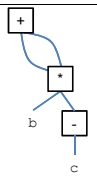
# Graph IRs

**Directed Acyclic Graphs**
Eliminate storage and calculation redundancy

**Example:**
```
a := b * (-c) + b * (-c)
```




May be used as a step from AST to IR or to convert to more efficient machine code.

Three address code of our example is now:

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

# Control Flow Graphs

The three-address IR looks like assembly language including control transfer instructions.

Labeling or numbering IR statements that are branch targets seems premature to do before code generation time because we may move or remove IR statements during the optimization phase.

We can create a hybrid IR that uses three-address code for straight-line portions of code and replace the control transfer instructions with a graph representation.

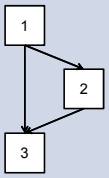
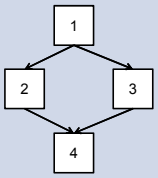
The graph representing the runtime flow of the program is called a **control flow graph**.

A control flow graph is a graph G = (N, E) where each node n ∈ N is a basic block and each edge e ∈ E is a control flow transfer between blocks (branch or fall through).

# Control Flow Graphs

Control flow graphs (**CFG**s – which we also use to abbreviate Context Free Grammars) are the flowchart representation of program logic.

**Examples:**

| If-then Statement | If-then-else Statement |
|---|---|
|  |  |



# Basic Blocks

The nodes of our CFG are each a basic block.

A **basic block** is a maximal unit of straight line code with no control transfers into it except at the start and no transfers out of the code except at the end.

Alternatively, it means:

- The first instruction in a basic block is the label of a branch/jump or a fall-through.
- The last instruction in a basic block is a branch, jump, return, or predicated instruction.

# Fall-throughs

Recall from our assembly-language that most branch instructions only encode one target:

```
        slt $t0, $s0, $s1
        bne $zero, $t0, L1
        addi $t1, $t1, 1
        …
L1:     …
```

This means that there is an implicit control transfer to the instruction following the branch in the case the condition evaluates to false. In the above code, we have 3 basic blocks.

# Fall-throughs

Recall from our assembly-language that most branch instructions only encode one target:

```
        slt $t0, $s0, $s1
        bne $zero, $t0, L1
        addi $t1, $t1, 1
        …
L1:     …
```

This means that there is an implicit control transfer to the instruction following the branch in the case the condition evaluates to false. In the above code, we have 3 basic blocks.

# Basic Blocks and Traces

When we generate code for a given CFG we are constructing a linear sequence of code that comes from a nonlinear CFG.

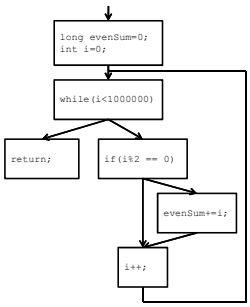
Any linearization of the CFG can result in proper operation, but are some better than others?

If we are using an assembly language with fall-through branches, we may be able to lay out several basic blocks in a row such that it is rare to take a branch.

This concatenation of basic blocks that could be executed together in sequence is called a **trace**.

Traces allow for the removal of unconditional jumps and to exploit architectures where there are significant performance penalties for branches.

# Building a CFG




```
long evenSum=0;
int i=0;

while(i<1000000) {
    if(i%2 == 0){
        evenSum+=i;
    }
    i++;
}

return;
```

# CFGs in Compilers

CFGs are used for a number of purposes in a compiler, mostly related to optimization.

Many of the algorithms are easier to implement if there is a single root node of the CFG and a single exit node.

We don't usually need to augment the CFG with a dummy entry node since most procedures only have one entry point already.

However, we may return from a procedure in several places in the code. We likely will wish to augment the CFG so that all blocks that contain a return instead transfer to a single block that contains the return point for the whole procedure.

# Naming Temporaries

| Source Code | Source Names | Value Names |
|---|---|---|
| a = b + c<br>b = a - d<br>c = b + c<br>d = a - d | t1 := b<br>t2 := c<br>t3 := t1 + t2<br>a := t3<br>t4 := d<br>t1 := t3 - t4<br>b := t1<br>t2 := t1 + t2<br>c := t2<br>t4 := t3 - t4<br>d := t4 | t1 := b<br>t2 := c<br>t3 := t1 + t2<br>a := t3<br>t4 := d<br>t5 := t3 - t4<br>**b := t5**<br>t6 := t5 + t2<br>c := t6<br>t5 := t3 - t4<br>**d := t5** |

**Source naming** uses fewer names than value naming and follows the source code names.

**Value naming** uses more names than source naming, however it ensures that textually identical expressions produce the same result

- `b` and `d` must receive the same value, something useful for optimization

# SSA

**Static Single Assignment** (SSA) was developed by R. Cytron, J. Ferrante, et al. in the 1980s.

Every variable is assigned exactly once, i.e., one **def** (definition)

Convert original variable name to $name_{version}$
e.g., x $\rightarrow x_1, x_2$ in different places as it is assigned to.

Use $\phi$-function to combine two defs of same original variable.

SSA is useful because it easily exposes several optimization opportunities.

# Phi Functions

| Source Code | SSA Form |
|---|---|
| x = 0;<br>y = 1;<br><br>while(x < 100) {<br>  x = x + 1;<br>  y = y + x;<br>} | $x_0$ := 0<br>$y_0$ := 1<br>if ($x_0$ > 100) goto next<br><br>loop: $x_1$ := $\phi(x_0, x_2)$<br>$y_1$ := $\phi(y_0, y_2)$<br>$x_2$ := $x_1$ + 1<br>$y_2$ := $y_1$ + $x_2$<br>if ($x_2$ < 100) goto loop<br><br>next: $x_3$ := $\phi(x_0,x_2)$<br>$y_3$ := $\phi(y_0,y_2)$ |

# Phi Functions

ϕ-functions are not three-address code.

- Need some alternate way to represent the variable number of arguments (one for each control-flow path to the block that assigns the variable).
- Perhaps use an extra data structure to hold the arguments

Where to insert ϕ-functions?

- Insert ϕ -functions for each value at the start of each basic block that has more than one predecessor in the CFG.
  - Too naïve, but it works
- Dominance Frontiers
  - Built upon several ideas, and is beyond the scope of this course.

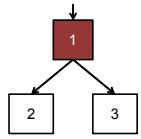
# Dominators

Certain blocks **dominate** other blocks in control flow graphs

- All paths from the root to a given basic block must go through the dominator

**Example:**

Block 1 dominates blocks 2 and 3




If a block A dominates another block B, then we do not need a ϕ-function as we know one of two things:

- The definitions of variables in A reach into B, unless
- A redefinition of a variable happens in the path between A and B