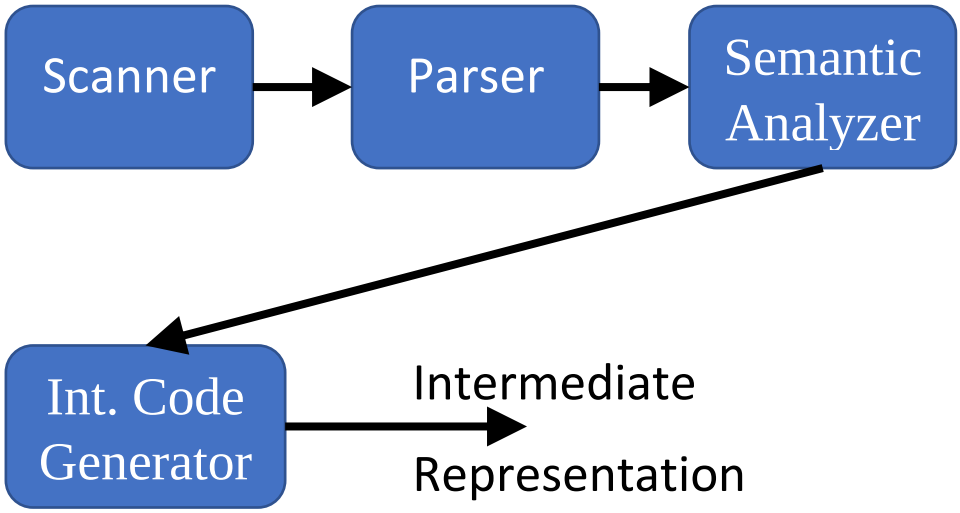
# Lecture Overview

## Intermediate Representations

- Graphical IR
- Linear IR
  - 1-address code
  - 3-address code
  - basic blocks
- Hybrid IR
  - constructing a control-flow graph
- Constructing an expression DAG
- Static Single-assignment form
  - constructing maximal SSA

[Chapter 5]

# Intermediate Representations




Compilers use the intermediate representation (IR) as the reference form of the input program for the later phases. (The original source is never consulted.) The properties of the IR have a direct effect on what the compiler can do with the code.

Most phases of compilation use the IR. Some compilers use more than one IR, sequentially.

An intermediate representation can be:

- graphical
- linear
- hybrid

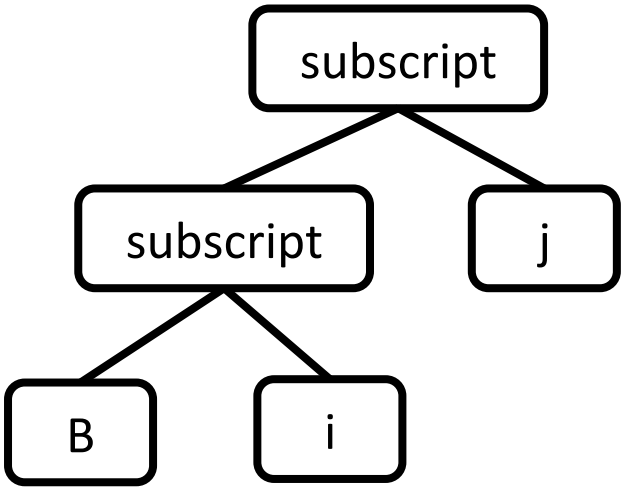
**Graphical IRs** encode the program mainly as a graph structure. Examples include the decorated AST, and expression DAGs.

**Linear IRs** encode the program as a linear sequence of instructions.

**Hybrid IRs** encode the program partly as graph, partly as linear instruction sequences.

Another axis of classification of IRs is **abstraction level**: a **high-level IR** represents concepts as abstractions, generally close to the input language. A **low-level IR** represents operations concretely, generally close to the machine level.

For example, consider that we have an array B[][] with four-byte elements and subelements, and with both subscripts running from 1 to 10, and we wish to represent the array reference B[i][j]. In a high-level tree representation, we could have:



Whereas in a low-level linear representation, we might have:

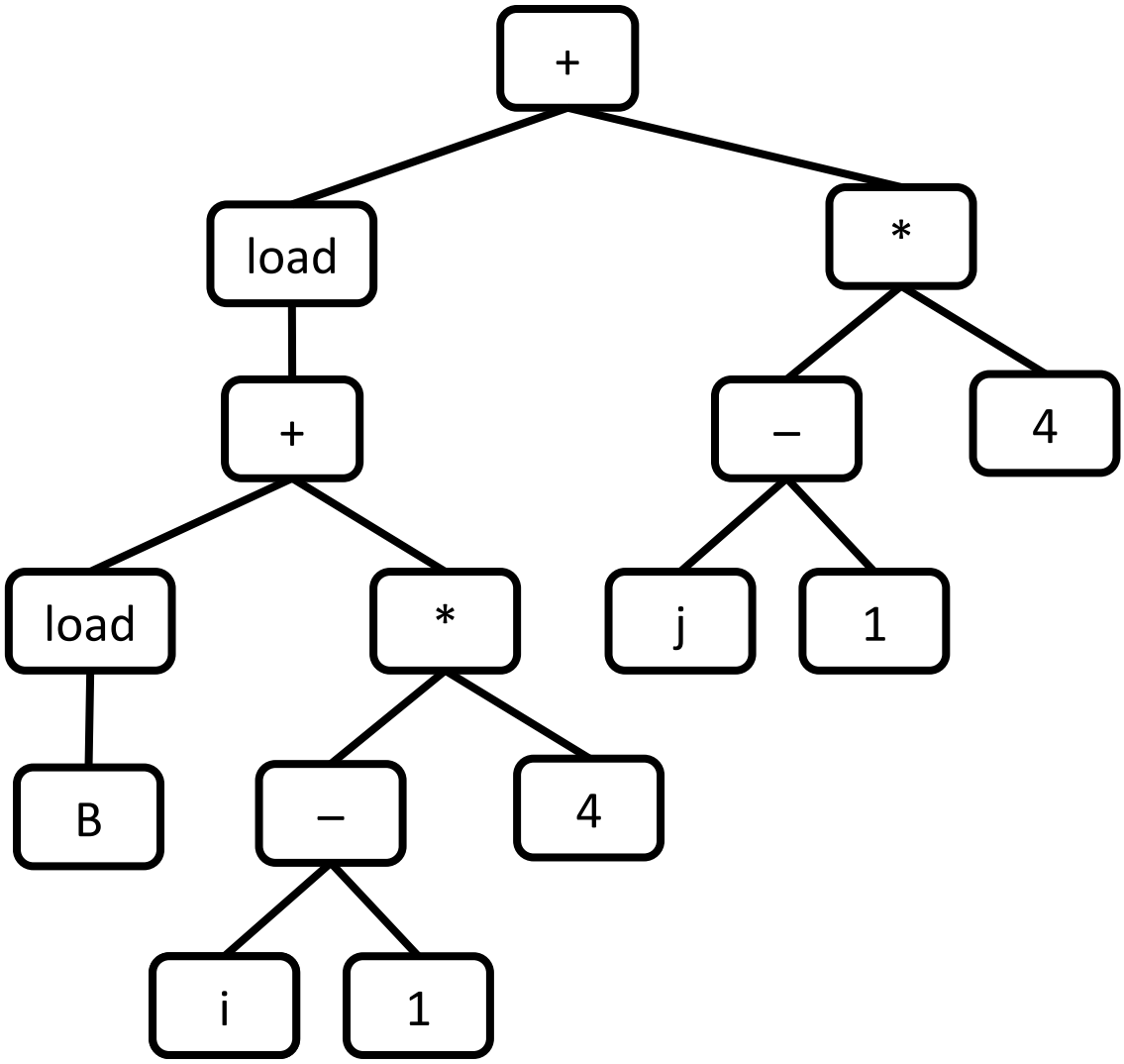
$r_1$ = load B
$r_2$ = i – 1
$r_3$ = $r_2$ * 4
$r_4$ = $r_1$ + $r_3$
$r_5$ = load $r_4$
$r_6$ = j – 1
$r_7$ = $r_6$ * 4
$r_8$ = $r_5$ + $r_7$

Note: this is simple address code, with no array headers or bounds checking. With headers and bounds checking, this code would be much longer and more obscure.

Here's a high-level linear representation:

$r_1$ = B[i]
$r_2$ = $r_1$[j]

And for completeness, here's a low-level tree IR:




Some tasks are best accomplished on high-level IR, and some on low-level IR. Generally a compiler will do all the high-level tasks on high-level IR, and then **lower** the IR to low-level to do low-level tasks.

An example of a high-level task is **alias analysis**: determining whether two variables possibly refer to the same memory location. An example low-level task is recognizing **loop-invariant expressions**. These are expressions whose value doesn't change over the iterations of a loop.

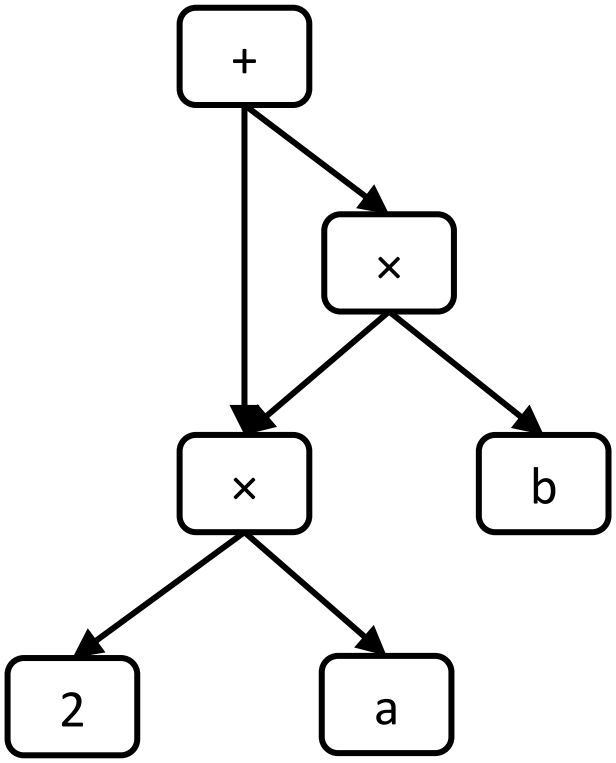
Another axis of difference between IRs is in the naming of values that are generated. Some IRs reuse names and some do not.

| | |
|---|---|
| $t_1 = b$ | $t_1 = b$ |
| $t_2 = 2 \times t_1$ | $t_1 = 2 \times t_1$ |
| $t_3 = a$ | $t_2 = a$ |
| $t_4 = t_3 - t_2$ | $t_1 = t_2 - t_1$ |

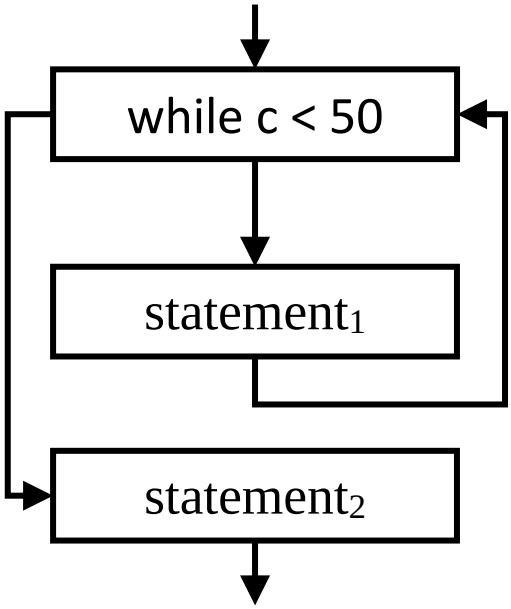
The right code uses fewer temporaries. However, if the code went on to use 2×b again, then the left code could reuse $t_2$ whereas the right code must again compute the expression.

# Graphical IRs

- **ASTs.** Can be high-level, as in our project, or low-level, revealing addressing and other calculations.
- **Expression DAGs.** For example, the expression 2a + 2ab could have the DAG:

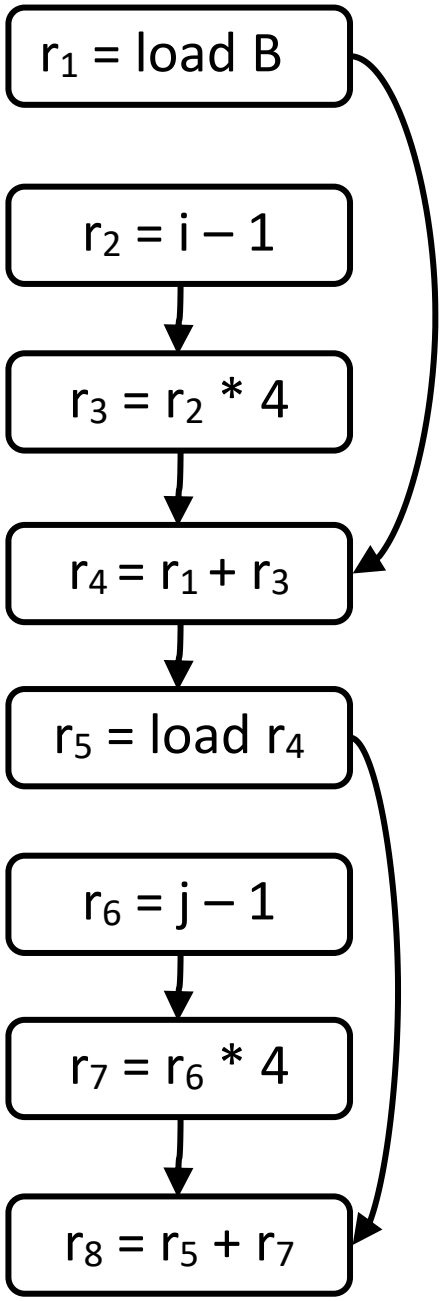


- **Control-flow graphs.** A graphical representation of possible paths of control flow at run time. A graph of which statements can follow each statement.




- **Call graphs.** A graph of which subroutines call which other subroutines.

- **Dependence graphs.** A graph of which statements depend on values created by each statement.



## Linear IRs

Linear IRs used in compilers resemble the assembly code for an abstract machine. They impose a clear and useful ordering on the sequence of operations.

Linear IRs must be able to model transfer of control from one part of the code to another. Most frequently it models the implementation of control flow on the target machine: conditional branches and jumps.

Linear IRs can be classified by the number of operands/addresses specified in any one instruction.

- **One-address code.** These model the behaviour of accumulator or stack machines. The resulting code is quite compact. Our ASM code is a one-address code. Java, Smalltalk, and Scala all compile to one-address code.
- **Two-address code.** These model a machine with destructive operations (operations take two operands and write the result over one of them). Not popular or useful nowadays.
- **Three-address code.** These model a machine where most operations take two operands and produce a result. The resulting code resembles code for a RISC machine. Very popular.

## Three-address code

In three-address code (3AC) most operations have the form

i = j **op** k

with operator **op**, two operands j and k, and result i. Some operations, such as loads and jumps, need fewer operands.

3AC is reasonably compact; most operations consist of four elements: an opcode and three names. A record that stores this information is often called a **quadruple.**

Both the opcode and the names are often drawn from limited sets. Opcodes generally require one or two bytes. Names are typically represented by integers or table indices; 4 bytes is generally enough for one. So 3AC can generally be represented in about 14 bytes/instruction.

Several methods of collecting quadruples together to form code have been used: arrays, arrays of pointers, and linked lists.

## Basic Blocks

A basic block is a maximal set of consecutive linear (say, 3AC) instructions that must be executed together. A basic block starts with a **leader** that is either the first instruction in a subroutine, a labelled instruction (which can be the target of a jump or branch), or the statement after a branch. A basic block ends at the first branch, jump, or subroutine call, return or other leader after its leader.

```
1                 s = 0
2                 j = 0
3                 c = 8
4     loop:       branch c == 0 exit
5                 s = s + c
6                 j = j + s
7                 c = c - 1
8                 jump loop
9     exit:       k = j
10                return
```

In this code, the leaders are lines 1, 4, 5, and 9.

The basic blocks are:

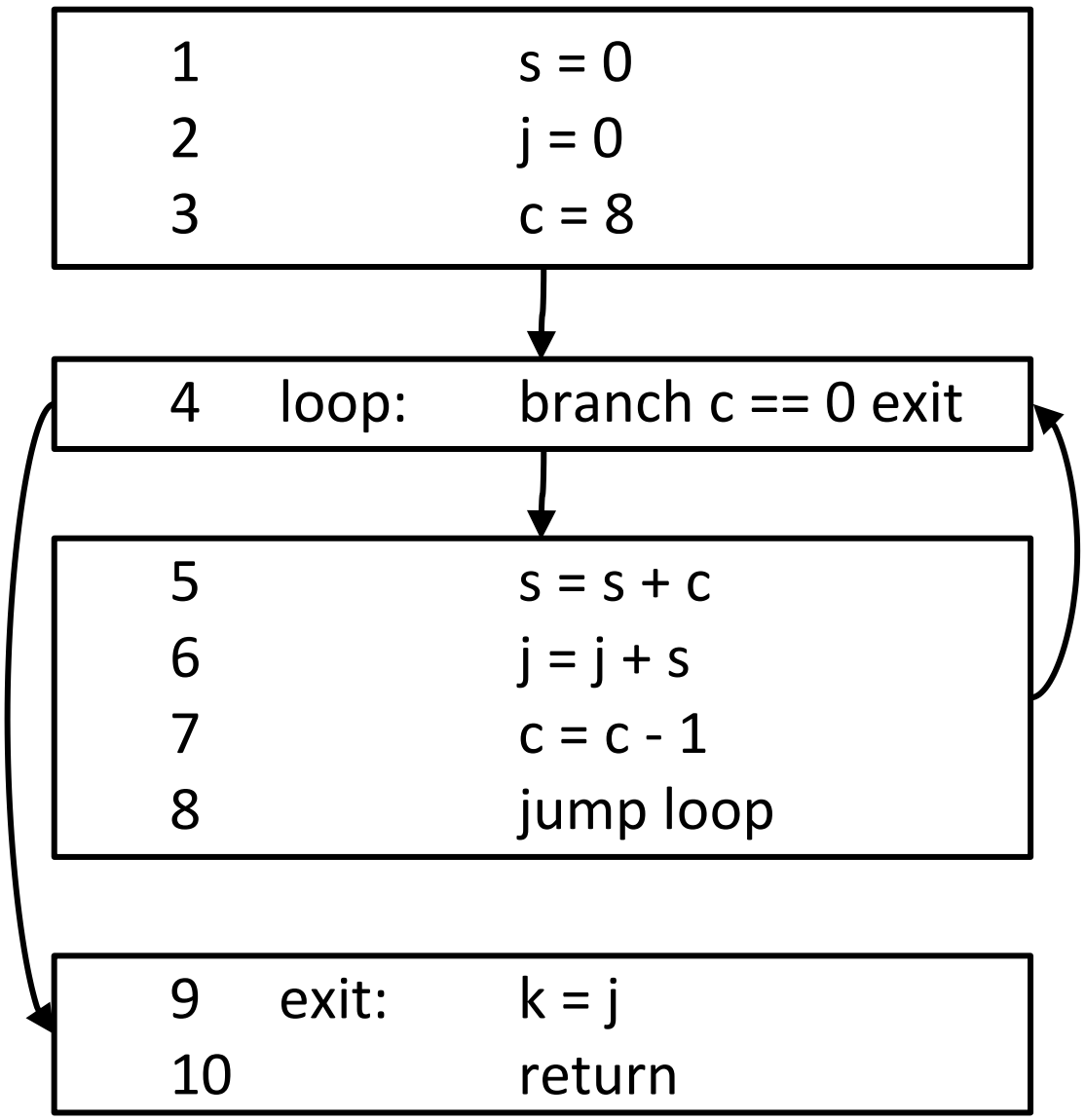
```
1           s = 0
2           j = 0
3           c = 8
```

```
4    loop:  branch c == 0 exit
```

```
5           s = s + c
6           j = j + s
7           c = c - 1
8           jump loop
```

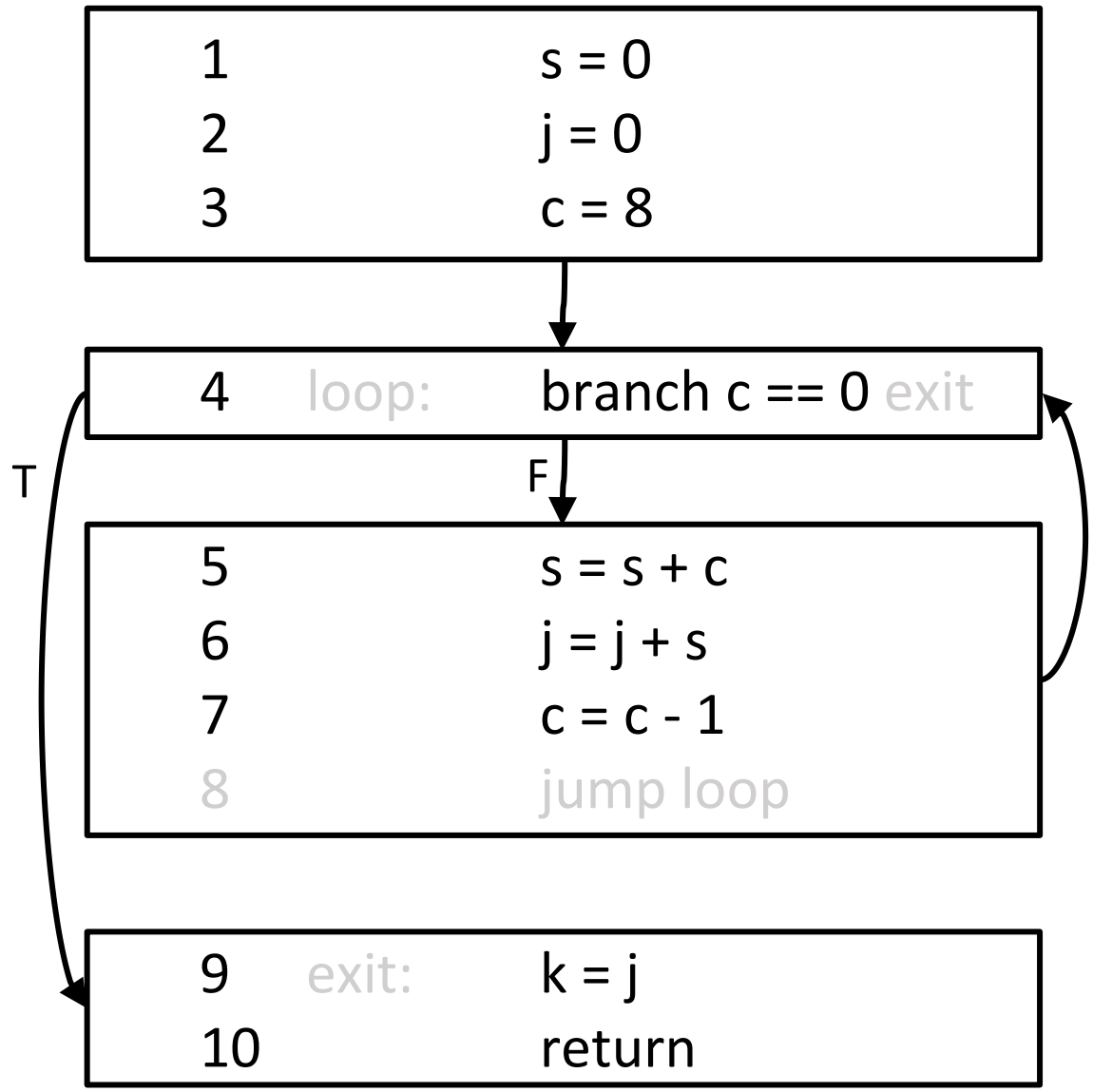
```
9    exit:  k = j
10          return
```

If we add directed edges to show where program control can go, we arrive at the **control flow graph** (CFG) of the subroutine.




The CFG is a very common **hybrid IR.**

With a CFG, if we note which branch targets are true and false, labels and jumps are not necessary.



In computationally intensive code, basic blocks can become very long. This is good because it allows many opportunities for optimization.

$r_1$ = load B
$r_2$ = i – 1
$r_3$ = $r_2$ * 4
$r_4$ = $r_1$ + $r_3$
$r_5$ = load $r_4$
$r_6$ = j – 1
$r_7$ = $r_6$ * 4
$r_8$ = $r_5$ + $r_7$
$r_9$ = load $r_8$
$r_{10}$ = load C
$r_{11}$ = i – 1
$r_{12}$ = $r_{11}$ * 4
$r_{13}$ = $r_{10}$ + $r_{12}$
$r_{14}$ = load $r_{13}$
$r_{15}$ = j – 1
$r_{16}$ = $r_{15}$ * 4
$r_{17}$ = $r_{14}$ + $r_{16}$
$r_{18}$ = load $r_{17}$
$r_{19}$ = $r_9$ + $r_{18}$
$r_{20}$ = load A
$r_{21}$ = i – 1
$r_{22}$ = $r_{21}$ * 4
$r_{23}$ = $r_{20}$ + $r_{22}$
$r_{24}$ = load $r_{23}$
$r_{25}$ = j – 1
$r_{26}$ = $r_{25}$ * 4
$r_{27}$ = $r_{24}$ + $r_{26}$
store $r_{27}$, $r_{19}$

A[i][j] = B[i][j] + C[i][j]

# Optimizations

Optimizations done on a single basic block are called **local** optimizations.

Optimizations done on a single procedure are called **global** optimizations.

Optimizations done on a bigger scale are called **interprocedural** optimization.

Some authors separate out **intermodular** optimizations as those done between files. Others group these with interprocedural.

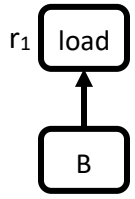
# Using expression DAGs for local optimization

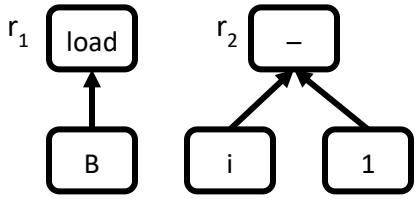
First we must construct a DAG from a basic block.

For each instruction:

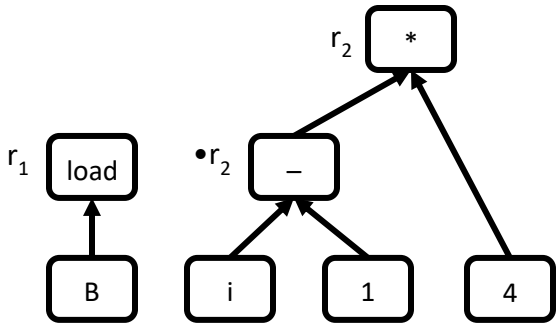
1. create node for each argument if the node does not already exist.
2. create node for operation on arguments if a node with the same operation and arguments does not already exist. Connect this new node to its arguments.
3. If a new node was created in (2), then call it by the result name. If not, add the result name to the existing node. If another node with the same name exists, deprecate that node so that the new one is henceforth used.

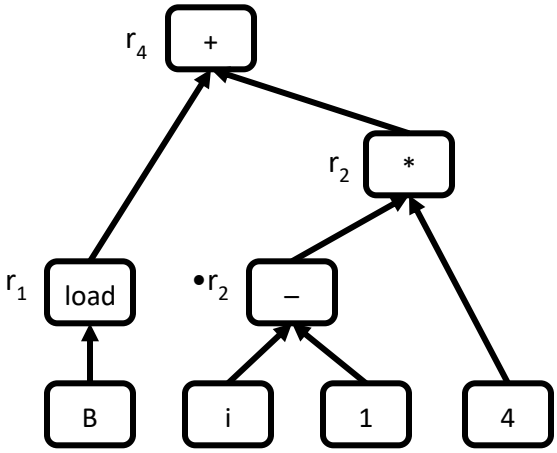
$r_1$ = load B
$r_2$ = i – 1

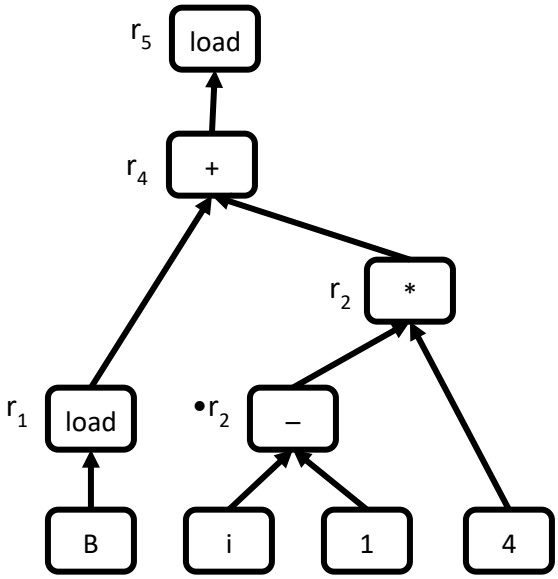


$r_2 = r_2 * 4$
$r_4 = r_1 + r_2$
$r_5 = \text{load } r_4$

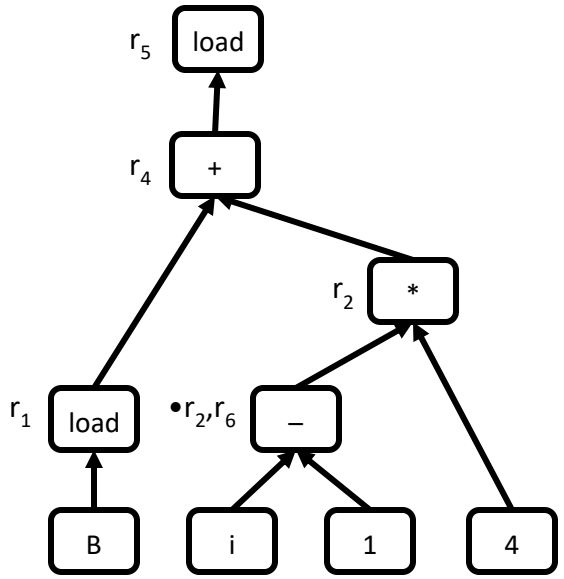


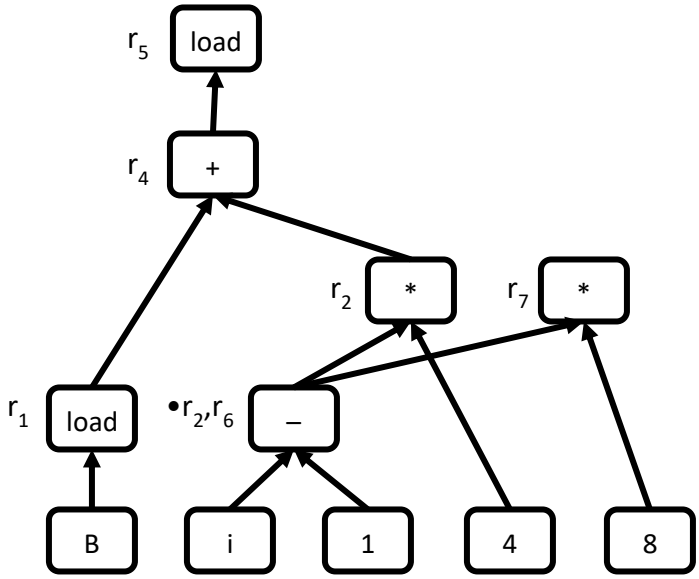



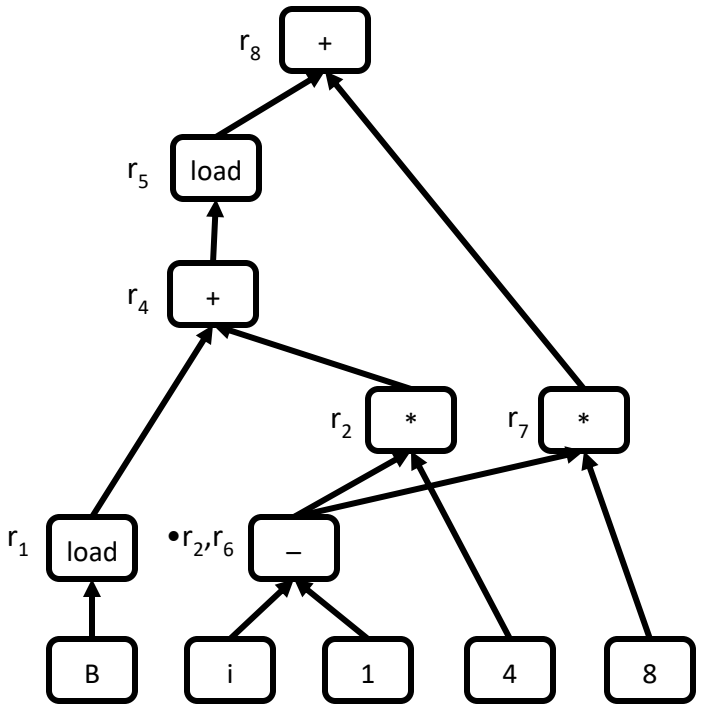
$r_6 = i - 1$

$r_7 = r_6 * 8$


r5
load
r4
+
r2
*
r1
load
•r2,r6
–
B
i
1
4


r5
load
r4
+
r2
*
r7
*
r1
load
•r2,r6
–
B
i
1
4
8

$r_8 = r_5 + r_7$




## Before DAG

$r_1 = \text{load } B$
$r_2 = i - 1$
$r_2 = r_2 * 4$
$r_4 = r_1 + r_2$
$r_5 = \text{load } r_4$
$r_6 = i - 1$
$r_7 = r_6 * 8$
$r_8 = r_5 + r_7$

## After DAG

$r_1 = \text{load } B$
$r_6 = i - 1$
$r_2 = r_6 * 4$
$r_4 = r_1 + r_2$
$r_5 = \text{load } r_4$
$r_7 = r_6 * 8$
$r_8 = r_5 + r_7$

# Static Single-Assignment form

Static Single-Assignment (SSA) is a naming and encoding discipline that encodes information about the flow of control and flow of data. In SSA, a name is only assigned to at one point in the code; each name is defined by one operation.

Each use of a variable therefore, through the name, encodes some information about where the variable was defined. To reconcile this discipline with the effects of control flow, SSA form contains special operations, called ϕ-functions, at points where control-flow paths meet.

| Original | SSA | |
|---|---|---|
| x = 0 | | $x_0 = 0$ |
| y = 0 | | $y_0 = 0$ |
| while( x < 100) | loop: | $x_1 = \phi(x_0, x_2)$ |
| x = x + 1 | | $y_1 = \phi(y_0, y_2)$ |
| y = y + x | | if $x_1 \geq 100$ goto next |
| | | $x_2 = x_1 + 1$ |
| | | $y_2 = y_1 + x_2$ |
| | | goto loop |
| | next: | |

A ϕ-function selects whichever of its arguments was last assigned to. In the code above, the first ϕ-function selects $x_0$ the first time it is executed, and it selects $x_2$ each time after that.

The convention is that all ϕ-functions at the beginning of a basic block execute concurrently. First they evaluate their arguments, then they define their result names. This convention allows compilers to ignore the ordering of ϕ-functions, which is helpful in converting SSA back to executable code.

SSA ϕ-functions are not of fixed arity. If five control flow paths converge at one point, one may find a five-argument ϕ-function there. They don't naturally fit into 3AC data structures.

## Building Maximal SSA form code

There are two steps:

1. **insert ϕ-functions.** At the start of each basic block with multiple predecessors, insert a ϕ-function such as y = ϕ(y, y) for each name y that the code either defines or uses in the current procedure. This rule inserts a ϕ-function in every case where one is needed, but also inserts many extraneous ϕ-functions.
2. **rename.** After ϕ-functions have been inserted, each definition of a variable can be given its own subscript. ϕ-functions are definitions. Then at each line of code, we determine which definition of each of its arguments reaches that line, and rename the arguments.

```
        x = 0
        y = 0
loop:   if x ≥ 100 goto next
        x = x + 1
        y = y + x
        goto loop
next:
```

```
        x = 0
        y = 0
loop:   x = ϕ(x, x)
        y = ϕ(y, y)
        if x ≥ 100 goto next
        x = x + 1
        y = y + x
        goto loop
next:
```

```
        x0 = 0
        y0 = 0
loop:   x1 = ϕ(x, x)
        y1 = ϕ(y, y)
        if x ≥ 100 goto next
        x2 = x + 1
        y2 = y + x
        goto loop
next:
```

```
        x0 = 0
        y0 = 0
loop:   x1 = ϕ(x0, x2)
        y1 = ϕ(y0, y2)
        if x1 ≥ 100 goto next
        x2 = x1 + 1
        y2 = y1 + x2
        goto loop
next:
```