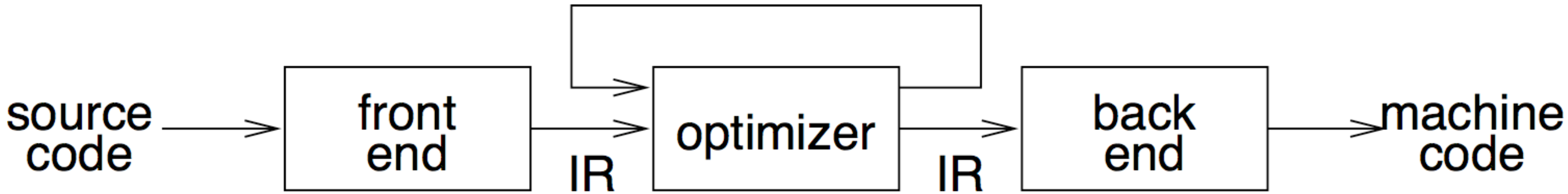
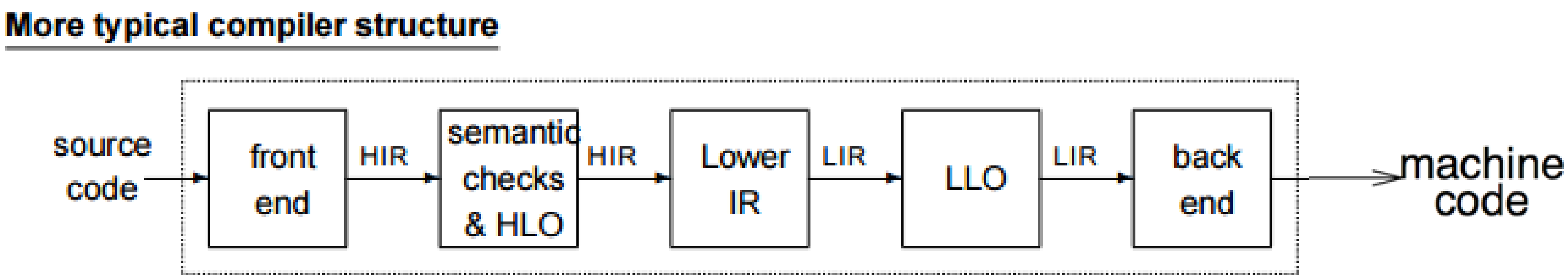
Intermediate Representation（IR）

IR scheme





- front end produces IR
- optimizer transforms IR to more efficient program
- back end transform IR to target code

Why use intermediate representations?

Components and Design Goals for an IR

Components of IR

- Code representation: actual statements or instructions
- Symbol table with links to/from code
- Analysis information with mapping to/from code
- Constants table: strings, initializers, ...
- Storage map: stack frame layout, register assignments

Design Goals for an IR?

- There is no universally good IR. Many forms of IR have been used.
- The right choice depends strongly on the goals of the compiler system.

Kinds of IR

- Abstract syntax trees (AST)
- Directed acyclic graphs (DAG)
- Control flow graphs (CFG)
- Data dependence graphs (DDG)
- Static single assignment form (SSA)
- 3-address code
- Hybrid combinations

Categories of IR

- Structural
  - — graphically oriented (trees, DAGs)
  - — nodes and edges tend to be large
  - — heavily used on source-to-source translators
  - — harder to rearrange

  Examples: AST, CFG, DDG, Expression DAG, Points-to graph
- Linear
  - — pseudo-code for abstract machine
  - — large variation in level of abstraction
  - — simple, compact data structures
  - — easier to rearrange

  Examples: 3-address, 2-address, accumulator, or stack code
- Hybrid
  - — CFG + 3-address code (SSA or non-SSA)
  - — CFG + 3-address code + expression DAG
  - — AST (for control flow) + 3-address code (for basic blocks)
  - — AST (for control flow) + expression DAG (for basic blocks)
  - — attempt to achieve best of both worlds

Important IR properties

- Ease of generation
- Ease of manipulation
- Cost of manipulation
- Level of abstraction
- Freedom of expression
- Size of typical procedure
- Original or derivative

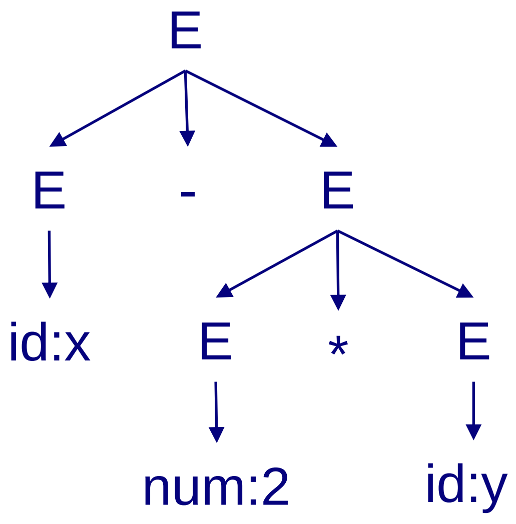
Subtle design decisions in the IR can have far-reaching effects on the speed and effectiveness of the compiler!
➔ *Degree of exposed detail can be crucial*

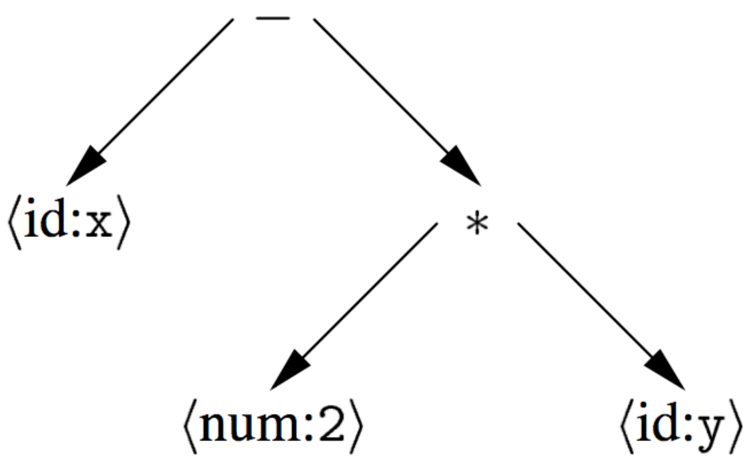
Abstract syntax tree

An AST is a simplified parse tree with nodes for most non-terminals removed.

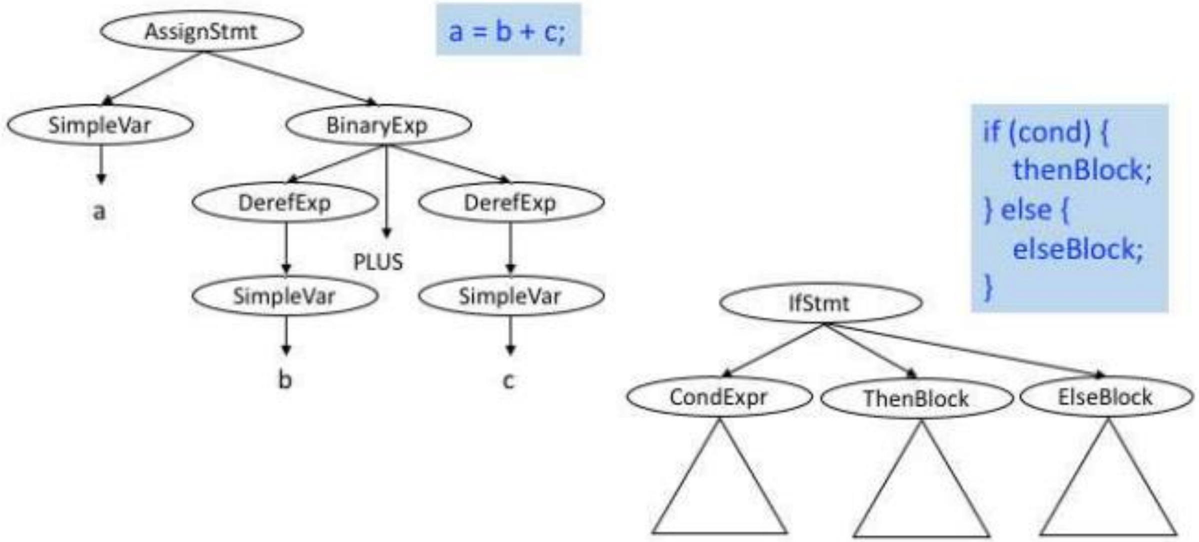
- It retains syntactic structure of code

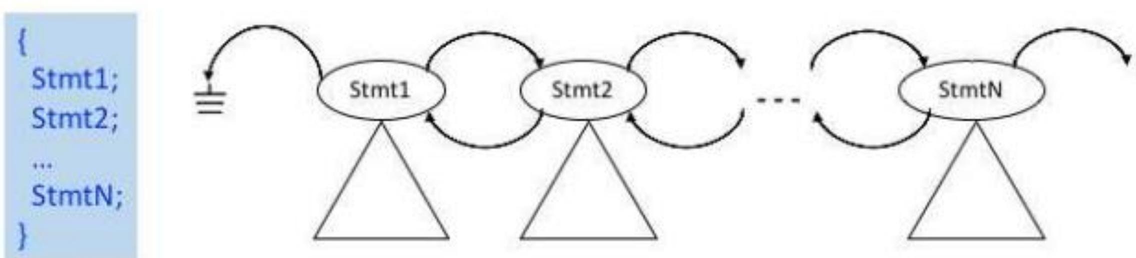
Since the program is already parsed, non-terminals needed to establish precedence and associativity can be collapsed!

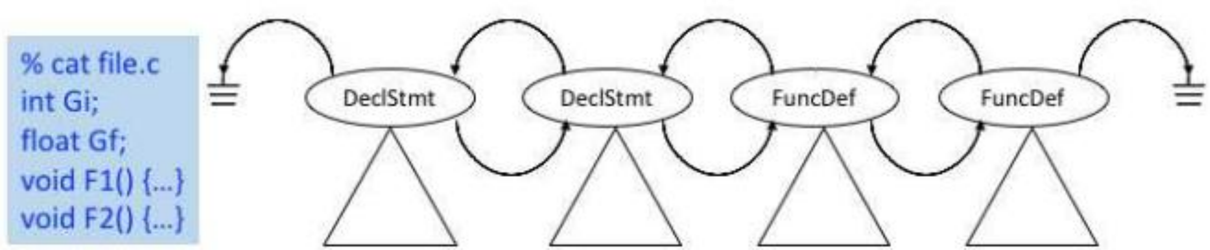




Abstract syntax tree: Examples







Abstract syntax tree

Advantage

- Well-suited for source code
- Widely used in source-source translators
- Captures both control flow constructs and straight-line code explicitly

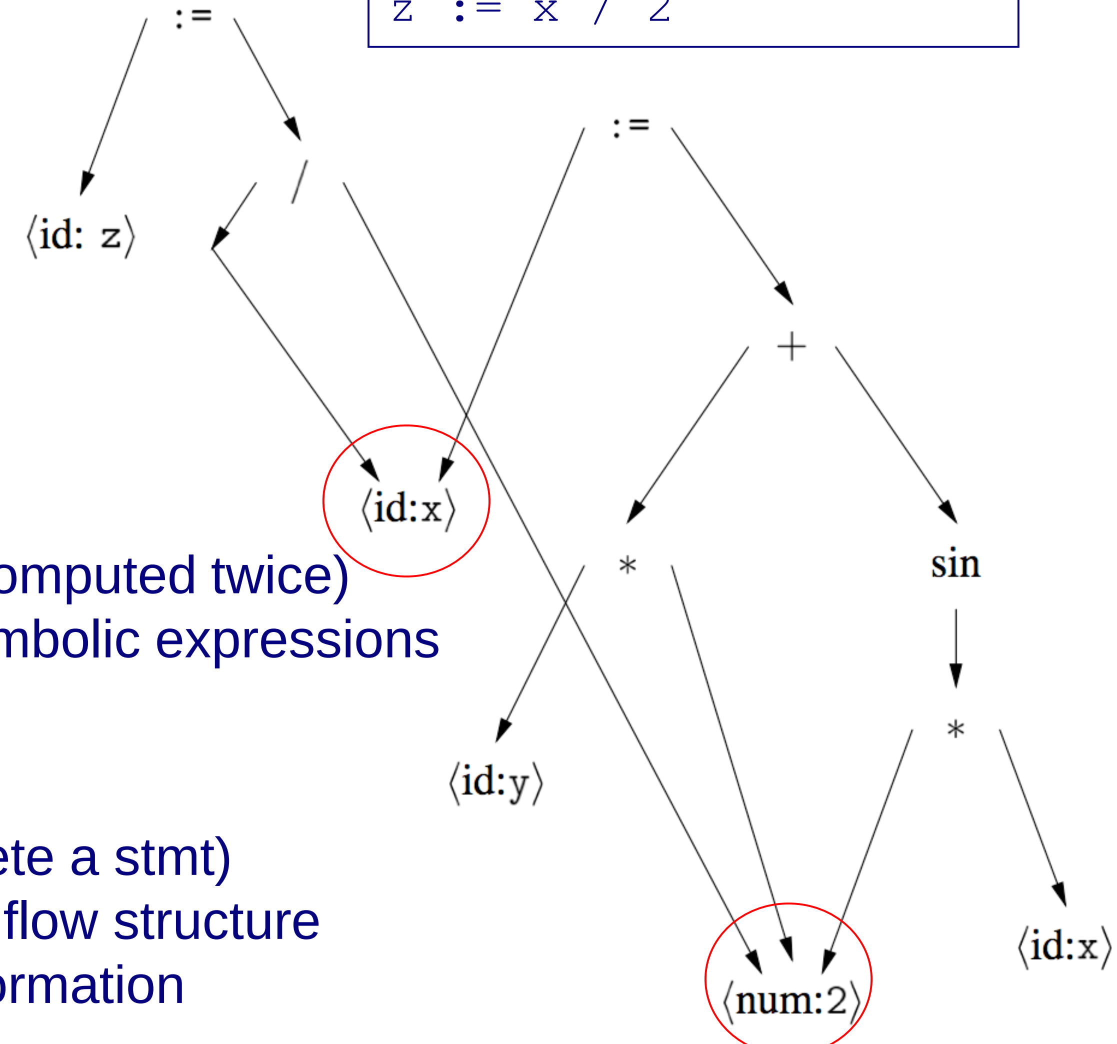
Disadvantage

- Traversal and transformations are both relatively expensive
- pointer-intensive
- transformations are memory-allocation-intensive

Directed acyclic graph

```
x := 2 * y + sin(2*x)
z := x / 2
```

A DAG is an AST with unique, shared nodes for each value.



Advantages

- sharing of values is explicit
- exposes redundancy (value computed twice)

⇒ powerful representation for symbolic expressions

Disadvantages

- difficult to transform (e.g., delete a stmt)
- not useful for showing control flow structure

⇒ Better for analysis than transformation

Control flow graph

Basic Block: a consecutive sequence of statements (or instructions) $S_1 . . . S_n$ such that

(a) the flow of control must enter the block at S_1, and

(b) if S_1 is executed, then $S_2 . . . S_n$ are all executed in that order (unless one of the statements causes the program to halt).

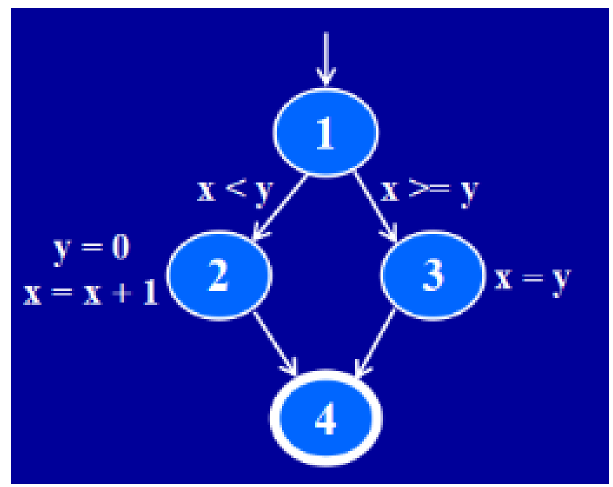
Leader: the first statement of a basic block

Maximal Basic Block ≡ a maximal-length basic block

Control flow graph

- A CFG models *transfer of control* in a program
  - nodes are *basic blocks* (straight-line blocks of code, Maximal Basic Block)
  - edges represent *control flow* (loops, if/else, goto …), there is an edge $b_1 \rightarrow b_2$ if control may flow from last stmt of b_1 to first stmt of b_2 in some execution

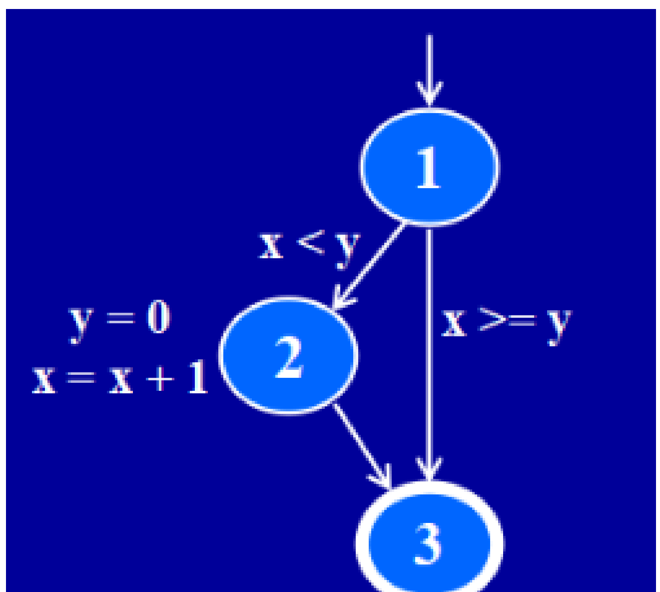
```
if (x < y)
{
    y = 0;
    x = x + 1;
}
else
{
    x = y;
}
```



A CFG is a conservative approximation of the control flow! Why?

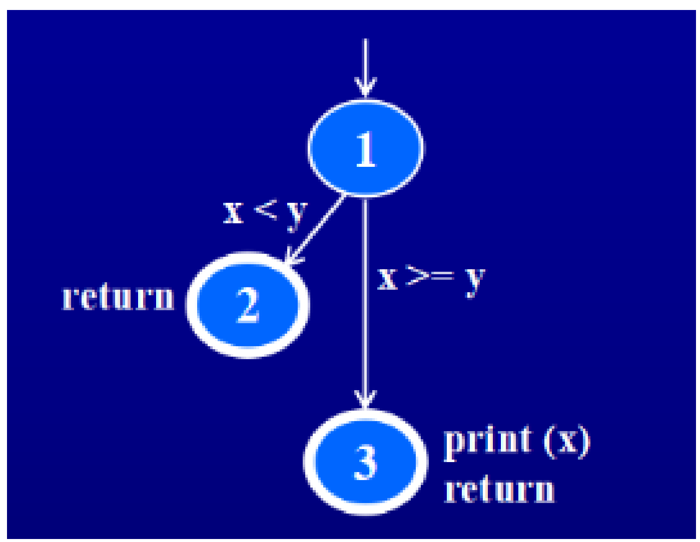
Control flow graph

```
if (x < y)
{
    y = 0;
    x = x + 1;
}
```



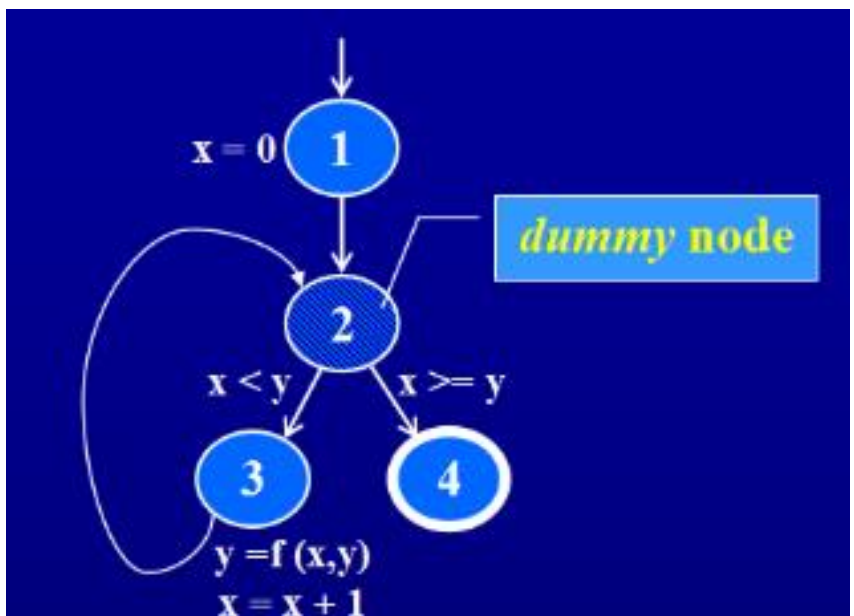
Control flow graph

```
if (x < y)
{
    return;
}
print (x);
return;
```



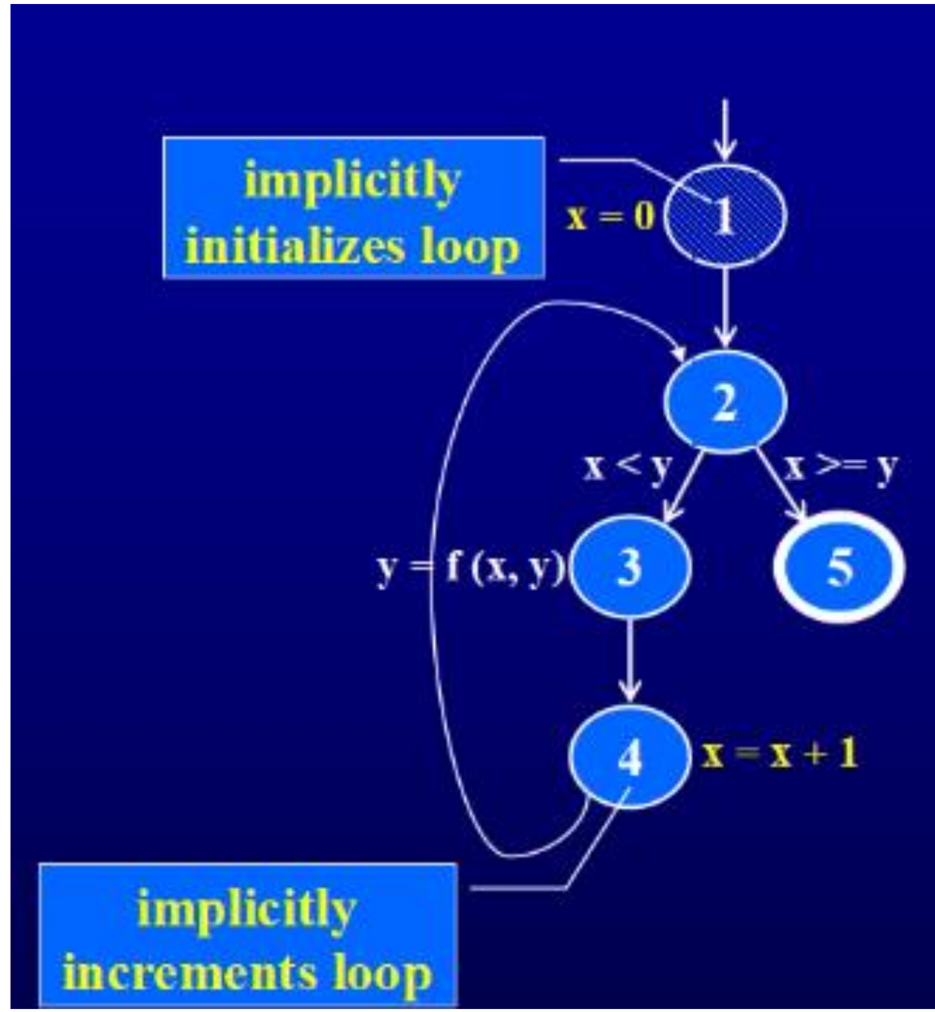
Control flow graph

```
x = 0;
while (x < y)
{
    y = f (x, y);
    x = x + 1;
}
```



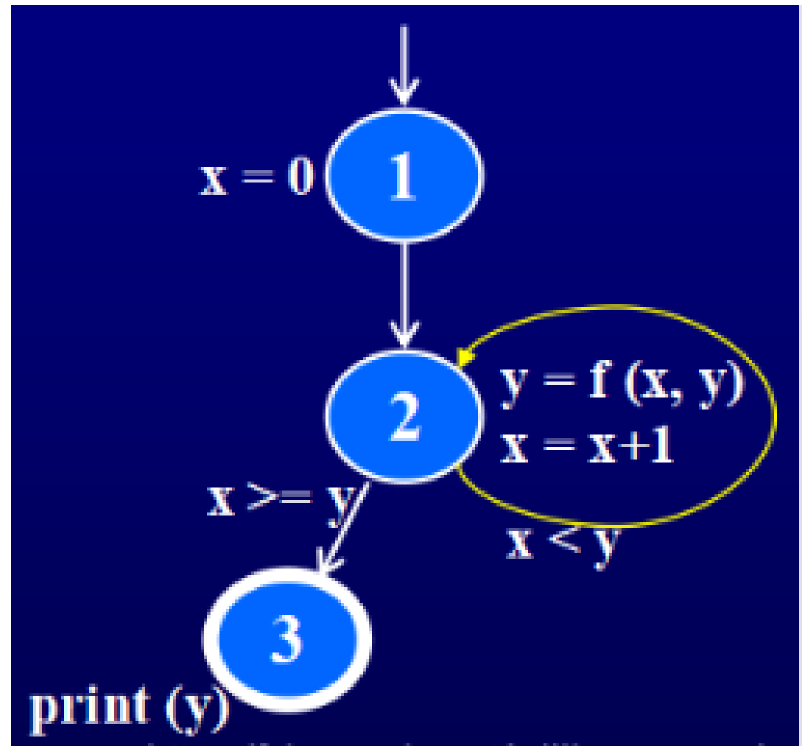
Control flow graph

```
for (x = 0; x < y; x++)
{
    y = f (x, y);
}
```



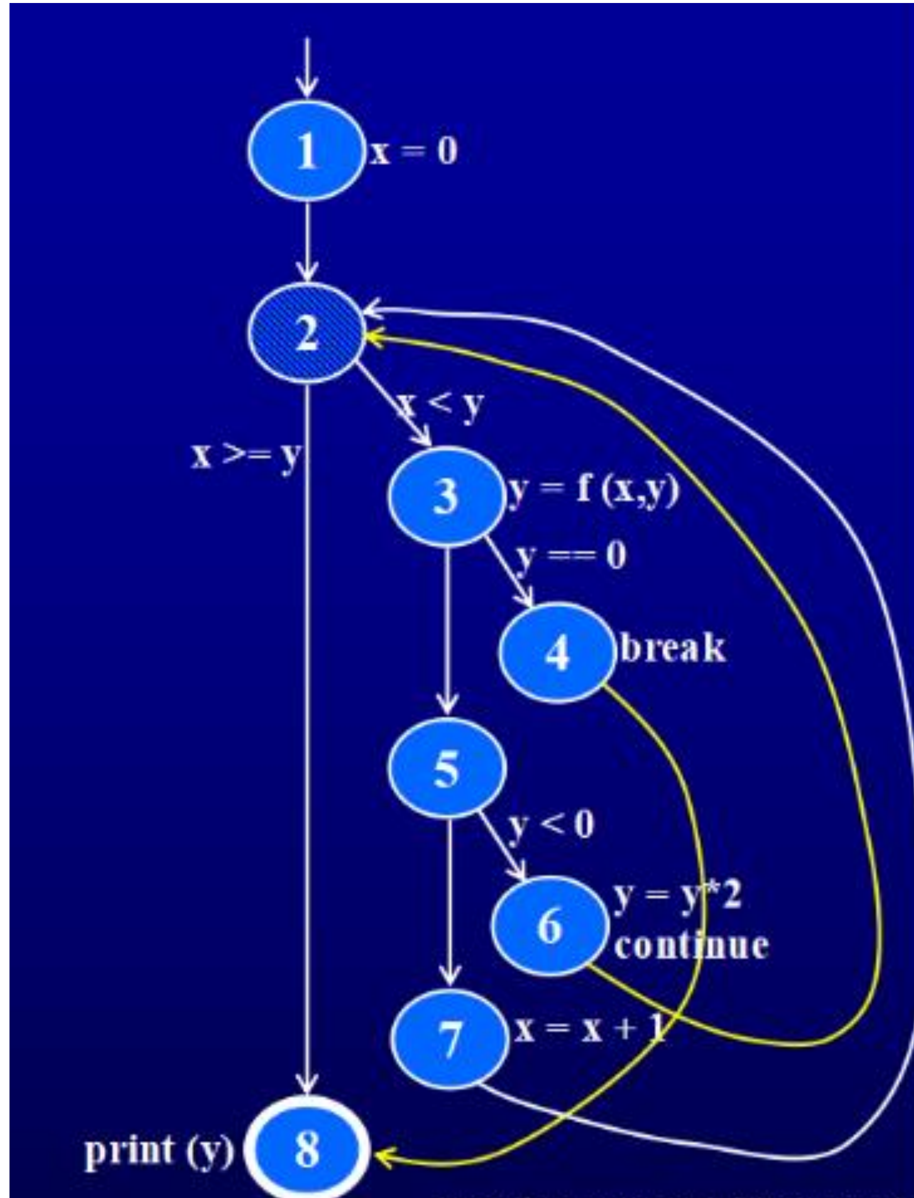
Control flow graph

```
x = 0;
do
{
    y = f (x, y);
    x = x + 1;
} while (x < y);
println (y)
```



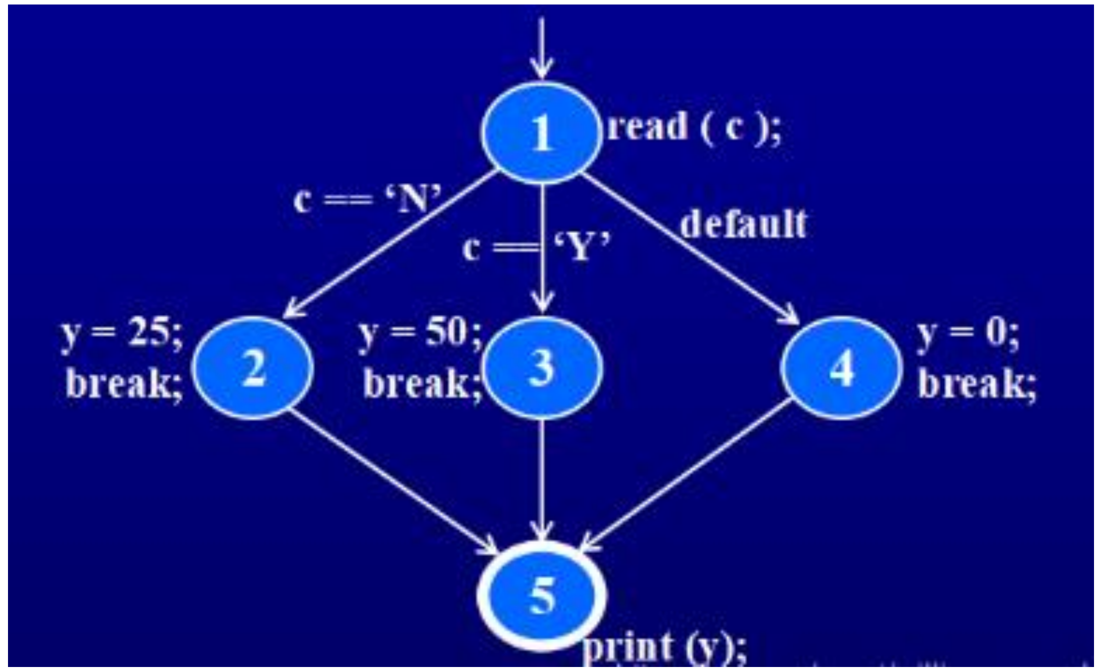
Control flow graph

```
x = 0;
while (x < y)
{
    y = f (x, y);
    if (y == 0)
    {
        break;
    } else if (y < 0)
    {
        y = y*2;
        continue;
    }
    x = x + 1;
}
print (y);
```



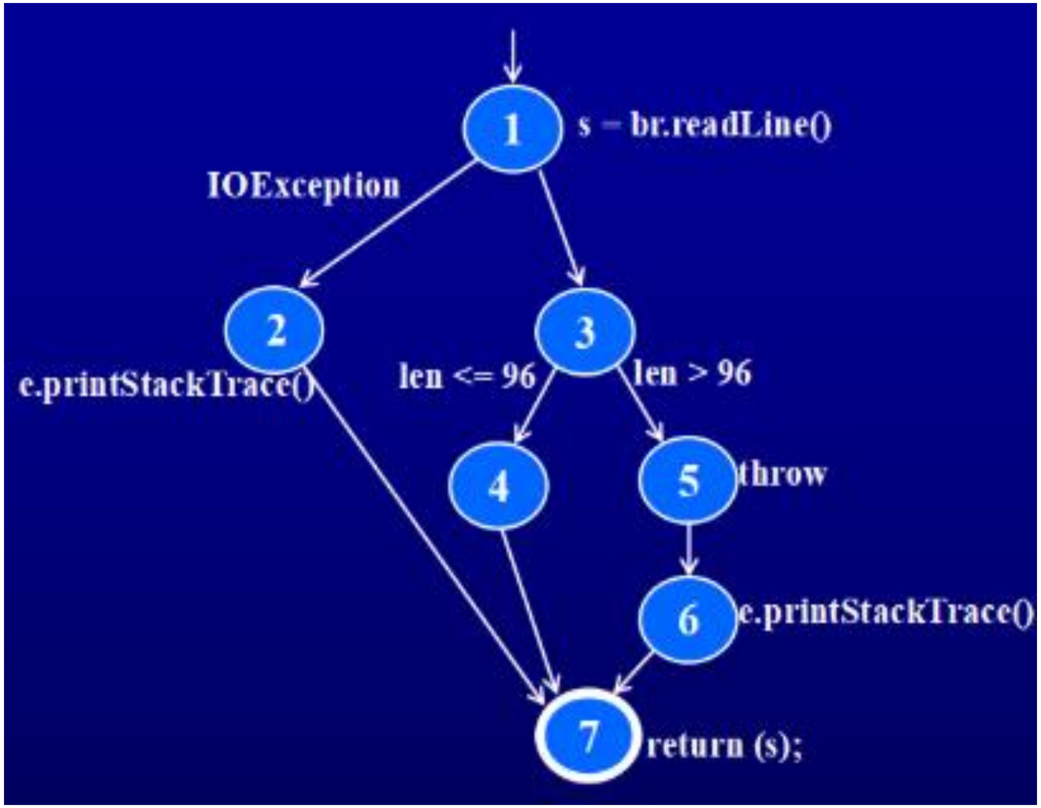
Control flow graph

```
read ( c) ;
switch ( c )
{
    case 'N':
        y = 25;
        break;
    case 'Y':
        y = 50;
        break;
    default:
        y = 0;
        break;
}
print (y);
```



Control flow graph

```
try
{
    s = br.readLine();
    if (s.length() > 96)
        throw new Exception
            (“too long”);
} catch IOException e) {
    e.printStackTrace();
} catch Exception e) {
    e.printStackTrace();
}
return (s);
```



Single static assignment (SSA)

> Each assignment to a temporary is given a unique name
  - All uses reached by that assignment are renamed
  - Compact representation
  - Useful for many kinds of compiler optimization …
  - Property: each variable is defined once

> Usage
  - constant propagation
  - value range propagation
  - sparse conditional constant propagation
  - dead code elimination, ...see list in wiki

```
x := 3;
x := x + 1;
x := 7;
x := x*2;
```

➔

x_1 := 3;
x_2 := x_1 + 1;
x_3 := 7;
x_4 := x_3*2;

What about flow of control?

Ron Cytron, et al., *"Efficiently computing static single assignment form and the control dependence graph,"* ACM TOPLAS., 1991. doi:10.1145/115372.115320

Single static assignment (SSA)

How do we know which one to use for X or j? φ-functions!

- *2-way branch*:

```
if (...)
        X = 5;
else
        X = 3;

Y = X;
```

$$
\begin{aligned}
&\text{if } (\ldots)\\
&\qquad X_0 = 5;\\
&\text{else}\\
&\qquad X_1 = 3;\\
&X_2 = \phi(X_0, X_1);\\
&Y_0 = X_2;
\end{aligned}
$$

- *While loop*:

```
    j = 1;
S: // while (j < x)
    if (j >= X)
        goto E;
    j = j+1;
    goto S
E:
    N = j;
```

$$
\begin{aligned}
&\qquad j_5 = 1;\\
&S:\qquad j_2 = \phi(j_5, j_4);\\
&\qquad \text{if } (j_2 >= X)\\
&\qquad\qquad \text{goto } E;\\
&\qquad j_4 = j_2 + 1;\\
&\qquad \text{goto } S\\
&E:\\
&\qquad N = j_2;
\end{aligned}
$$

will generate a new definition of X_0, X_1, by "choosing" either X_0 or X_1, depending on which arrow control arrived from

Single static assignment (SSA)

- **Definition (ϕ Functions)**:
  In a basic block B with N predecessors, $P_1,\ P_2,\ \ldots,\ P_N$,
  $$X = \phi(V_1, V_2, \ldots, V_N)$$
  assigns $X = V_j$ if control enters block B from P_j, $1 \leq j \leq N$.

- *Properties of ϕ-functions*:
  - ϕ is <u>not</u> an executable operation.
  - ϕ has exactly as many arguments as the number of incoming BB edges
  - Think about ϕ argument V_i as being evaluated on CFG edge from predecessor P_i to B

How to place φ-function?

If basic block B contains an assignment to a variable V, then a ϕ must be inserted in each basic block Z such that all of these are true:

1. there is a non-empty path B $\rightarrow^+$ Z;
2. there is a path from ENTRY to Z that does not go through B;
3. Z is the first node on the path B $\rightarrow^+$ Z that satisfies (2).

These conditions must be reapplied for every Φ inserted in the code!

Stack machine code

Used in compilers for stack architectures: B5500, B1700, P-code, BCPL Popular again for bytecode languages: JVM, MSIL

Advantages

- compact form
- introduced names are implicit, not explicit
- simple to generate & execute code

Disadvantages

- does not match current architectures
- many spurious dependences due to stack:
  ⇒ difficult to do reordering transformations
- cannot "reuse" expressions easily (must store and re-load)

⇒ difficult to express optimized code

Example

$$x - 2 * y - 2 * z$$

Stack machine code:

```
push x
push 2
push y
multiply
push 2
push z
multiply
add
subtract
```

3-address code

A term used to describe many different representations

Each statement ≡ single operator + at most three operands

> Statements take the form: `x = y op z`
>
> — single operator and at most three names

```
x - 2 * y
```

→

```
t1 = 2 * y
t2 = x - t1
```

Advantages:

- — compact form
- — makes intermediates values explicit
- — suitable for many levels (high, mid, low):
  - high-level: e.g., array refs, min / max ops
  - mid-level : e.g., virtual regs, simple ops
  - low-level : close to assembly code

Disadvantages

- Large name space (due to temporaries)
- Loses syntactic structure of source

Typical 3-address codes

assignments	`x = y op z`
	`x = op y`
	`x = y[i]`
	`x = y`

branches	`goto L`
conditional branches	`if x relop y goto L`

procedure calls	`param x`
	`param y`
	`call p`

address and pointer assignments	`x = &y` `*y = z`

3-address code — two variants

Quadruples

`x - 2 * y`

(1)	load	t1	y	
(2)	loadi	t2	2	
(3)	mult	t3	t2	t1
(4)	load	t4	x	
(5)	sub	t5	t4	t3

- simple record structure
- easy to reorder
- explicit names

Triples

`x - 2 * y`

(1)	load	y	
(2)	loadi	2	
(3)	mult	(1)	(2)
(4)	load	x	
(5)	sub	(4)	(3)

- table index is implicit name
- only 3 fields
- harder to reorder

IR choices

- Other hybrids exist
  - combinations of graphs and linear codes
  - CFG with 3-address code for basic blocks
- Many variants used in practice
  - no widespread agreement
  - compilers may need several different IRs!
- Advice:
  - choose IR with right level of detail
  - keep manipulation costs in mind

Compilation Strategies

High-level Model

- Retain high-level data types: Structs, Arrays, Pointers, Classes
- Retain high-level control constructs (AST) OR 3-address code
- Generally operate directly on program variables (i.e., no registers)

Mid-level Model

- Retain some high-level data types: Structs, Arrays, Pointers
- Linear 3-address code + CFG
- Distinguish virtual regs from memory
- No low-level architectural details

Low-level Model

- Linear memory model (no high-level data types)
- Distinguish virtual registers from memory
- Low-level 3-address code + CFG
- Explicit addressing arithmetic
- Expose all low-level architectural details: Addressing modes, stack frame, calling conventions, data layout

Example of the Real Compiler LLVM

LLVM Compiler (C, C++, . . .)

Code ≡ CFG + Mostly 3-address IR in SSA form

Analysis info ≡ Value Numbering + Points-to graph + Call graph

Basic blocks: doubly linked list of LLVM instructions