Intermediate Code

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

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

Multiple IRs

- Some optimizations require high-level structure
- Others more appropriate on low-level code

Multiple IRs

- Some optimizations require high-level structure
- Others more appropriate on low-level code
- Solution: use multiple IR stages
Machine Optimizations

- ... some other optimizations take advantage of the features of the target machine
- Machine-specific optimizations

AST → HIR → LIR

optimize

Pentium

optimize

Java bytecode

optimize

Sparc

Next Lectures

- Next few lectures: intermediate representation
- Optimizations covered later

AST → HIR → LIR

Pentium

Java bytecode

Sparc

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

Pentium

Java bytecode

Sparc

Pascal

Multiple IRs

- Another benefit: a significant part of the translation from high-level to low-level is
  - Language-independent
  - Machine-independent

C

Fortran → HIR → LIR

Pentium

Java bytecode

Sparc

Pascal

High-Level IR

- High-level intermediate representation is essentially the AST
  - Must be expressive for all input languages

- Preserves high-level language constructs
  - Structured control flow: if, while, for, switch, etc.
  - variables, methods

- Allows high-level optimizations
  - E.g., optimizations of nested “for” loops

Low-Level IR

- Low-level representation is essentially an abstract machine

- Has low-level constructs
  - Unstructured jumps, instructions

- Allows optimizations specific to these constructs (e.g. register allocation, branch prediction)
Low-Level IR

- Alternatives for low-level IR:
  - Three-address code or quadruples (Dragon Book):
    \[ a = b \OP c \]
  - Tree representation (Appel Book)
  - Mixed: three address for expressions and flat representation of control-flow
  - Stack machine (similar to Java bytecodes)

- Advantages:
  - Three-address code: easier dataflow analysis
  - Tree IR: easier instruction selection
  - Stack machine: better if the target has a stack model

Three-Address Code

- In this class: three-address code
  \[ a = b \OP c \]

- Has at most three addresses (may have fewer)
- Also named quadruples because can be represented as: \((a, b, c, \OP)\)

- Example:
  \[ a = (b+c)*(-e); \quad t1 = b + c \]
  \[ t2 = -e \]
  \[ a = t1 * t2 \]

Arithmetic / Logic Instructions

- Abstract machine supports a variety of different operations
  \[ a = b \OP c \quad a = \OP b \]

- 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:
  \[ a = \#b \quad \#a = b \]
  - Models a load/store machine

- Address-of instruction (if language supports it):
  \[ a = \&b \]

- Array accesses:
  \[ a = b[i] \quad a[i] = b \]

- Field accesses:
  \[ a = b.f \quad a.f = b \]

Call Instruction

- Supports function call statements
  \[ \text{call } f(a_1, ..., a_n) \]

- ... and function call assignments:
  \[ a = \text{call } f(a_1, ..., a_n) \]

- No explicit representation of argument passing, stack frame setup, etc.
Example

```plaintext
n = 0;
label test
while (n < 10) {
  n = n + 1
}
```

Another Example

```plaintext
m = 0;
t1 = c == 0
  cjump t1 trueb
m = m + n
} else {
  m = m + n;
}
```

How To Translate?

- May have nested language constructs
  - Nested if and while statements
- Solution: syntax-directed translation
  - 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 code for high-level IR construct $e$
  - $T[i]$ is a sequence of Low-level IR instructions
- If $e$ is an expression, 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 $t = v$
- Temporary variables = new locations
  - Use temporary variables to store intermediate values during this translation

Translating Expressions

- Binary operations: $t := T[ e_1 \text{ OP } e_2 ]$
  (arithmetic operations and comparisons)
  ```plaintext
t1 := T[ e1 ]
t2 := T[ e2 ]
t = t1 \text{ OP } t2
```
- Unary operations: $t = T[ \text{ OP } e ]$
  ```plaintext
t1 := T[ e ]
t = \text{ OP } t1
```

Translating Boolean Expressions

- $t := T[ e_1 \text{ OR } e_2 ]$
  ```plaintext
t1 := T[ e1 ]
t2 := T[ e2 ]
t = t1 \text{ OR } t2
```
  - ... how about short-circuit OR?
  - Should compute $e_2$ only if $e_1$ evaluates to false
Translating Short-Circuit OR

- Short-circuit OR: \( t := T[ e1 \ SC-OR \ e2 ] \)
  
  \[
  \begin{align*}
  t := T[ e1 ] \\
  tjump \ t \ \text{Lend} \quad \text{SC-OR} \\
  t := T[ e2 ] \\
  \text{label Lend}
  \end{align*}
  \]

- ... how about short-circuit AND?

Array and Field Accesses

- Array access: \( t := T[ v[e] ] \)
  
  \[
  \begin{align*}
  t := T[ e ] \\
  t := v[t1]
  \end{align*}
  \]

- Field access: \( t := T[ e1.f ] \)
  
  \[
  \begin{align*}
  t1 := T[ e1 ] \\
  t := t1.f
  \end{align*}
  \]

Translating Statements

- Statement sequence: \( T[ s1; s2; \ldots; sN ] \)
  
  \[
  \begin{align*}
  T[ s1 ] \\
  T[ s2 ] \\
  \ldots \\
  T[ sN ]
  \end{align*}
  \]

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

Assignment Statements

- Variable assignment: \( T[ v = e ] \)
  
  \[
  \begin{align*}
  v := T[ e ]
  \end{align*}
  \]

- Array assignment: \( T[ v[e1] = e2 ] \)
  
  \[
  \begin{align*}
  t1 := T[ e1 ] \\
  t2 := T[ e2 ] \\
  v[t1] = t2
  \end{align*}
  \]

Translating If-Then-Else

- \( T[ \text{if} \ (e) \ \{ \ s1 \} \ \text{else} \ { \ s2 } ] \)
  
  \[
  \begin{align*}
  t1 := T[ e ] \\
  fjump \ t1 \ \text{Lfalse} \\
  T[ s1 ] \\
  \text{if-then-else} \\
  \text{jump Lend} \\
  \text{Lfalse} \\
  T[ s2 ] \\
  \text{label Lend}
  \end{align*}
  \]

Translating If-Then

- \( T[ \text{if} \ (e) \ \{ \ s \} ] \)
  
  \[
  \begin{align*}
  t1 := T[ e ] \\
  fjump \ t1 \ \text{Lend} \\
  T[ s ] \\
  \text{if-then} \\
  \text{label Lend}
  \end{align*}
  \]
While Statements

- $T[\text{ while (e) \{ s \} }]$
  
  label Ltest
  $t_1 := T[ e ]$
  fjump $t_1$ Lend
  $T[ s ]$
  jump Ltest
  label Lend

Call and Return Statements

- $T[\text{ call f(e1, e2, ..., en) } ]$
  $t_1 := T[ e_1 ]$
  $t_2 := T[ e_2 ]$
  ...
  $t_n := T[ e_n ]$
  call $f(t_1, t_2, ..., t_n)$

- $T[\text{ return e } ]$
  $t := T[ e ]$
  return $t$

\[ \text{call} \]
\[ \text{return} \]