2 \]

Low IR Instructions

- Assignment instructions:
  - Binary operations: \( a = b \text{ OP } c \)
  - arithmetic: ADD, SUB, MUL, DIV, MOD
  - logic: AND, OR, XOR
  - comparisons: EQ, NEQ, LT, GT, LEQ, GEQ
  - Unary operation \( a = \text{ OP } b \)
  - Arithmetic MINUS or logic NEG
- Copy instruction: \( a = b \)
- Load/store: \( a = *b \), \( *a = b \)
- Other data movement instructions

Low IR Instructions (Ctd)

- Flow of control instructions:
  - label \( L \): label instruction
  - jump \( L \): Unconditional jump
  - jump \( a \): conditional jump
- Function call
  - call \( f(a_1, \ldots, a_n) \)
  - \( a = \text{ call } f(a_1, \ldots, a_n) \)
  - Is an extension to quads
- ... IR describes the Instruction Set of an abstract machine

Temporary Variables

- The operands in the quadruples can be:
  - Program variables
  - Integer constants
  - Temporary variables
- Temporary variables = new locations
  - Use temporary variables to store intermediate values

Arithmetic / Logic Instructions

- Abstract machine supports a variety of different operations
  \[ a = b \text{ OP } c \quad a = \text{ OP } b \]
- Arithmetic operations: ADD, SUB, DIV, MUL
- Logic operations: AND, OR, XOR
- Comparisons: EQ, NEQ, LE, LEQ, GE, GEQ
- Unary operations: MINUS, NEG
Data Movement

- Copy instruction: \( a = b \)
- Load/store instructions:
  - \( a = \* b \)
  - \( \* a = b \)
  - Models a load/store machine
- Address-of instruction: \( a = \& b \)
- Array accesses:
  - \( a = b[i] \)
  - \( a[i] = b \)
- Field accesses:
  - \( a = b.f \)
  - \( a.f = b \)

Branch Instructions

- Label instruction: \( \text{label L} \)
- Unconditional jump: go to statement after label L
  - \( \text{jump L} \)
- Conditional jump: test condition variable \( a \); if true, jump to label L
  - \( \text{cjump a L} \)
- Alternative: two conditional jumps:
  - \( \text{tjump a L} \)
  - \( \text{fjump a L} \)

Call Instruction

- Supports function call statements
  - \( \text{call f}(a_1, \ldots, a_n) \)
- ... and function call assignments:
  - \( a = \text{call f}(a_1, \ldots, a_n) \)
- No explicit representation of argument passing, stack frame setup, etc.

Example

\[
\begin{align*}
  n &= 0; \\
  \text{while } (n < 10) \{ \\
  &\quad n = n + 1 \\
  \} \\
\end{align*}
\]

Another Example

\[
\begin{align*}
  m &= 0; \\
  \text{if } (c == 0) \{ \\
  &\quad m = m + n \times n; \\
  \} \quad \text{else} \{ \\
  &\quad m = m + n; \\
\end{align*}
\]

How To Translate?

- May have nested language constructs
  - Nested if and while statements
- Need an algorithmic way to translate
- Solution:
  - Start from the AST representation
  - Define translation for each node in the AST
  - Recursively translate nodes in the AST
Notation

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

Translating Expressions

- Binary operations: \( t = T[e1 \ OP \ e2] \)
  (arithmetic operations and comparisons)
  \[
  \begin{align*}
  t1 &= T[e1] \\
  t2 &= T[e2] \\
  t &= t1 \ OP \ t2
  \end{align*}
  \]
- Unary operations: \( t = T[\ OP \ e] \)
  \[
  \begin{align*}
  t1 &= T[e] \\
  t &= OP \ t1
  \end{align*}
  \]

Translating Boolean Expressions

- \( t = T[e1 \ OR \ e2] \)
  \[
  \begin{align*}
  t1 &= T[e1] \\
  t2 &= T[e2] \\
  t &= t1 \ OR \ t2
  \end{align*}
  \]
- ... how about short-circuit OR?
- Should compute \( e2 \) only if \( e1 \) evaluates to false

Translating Short-Circuit OR

- Short-circuit OR: \( t = T[e1 \ SC-OR \ e2] \)
  \[
  \begin{align*}
  t &= T[e1] \\
  t &= T[e2] \\
  t &= T[e2] \\
  &\ text{label Lend}
  \end{align*}
  \]
- ... how about short-circuit AND?

Translating Short-Circuit AND

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

Another Translation

- Short-circuit AND: \( t = T[e1 \ SC-AND \ e2] \)
  \[
  \begin{align*}
  t &= T[e1] \\
  t &= T[e2] \\
  &\ text{label Lend}
  \end{align*}
  \]
  
  \[
  \begin{align*}
  t &= T[e1] \\
  &\ text{jump t Lnext}
  \end{align*}
  \]
Array and Field Accesses

- **Array access**: \( t = T[ v[e] ] \)
  \( t1 = T[ e ] \)
  \( t = v[t1] \)

- **Field access**: \( t = T[ e1.e2 ] \)
  \( t1 = T[ e1 ] \)
  \( t2 = T[ e2 ] \)
  \( t = t1.t2 \)

Translating Statements

- **Statement sequence**: \( T[ s1; s2; \ldots; sN ] \)
  \( T[ s1 ] \)
  \( T[ s2 ] \)
  \( \ldots \)
  \( T[ sN ] \)
  \( s1 \)
  \( s2 \)
  \( \ldots \)
  \( sN \)

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

Assignment Statements

- **Variable assignment**: \( T[ v = e ] \)
  \( v = T[ e ] \)

- **Array assignment**: \( T[ v[e1] = e2 ] \)
  \( t1 = T[ e1 ] \)
  \( t2 = T[ e2 ] \)
  \( v[t1] = t2 \)

Translating If-Then-Else

- **\( T[ if (e) then s1 else s2 ] \)**
  \( t1 = T[ e ] \)
  \( fjump t1 Lfalse \)
  \( T[ s1 ] \)
  \( jump Lend \)
  \( Lfalse \)
  \( T[ s2 ] \)
  \( Lend \)

  \( if-then-else \)
  \( e \)
  \( s1 \)
  \( s2 \)

Translating If-Then

- **\( T[ if (e) then s ] \)**
  \( t1 = T[ e ] \)
  \( fjump t1 Lfalse \)
  \( T[ s ] \)
  \( Lfalse \)
  \( t1 Lend \)
  \( label Lend \)

  \( if-then \)
  \( e \)
  \( s \)

While Statements

- **\( T[ while (e) \{ s \} ] \)**
  \( label Ltest \)
  \( t1 = T[ e ] \)
  \( fjump t1 Lfalse \)
  \( T[ s ] \)
  \( jump Ltest \)
  \( Lfalse \)
  \( s \)
  \( Ltest \)
  \( Lend \)
Switch Statements

• \( \text{T[ switch (e) { case v1: s1, ..., case vN: sN } ]} \)

\[
\begin{align*}
t &= \text{T[ e ]} \\
c &= t =: v1 \\
\text{jump c L2} \\
\text{T[ s1 ]} \\
\text{jump Lend} \\
\text{label L2} \\
c &= t =: v2 \\
\text{jump c L3} \\
\text{T[ s2 ]} \\
\text{jump Lend} \\
\text{label L3} \\
c &= t =: vN \\
\text{jump c Lend} \\
\text{T[ sN ]} \\
\text{label Lend}
\end{align*}
\]

Call and Return Statements

• \( \text{T[ call f(e1, e2, ..., eN) ]} \)

\[
\begin{align*}
t1 &= \text{T[ e1 ]} \\
t2 &= \text{T[ e2 ]} \\
\vdots \\
tN &= \text{T[ eN ]} \\
\text{call f(t1, t2, ..., tN)}
\end{align*}
\]

• \( \text{T[ return e ]} \)

\[
\begin{align*}
t &= \text{T[ e ]} \\
\text{return t} \\
\text{return e}
\end{align*}
\]

Statement Expressions

• So far: statements which do not return values
  • Easy extensions for statement expressions:
    - Block statements
    - If-then-else
    - Assignment statements

• \( t = \text{T[ s ]} \) is the sequence of low IR code for statement \( s \), whose result is stored in \( t \)

Statement Expressions

• \( t = \text{T[ if (e) then s1 else s2 ]} \)

\[
\begin{align*}
t1 &= \text{T[ e ]} \\
\text{jump t1 Ltrue} \\
t &= \text{T[ s2 ]} \\
\text{jump Lend} \\
lable Ltrue \\
t &= \text{T[ s1 ]} \\
\text{label Lend}
\end{align*}
\]

Block Statements

• \( t = [ s1; s2; ...; sN ] \)

\[
\begin{align*}
\text{[ s1 ]} \\
\text{[ s2 ]} \\
\text{...}
\end{align*}
\]

• Result value of a block statement = value of last statement in the sequence

Assignment Statements

• \( t = [ v = e ] \)

\[
\begin{align*}
v &= [ e ] \\
t &= v
\end{align*}
\]

• Result value of an assignment statement = value of the assigned expression