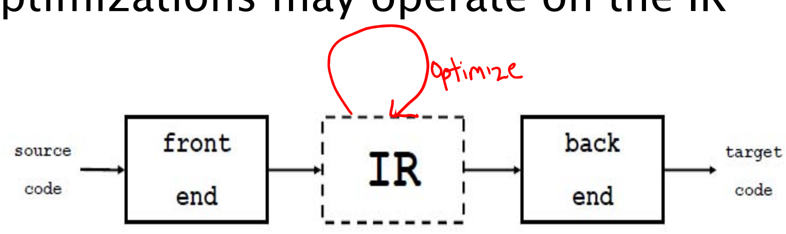
Intermediate Representations

(Slides adapted from http://moodle.bracu.ac.bd/course/view.php?id=90)

COMP 621 – McGill University – Laurie Hendren

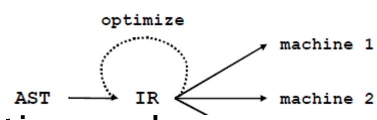
Intermediate code

- 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

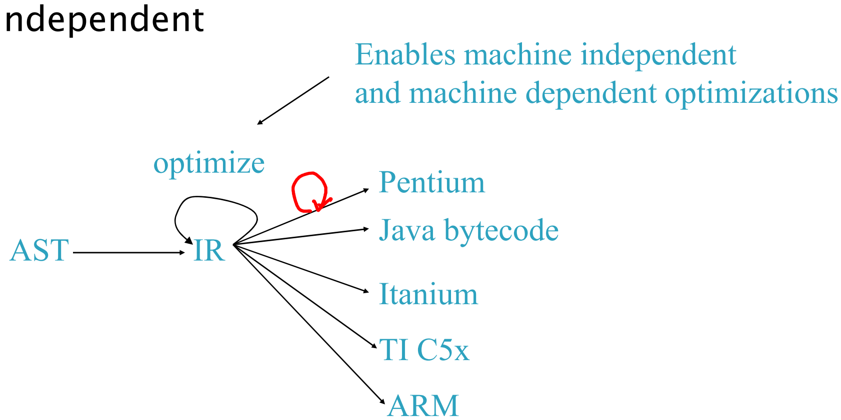
- Break compiler into manageable pieces
  - simpler pieces
  - more **modularity**
- Easier re-targeting



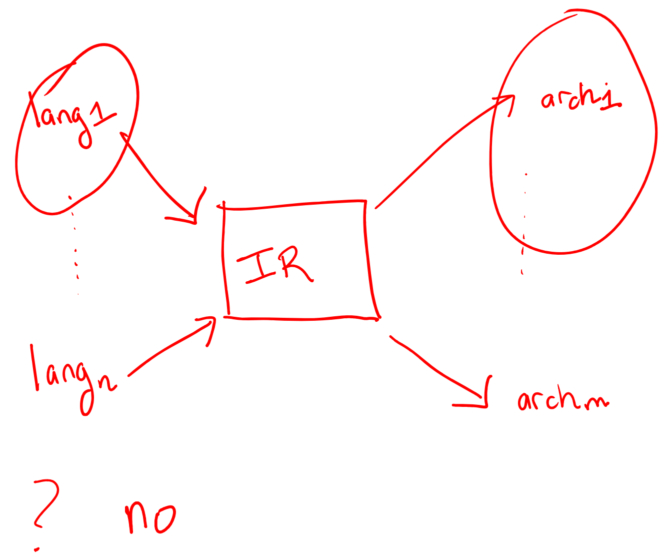
- Complete pass before emitting code
  - => better code
- Allows for language-independent and machine-independent optimizations
- Drawback: Another step => **loss in efficiency**

Intermediate Representation (IR)

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



Can there be one general purpose IR?

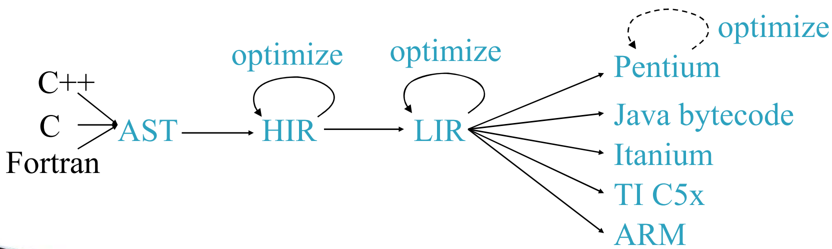


What Makes a Good IR?

- 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

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

IR Category example

Source

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

Low ←→ High

High-level IR

```
t1 = a[i, j+2]
```

Middle-level IR

```
t1 = j + 2
t2 = i * 20
t3 = t1 + t2
t4 = 4 * t3
t5 = addr a
t6 = t5 + t4
t7 = *t6
```

Low-level IR

```
r1 = [fp - 4]
r2 = [r1 + 2]
r3 = [fp - 8]
r4 = r3 * 20
r5 = r4 + r2
r6 = 4 * r5
r7 = fp -216
f1 = [r7 + r6]
```


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

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
  - OP ↑ operation; y, z ↑ operands; x ↑ result

- Has three names/addresses (x,y,z), or less
- A single operator (OP)

- We will write as: x ← y OP z

<u>Example:</u>

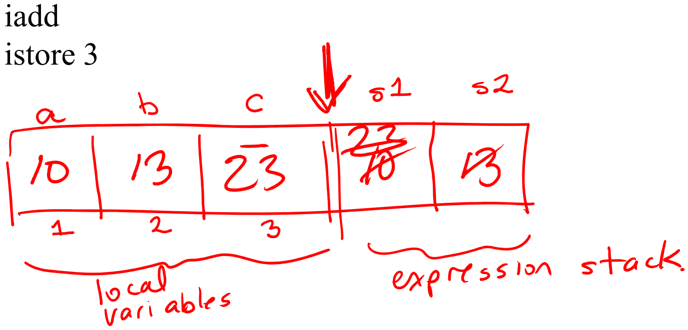
```
x ← (y + z)*(-r);
```

⇨

```
t1 ← y + z
t2 ← -r
t3 ← t1 * t2
```

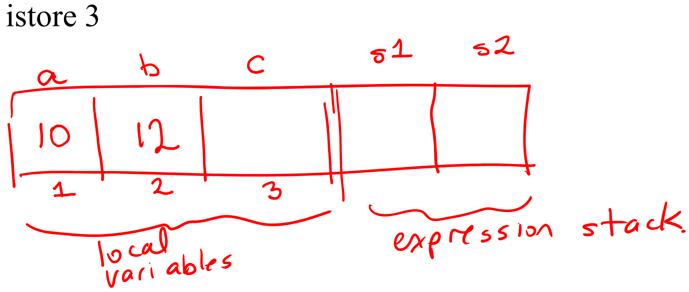

Stack-based bytecode versus 3-address code (c = a + b)

```
iload 1
iload 2
iadd
istore 3
```



Suppose we are doing constant propogation, and we know that a is 10, and b is 12?

```
iload 1
iload 2
iadd
istore 3
```



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 * c + d
  - May have fewer operands
- Also called quadruples: (a,b,c,OP)
- Example

a = (b+c) * (-e)

```
t1 = b + c
t2 = -e
a = t1 * t2
```

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 ← y OP z : binary OP
    - Arithmetic: +, -, *, /, mod
    - Logic: AND, OR, XOR
    - Comparisons: =, !=, <, >, >=, =<
  - x ← OP y : unary OP
    - Arithmetic: -
    - Logic: NOT

Typical Statements

- Data movement:
  - Copy/Move   x ← y
  - Load: x ← [y]
  - Store: [x] ← y

  } [] = contents of

  - "address of":

    x ← addr y

Copy stmt

alternative.
x ← ld y
st x ← y

x ← $ y

Typical Statements

- Flow of control (branch)
  - `label L` : define a label (= a point in LIR)
  - `jump L` : unconditional jump (goto L)
  - `cjmp c L` : conditional jump (jump to `L` if `c` `TRUE`)
    (if (x op y) goto L)
- Function call
  - `call f(a1,a2,. . .,an)`
  - `x ← call f(a1,a2,. . .,an)`
  - Can/should add explicit representation of setup for passing function arguments

IR Example

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

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

Laurie's version

```
    n ← 0
ltest:
    if (n ≥ 10) goto lend
lbody:
    n ← n+1
    goto ltest
lend:
```


Another IR Example

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

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

Laurie's Version

```
    m ← 0
    if (c == 0) goto ltrue
    m ← m+n
    goto lend
ltrue:
    t2 ← n*n
    m ← m + t2
lend:
```

IR Instructions

- Assignment instructions
  - a = b OP C (binary op)
    - arithmetic: ADD, SUB, MUL, DIV, MOD
    - logic: AND, OR, XOR
    - comparisons: EQ, NEQ, LT, GT, LEQ, GEQ
  - a = OP b (unary op)
    - arithmetic MINUS, logical NEG
  - a = b : copy instruction
  - a = ld b : load instruction
  - 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(a1, ..., an)
  - a = call f(a1, ..., an)
- IR describes the instruction set of an abstract machine

t := a[i];

a[i] := t;

return (goto end)

alloca, malloc

*x = z [x] := z

z = *x z := [x];

x.y := z;

z := x.y;

What's missing?

COMP 621 - McGill University - Laurie Hendren 22

Class Problem

Convert the following code segment to assembly code

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

COMP 621 - McGill University - Laurie Hendren 23