Intermediate Code
Where We Are

Source code (character stream)

Lexical analysis

Syntactic Analysis

Token stream

Abstract syntax tree

Semantic Analysis

Abstract syntax tree + symbol tables, types

Intermediate Code Generation

Intermediate Code

regular expressions

grammars

static semantics
Intermediate Code

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

  - HIR
  - LIR

  - C
  - Fortran
  - Pascal
  - Pentium
  - Java bytecode
  - PowerPC
High-level IR

- **Tree node structure**, essentially **ASTs**
- High-level constructs common to many languages
  - Expression nodes
  - Statement nodes

- **Expression nodes for:**
  - Integers and program variables
  - Binary operations: `e1 OP e2`
    - Arithmetic operations
    - Logic operations
    - Comparisons
  - Unary operations: `OP e`
  - Array accesses: `e1[e2]`
High-level IR

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

- May also contain:
  - For loop statements: \(\text{for}(v = e_1 \text{ to } e_2) \ s\)
  - Break and continue statements
  - Switch statements: \(\text{switch}(e) \{ \ v_1: s_1, ..., v_N: s_N \ \}\)
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: \( a = b \ OP \ c \)
    • arithmetic: ADD, SUB, MUL, DIV, MOD
    • logic: AND, OR, XOR
    • comparisons: EQ, NEQ, LT, GT, LEQ, GEQ
  – Unary operation \( a = \ OP \ b \)
    • Arithmetic MINUS or logic NEG
  – Copy instruction: \( a = b \)
  – Load /store: \( a = *b, *a = b \)
  – Other data movement instructions
Low IR Instructions, cont.

- Flow of control instructions:
  - label L : label instruction
  - jump L : Unconditional jump
  - tjump a L : conditional jump if a is true
  - fjump a L : conditional jump if a is false

- Function call
  - call f(a_1, ..., a_n)
  - a = call f(a_1, ..., a_n)
  - is an extension to quads

- ... IR describes the Instruction Set of an abstract machine
Example

```c
m = 0;
if (c == 0) {
    m = m + n*n;
} else {
    m = m + n;
}
```

```assembly
m = 0
m = n + m
m = m + (n * n)
fjump t1 falseb
m = m + n
label falseb
```

```assembly
m = 0
t1 = (c == 0)
t2 = n * n
m = m + t2
jump end
label falseb
m = m + n
label end
```
LLVM IR

• Go to this website for example
  http://ellcc.org/demo/index.cgi
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 in terms of a (recursive) translation of its constituents
Notation

- Use the notation $T[e] = \text{low-level IR of high-level IR construct } e$
- $T[e]$ is sequence of low-level IR instructions
- If $e$ is expression (or statement expression), $T[e]$ represents a value
- Denote by $t = T[e]$ the low-level IR of $e$, whose result value is stored in $t$
- For variable $v$, define $T[v]$ to be $v$, i.e., $t = T[v]$ is copy instruction $t = v$
Translating Expressions

- **Binary operations**: \( t = T[e_1 \ OP \ e_2] \)
  (arithmetic operations and comparisons)
  \[
  \begin{align*}
  t_1 &= T[e_1] \\
  t_2 &= T[e_2] \\
  t &= t_1 \ OP \ t_2
  \end{align*}
  \]

- **Unary operations**: \( t = T[\ OP \ e] \)
  \[
  \begin{align*}
  t_1 &= T[e] \\
  t &= \ OP \ t_1
  \end{align*}
  \]
Translating Boolean Expressions

- \( t = T[e_1 \text{ OR } e_2] \)

  \[
  \begin{align*}
  t_1 &= T[e_1] \\
  t_2 &= T[e_2] \\
  t &= t_1 \text{ OR } t_2
  \end{align*}
  \]

- ... but how about short-circuit OR, for which we should compute \( e_2 \) only if \( e_1 \) evaluates to false
Translating Short-Circuit OR

• Short-circuit OR: \( t = T[ e_1 \text{ SC-OR } e_2 ] \)

\[
\begin{align*}
t & = T[ e_1 ] \\
t & \text{jump } t \text{ Lend} \\
t & = T[ e_2 ] \\
\text{label Lend}
\end{align*}
\]

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

• Short-circuit AND: \( t = T[ \text{e1 SC-AND e2} ] \)

\[
\begin{align*}
t &= T[ \text{e1} ] \\
fjump t \text{ Lend} \\
t &= T[ \text{e2} ] \\
\text{label Lend}
\end{align*}
\]
Array and Field Accesses

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

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

• In these translations, expressions may be nested;
• Translation recurses on the expression structure

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

\[
\begin{align*}
  t1 &= a \\
  t2 &= b \\
  t3 &= t1 - t2 \\
  t4 &= b \\
  t5 &= c \\
  t5 &= t4 + t5 \\
  t &= t3 \times t5
\end{align*}
\]

\( T[ (a - b) ] \)

\( T[ (c + d) ] \)

\( T[ (a - b) \times (c + d) ] \)
Translating Statements

• Statement sequence: \[ T[ s1; s2; \ldots; sN ] \]

\[ T[ s1 ] \]
\[ T[ s2 ] \]
\[ \ldots \]
\[ T[ sN ] \]

• IR instructions of a statement sequence = concatenation of IR instructions of statements
Assignment Statements

• Variable assignment: $T[v = e]$  
  
  $t = T[e]$  
  
  $v = t$  
  
  [alternatively]  
  
  $v = T[e]$  

• Array assignment: $T[v[e_1] = e_2]$  
  
  $t_1 = T[e_1]$  
  
  $t_2 = T[e_2]$  
  
  $v[t_1] = t_2$
Translating If-Then-Else

- \[ T[ \text{if (e) then s1 else s2} ] \]

  \[
t1 = T[ e ] \\
fjump t1 Lfalse \\
T[ s1 ] \\
jump Lend \\
label Lfalse \\
T[ s2 ] \\
label Lend
\]
Translating If-Then

- $T[\text{if (e) then s}]$

  $$t1 = T[e]$$
  $$\text{fjump t1 Lend}$$
  $$T[s]$$
  $$\text{label Lend}$$
While Statements

- \( T[ \text{while} \text{ (e)} \{ \text{s} \} ] \)

  - label Ltest
  - \( t1 = T[ e ] \)
  - fjump t1 Lend
  - \( T[ s ] \)
  - jump Ltest
  - label Lend
Switch Statements

- \( T[\ switch (e) \{ \ case v1: s1, \ldots, \ case vN: sN \} \] \)

\[
\begin{align*}
  t &= T[ e ] \\
  c &= t \neq v1 \\
  &\text{tjump} \ c \ L2 \\
  &T[ \ s1 ] \\
  &\text{jump} \ L2 \\
  &\text{label} \ L2 \\
  c &= t \neq v2 \\
  &\text{tjump} \ c \ L3 \\
  &T[ \ s2 ] \\
  &\text{jump} \ Lend \\
  \ldots \\
  c &= t \neq vN \\
  &\text{tjump} \ c \ Lend \\
  &T[ \ sN ] \\
  &\text{label} \ Lend
\end{align*}
\]
Call and Return Statements

- \( T[ \text{call } f(e_1, e_2, \ldots, e_N) ] \)
  
  \[
  \begin{align*}
  t_1 &= T[e_1] \\
  t_2 &= T[e_2] \\
  \vdots \\
  t_N &= T[e_N] \\
  \text{call } f(t_1, t_2, \ldots, t_N)
  \end{align*}
  \]

- \( T[\text{return } e] \)
  
  \[
  \begin{align*}
  t &= T[e] \\
  \text{return } t
  \end{align*}
  \]
Nested Statements

- Same for statements as expressions: recursive translation

Example: $T[\text{if } c \text{ then if } d \text{ then } a = b ]$

$$
t_1 = c
fjump t_1 \text{ Lend1}
t_2 = d
fjump t_2 \text{ Lend2}
t_3 = b
a = t_3
label \text{ Lend2}
label \text{ Lend1}
$$
IR Lowering Efficiency

t1 = c
fjump t1 Lend1
t2 = d
fjump t2 Lend2
t3 = b
a = t3
label Lend2
label Lend1

c if-then
d =
a b

fjump c Lend
fjump d Lend
a = b
Label Lend