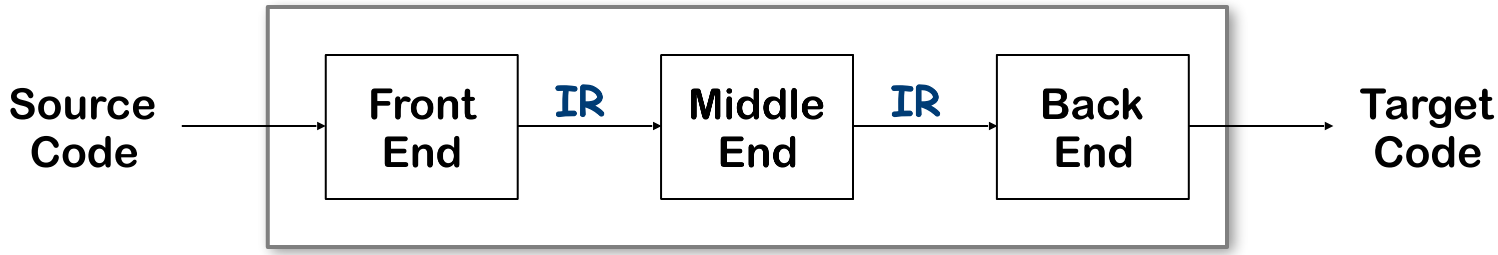
Intermediate Representations

Where In The Course Are We?

- We are on the cusp of the art, science, & engineering of compilation
- Scanning & parsing are applications of automata theory
- Context-sensitive analysis, as covered in class, is mostly software engineering
- The mid-section of the course will focus on issues where the compiler writer needs to choose among alternatives
  - The choices matter; they affect the quality of compiled code
  - There may be no "best answer" or "best practice"

The fun begins at this point

Intermediate Representations



- 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

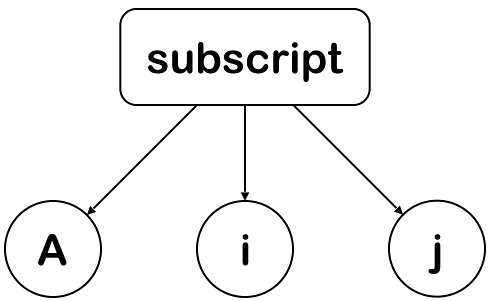
- Hybrid
  - Combination of graphs and linear code

Example:
Control-flow graph

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:

array A[1…10,1…10] of 1 byte memorised in row-major order



High level AST:
Good for memory disambiguation

$\texttt{loadI}\ 1 \Rightarrow r_1$

$\texttt{sub}\ r_j, r_1 \Rightarrow r_2$

$\texttt{loadI}\ 10 \Rightarrow r_3$

$\texttt{mult}\ r_2, r_3 \Rightarrow r_4$

$\texttt{sub}\ r_i, r_1 \Rightarrow r_5$

$\texttt{add}\ r_4, r_5 \Rightarrow r_6$

$\texttt{loadI}\ @A \Rightarrow r_7$

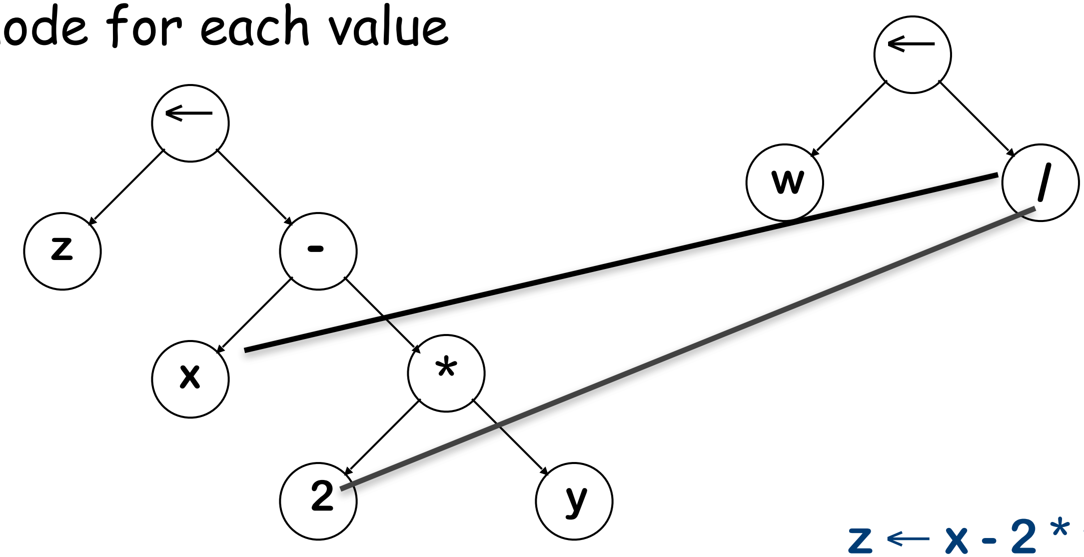
$\texttt{add}\ r_7, r_6 \Rightarrow r_8$

$\texttt{load}\ r_8 \Rightarrow r_{Aij}$

Low level linear code:
Good for address calculation

Directed Acyclic Graph

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



z ← x - 2 * y
w ← x / 2

- 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

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

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

Three Address Code

Several different representations of three address code

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

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

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

Example:

z ← 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)

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

Implicit names occupy no space

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

- Major tradeoff between quads and triples is compactness versus ease of manipulation
  - In the past compile-time space was critical
  - Today, speed may be more important

Two Address Code

- Allows statements of the form

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

  Has 1 operator ($\underline{op}$) 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

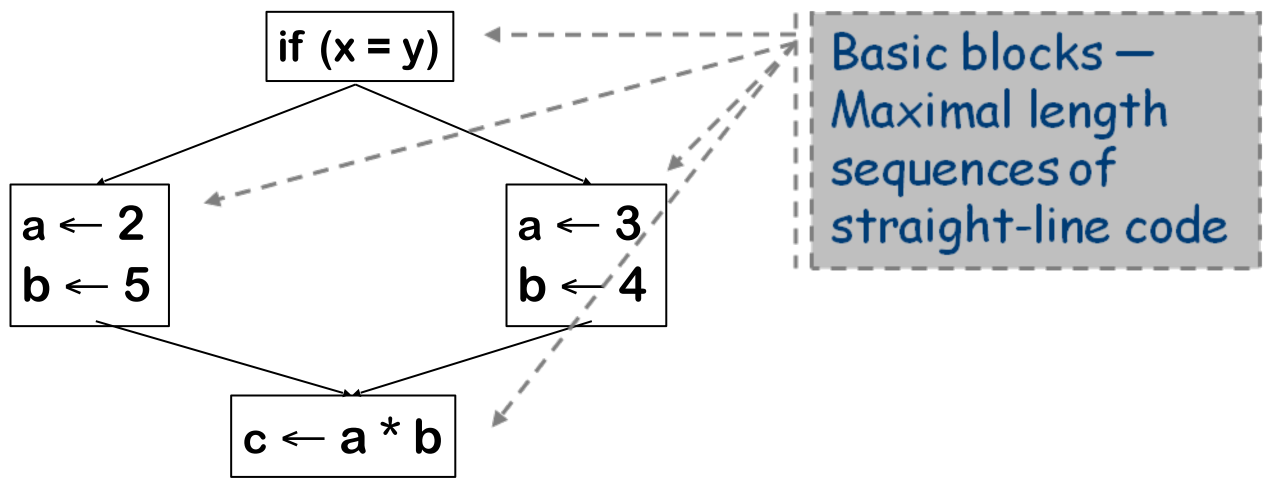
- Machines no longer rely on destructive operations
- Difficult name space
  - Destructive operations make reuse hard
  - Good model for machines with destructive ops (PDP-11)

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



Static Single Assignment Form

- The main idea: each name is defined exactly once
- The name refers to the variable in some program point
- Encodes both control and value flow
- Introduce ϕ-functions to make it work

Original

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

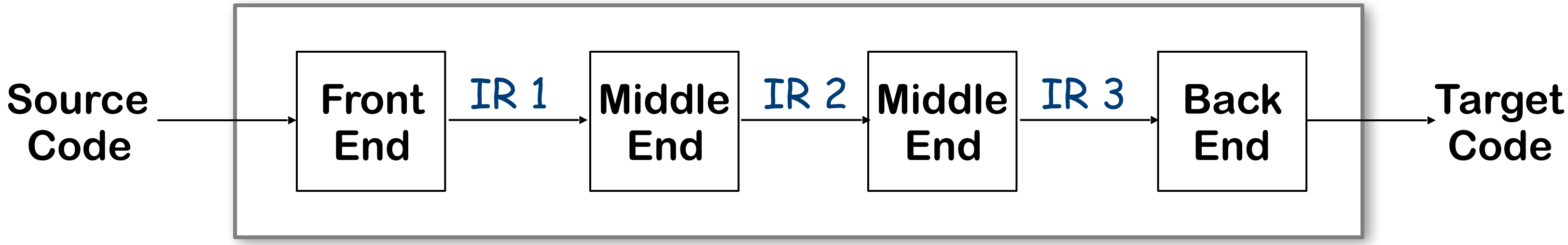
SSA-form

```
          x0 ← …
          y0 ← …
          if (x0 >= k) goto next
loop:        x1 ← φ(x0,x2)
             y1 ← φ(y0,y2)
             x2 ← x1 + 1
             y2 ← y1 + x2
             if (x2 < k) goto loop
next:         …
```

Strengths of SSA-form

- each use refers to a single definition
- (sometimes) faster algorithms

Using Multiple Representations



- Repeatedly lower the level of the intermediate representation
  - Each intermediate representation is suited towards certain optimizations
- Example: the Open64 compiler
  - WHIRL intermediate format
    - → Consists of 5 different IRs that are progressively more detailed and less abstract

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

use virtual registers!

- 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
- Compilers for RISC machines usually use register-to-register
  - Easier to determine when registers are used

The Rest of the Story…

Representing the code is only part of an IR

The compiler must discover and store many distinct kinds of information

For a variable it has to store its data type, storage class, the level of its declaring procedure, and a base and offset in memory

It often uses centralised

- Symbol table
- Constant table
  - Representation, type
  - Storage class, offset

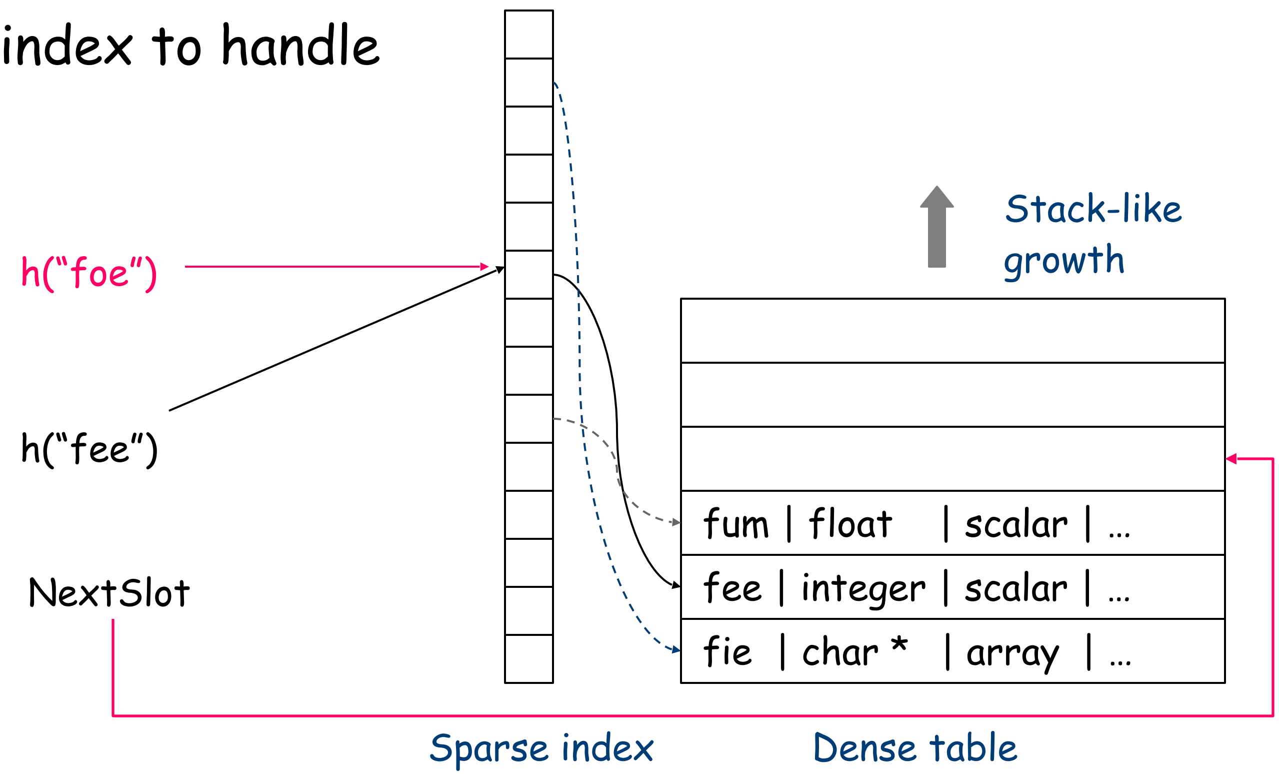
It needs an efficient and expandable way to realise them!

Symbol Tables

Classic approach to building a symbol table uses hashing

- A two-table scheme
  - Sparse index to reduce chance of collisions
  - Dense table to hold actual data
    - → Easy to expand, to traverse, to read & write from/to files
- Use chains in index to handle collisions

Collision occurs when h() returns a slot in the sparse index that is already full.



Hash-less Symbol Tables

Classic approach to building a symbol table uses hashing

- Some concern about worst-case behavior
  - Collisions in the hash function can lead to linear search
  - Some authors advocate "perfect" hash for keyword lookup
- Automata theory lets us avoid worst-case behavior

Collision occurs when h() returns a slot in the sparse index that is already full.