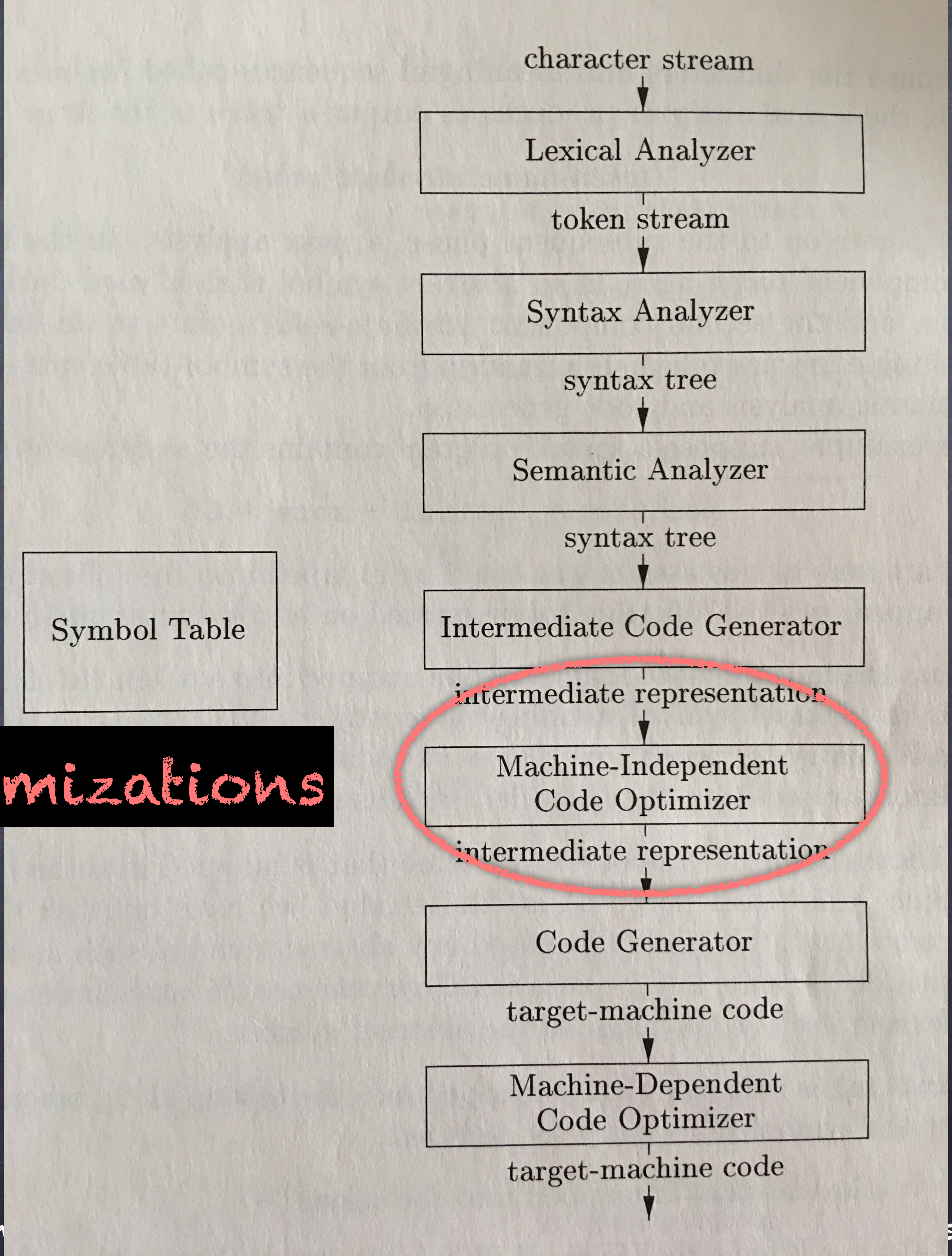
CSE443
Compilers

Dr. Carl Alphonce
alphonce@buffalo.edu
343 Davis Hall

© 2021 Carl Alphonce - Reproduction of this material is prohibited without the author's consent

Phases of a compiler



Optimizations

Figure 1.6, page 5 of text

© 2021 Carl Alphonce - Reproduction sent

Algebraic Identities [p. 536]

$$x + 0 = 0 + x = x$$

$$x * 1 = 1 * x = x$$

$$x - 0 = x$$

$$x / 1 = x$$

© 2021 Carl Alphonce - Reproduction of this material is prohibited without the author's consent

Algebraic Identities [p. 536]

$$x^2 = x * x$$

$$2 * x = x + x$$

$$x / 2 = x * 0.5$$

Can use left and right shift for integers

But see next slide and these links:

https://en.wikipedia.org/wiki/Arithmetic_shift

https://stackoverflow.com/questions/19517868/integer-division-by-negative-number

https://www.microsoft.com/en-us/research/wp-content/uploads/2016/02/divmodnote-letter.pdf

© 2020 Carl Alphonce - Reproduction of this material is prohibited without the author's consent

4-bit examples
non-negative values

0011 -> +3

Logical shifts

right: 0001 -> +1 $\lfloor 3/2 \rfloor = 1$

left: 0110 -> +6

Arithmetic shifts

right: 0001 -> +1

left: 0110 -> +6

© 2020 Carl Alphonce - Reproduction of this material is prohibited without the author's consent

4-bit examples
negative values

Logical shifts

Arithmetic shifts

right shift sign extension:

unexpected result?

1101 -> -3

Logical shifts

right: 0110 -> +6

left: 1010 -> -6

Arithmetic shifts

right: 1110 -> -2 $\lfloor -3/2 \rfloor = -2$

left: 1010 -> -6

© 2020 Carl Alphonce - Reproduction of this material is prohibited without the author's consent

C vs Python

```c
#include <stdio.h>
void printQuotientRemainder(int a, int b) {
  int q = a/b;
  int r = a%b;
  printf("%d/%d = %d remainder %d\t\t",a,b,q,r);
  printf("%d*%d + %d => %d = %d\n",b,q,r,(b*q+r),a);
}

int main(void) {
  printQuotientRemainder( 5, 2);
  printQuotientRemainder(-5, 2);
  printQuotientRemainder( 5,-2);
  printQuotientRemainder(-5,-2);
  printQuotientRemainder( 2, 5);
  printQuotientRemainder(-2, 5);
  printQuotientRemainder( 2,-5);
  printQuotientRemainder(-2,-5);
  return 0;
}
```

```python
def printQuotientRemainder(a,b):
  q = a//b
  r = a%b
  print("%d/%d = %d remainder %d\t\t" % (a,b,q,r),
end='')
  print("%d*%d + %d => %d = %d" % (b,q,r,(b*q+r),a))

def main():
  printQuotientRemainder( 5, 2)
  printQuotientRemainder(-5, 2)
  printQuotientRemainder( 5,-2)
  printQuotientRemainder(-5,-2)
  printQuotientRemainder( 2, 5)
  printQuotientRemainder(-2, 5)
  printQuotientRemainder( 2,-5)
  printQuotientRemainder(-2,-5)

main()
```

```
 5/ 2 =  2 remainder  1        2* 2 +  1 =>  5 =  5
-5/ 2 = -2 remainder -1        2*-2 + -1 => -5 = -5
 5/-2 = -2 remainder  1       -2*-2 +  1 =>  5 =  5
-5/-2 =  2 remainder -1       -2* 2 + -1 => -5 = -5
 2/ 5 =  0 remainder  2        5* 0 +  2 =>  2 =  2
-2/ 5 =  0 remainder -2        5* 0 + -2 => -2 = -2
 2/-5 =  0 remainder  2       -5* 0 +  2 =>  2 =  2
-2/-5 =  0 remainder -2       -5* 0 + -2 => -2 = -2
```

```
 5/ 2 =  2 remainder  1        2* 2 +  1 =>  5 =  5
-5/ 2 = -3 remainder  1        2*-3 +  1 => -5 = -5
 5/-2 = -3 remainder -1       -2*-3 + -1 =>  5 =  5
-5/-2 =  2 remainder -1       -2* 2 + -1 => -5 = -5
 2/ 5 =  0 remainder  2        5* 0 +  2 =>  2 =  2
-2/ 5 = -1 remainder  3        5*-1 +  3 => -2 = -2
 2/-5 = -1 remainder -3       -5*-1 + -3 =>  2 =  2
-2/-5 =  0 remainder -2       -5* 0 + -2 => -2 = -2
```

© 2021 Carl Alphonce - Reproduction of this material is prohibited without the author's consent

Algebraic Identities [p. 536]

Constant folding

"…evaluate constant expressions at compile time and replace the constant expressions by their values."

© 2021 Carl Alphonce - Reproduction of this material is prohibited without the author's consent

Algebraic Identities [p. 536]

See footnote 2:

"Arithmetic expressions should be evaluated the same way at compile time as they are at run time. K. Thompson has suggested an elegant solution to constant folding: compile the constant expression, execute the target code on the spot, and replace the expression with the result. Thus, the compiler does not need to contain an interpreter."

© 2021 Carl Alphonce - Reproduction of this material is prohibited without the author's consent

Peephole optimization
[p 549]

"The peephole is a small, sliding window on a program." [p. 549]

"In general, repeated passes over the target code are necessary to get the maximum benefit." [p. 550]

© 2021 Carl Alphonce - Reproduction of this material is prohibited without the author's consent

Peephole optimization: redundant LD/ST

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

If the ST instruction has a label, cannot remove it. (If instructions are in the same block we're OK.)

© 2021 Carl Alphonce - Reproduction of this material is prohibited without the author's consent

Peephole optimization: unreachable code

This case takes several slides...

```
    if E=K goto L1
    goto L2
L1: ...
    ...
L2: ...
    ...
```

Suppose K is a constant.

© 2021 Carl Alphonce - Reproduction of this material is prohibited without the author's consent

Peephole optimization: unreachable code

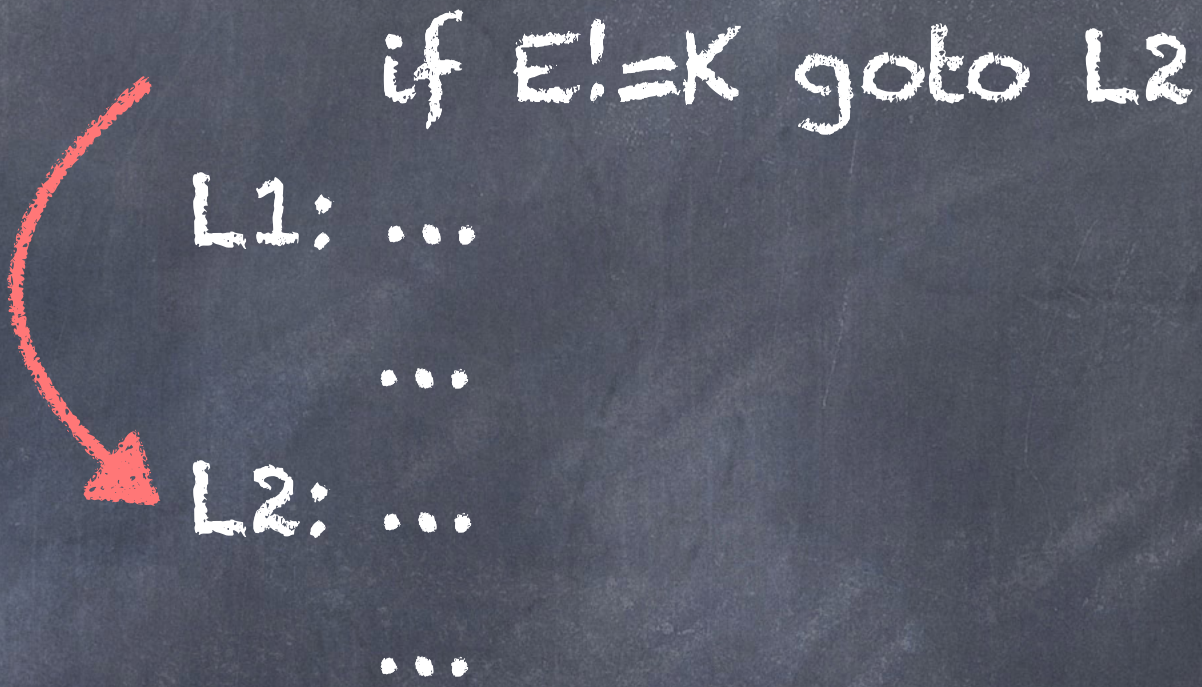
```
    if E=K goto L1
    goto L2
L1: ...do something...
...
L2: ...do something...
...
```

Eliminate jumps over jumps

© 2021 Carl Alphonce - Reproduction of this material is prohibited without the author's consent

Peephole optimization: unreachable code

```
    if E=K goto L1
    goto L2
L1: ...
    ...
L2: ...
    ...
```



```
    if E!=K goto L2
L1: ...
    ...
L2: ...
    ...
```

Eliminate jumps over jumps

© 2021 Carl Alphonce - Reproduction of this material is prohibited without the author's consent

Peephole optimization: unreachable code

```
    if E=K goto L1
    goto L2
L1: ...
    ...
L2: ...
    ...
```

```
    if E!=K goto L2
    ...
    ...
L2: ...
    ...
```

If there are no jumps to L1, we can remove label

© 2021 Carl Alphonce - Reproduction of this material is prohibited without the author's consent

Peephole optimization: unreachable code

```
    if E=K goto L1
    goto L2
L1: ...
    ...
L2: ...
    ...
```

```
    if E!=K goto L2
    ...
    ...
L2: ...
    ...
```

If E is set to a constant value other than K, then...

© 2021 Carl Alphonce - Reproduction of this material is prohibited without the author's consent

Peephole optimization: unreachable code

```
    if E=K goto L1
    goto L2
L1: ...
    ...
L2: ...
    ...
```

```
    if true goto L2
    ...
    ...
L2: ...
    ...
```

...conditional jump becomes unconditional...

© 2021 Carl Alphonce - Reproduction of this material is prohibited without the author's consent

Peephole optimization: unreachable code

```
    if E=K goto L1
    goto L2
L1: ...
    ...
L2: ...
    ...
```

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

...and the unreachable code can be removed.

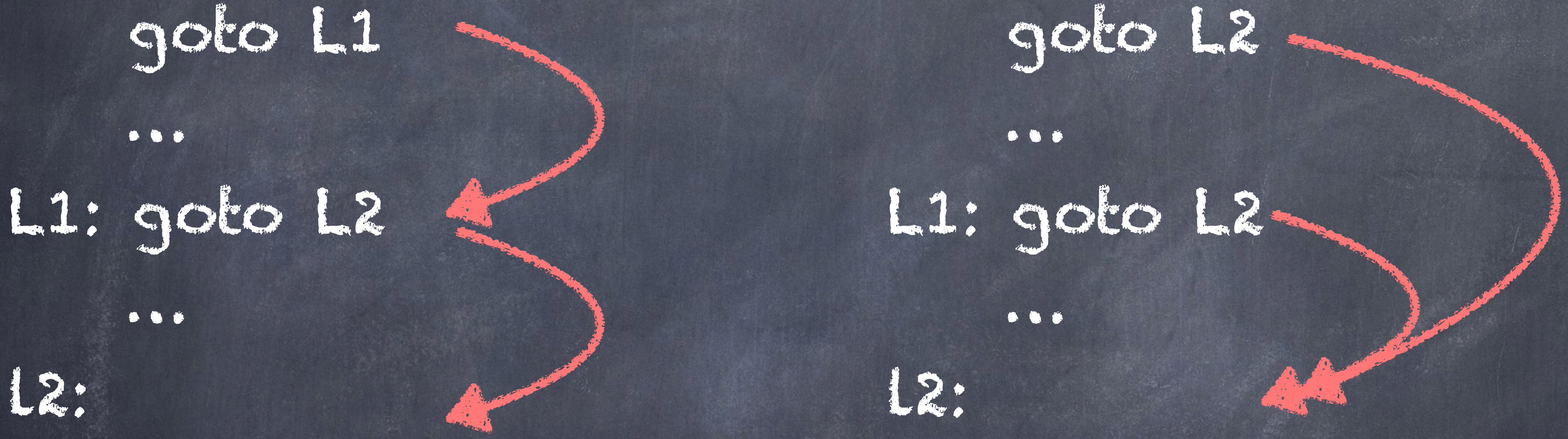
© 2021 Carl Alphonce - Reproduction of this material is prohibited without the author's consent

Peephole optimization: flow-of-control

```
    goto L1
    ...
L1: goto L2
    ...
L2:
```

© 2021 Carl Alphonce - Reproduction of this material is prohibited without the author's consent

Peephole optimization: flow-of-control

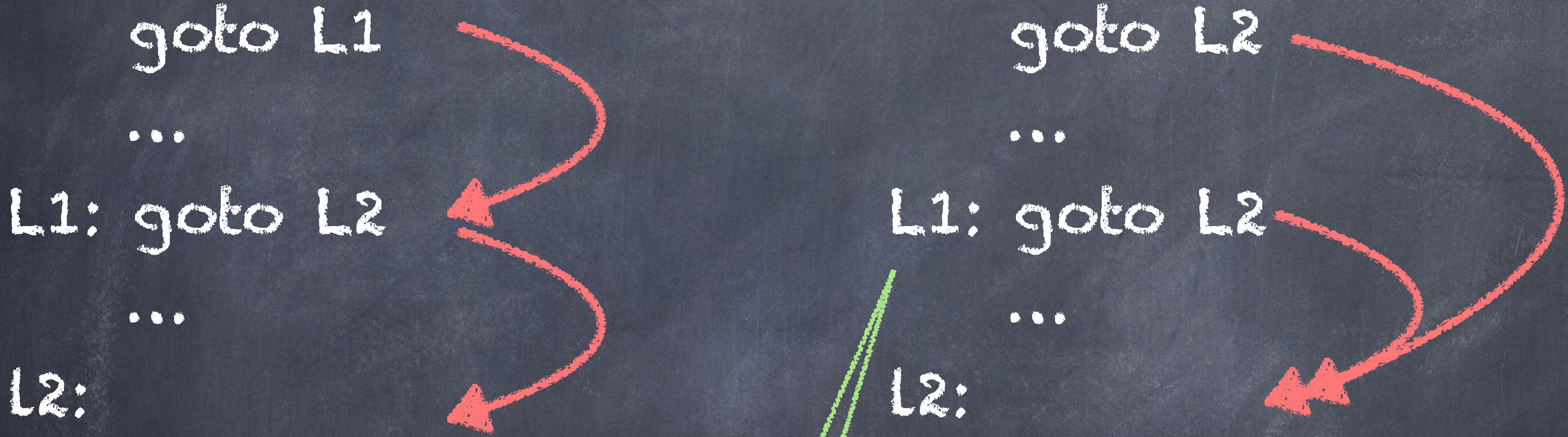


```
    goto L1
    ...
L1: goto L2
    ...
l2:
```

```
    goto L2
    ...
L1: goto L2
    ...
l2:
```

© 2021 Carl Alphonce - Reproduction of this material is prohibited without the author's consent

Peephole optimization: flow-of-control



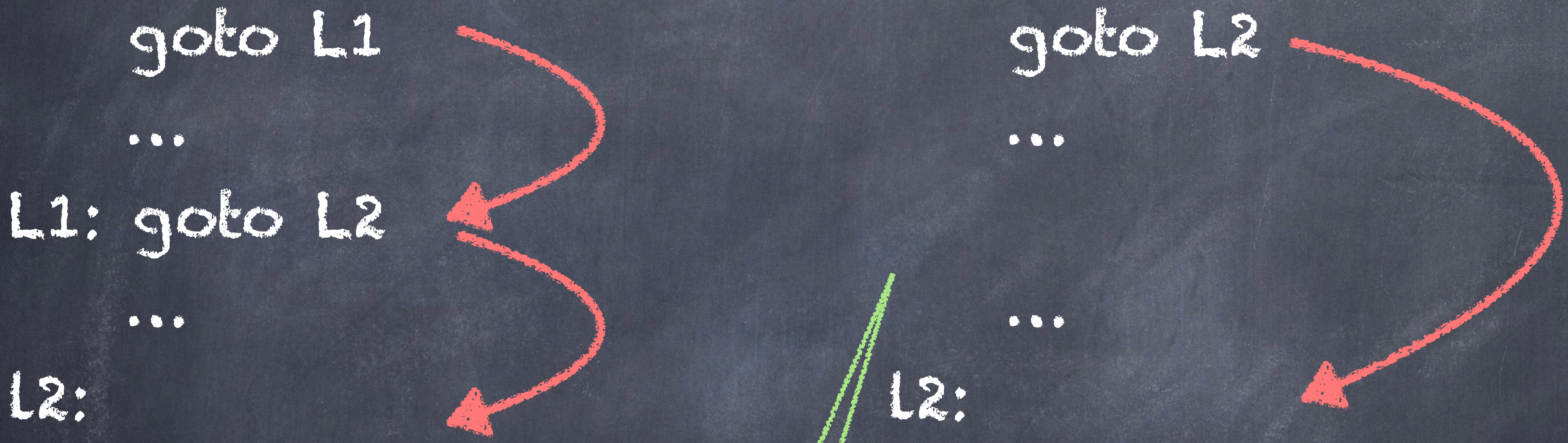
```
    goto L1
    ...
L1: goto L2
    ...
L2:
```

```
    goto L2
    ...
L1: goto L2
    ...
L2:
```

If there are no jumps to L1,
and L1 is preceded by an unconditional
jump...

© 2021 Carl Alphonce - Reproduction of this material is prohibited without the author's consent

Peephole optimization: flow-of-control



```
    goto L1
    ...
L1: goto L2
    ...
L2:
```

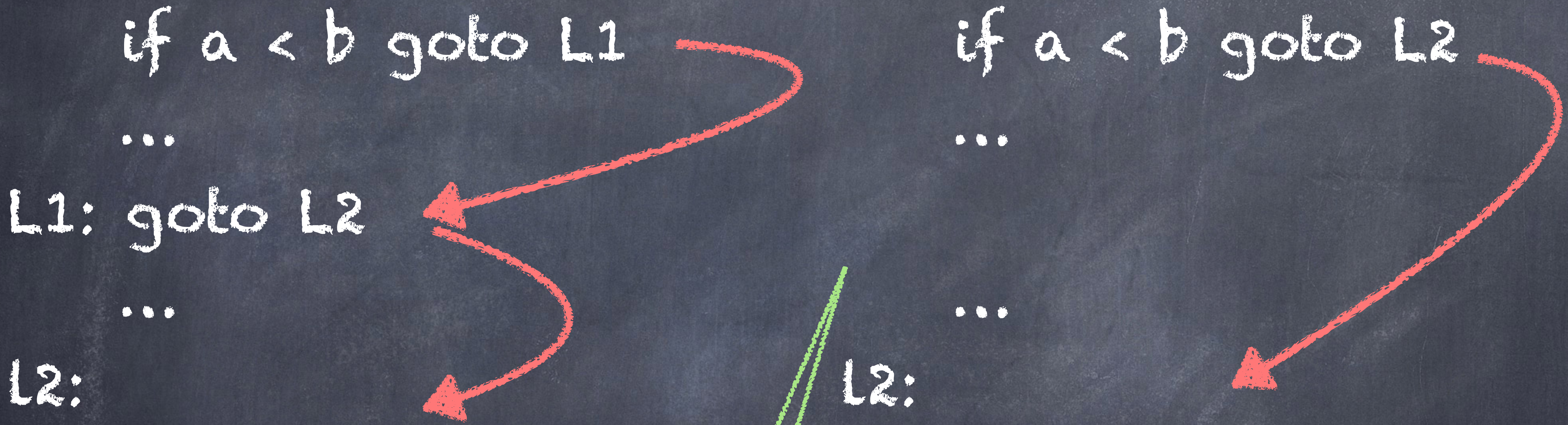
```
    goto L2
    ...

    ...
L2:
```

...then we can eliminate the statement labelled L1

© 2021 Carl Alphonce - Reproduction of this material is prohibited without the author's consent

Peephole optimization: flow-of-control



```
    if a < b goto L1
    ...
L1: goto L2
    ...
L2:
```

```
    if a < b goto L2
    ...

    ...
L2:
```

...similar arguments can be made for conditional jumps.

© 2021 Carl Alphonce - Reproduction of this material is prohibited without the author's consent

Optimization

- The semantics of a program **must** be preserved by optimizations.
- The compiler does not know a programmer's intent - it can only reason about the program as written.

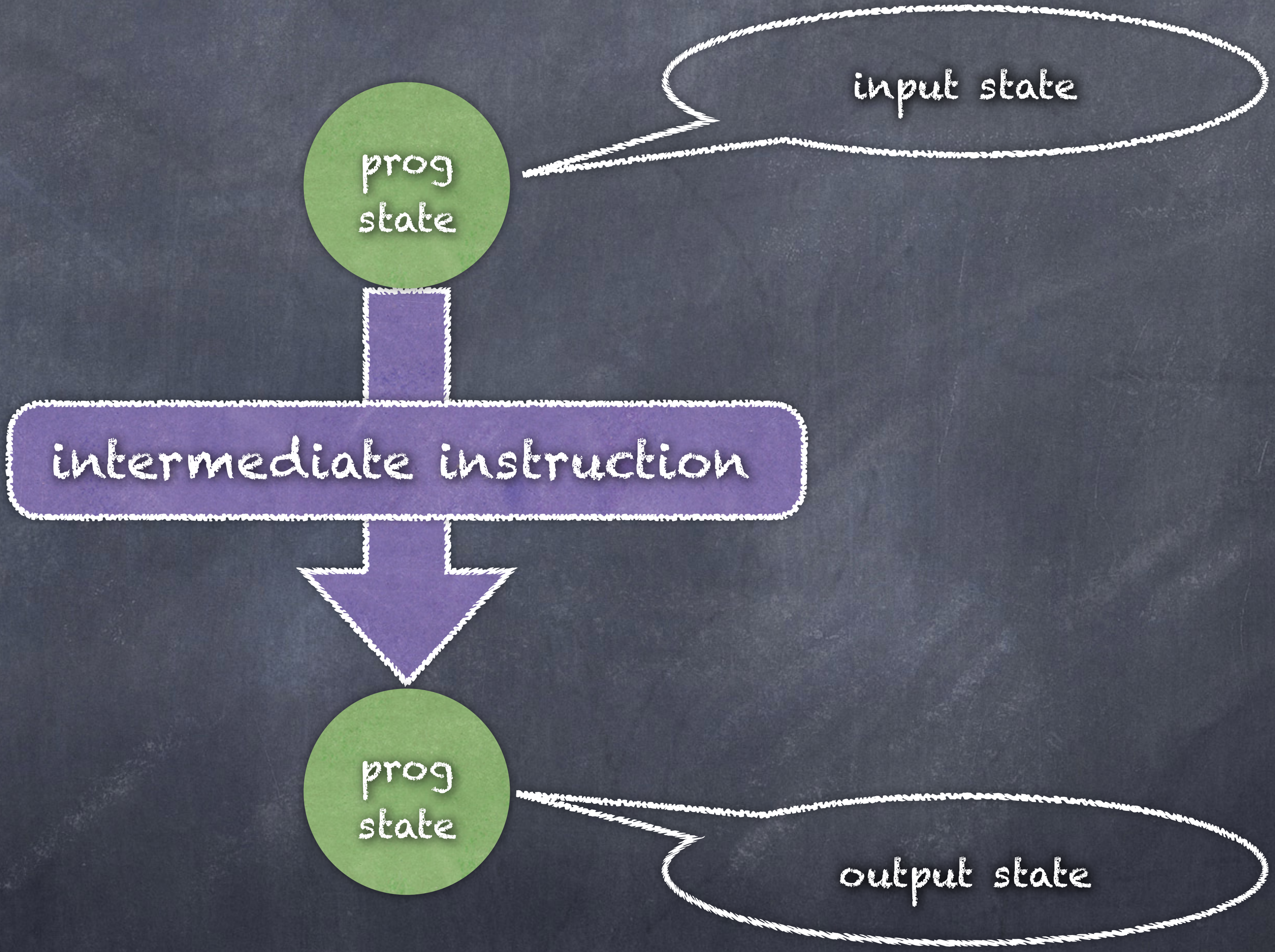
© 2020 Carl Alphonce - Reproduction of this material is prohibited without the author's consent

Data-flow analysis

- View program execution as a sequence of state transformations.
- Each program state consists of all the variables in the program along with their current values.

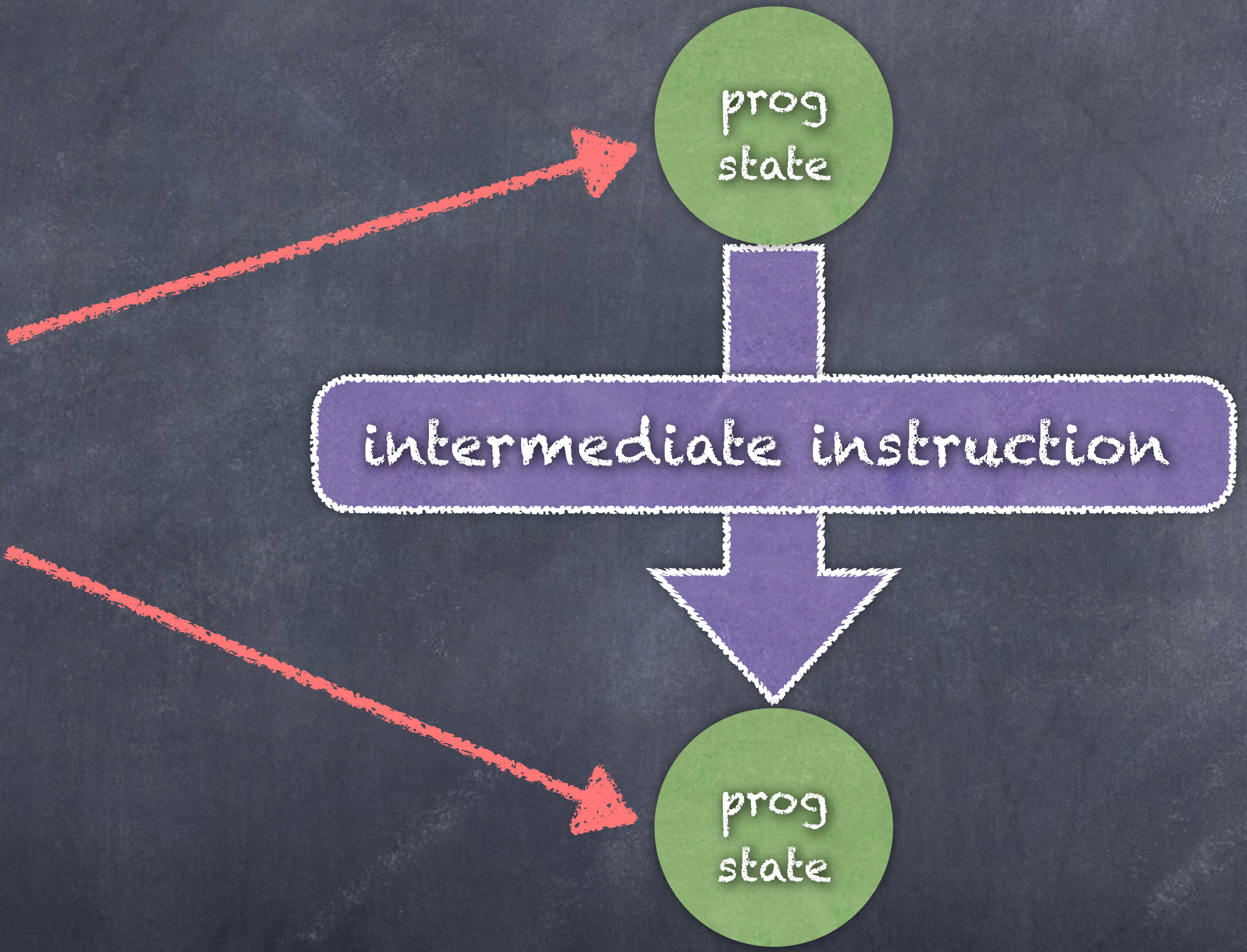
© 2020 Carl Alphonce - Reproduction of this material is prohibited without the author's consent

State transformation



© 2020 Carl Alphonce - Reproduction of this material is prohibited without the author's consent

State transformation



Program states are called program points.

A sequence of program points is called a path.

© 2020 Carl Alphonce - Reproduction of this material is prohibited without the author's consent

Data-flow analysis

- Begin by considering only the flow graph for a single function.

© 2020 Carl Alphonce - Reproduction of this material is prohibited without the author's consent

Properties

- Within a basic block:
  - Program point after a statement is same as program point before the next statement.
  - Why?

© 2020 Carl Alphonce - Reproduction of this material is prohibited without the author's consent

Properties

- Between basic blocks:
  - "If there is an edge from block B1 to block B2, then the program point after the last statement of B1 may be followed immediately by the program point before the first statement of B2."

[p. 597]

© 2020 Carl Alphonce - Reproduction of this material is prohibited without the author's consent

Execution path

"An execution path (or just path) from point p_1 to point p_n [is] a sequence of points p_1, p_2, …, p_n such that for each $i = 1,2,\ldots,n-1$, either

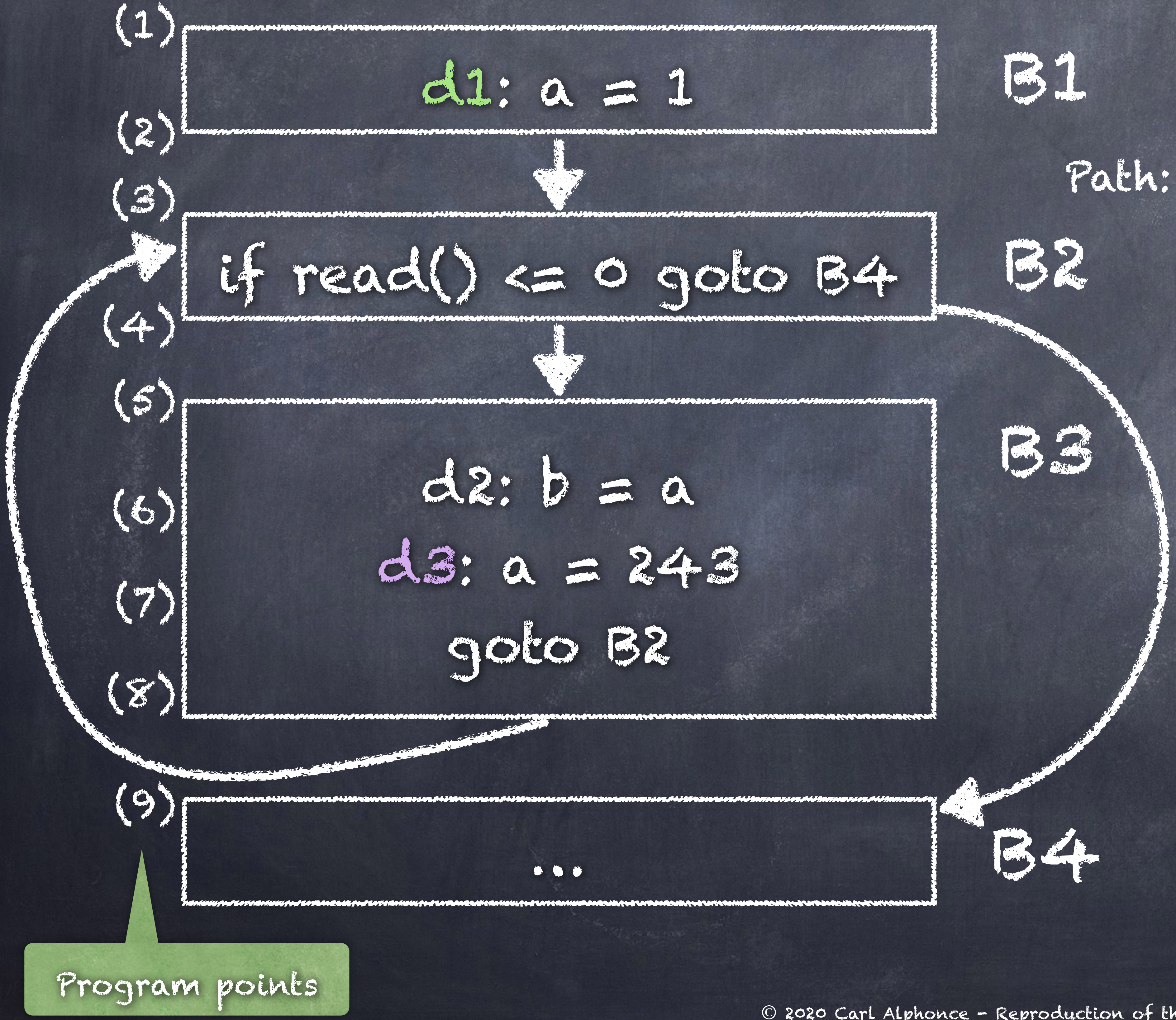
1. p_i is the point immediately preceding a statement and p_{i+1} is the point immediately following that same statement, or

2. p_i is the end of some block and p_{i+1} is the beginning of a successor block."

[p. 597]

© 2020 Carl Alphonce - Reproduction of this material is prohibited without the author's consent

Example 9.8 (p. 598)



(1)

B1

d1: a = 1

(2)

(3)

B2

if read() <= 0 goto B4

(4)

(5)

B3

(6) d2: b = a

(7) d3: a = 243

(8) goto B2

(9)

B4

...

Program points

Path: (1,2,3,4,9)

Path: (1,2,3,4,5,6,7,8,3,4,9)

a has value 1 first time (5) is executed.

d1 reaches (5) on the first iteration.

a has value 243 at (5) on the second and subsequent iterations.

d3 reaches (5) on those iterations.

© 2020 Carl Alphonce - Reproduction of this material is prohibited without the author's consent

Reaching definitions

"The definitions that may reach a program point along some path are known as reaching definitions."

[p. 598]

© 2020 Carl Alphonce - Reproduction of this material is prohibited without the author's consent

Gathering different data for different uses

to determine possible values

"… at point (5) … the value of a is one of { 1 , 243 } and … it may be defined by one of { d1 , d3 }."

[p. 598]

"… at point (5) … there is no definition that must be the definition of a at that point, so this set is empty for a at point (5). Even if a variable has a unique definition at a point, that definition must assign a constant to the variable. Thus, we may simply describe certain variables as 'not a constant', instead of collecting all their possible values or all their possible definitions."

[p. 599]

for 'constant folding'

© 2020 Carl Alphonce - Reproduction of this material is prohibited without the author's consent

9.2.2 Data-flow analysis schema

"In each application of data-flow analysis, we associate with every program point a data-flow value that represents an abstraction of the set of all possible program states that can be observed at that point." [p. 599]

"The set of possible data-flow values is the domain..." [p. 599]

"We denote the data-flow values before and after each statement s by IN[s] and OUT[s], respectively." [p. 599]

© 2020 Carl Alphonce - Reproduction of this material is prohibited without the author's consent

9.2.2 Data-flow analysis schema

"The data-flow problem is to find a solution to a set of constraints on the IN[s]'s and OUT[s]'s, for all statements s. There are two sets of constraints: those based on the semantics of the statements ("transfer functions") and those based on the flow of control." [p. 599]

© 2020 Carl Alphonce - Reproduction of this material is prohibited without the author's consent

Transfer functions

Information can flow forwards or backwards.

Forward flow: $OUT[s] = f_s (IN[s])$

Backward flow: $IN[s] = g_s (OUT[s])$

© 2020 Carl Alphonce - Reproduction of this material is prohibited without the author's consent

Control flow constraints

In a sequence $s_1, s_2, \ldots, s_n$ without jumps,

$IN[s_{i+1}] = OUT[s_i]$ for all $i=1,2,\ldots,n-1$

For data-flow between blocks, take "the union of the definitions after last statements of each of the predecessor blocks." [p. 600]

© 2020 Carl Alphonce - Reproduction of this material is prohibited without the author's consent

9.2.3 Data-flow schemas on basic blocks

Suppose a basic block B consists of the sequence of statements s_1, s_2, …,s_n. Define IN[B] = IN[s_1] and OUT[B] = OUT[s_n].

The transfer function of B:

$$f_B = f_{sn} \circ \dots \circ f_{s2} \circ f_{s1}$$

The transfer function of B:

$$OUT[B] = f_B(IN[B])$$

© 2020 Carl Alphonce - Reproduction of this material is prohibited without the author's consent

9.2.3 Data-flow schemas on basic blocks

Forward flow problem

$\text{OUT}[B] = f_B(\ \text{IN}[B]\ )$

$\text{IN}[B] = \cup_{P \text{ a predecessor of } B}\ \text{OUT}[P]$

Backward flow problem

$\text{IN}[B] = g_B(\ \text{OUT}[B]\ )$

$\text{OUT}[B] = \cup_{S \text{ a successor of } B}\ \text{IN}[S]$

© 2020 Carl Alphonce - Reproduction of this material is prohibited without the author's consent