Intermediate code

- Intermediate code provides an abstraction which can be produced by the front-end, and consumed by the back-end.
- **Front end** – produces IR of source program
- **Back end** – generates target code from IR
- Optimizations may operate on the IR
**IR benefits and drawbacks**

- Break compiler into manageable pieces
  - simpler pieces
  - more modularity
- Easier re-targeting
- Complete pass before emitting code
  - \(\Rightarrow\) better code
- Allows for language-independent and machine-independent optimizations
- **Drawback:** Another step \(\Rightarrow\) loss in efficiency

**Intermediate Representation (IR)**

- The compilers internal representation
  - Is language-independent and machine-independent

Enables machine independent and machine dependent optimizations

- AST
- IR
- optimize
- Pentium
- Java bytecode
- Itanium
- TI C5x
- ARM
Can there be one general purpose IR?

- Captures high-level language constructs
  - Easy to translate from AST
  - Supports high-level optimizations
- Captures low-level machine features
  - Easy to translate to assembly
  - Supports machine-dependent optimizations
- Narrow interface: small number of node types (instructions)
  - Easy to optimize
  - Easy to retarget

What Makes a Good IR?
Multiple IRs

- Most compilers use 2 IRs:
  - High-level IR (HIR): Language independent but closer to the language
  - Low-level IR (LIR): Machine independent but closer to the machine
  - A significant part of the compiler is both language and machine independent!

Intermediate Representation Categories

- Structural (High-level IR)
  - graph-based or tree-based
  - convenient for high-level transformations
  - may require more storage space

- Linear (Low-level IR)
  - pseudo-code for abstract machine
    - e.g., stack machine, RTL from gcc (Register Transfer Language)
  - large variation in level of abstraction
  - simple, compact data structures

- Hybrids
  - combination of graph & linear code
  - examples: control flow graphs
High–Level IR

- HIR is essentially the AST
  - Must be expressive for all input languages
- Preserves high-level language constructs
  - Structured control flow: if, while, for, switch
  - Variables, expressions, statements, functions
- Allows high-level optimizations based on properties of source language
  - Function inlining, memory dependence analysis, loop transformations

**Source**
```c
float a[10][20];
a[i][j+2];
```

**High-level IR**
- \( t_1 = a[i, j+2] \)

**Middle-level IR**
- \( t_1 = j + 2 \)
- \( t_2 = i * 20 \)
- \( t_3 = t_1 + t_2 \)
- \( t_4 = 4 + t_3 \)
- \( t_5 = \text{addr} a \)
- \( t_6 = t_5 + t_4 \)
- \( t_7 = \ast t_6 \)

**Low-level IR**
- \( r_1 = \lfloor f_p - 4 \rfloor \)
- \( r_2 = \lfloor r_1 + 2 \rfloor \)
- \( r_3 = \lfloor f_p - 8 \rfloor \)
- \( r_4 = r_3 * 20 \)
- \( r_5 = r_4 + r_2 \)
- \( r_6 = 4 * r_5 \)
- \( r_7 = f_p - 216 \)
- \( f_1 = \lfloor r_7 + r_6 \rfloor \)
Low-Level IR

- A set of instructions which emulates an abstract machine (typically RISC: Reduced instruction set computing)
- Has low-level constructs
  - Unstructured jumps, registers, memory locations
- Types of instructions
  - Arithmetic/logic (a = b OP c), unary operations, data movement (move, load, store), function call/return, branches
- Allows for machine-specific optimizations
  - E.g., register allocation

Alternatives for LIR

- 3 general alternatives
  - Three-address code or quadruples
    - a = b OP c
    - Advantage: Makes compiler analysis/opt easier
  - Low-level tree representation
    - Was popular for CISC (complex instruction set computer) architectures
    - Advantage: Easier to generate machine code
  - Stack machine
    - Like Java bytecode
    - Advantage: Easy to generate, compact representation
    - Disadvantage: Difficult to optimize directly
Three-Address Code (Quadruples)

- OP: y, z, x
  - operation
  - operands: x, y, z
  - result

- Has three namesAddresses (x, y, z), or less
- A single operator (OP)
- We will write as: x ← y OP z

**Example:**

\[
x \leftarrow (y + z) \times (-r);
\]

\[
\begin{align*}
t1 & \leftarrow y + z \\
t2 & \leftarrow -r \\
t3 & \leftarrow t1 \times t2
\end{align*}
\]

Stack-based bytecode versus 3-address code (c = a + b)

iload 1
iload 2
iadd
istore 3
Suppose we are doing constant propagation, and we know that \( a \) is 10, and \( b \) is 12?

\[
\text{iload 1} \\
\text{iload 2} \\
\text{iadd} \\
\text{istore 3}
\]

\[
\begin{array}{c|c|c|c|c|c|}
\text{a} & \text{b} & \text{c} & \text{t1} & \text{t2} \\
10 & 12 & 3 & & \\
\end{array}
\]

Three–Address Code

- \( a = b \ OP c \)
  - Originally, because instruction had at most 3 addresses or operands
    - This is not enforced today, ie MAC: \( a = b \ast c + d \)
  - May have fewer operands
- Also called quadruples: \((a,b,c,OP)\)
- Example

\[
\begin{align*}
a &= (b+c) \ast (-e) \\
t1 &= b + c \\
t2 &= -e \\
a &= t1 \ast t2
\end{align*}
\]

Compiler-generated temporary variable
IR Operands

- The operands in 3-address code can be:
  - Program variables
  - Constants or literals
  - Temporary variables
- Temporary variables = new locations
  - Used to store intermediate values
  - Needed because 3-address code not as expressive as high-level languages
- Often introduce lots of temporaries and then simplify to remove spurious ones.

Typical Statements

- **Assignments:**
  - \( x \leftarrow y \text{ OP } z : \text{binary OP} \)
    - Arithmetic: +, -, *, /, mod
    - Logic: AND, OR, XOR
    - Comparisons: =, ! =, <, >, >=, <=
  - \( x \leftarrow \text{OP } y : \text{unary OP} \)
    - Arithmetic: -
    - Logic: NOT
Typical Statements

- Data movement:
  - Copy/Move: \( x \leftarrow y \)
  - Load: \( x \leftarrow [y] \)
  - Store: \( [x] \leftarrow y \)
  - "address of": \( x \leftarrow \text{addr } y \)

Flow of control (branch):
- Label \( L \): define a label (a point in LIR)
- Jump \( L \): unconditional jump \( (\text{goto } L) \)
- Cjmp \( c \): conditional jump (jump to \( L \) if \( c \) TRUE)
  \( (\text{if } (x \text{ op } y) \text{ goto } L) \)

Function call:
- Call \( f(a_1,a_2,\ldots,a_n) \)
- \( x \leftarrow \text{call } f(a_1,a_2,\ldots,a_n) \)

Can/should add explicit representation of setup for passing function arguments
IR Example

```c
n = 0;
while (n < 10) {
    n = n + 1;
}
```

Lauren's Version

```
n = 0
label lTEST
t2 = n < 10
t3 = NOT t2
cjmp t3 lEND
label lBODY
n = n + 1
jump lTEST
label lEND
```

Another IR Example

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

Lauren's Version

```
m = 0
t1 = c == 0
cjmp t1 lTRUE
m = m + n
jump lEND
label lTRUE
t2 = n * n
m = m + t2
label lEND
```

```
m <= 0
if (c == 0) goto &true
m <= m + n
goto &lend
ltrue:
t2 <= n * n
m <= m + t2
lend:
```
IR Instructions

- **Assignment instructions**
  - $a = b \text{ OP } c$ (binary op)
    - arithmetic: ADD, SUB, MUL, DIV, MOD
    - logic: AND, OR, XOR
    - comparisons: EQ, NEQ, LT, GT, LEQ, GEQ
  - $a = \text{ OP } b$ (unary op)
    - arithmetic MINUS, logical NEG
  - $a = b$ : copy instruction
  - $a = \text{ ld } b$ : load instruction
  - $a = \text{ st } a = b$ : store instruction
  - $a = \& b$ : symbolic address

- **Flow of control**
  - label L: label instruction
  - goto L: unconditional jump
  - if (a op b) goto L: cond. jump

- **Function call**
  - call $f(a_1, ..., an)$
  - $a = \text{ call } f(a_1, ..., an)$

- **IR describes the instruction set of an abstract machine**

Class Problem

Convert the following code segment to assembly code

```plaintext
n = 0;
sum = 0;
while (n < 10) {
    if (n % 2 == 0)
        sum = n+1;
}
print(sum);
```