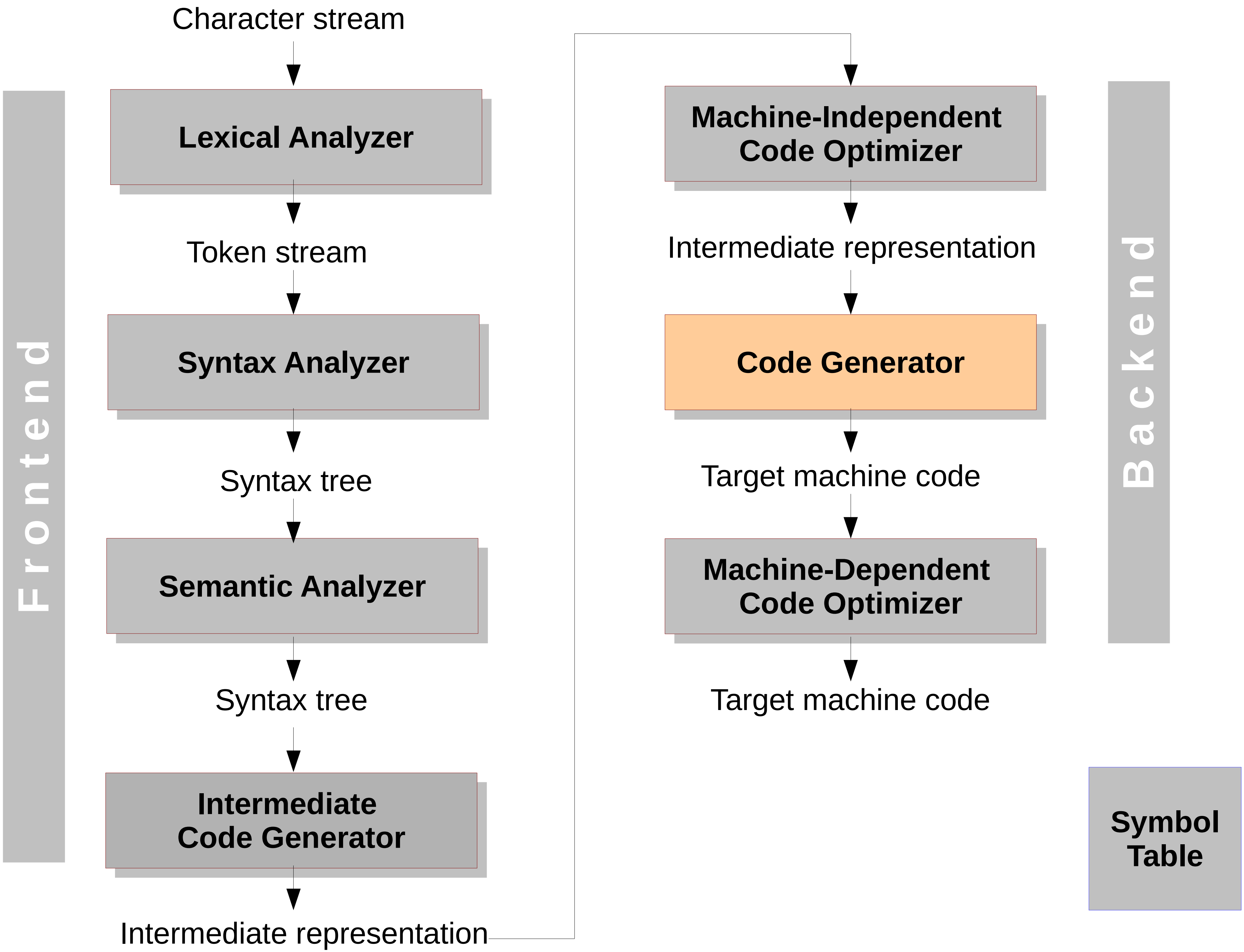
Code Generation

Rupesh Nasre.

CS3300 Compiler Design
IIT Madras
Jul 2018



Frontend

Character stream

Lexical Analyzer

Token stream

Syntax Analyzer

Syntax tree

Semantic Analyzer

Syntax tree

Intermediate
Code Generator

Intermediate representation

Machine-Independent
Code Optimizer

Intermediate representation

Code Generator

Target machine code

Machine-Dependent
Code Optimizer

Target machine code

Backend

Symbol
Table

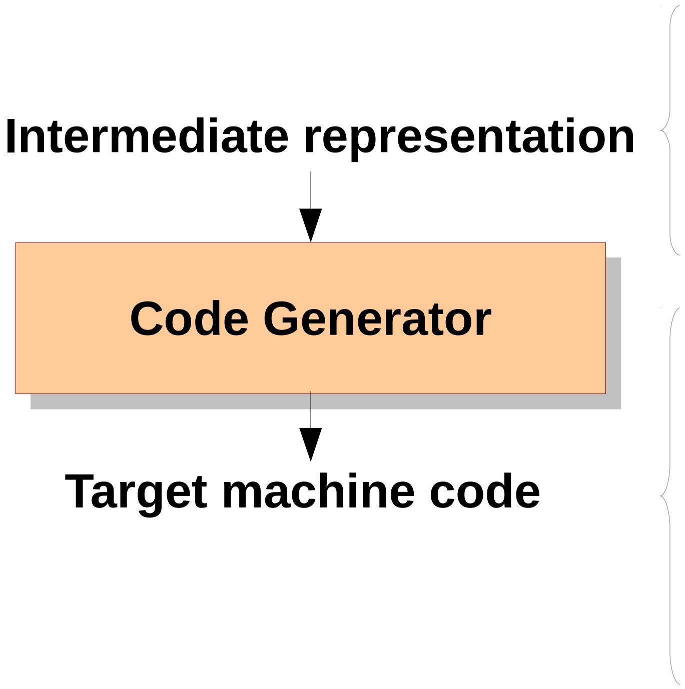
Role of Code Generator

- From IR to target program.
- Must preserve the semantics of the source program.
  - Meaning *intended* by the programmer in the original source program should carry forward in each compilation stage until code-generation.
- Target code should be of high quality
  - execution time or space or energy or …
- Code generator itself should run efficiently.
  1. instruction selection,
  2. register allocation and
  3. instruction ordering.

Code Generator in Reality

- The problem of generating an optimal target program is undecidable.
- Several subproblems are NP-Hard (such as register allocation).
- Need to depend upon
  - Approximation algorithms
  - Heuristics
  - Conservative estimates

Input + Output



Intermediate representation

Code Generator

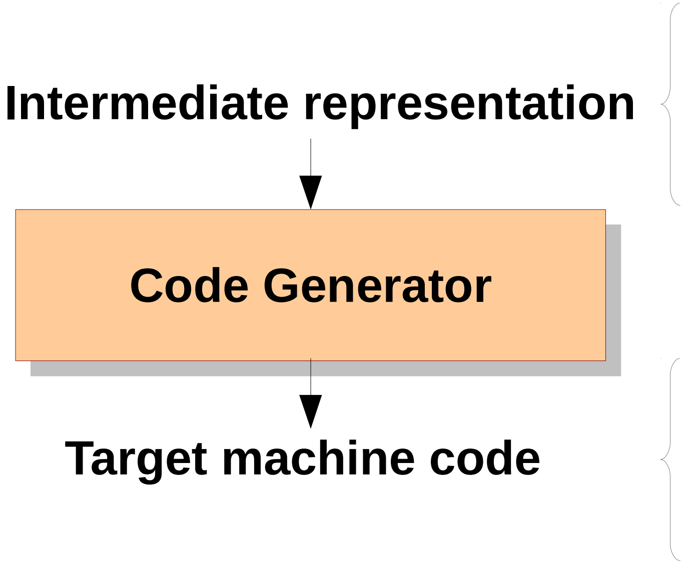
Target machine code

- **3AC** (Quadruples / Triples / Indirect triples)
- **VM** instructions (bytecodes / stack machine codes)
- **Linear** representations (postfix)
- **Graphical** representation (syntax trees / DAGs)

- **RISC** (many registers, 3AC, simple addressing modes, simple ISA)
- **CISC** (few registers, 2AC, variety of addressing modes, several register classes, variable length instructions, instructions with side-effects)
- **Stack** machine (push / pop, stack top uses registers, used in JVM, JIT compilation)

It helps to assume an assembler. Imagine if in A3 you had to generate machine code and manipulate bits rather than generating x86 assembly.

IR and Target Code

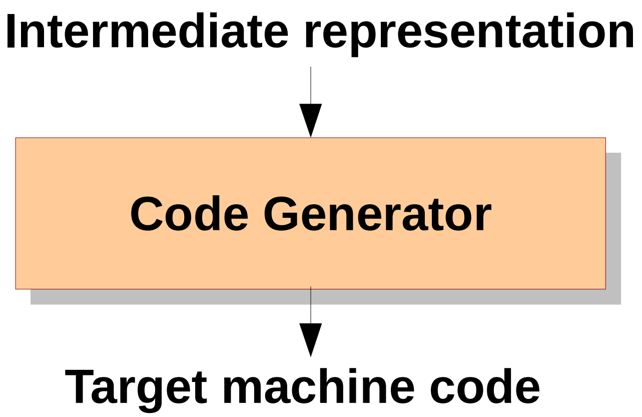


```
R0 = y
R0 = R0 + z
x = R0
```

```
LD R0, y
ADD R0, R0, z
ST x, R0
```

What is the issue with this kind of code generation?

IR and Target Code


Intermediate representation

Code Generator

Target machine code

```
R0 = y
R0 = R0 + z
x = R0
```

```
LD R0, y
ADD R0, R0, z
ST x, R0
```

Generate code for

a = b + c
d = a + e

LD R0, b
ADD R0, R0, c
ST a, R0
LD R0, a
ADD R0, R0, e
ST d, R0

1 Instruction Selection

- Complexity of instruction selection depends upon
  - Level of the IR

    Low-level IR can help generate more efficient code.

    e.g., *intsize* versus *4*.
  - Nature of the ISA

    Uniformity and completeness of ISA affects the code.

    e.g., floats required to be loaded in special registers.
  - Desired quality of the generated code

    Context and amount of information to process affects the code quality.

    e.g., INC *a* versus *LD R0, a; ADD R0, R0, #1; ST a, R0*

2

Register Allocation

- Register allocation involves
  - *Allocation*: which variables to be put into registers
  - *Assignment*: which register to use for a variable
- Finding an optimal assignment of registers to variables is NP-Complete.
- Architectural conventions complicate matters.
  - Combination of registers used for double-precision arithmetic.
  - Result is stored in accumulator.
  - Registers are reserved for special instructions.
  - ...

Thought exercise: How to use graph coloring for register allocation?

3

Instruction Ordering

- Instruction order affects execution efficiency.
- Picking the best order is NP-complete.
- Optimizer / Code generator needs to look at multiple instructions at a time.
- **Classwork**: Create an example IR whose generated code results in the same meaning but different efficiency for different orders.

```
1: R0 = a
2: R1 = b
3: R2 = c
4: R3 = R0 + R1
5: R4 = R2 + R3
6: d = R4
```

```
1: R0 = a
2: R1 = b
4: R2 = R0 + R1
3: R0 = c
5: R3 = R2 + R0
6: d = R3
```

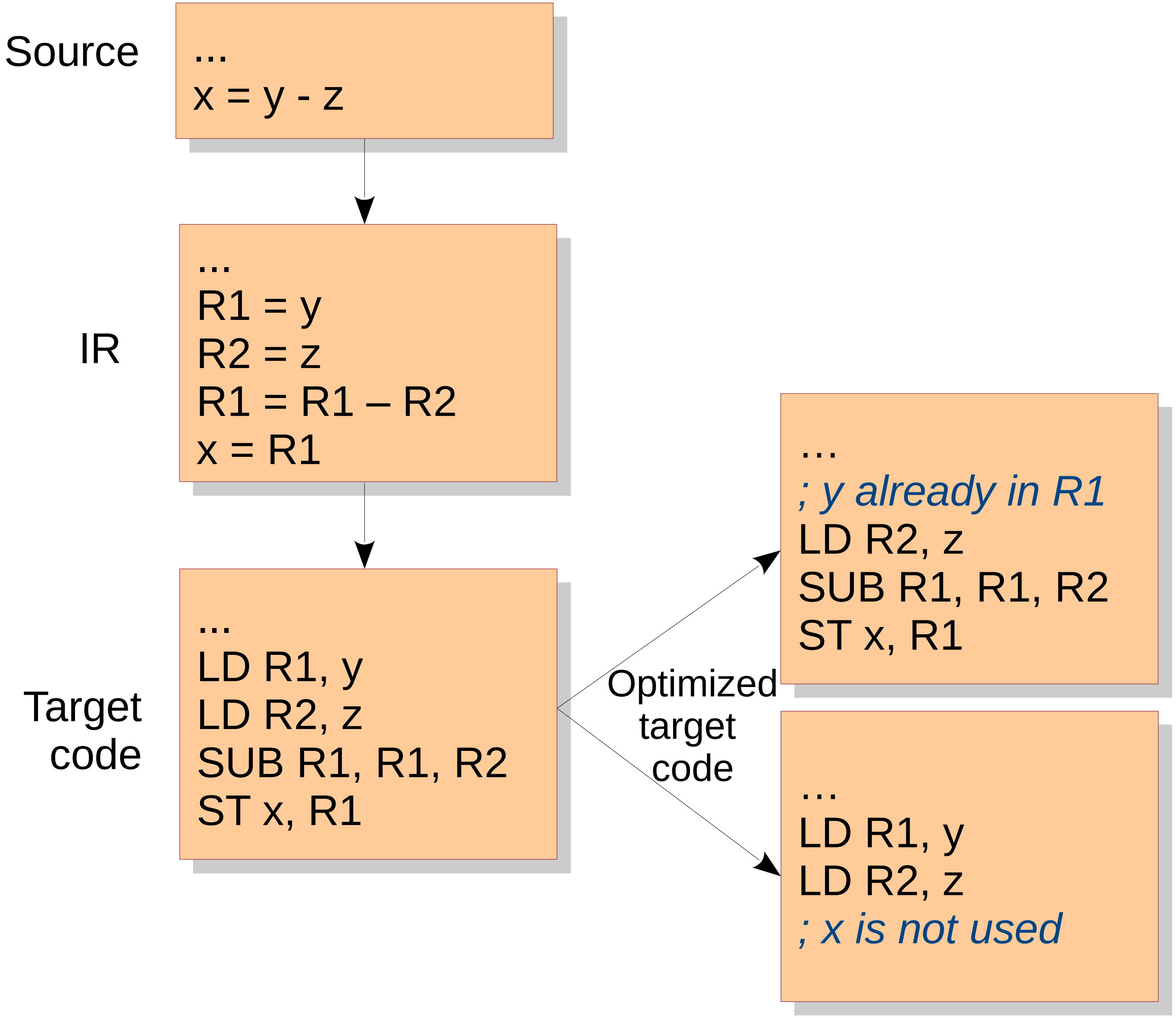
A Typical Target Machine Model

Instruction Type	Example
Load	LD R1, x
Store	ST R1, x
Computation	SUB R1, R2, R3
Unconditional Jump	BR main
Conditional Jump	BLTZ R1, main

Addressing Mode	Example
Direct	LD R1, 100000
Named / Variable	LD R1, x
Variable Indexed	LD R1, a(R2)
Immediate Indexed	LD R1, 100(R2)
Indirect	LD R1, *100(R2)
Immediate	LD R1, #100

Example Code Generation

using our Target Machine Model



Source

```
...
x = y - z
```

IR

```
...
R1 = y
R2 = z
R1 = R1 – R2
x = R1
```

Target code

```
...
LD R1, y
LD R2, z
SUB R1, R1, R2
ST x, R1
```

Optimized target code

```
...
; y already in R1
LD R2, z
SUB R1, R1, R2
ST x, R1
```

```
...
LD R1, y
LD R2, z
; x is not used
```

Optimization and Code Generation are often run together multiple-times.

Homework

- Exercises 8.2.3 from ALSU book.

Basic Blocks and CFG

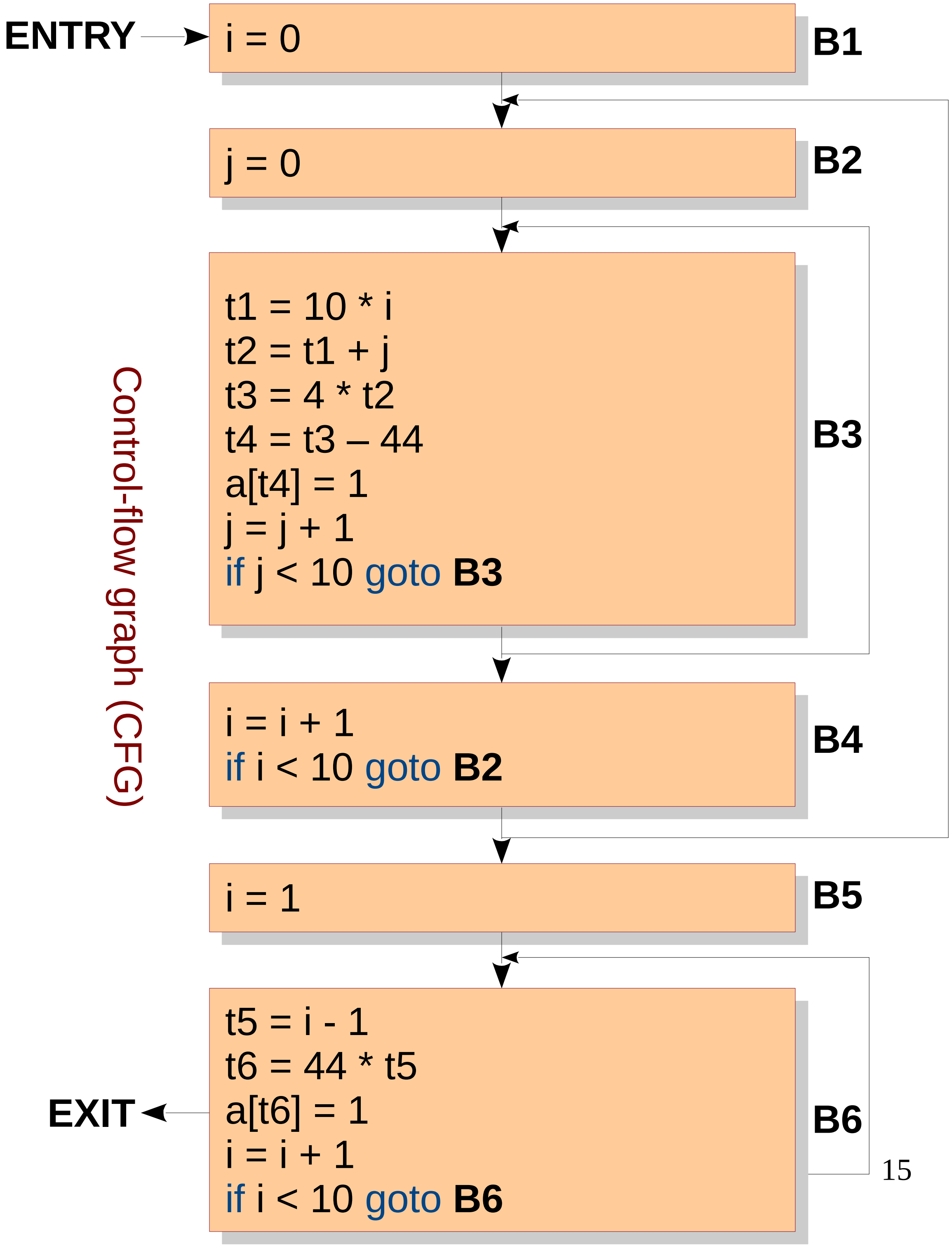
- A basic block is a maximal sequence of consecutive 3AC instructions such that
  - Single-entry: Control-flow enters the basic-block through only the first instructions in the block.
  - Single-exit: Control leaves the block only after the last instruction.
- Thus, if control reaches a basic block, all instructions in it are executed in sequence.
  - No branching from in-between or no jumps to in-between instructions.
- Basic-blocks together form a control-flow graph.

```
for (ii = 0; ii < 10; ++ii) {
        for (jj = 0; jj < 10; ++jj) {
                a[ii][jj] = 0;
        }
}
for (ii = 0; ii < 10; ++ii)
        a[ii][ii] = 1;
```

Source

```
i = 0
L2:
j = 0
L1:
t1 = 10 * i
t2 = t1 + j
t3 = 4 * t2
t4 = t3 – 44
a[t4] = 1
j = j + 1
if j < 10 goto L1
i = i + 1
if i < 10 goto L2
i = 1
L3:
t5 = i - 1
t6 = 44 * t5
a[t6] = 1
i = i + 1
if i < 10 goto L3
```

Intermediate representation



Control-flow graph (CFG)

B1:
```
i = 0
```

B2:
```
j = 0
```

B3:
```
t1 = 10 * i
t2 = t1 + j
t3 = 4 * t2
t4 = t3 – 44
a[t4] = 1
j = j + 1
if j < 10 goto B3
```

B4:
```
i = i + 1
if i < 10 goto B2
```

B5:
```
i = 1
```

B6:
```
t5 = i - 1
t6 = 44 * t5
a[t6] = 1
i = i + 1
if i < 10 goto B6
```

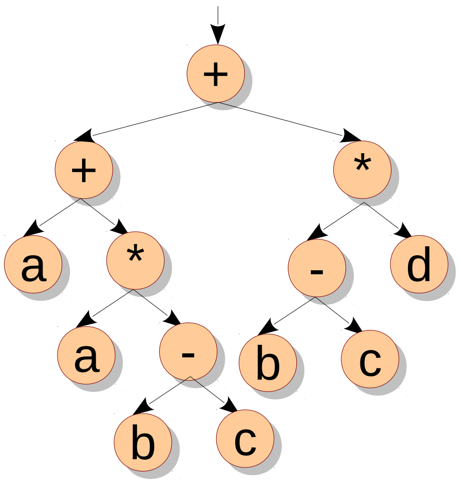
Optimizations using CFG

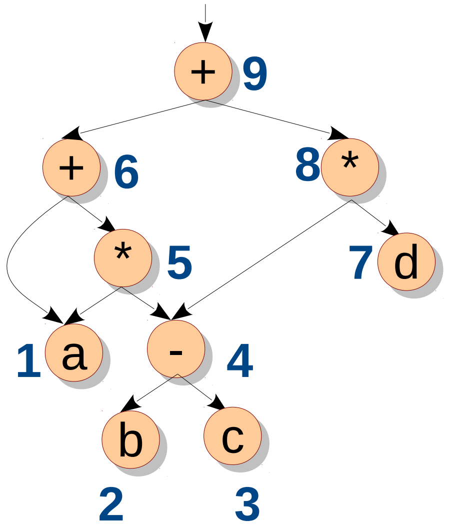
- **Local**: within a basic-block
  - Local common sub-expressions
  - Dead-code elimination
  - Use of algebraic identities
- **Global**: across blocks
  - Common sub-expression elimination
  - Strength reduction
  - Data-flow analysis

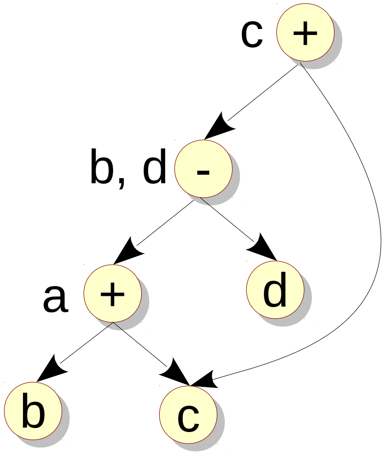
Local Common Sub-expressions Elimination

a + a * (b – c) + (b – c) * d

a = b + c
b = a – d
c = b + c
d = a - d







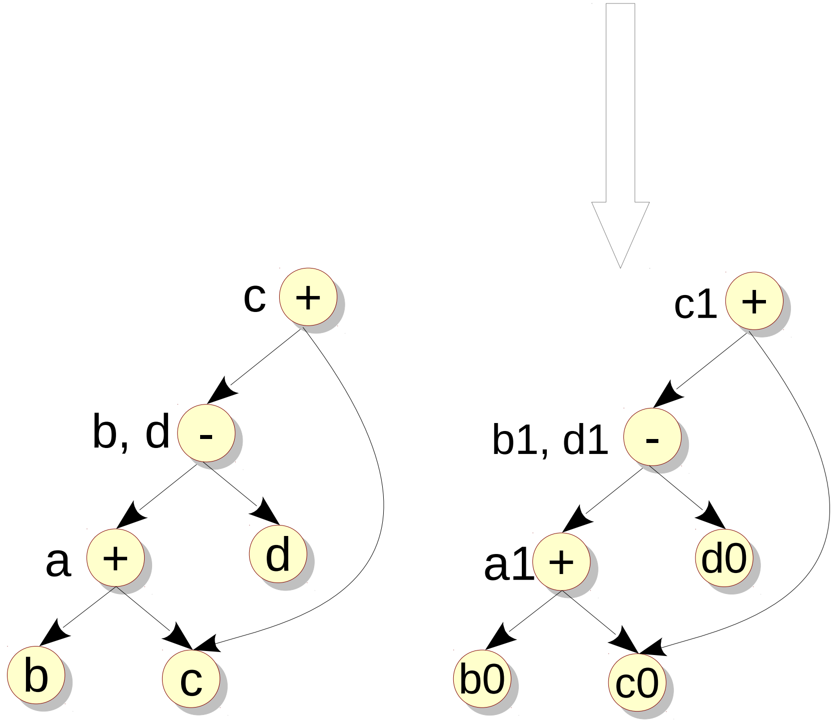
- Does not distinguish properly between different variable instances.
- It is unclear why certain variable should be used or a new one should be formed.
- We need use-def information.

Local Common Sub-expressions Elimination

```
a = b + c
b = a – d
c = b + c
d = a - d
```

```
a1 = b0 + c0
b1 = a1 – d0
c1 = b1 + c0
d1 = a1 - d0
```

- Variables have initial DEFs.
- Each DEF creates a new instance of the variable (recall SSA).
- Each USE refers to the latest DEF.



Local Common Sub-expressions Elimination

No common expressions

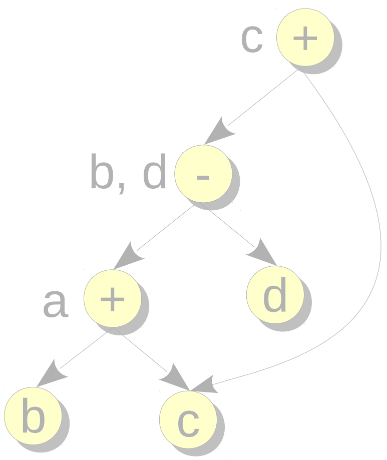
```
a = b + c
b = a – d
c = b + c
d = a - d
```

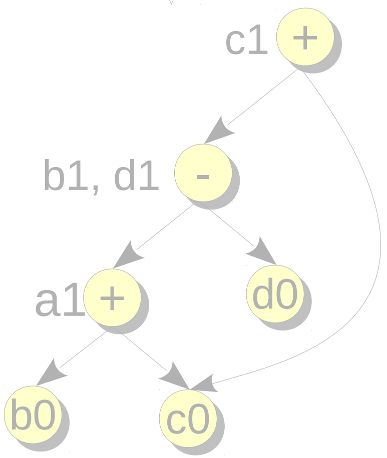
```
a1 = b0 + c0
b1 = a1 – d0
c1 = b1 + c0
d1 = a1 - d0
```

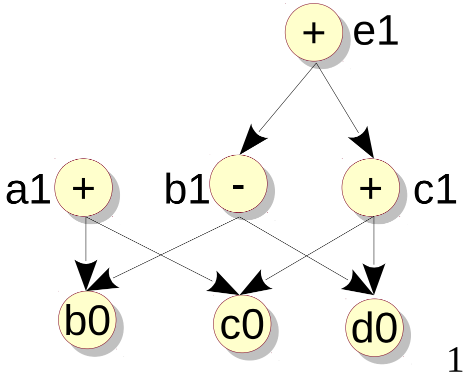
```
a = b + c
b = b – d
c = c + d
e = b + c
```

```
a1 = b0 + c0
b1 = b0 – d0
c1 = c0 + d0
e1 = b1 + c1
```

Classwork: Find the Basic Block DAG (expression DAG) for the above Basic Block.





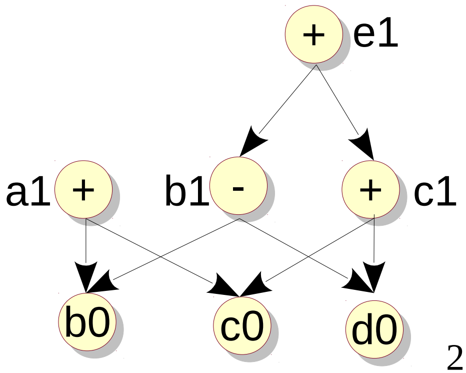


Dead-code Elimination

- Remove root from the DAG that have no live variables attached.
  - There could be multiple roots in the DAG.
  - We may be able to apply this repeatedly.

Assuming a and b are live (used later)
while c and e are not, then

- We can remove e1.
- Once e1 is removed, c1 can also be removed.



Algebraic Identities

- Algebraic properties
  - $x + 0 = 0 + x = x$ $\quad x – 0 = x$
  - $x * 1 = 1 * x = x$ $\quad x / 1 = x$
- Strength reduction
  - $x^2 = x * x$
  - $2 * x = x + x$
  - $x / 2 = x * 0.5$
- Constant folding
  - $2 * 3.14 = 6.28$

Algebraic Identities

- Commutativity and Associativity
  - DAG construction can help us here.
  - Apart from checking *left op right*, we could also check *right op left* for commutativity.

    e.g., (a + b) + (b + a).

    e.g., a = b + c; e = c + d + b;
- Some algebraic laws are not obvious.
  - e.g., Can you optimize *if (x > y) a = b + x + c – y*?

    However, we need to worry about underflows.

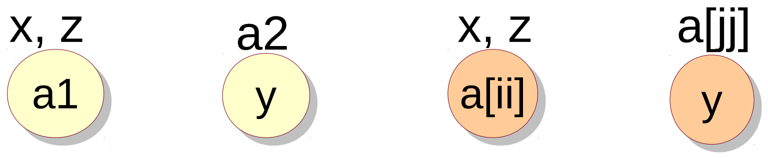
Array References

- Array references cannot be treated like usual variables.

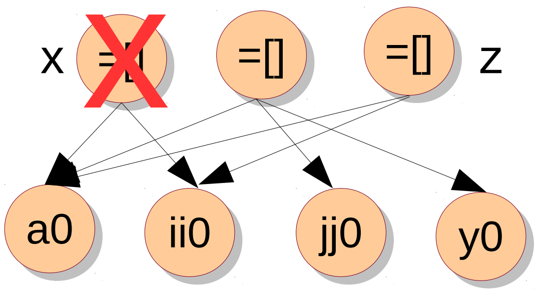
```
x = a1
a2 = y
z = a1
```

```
x = a[ii]
a[jj] = y
z = a[ii]
```

We represent a[ii] as a node with **two or three children** depending upon whether it is rvalue or lvalue.



wrong



correct

How do you decide the order in which assignments are executed?

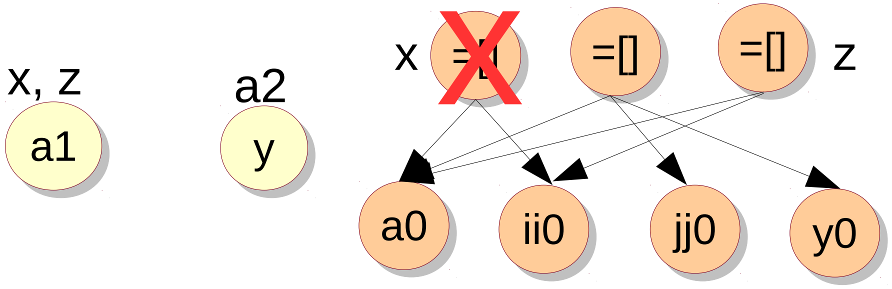
Array References

- Array references cannot be treated like usual variables.

```
x = a1
a2 = y
z = a1
```

```
x = a[ii]
a[jj] = y
z = a[ii]
```

```
x = a[ii]
b[jj] = y
z = a[ii]
```



Depending upon how much time a compiler can afford,

- it would either analyze if a[ii] and b[jj] are referring to the same memory location **OR**
- conservatively assume that they *MAY* be referring to the same location.

Aliasing

- The issue with array references is called aliasing.
- Two expressions may refer to the same memory location at the execution time.
  - a[ii] and a[jj]
  - *p and *q
  - Pass by reference variables
- Local processing may fail to identify aliasing
  - Precise alias analysis is computationally difficult.

Classwork: Aliasing

- Find all the aliases in this C++ program.

```
#include <iostream>
int g = 1;

void fun(int &p, int *q, int r, int a, int *s) {
        int &x = g;
        int *y = &p;
        std::cout << p << *q << r << a << *s << x << *y << g << std::endl;
}
int main() {
        int a = g;
        int &b = g;
        int *c = &g;

        fun(b, &g, g, a, c);
        return 0;
}
```

Peephole Optimization

- Consider a sliding window of instructions and optimize it.
- Repeated passes are often helpful.
  - Redundant load/store elimination
  - Dead-code elimination
  - Control-flow optimization
  - Algebraic simplifications
  - Use of machine idioms
  - ...

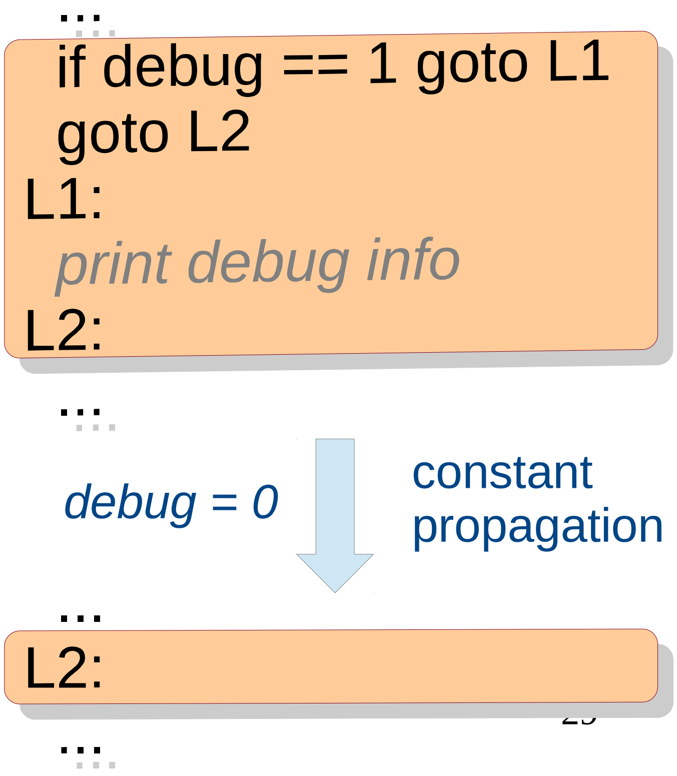
Peephole Optimization

- Consider a sliding window of instructions and optimize it.
- Repeated passes are often helpful.
  - Redundant load/store elimination
  - Dead-code elimination
  - Control-flow optimization
  - Algebraic simplifications
  - Use of machine idioms
  - ...

```
...
LD R0, a
ST a, R0
...
```

Peephole Optimization

- Consider a sliding window of instructions and optimize it.
- Repeated passes are often helpful.
  - Redundant load/store elimination
  - Dead-code elimination
  - Control-flow optimization
  - Algebraic simplifications
  - Use of machine idioms
  - ...

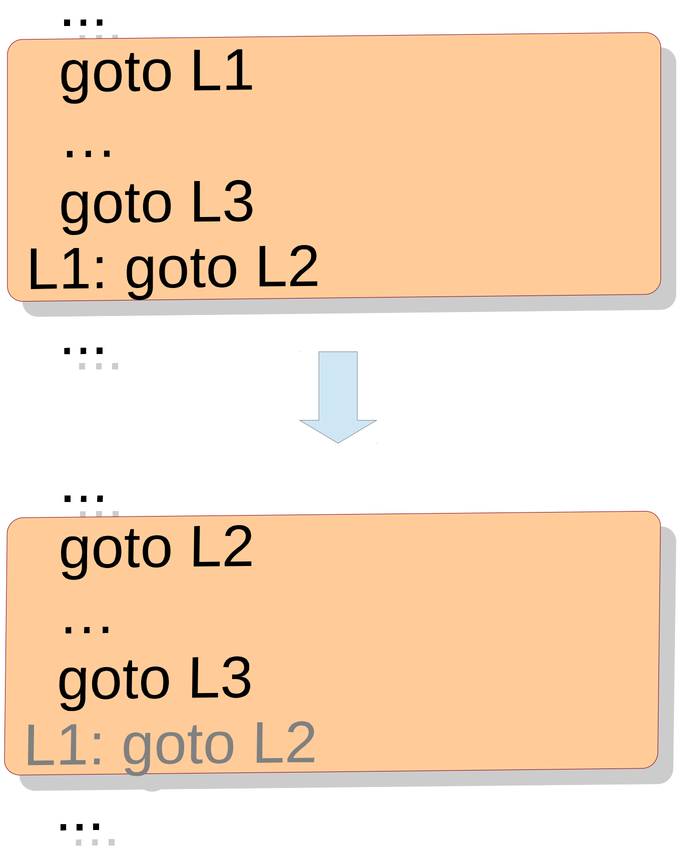


```
...
  if debug == 1 goto L1
  goto L2
L1:
  print debug info
L2:
...
```

```
...
L2:
...
```

Peephole Optimization

- Consider a sliding window of instructions and optimize it.
- Repeated passes are often helpful.
  - Redundant load/store elimination
  - Dead-code elimination
  - Control-flow optimization
  - Algebraic simplifications
  - Use of machine idioms
  - …



```
  ...
  goto L1
  …
  goto L3
L1: goto L2
  ...
```

```
  ...
  goto L2
  …
  goto L3
L1: goto L2
  ...
```

Remove “L1: goto L2” if no jumps to it.
Can be generalized to conditional jump to L1.

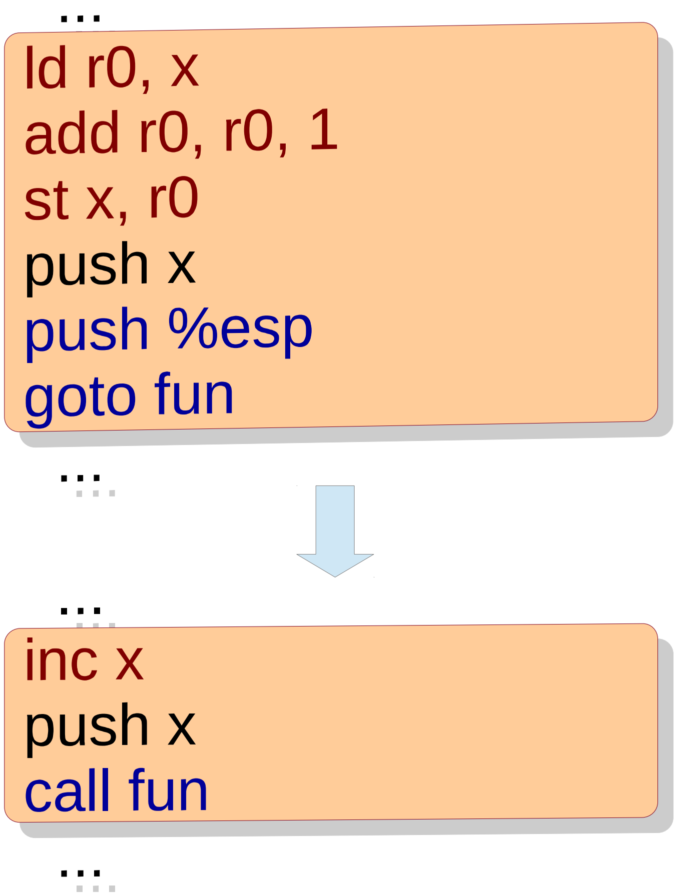
Peephole Optimization

- Consider a sliding window of instructions and optimize it.
- Repeated passes are often helpful.
  - Redundant load/store elimination
  - Dead-code elimination
  - Control-flow optimization
  - Algebraic simplifications
  - Use of machine idioms
  - ...

```
...
x = x + 0
z = y * 32
...
```

Peephole Optimization

- Consider a sliding window of instructions and optimize it.
- Repeated passes are often helpful.
  - Redundant load/store elimination
  - Dead-code elimination
  - Control-flow optimization
  - Algebraic simplifications
  - Use of machine idioms
  - ...



```
...
ld r0, x
add r0, r0, 1
st x, r0
push x
push %esp
goto fun
...
```

```
...
inc x
push x
call fun
...
```

Register Allocation

- Memory hierarchy: Network, File system, Main memory, L3 cache, L2, L1, Registers.
  - Capacity reduces, access time reduces.
- Critical to allocate and assign registers for efficiency.
  - Register versus Memory could be ~10x performance difference.
- C allows register variables.
  - register int a;   // not always a good idea.
  - register int a asm(“r12”);   // tries a specific register.
  - gcc -ffixed-r12 …   // reserve r12.

Register Allocation

Classwork: Allocate registers for the following code.

```
while (b) {
  a = b + c
  d = d – b
  e = a + f
  if (e) {
      b = d + f
      e = a – c
      if (b) goto fun
  } else {
      f = a - d
  }
  b = d + c
}
print b, c, d, e, f
```

- First-Come-First-Served way is often not the best policy for register allocation.
- We need to perform some analysis to find out the benefit of allocating registers to variables.
- We may have to assign cost / benefit to various operations within a loop.

- What if we say that K registers would be allocated to the top K variables that have the maximum number of uses?
- By paying a small spilling cost, we may be able to increase the benefit of K registers to more than K variables.

benefit(x, B) = F(use(x, B), live(x, B))

Variable x, Basic block B
use returns the number of uses.
live returns 0 or 1 based on if x is live after leaving B.

Liveness

benefit(x, B) =
F(use(x, B), live(x, B))

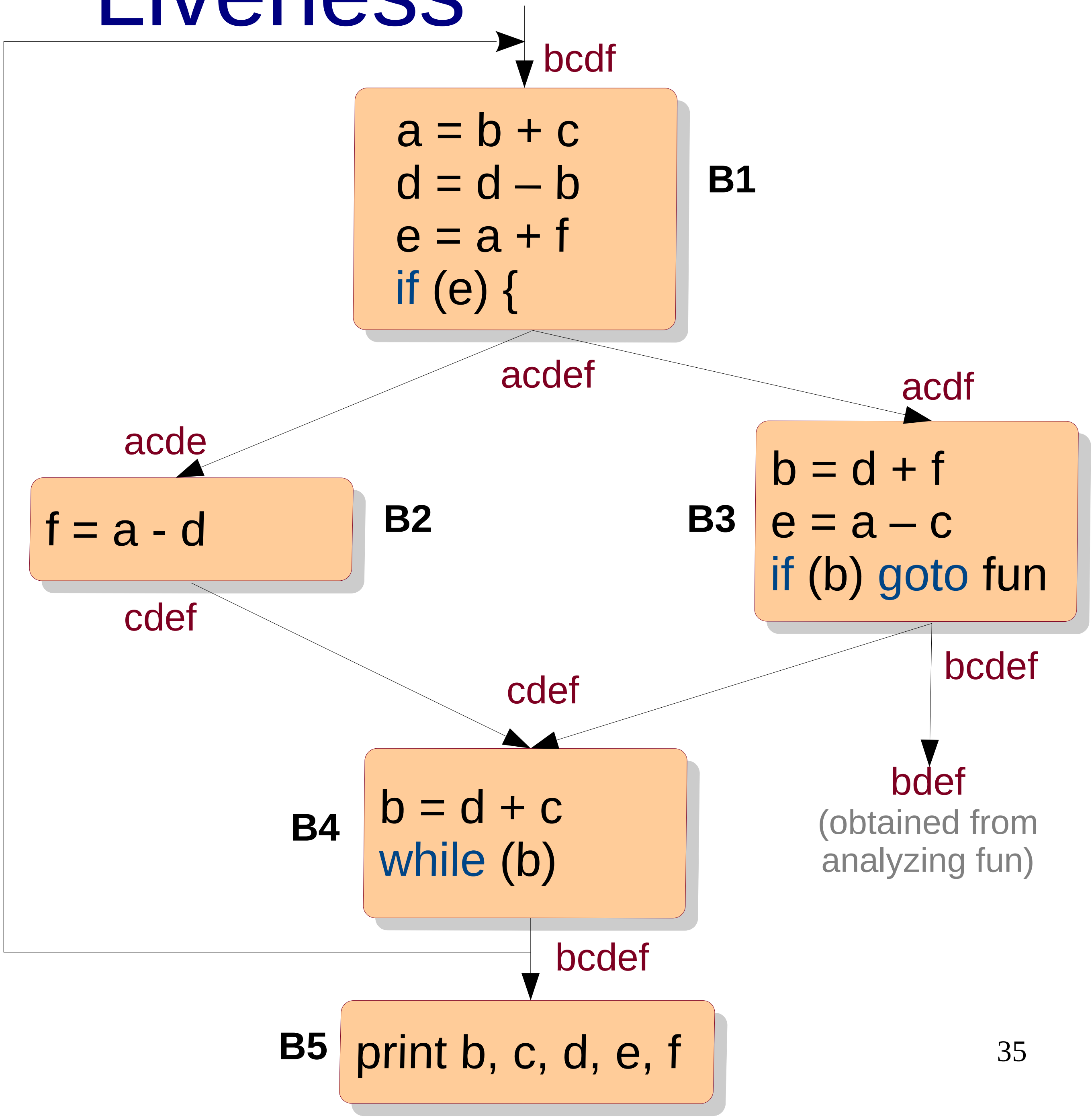
use returns number of uses.
live returns 0 or 1 based on if x is live after leaving B.

use(a, B1) = 1, live(a, B1) = 1
use(a, B2) = 1, live(a, B2) = 0
use(b, B3) = 1, live(b, B3) = 1
use(c, B2) = 0, live(c, B2) = 1
use(a, B4) = 0, live(a, B4) = 0
...

***Overall benefit S*(x) =**
***sum*(benefit(x, B))** for all B

Say, S(a) = 4, S(b) = 5, S(c) = 3,
S(d) = 6, S(e) = 4, S(f) = 4.

- Assign R0, R1, R2 to a, b and d globally (global allocation).
- Use remaining register R3 inside blocks (local allocation).



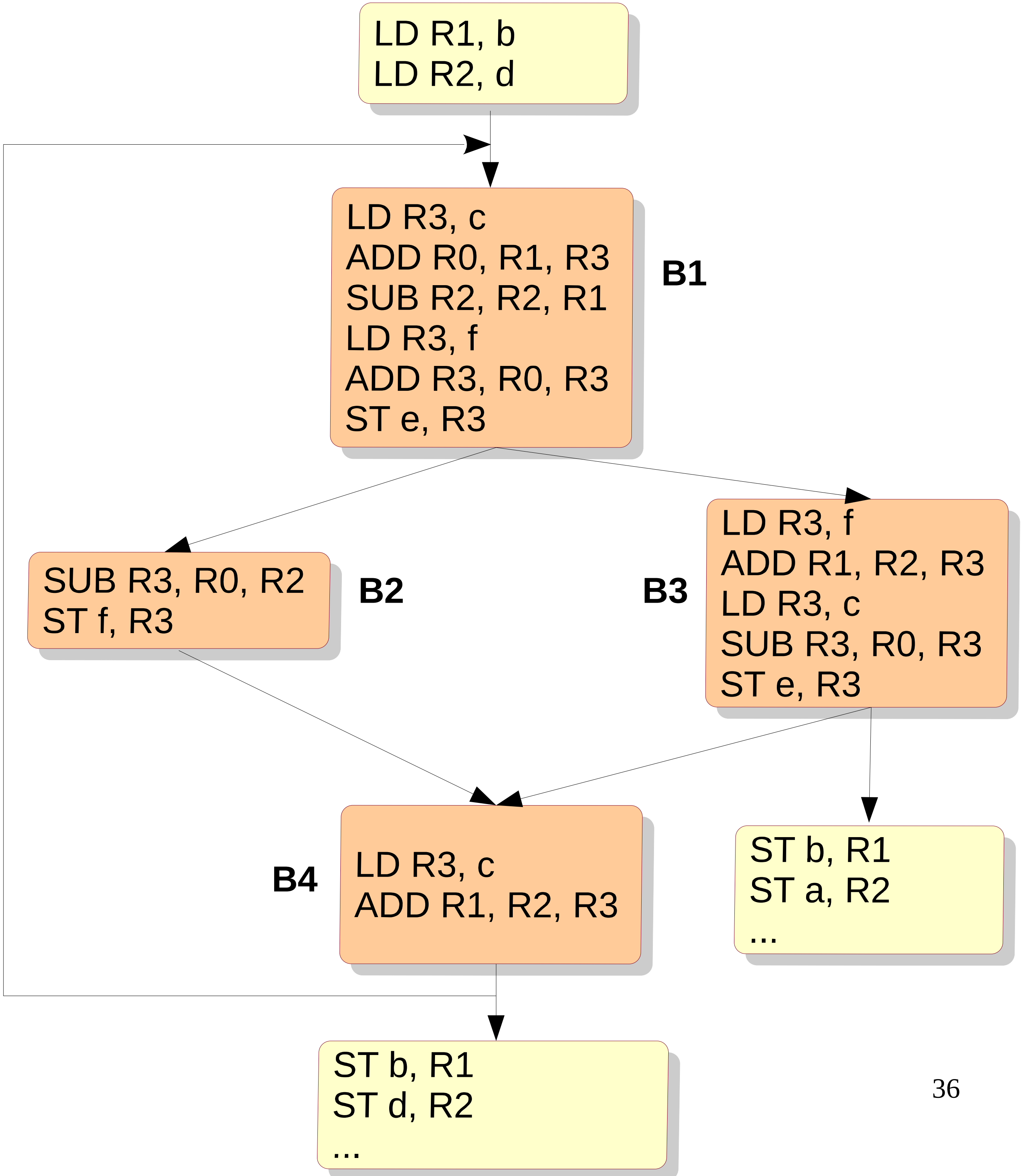
Allocation

- R1 and R2 remain assigned to b and d throughout.
- R3 is loaded repeatedly inside the loop as an auxiliary register.
- a is not live at the start, hence it is not loaded initially.
- At the end of the loop, the register values are stored back.

***Overall benefit S*(x) = *sum*(benefit(x, B))** for all B

Say, S(a) = 4, S(b) = 5, S(c) = 3, S(d) = 6, S(e) = 4, S(f) = 4.

- Assign R0, R1, R2 to a, b and d globally (global allocation).
- Use remaining register R3 inside blocks (local allocation).



```
LD R1, b
LD R2, d
```

B1

```
LD R3, c
ADD R0, R1, R3
SUB R2, R2, R1
LD R3, f
ADD R3, R0, R3
ST e, R3
```

B2

```
SUB R3, R0, R2
ST f, R3
```

B3

```
LD R3, f
ADD R1, R2, R3
LD R3, c
SUB R3, R0, R3
ST e, R3
```

B4

```
LD R3, c
ADD R1, R2, R3
```

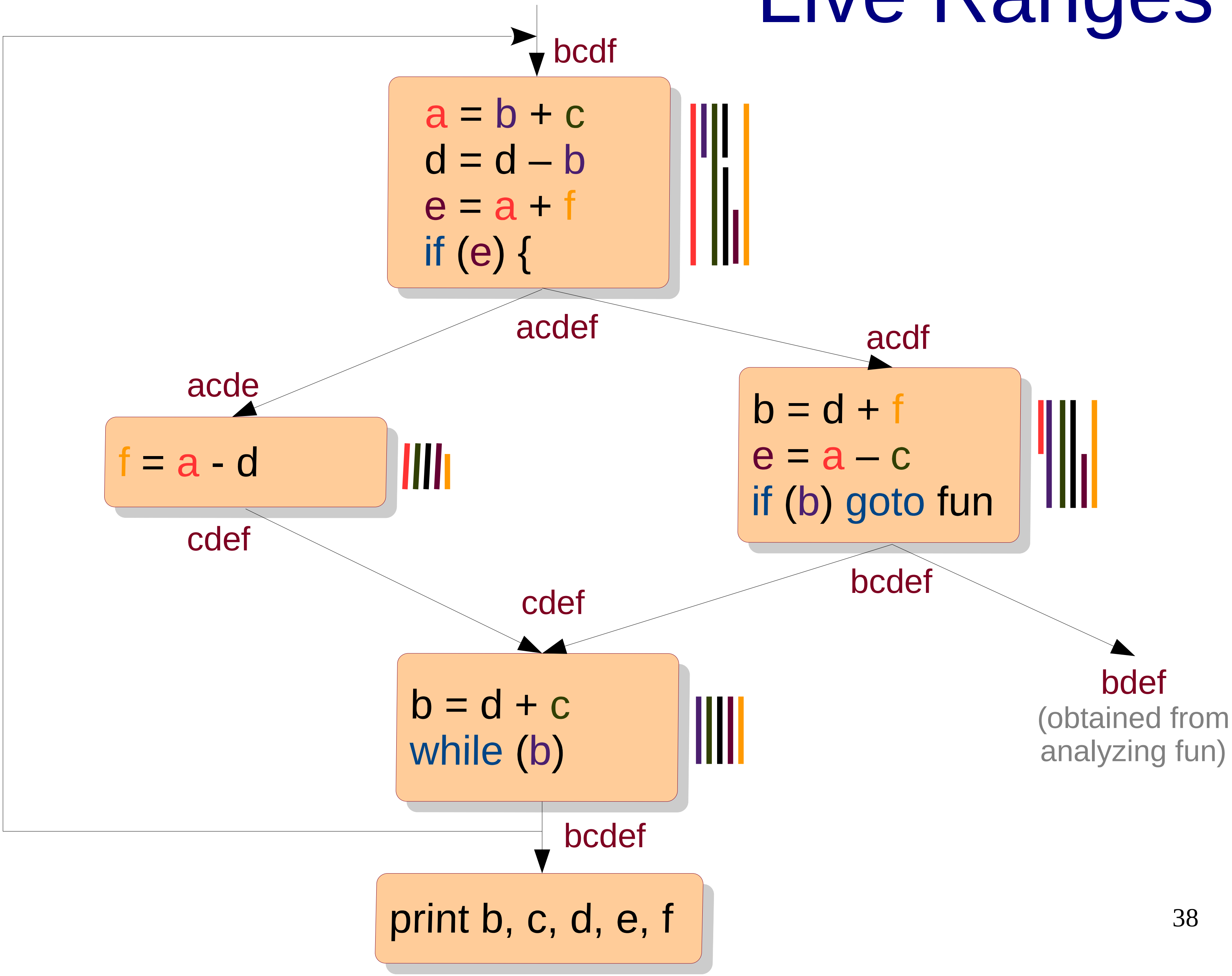
```
ST b, R1
ST a, R2
...
```

```
ST b, R1
ST d, R2
...
```

Register Allocation as Graph Coloring

- Vertices? Edges?
- Vertices: Variables (or their instances)
- Edges: Co-Live information
  - If x and y are live at the same program point, add an (undirected) edge between x and y.
- Vertex coloring colors neighbors differently.
  - Thus, vertex coloring colors x and y differently, if they are live at the same program point.
  - This means, x and y should not use the same register.
  - **Corollary**: if x and z have the same color, they can reuse the register (at different program points).

Live Ranges



bcdf

```
a = b + c
d = d – b
e = a + f
if (e) {
```

acdef

acde

```
f = a - d
```

cdef

acdf

```
b = d + f
e = a – c
if (b) goto fun
```

bcdef

cdef

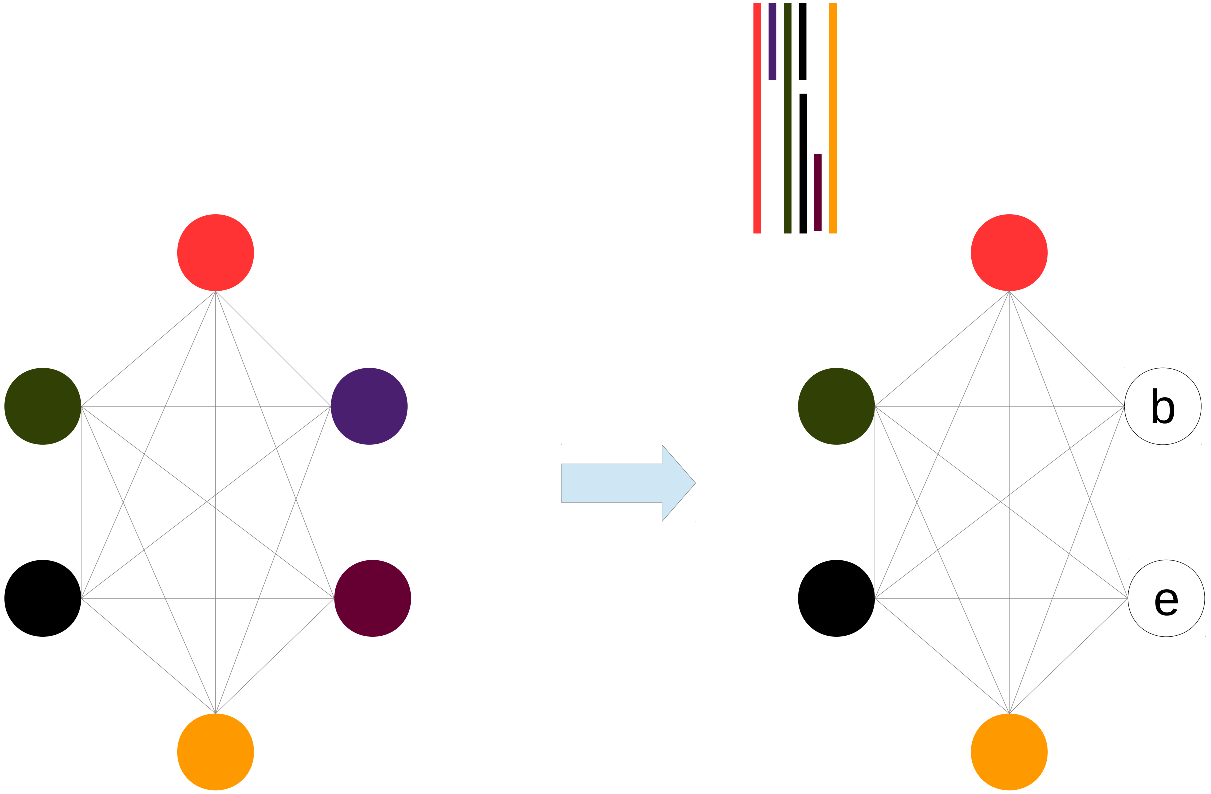
```
b = d + c
while (b)
```

bdef
(obtained from
analyzing fun)

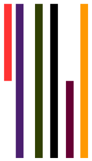
bcdef

```
print b, c, d, e, f
```

Coloring



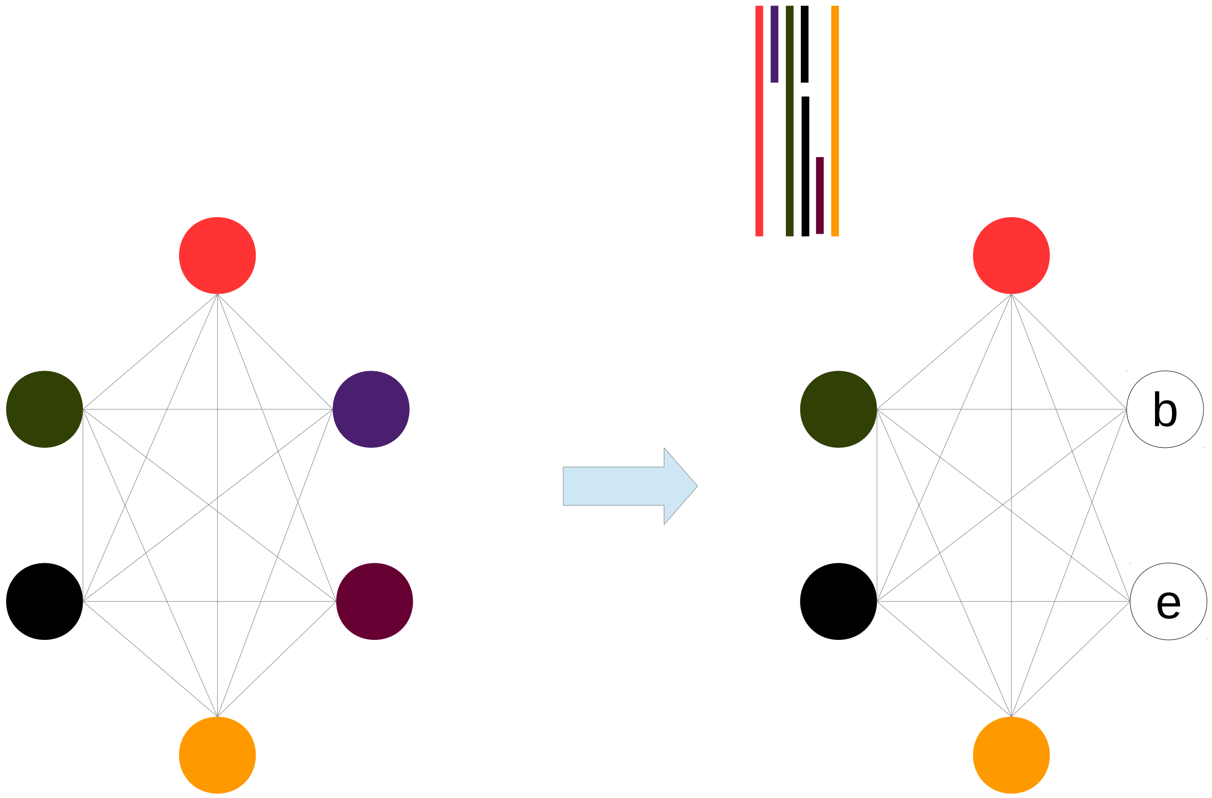
This means, in basic block B1, b and e could use the same register.

Classwork: Try it for  .

```
a = b + c
d = d – b
e = a + f
if (e) {
```

What is the issue with what we did?

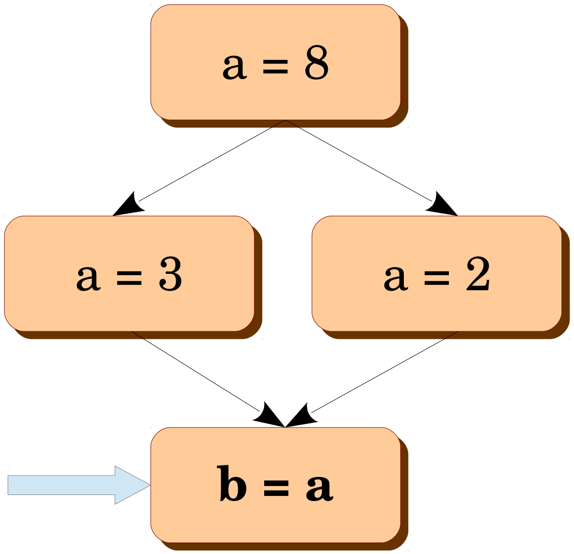
Coloring



- Coloring gave us the maximum number of registers required for the program.
- However, in practice, the number of registers is fixed.
- Therefore, we need to generate spill code for storing a variable into memory (ST x, R) and then reload the register with the next variable (LD R, y)

Data Flow Analysis

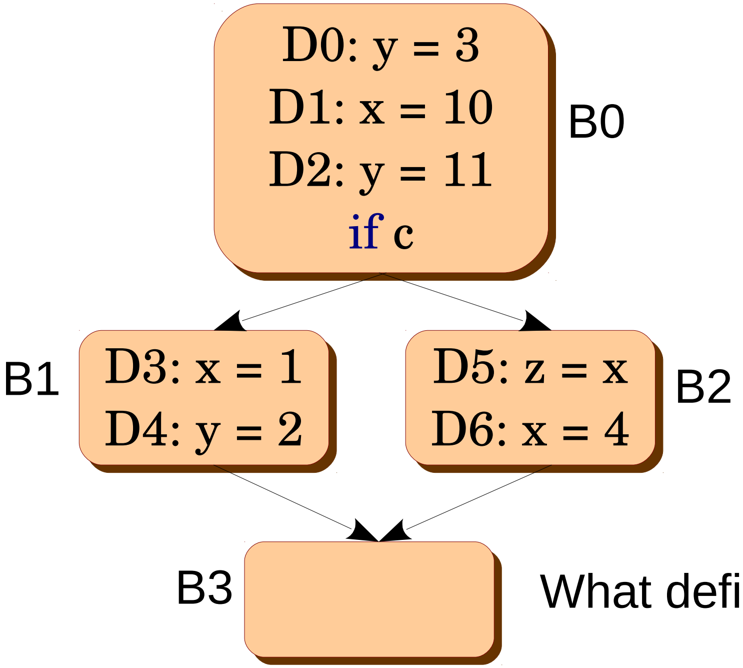
- Flow-sensitive: Considers the control-flow in a function
- Operates on a flow-graph with nodes as basic-blocks and edges as the control-flow
- Examples
  - Constant propagation
  - Common subexpression elimination
  - Dead code elimination



What is the value of b?

Reaching Definitions

- Every assignment is a definition.
- A definition d reaches a program point p if there exists a path from the point immediately following d to p such that d is not killed along the path.



B0

D0: y = 3
D1: x = 10
D2: y = 11
if c

B1

D3: x = 1
D4: y = 2

B2

D5: z = x
D6: x = 4

B3

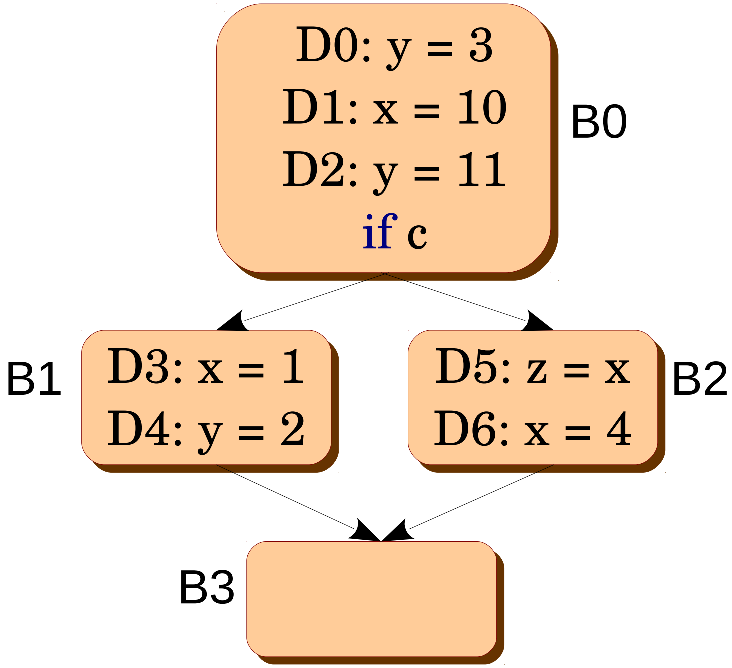
What definitions reach B3?

DFA Equations

- in(B) = set of data flow facts entering block B
- out(B) = …
- gen(B) = set of data flow facts generated in B
- kill(B) = set of data flow facts from the other blocks killed in B

DFA for Reaching Definitions

- in(B) = U out(P) where P is a predecessor of B
- out(B) = gen(B) U (in(B) – kill(B))
- Initially, out(B) = { }



D0: y = 3
D1: x = 10
D2: y = 11
if c

B0

B1

D3: x = 1
D4: y = 2

D5: z = x
D6: x = 4

B2

B3

gen(B0) = {D1, D2}   kill(B0) = {D3, D4, D6}
gen(B1) = {D3, D4}   kill(B1) = {D0, D1, D2, D6}
gen(B2) = {D5, D6}   kill(B2) = {D1, D3}
gen(B3) = { }   kill(B3) = { }

	in1	out1	in2	out2	in3	out3
B0	{}	{D1, D2}	{}	{D1, D2}	{}	{D1, D2}
B1	{}	{D3, D4}	{D1, D2}	{D3, D4}	{D1, D2}	{D3, D4}
B2	{}	{D5, D6}	{D1, D2}	{D2, D5, D6}	{D1, D2}	{D2, D5, D6}
B3	{}	{}	{D3, D4, D5, D6}	{D3, D4, D5, D6}	{D2, D3, D4, D5, D6}	{D2, D3, D4, D5, D6}

Algorithm for Reaching Definitions

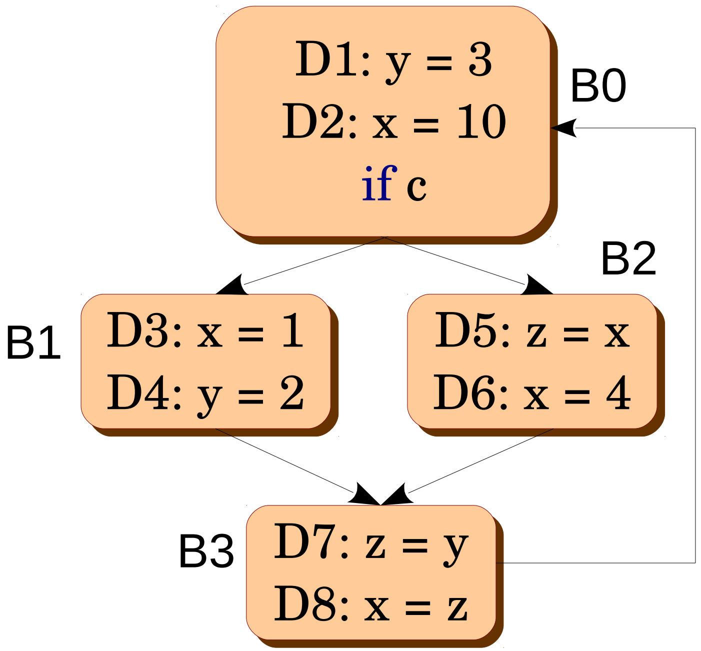
```
for each basic block B
  compute gen(B) and kill(B)
  out(B) = {}

do {
  for each basic block B
      in(B) = U out(P) where P \in pred(B)
      out(B) = gen(B) U (in(B) - kill(B))
} while in(B) changes for any basic block B
```

Classwork

- in(B) = U out(P) where P is a predecessor of B
- out(B) = gen(B) U (in(B) – kill(B))

- Initially, out(B) = { }



B0
D1: y = 3
D2: x = 10
if c

B1
D3: x = 1
D4: y = 2

B2
D5: z = x
D6: x = 4

B3
D7: z = y
D8: x = z

gen	kill
gen(B0) = {D1, D2}	kill(B0) = {D3, D4, D6, D8}
gen(B1) = {D3, D4}	kill(B1) = {D1, D2, D6, D8}
gen(B2) = {D5, D6}	kill(B2) = {D2, D3, D7, D8}
gen(B3) = {D7, D8}	kill(B3) = {D2, D3, D5, D6}

	in1	out1	in2	out2	in3	out3	in4	out4
B0	{}	{D1,D2}	{D7,D8}	{D1,D2, D7}	{D4,D7,D8}	{D1,D2,D7}	{D1,4,7}	{D1,2,7}
B1	{}	{D3,D4}	{D1,D2}	{D3,D4}	{D1,D2,D7}	{D3,D4,D7}	{D1,2,7}	{D3,4,7}
B2	{}	{D5,D6}	{D1,D2}	{D1,D5,D6}	{D1,D2,D7}	{D1,D5,D6}	{D1,2,7}	{D1,5,6}
B3	{}	{D78}	{D3456}	{D478}	{D13456}	{D1478}	{D134567}	{D1478}

DFA for Reaching Definitions

Domain	Sets of definitions

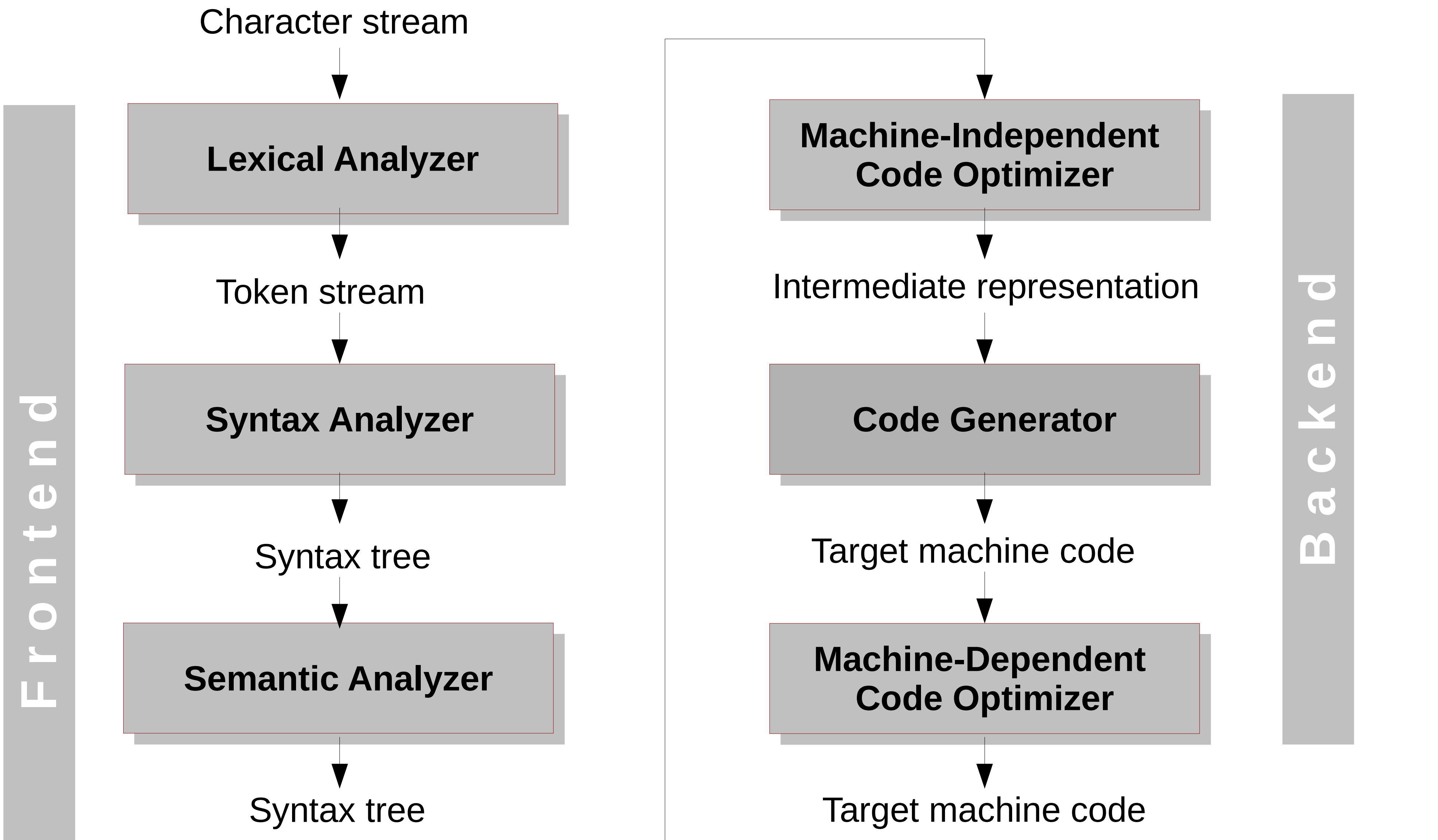
Transfer function	in(B) = U out(P)
	out(B) = gen(B) U (in(B) - kill(B))

Direction	Forward
Meet / confluence operator	U
Initialization	out(B) = { }

DFA for Live Variables

Domain	Sets of variables
Transfer function	in(B) = use(B) U (out(B) - def(B)) out(B) = U in(S) where S is a successor of B
Direction	Backward
Meet / confluence operator	U
Initialization	in(B) = { }

A variable v is live at a program point p if v is used along some path in the flow graph starting at p.
Otherwise, the variable v is dead.



Character stream

Lexical Analyzer

Token stream

Syntax Analyzer

Syntax tree

Semantic Analyzer

Syntax tree

Intermediate Code Generator

Intermediate representation

Frontend

Machine-Independent Code Optimizer

Intermediate representation

Code Generator

Target machine code

Machine-Dependent Code Optimizer

Target machine code

Backend

Symbol Table