

CPEN400P: Intermediate Representations

Lecture 2

Karthik Pattabiraman, UBC

Most of the material in
this lecture comes
from Chapter 5 of EaC

Copyright 2010, Keith D. Cooper & Linda Torczon, all rights reserved.

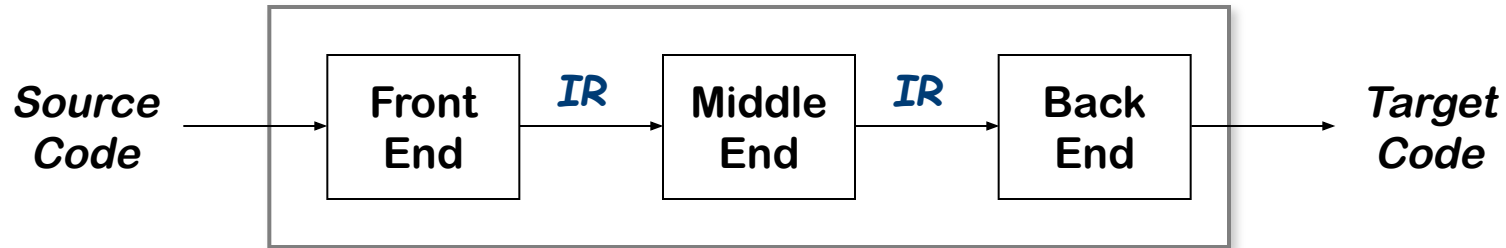
Students enrolled in Comp 412 at Rice University have explicit permission to make copies of these materials for their personal use.

Faculty from other educational institutions may use these materials for nonprofit educational purposes, provided this copyright notice is preserved.

Learning Objectives

- List the desirable features of intermediate representations
- List the types of Intermediate Representations and their types of abstractions
- Understand the pros and cons of different intermediate representations
- Understand SSA form and its advantages
- List the pros and cons of various memory models

Intermediate Representations: 3-pass compiler



- Front end - produces an intermediate representation (*IR*)
- Middle end - transforms the *IR* into an equivalent *IR* that runs more efficiently
- Back end - transforms the *IR* into native code
- *IR* encodes the compiler's knowledge of the program
- Middle end usually consists of several passes

Intermediate Representations

- Decisions in *IR* design affect the speed and efficiency of the compiler
- Some important *IR* properties
 - Ease of generation
 - Ease of manipulation
 - Procedure size
 - Freedom of expression
 - Level of abstraction
- The importance of different properties varies between compilers
 - Selecting an appropriate *IR* for a compiler is critical

Types of Intermediate Representations

Three major categories

- Structural

- Graphically oriented
- Heavily used in source-to-source translators
- Tend to be large

Examples:
Trees, DAGs

- Linear

- Pseudo-code for an abstract machine
- Level of abstraction varies
- Simple, compact data structures
- Easier to rearrange

Examples:
3 address code
Stack machine code

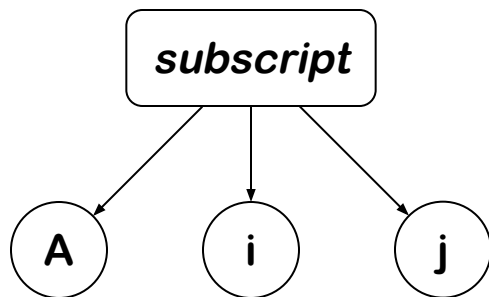
- Hybrid

- Combination of graphs and linear code
- Example: control-flow graph

Example:
Control-flow graph,
SSA form

Level of Abstraction

- The level of detail exposed in an *IR* influences the profitability and feasibility of different optimizations.
- Two different representations of an array reference:



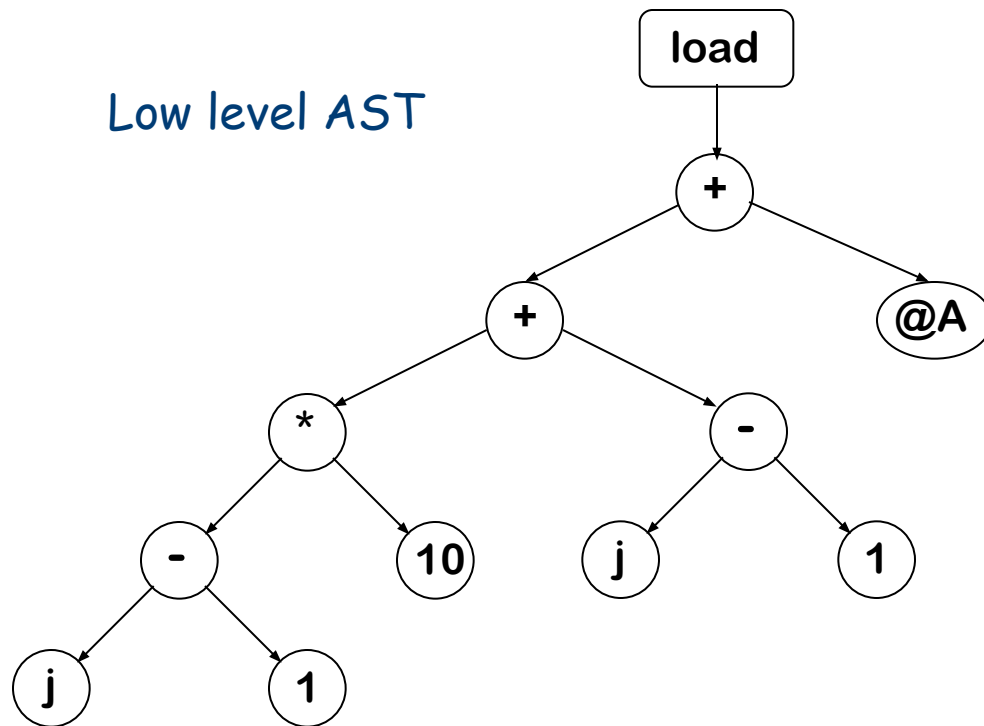
High level AST:
Good for memory
disambiguation

```
loadI 1      => r1
sub    rj, r1 => r2
loadI 10     => r3
mult   r2, r3 => r4
sub    ri, r1 => r5
add    r4, r5 => r6
loadI @A     => r7
add    r7, r6 => r8
load   r8    => rAij
```

Low level linear code:
Good for address calculation

Level of Abstraction

- Structural *IRs* are usually considered high-level
- Linear *IRs* are usually considered low-level
- Not necessarily true - see example below



loadArray A, i, j

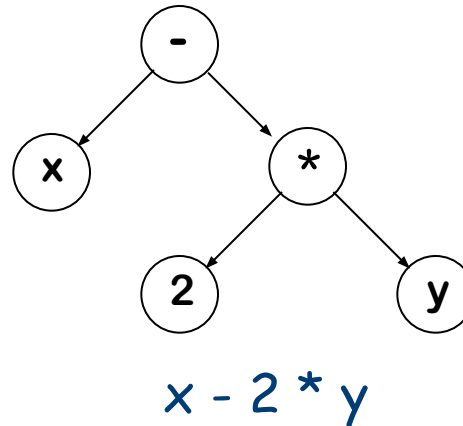
High level linear code

Learning Objectives

- List the desirable features of intermediate representations
- List the types of Intermediate Representations and their types of abstractions
- Understand the pros and cons of different intermediate representations
- Understand SSA form and its advantages
- List the pros and cons of various memory models

Abstract Syntax Tree

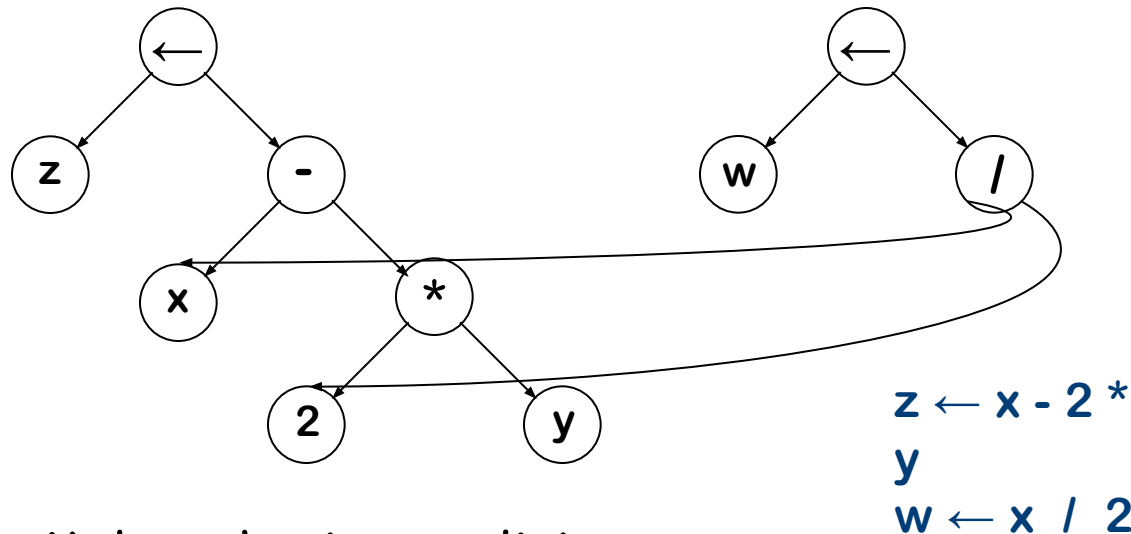
An abstract syntax tree is the procedure's parse tree with the nodes for most non-terminal nodes removed



- Can use linearized form of the tree
 - Easier to manipulate than pointers
 - $x \ 2 \ y \ * \ -$ in postfix form
 - $- \ * \ 2 \ y \ x$ in prefix form
- S-expressions (Scheme, Lisp) are (essentially) ASTs

Directed Acyclic Graph

A directed acyclic graph (DAG) is an AST with a unique node for each value



- Makes sharing explicit
- Encodes redundancy

With two copies of the same expression, the compiler might be able to arrange the code to evaluate it only once.

Stack Machine Code

Originally used for stack-based computers, now Java

- Example:

$x - 2 * y$ becomes

```
push x
push 2
push y
multiply
subtract
```

Advantages

- Compact form
- Introduced names are *implicit*, not *explicit*
- Simple to generate and execute code

Useful where code is transmitted
over slow communication links (*the net*)

Implicit names take up
no space, where explicit
ones do!

Three Address Code

Several different representations of three address code

- In general, three address code has statements of the form:

$$x \leftarrow y \text{ op } z$$

With 1 operator (op) and, at most, 3 names (x, y, & z)

Example:

$z \leftarrow x - 2 * y$ becomes

| | | | | | |
|--|---|---|---|---|---|
| | t | ← | 2 | * | y |
| | z | ← | x | - | t |

Advantages:

- Resembles many real machines
- Introduces a new set of names*
- Compact form

Three Address Code: Quadruples

Naïve representation of three address code

- Table of $k * 4$ small integers
- Simple record structure
- Easy to reorder
- Explicit names

The original FORTRAN compiler used "quads"

```
load  r1, y
loadI r2, 2
mult  r3, r2, r1
load  r4, x
sub   r5, r4, r3
```

RISC assembly code

| | | | |
|-------|---|---|---|
| load | 1 | y | |
| loadi | 2 | 2 | |
| mult | 3 | 2 | 1 |
| load | 4 | x | |
| sub | 5 | 4 | 3 |

Quadruples

Three Address Code: Triples

- Index used as implicit name
- 25% less space consumed than quads
- Much harder to reorder

| | | | |
|-----|-------|-----|-----|
| (1) | load | y | |
| (2) | loadI | 2 | |
| (3) | mult | (1) | (2) |
| (4) | load | x | |
| (5) | sub | (4) | (3) |

Implicit names occupy no space

Remember, for a long time, 640Kb was a lot of RAM

Two Address Code

- Allows statements of the form

$x \leftarrow x \text{ op } y$

Has 1 operator (op) and, at most, 2 names (x and y)

Example:

$z \leftarrow x - 2 * y$ becomes

```
t1 ← 2
t2 ← load y
t2 ← t2 * t1
z ← load x
z ← z - t2
```

- Can be very compact

Problems

- Machines no longer rely on destructive operations
- Difficult name space
 - Destructive operations make reuse hard
 - Good model for machines with destructive ops (PDP-11)

Learning Objectives

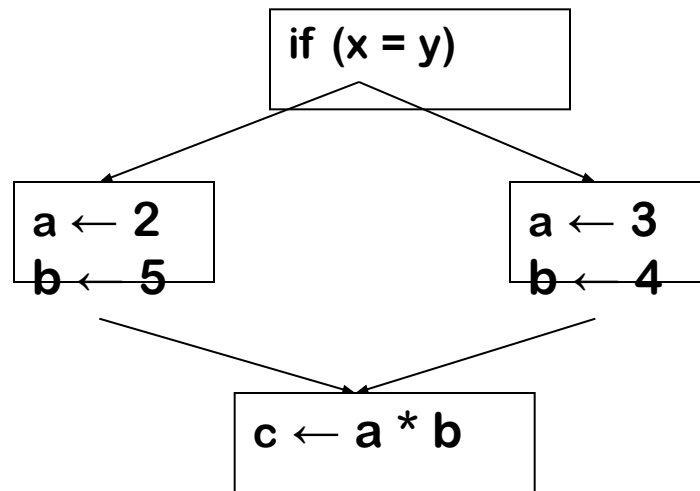
- List the desirable features of intermediate representations
- List the types of Intermediate Representations and their types of abstractions
- Understand the pros and cons of different intermediate representations
- Understand SSA form and its advantages
- List the pros and cons of various memory models

Control-flow Graph

Models the transfer of control in the procedure

- Nodes in the graph are basic blocks
 - Can be represented with quads or any other linear representation
- Edges in the graph represent control flow

Example



Algorithm for converting linear code to CFG

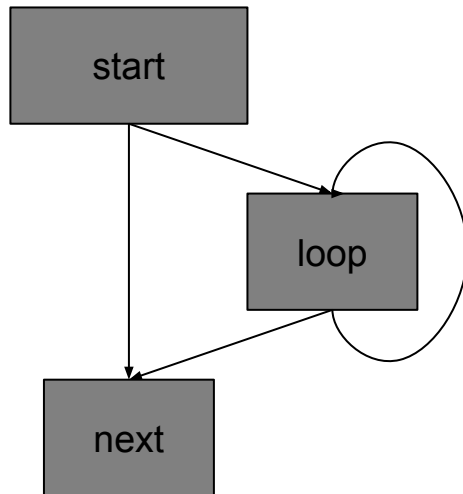
- Find leaders
 - Identify all nodes that are labels/targets for jumps as leaders
- Break CFG into (leader, last) pairs
 - For each leader, traverse program until we come to another leader. Terminate the block and record (leader, last) pair.
 - If the last instruction is a conditional branch to l1, l2, then record edges from the current block to blocks l1 and l2
 - If the last instruction is a unconditional branch to label 'l', then add an edge from the current block to 'l'
 - If the last instruction is an indirect jump (e.g., jmp r1), then
 - Strategy 1 (conservative): Add an edge to every basic block
 - Strategy 2 (precise): Add an edge to only those basic blocks that are targets of the jump - requires pointer/alias analysis to resolve

Add an entry node and exit node and the corresponding arcs if needed

Example of CFG Algorithm

Original

```
x ← ...  
y ← ...  
while (x < k)  
    x ← x + 1  
    y ← y + x
```



BBs With explicit branches

```
start:  
    x ← ...  
    y ← ...  
    if (x>k) goto next
```

```
loop:  
    x ← x + 1  
    y ← y + x  
    if (x<k) goto loop
```

```
next: ..
```

Class Activity

Draw the CFG for the following code applying the algorithm in the previous slide for CFG construction (Figure 5.14 in EaC).

■ FIGURE 5.13 Code Fragment for Exercise 4.

```
L01: add    ra, rb    ⇒ r1
      add    rc, rd    ⇒ r2
      add    r1, r2    ⇒ r3
      add    ra, rb    ⇒ r4
      cmp_LT r1, r2    ⇒ r5
      cbr    r5        → L02, L04
L02: add    ra, rb    ⇒ r6
      multI  r6, 17    ⇒ r7
      jumpI          → L03
L03: add    ra, rb    ⇒ r22
      multI  r22, 17   ⇒ r23
      jumpI          → L07
L04: add    rc, rd    ⇒ r8
      i2i    ra        ⇒ r9
      cmp_LT r9, rd    ⇒ r10
      cbr    r10       → L05, L06
L05: add    r9, rb    ⇒ r11
      add    ra, rb    ⇒ r12
      add    rc, rd    ⇒ r13
      i2i    ra        ⇒ r13
      add    r13, rb    ⇒ r14
      multI  r12, 17   ⇒ r15
      jumpI          → L03
L06: add    r1, r2    ⇒ r16
      i2i    r2        ⇒ r17
      i2i    r1        ⇒ r18
      add    r17, r18    ⇒ r19
      add    r18, r17    ⇒ r20
      multI  r1, 17    ⇒ r21
      jumpI          → L03
L07: nop
```

Static Single Assignment (SSA) Form

- The main idea: each name defined exactly once
- Introduce ϕ -functions to make it work

Original

```
x ← ...
y ← ...
while (x < k)
  x ← x + 1
  y ← y + x
```

SSA-form

```
x0 ← ...
y0 ← ...
if (x0 ≥ k) goto next
loop:
  x1 ←  $\phi(x_0, x_2)$ 
  y1 ←  $\phi(y_0, y_2)$ 
  x2 ← x1 + 1
  y2 ← y1 + x2
  if (x2 < k) goto loop
next:
  ...
```

Strengths of SSA-form

- Sharper analysis
- ϕ -functions give hints about placement
- Some facts are obvious (e.g., live variables)
- Compact representation of data-flow facts

Algorithm to convert code to SSA (Naïve algorithm)

- **Non-obvious construction algorithm**
 - Will examine this algorithm later in this course
 - Naïve algorithm inserts too many redundant phi functions
- **Naïve algorithm**
 - Traverse the code in linear order
 - The first time you come to a variable, no action is needed
 - When you come to a previously defined variable, rename it to a unique name and replace all references with the new name
 - If you come to a Y-point in the code (i.e., control flow convergence), insert a phi node for every variable defined and used in the downstream computations, and add the predecessor block's latest values as operands of the phi instruction
 - Not as simple as it looks

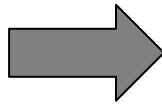
Class Activity

Draw the CFG of the following code and convert it to SSA form
(Example based on Figure 5.13 in EaC)

```
x ← ...  
y ← ..  
a ← y + 2  
b ← 0
```

```
while (x < a) {  
    if (y < x) {  
        x ← y + 1  
        y ← b * 2  
    } else {  
        x ← y + 2  
        y ← a + 2;  
    }  
}
```

```
w ← x + 2  
z ← y * a  
y ← y + 1
```



```
B0: x ← ...  
    y ← ...  
    a ← y + 2  
    b ← 0  
    if (x >= a) jump B5
```

```
B2: if (y >= x) jump B4
```

```
B3: x ← y + 1  
    y ← b * 2  
    jump B5
```

```
B4: x ← y + 2  
    y ← a + 1
```

```
B5: if (x >= a) jump B2
```

```
B6: w ← x + 2  
    z ← y * a  
    y ← y + 1
```

Learning Objectives

- List the desirable features of intermediate representations
- List the types of Intermediate Representations and their types of abstractions
- Understand the pros and cons of different intermediate representations
- Understand SSA form and its advantages
- List the pros and cons of various memory models

Memory Models

Two major models

- Register-to-register model
 - Keep all values that can legally be stored in a register in registers
 - Ignore machine limitations on number of registers
 - Compiler back-end must insert loads and stores
- Memory-to-memory model
 - Keep all values in memory
 - Only promote values to registers directly before they are used
 - Compiler back-end can remove loads and stores

Pros of Register-to-Register memory Model

- Compilers for RISC machines usually use register-to-register
 - Reflects programming model
 - Easier to determine when registers are used
 - Does not limit number of registers in the target
 - Easy to represent data-flow facts about the program (i.e., a value that is safe to move to a register is one that is not indirectly modified, through a pointer, for example)

Cons of Register-to-Register Memory Model

- Pressure on downstream passes and register allocator
 - Additional loads/stores required
- Can provide misleading information about program's performance at the intermediate code level
 - Cannot reason about memory pressure, cache misses etc.
 - Not a good match for register-constrained machines
- Handling memory references may be cumbersome at the IR level as the default mode is to operate on registers
 - Need to ensure that any variable whose address can be taken cannot have multiple references to it (i.e., alias analysis)
 - Such variables cannot be stored in registers

Learning Objectives

- List the desirable features of intermediate representations
- List the types of Intermediate Representations and their types of abstractions
- Understand the pros and cons of different intermediate representations
- List the pros and cons of various memory models

TODOs in the near term

Find a partner to do the assignments with and send us a private note on Piazza by Jan 20th

Try tutorial 1 - installation and writing a simple pass in LLVM

- Needed for doing assignment 1 (Jan 21st release)
- You can get help from TAs during the lab session

Prepare for the PPT on Thursday (Jan 20th)

- Simple C++ programming problems
- Done online in the HR platform

Decide if you want to stay in the course or drop it (Jan 21)