CS202: Software Tools and Techniques for CSE

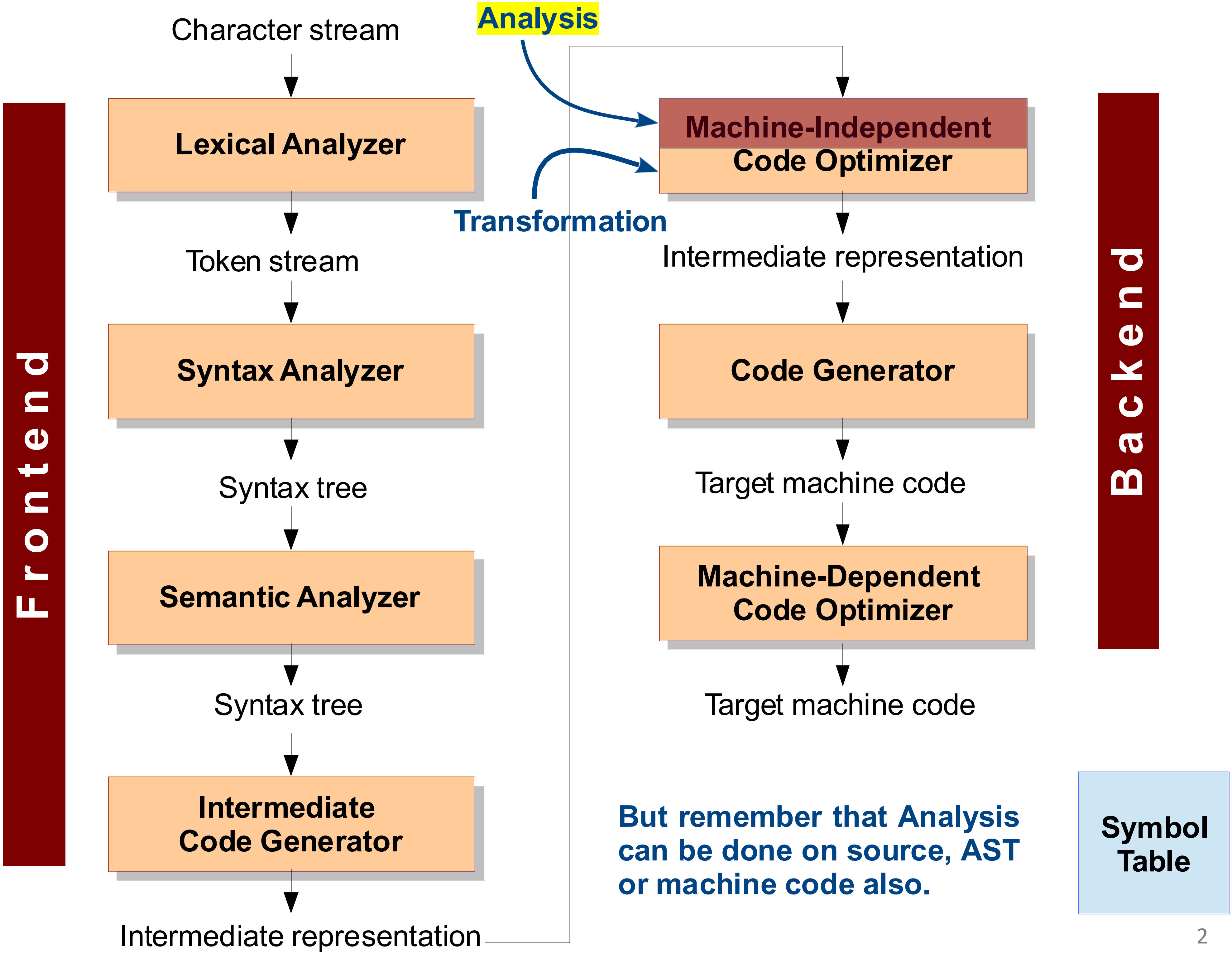
Lecture 7

Shouvick Mondal

shouvick.mondal@iitgn.ac.in

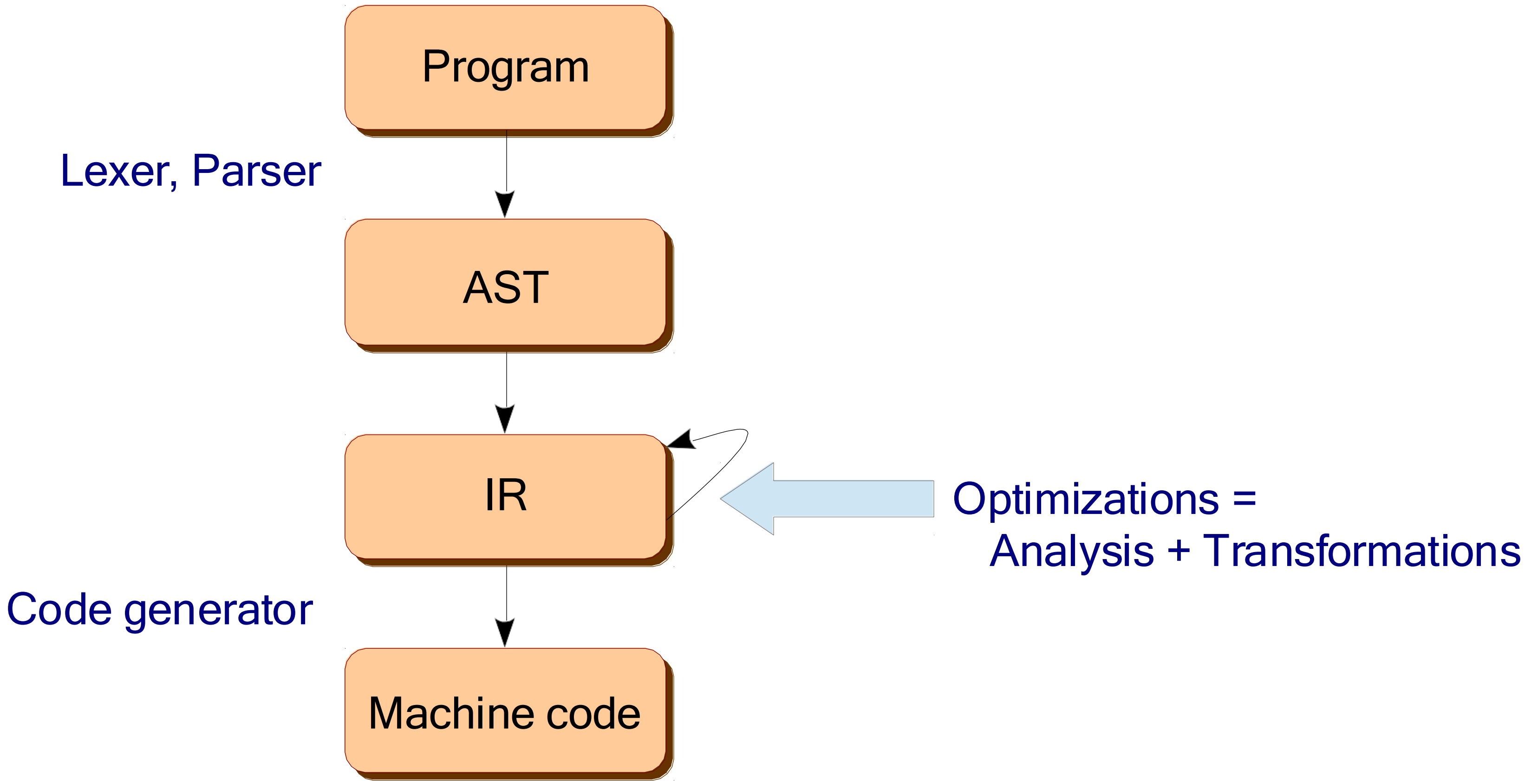
August 2025

Slides obtained from: http://www.cse.iitm.ac.in/~rupesh/teaching/pa/jan19



But remember that Analysis can be done on source, AST or machine code also.

Compiler Organization



Why are we doing this?

- To apply data-flow analysis and its variants on input programs and collect relevant information
  - reaching definitions, points-to information, etc.

- To design and implement analyses for new problems

Example

```
void main() {
  int a, b, c, d, *p;

  p = &a;
  c = a + b;
  d = *p + b;
}
```

Can this computation be avoided?
(*common subexpression elimination*)

```
void main() {
  int a, b, c, d, *p;

  p = &a;
  int t = a + b;
  c = t;
  d = t;
}
```

This requires a program analysis called *pointer analysis*.

This requires another analysis called *type analysis*.

What is Pointer Analysis?

```
a = &x;
b = a;
if (b == *p) {
    …
} else {
    …
}
```

What is Points-to Analysis?

```
a = &x;
b = a;
if (b == *p) {
    …
} else {
    …
}
```

a points to x

What is Points-to Analysis?

```
a = &x;
b = a;
if (b == *p) {
    …
} else {
    …
}
```

a points to x

a and b are aliases

What is Points-to Analysis?

```
a = &x;
b = a;
if (b == *p) {
    …
} else {
    …
}
```

a points to x

a and b are aliases

Is this condition always sa

Points-to information

a → {x, y}
b → {y, z}
c → {z}

Aliasing information

	a	b	c
a	--	Yes	No
b	--	--	Yes
c	--	--	--

Pointer Analysis is a mechanism to **statically** find out run-time values of a pointer.

Why Points-to Analysis?

- for Parallelization
  - `fun(p) || fun(q)`
- for Optimization
  - `a = p + 2;`
  - `b = q + 2;`
- for Bug-Finding
- for Program Understanding
- ...

$\Bigg\}$ **Clients of Points-to Analysis**

Compiler Basics

- Program as Data
- Control-Flow Graph (CFG)
- Basic Blocks
- Optimizations
  - *gcc -O2 prog.c*

```
int main() {
  int x = 1;
  if (x > 0)
      ++x;
  else
      x = 100;
  printf("%d\n", x);
}
```

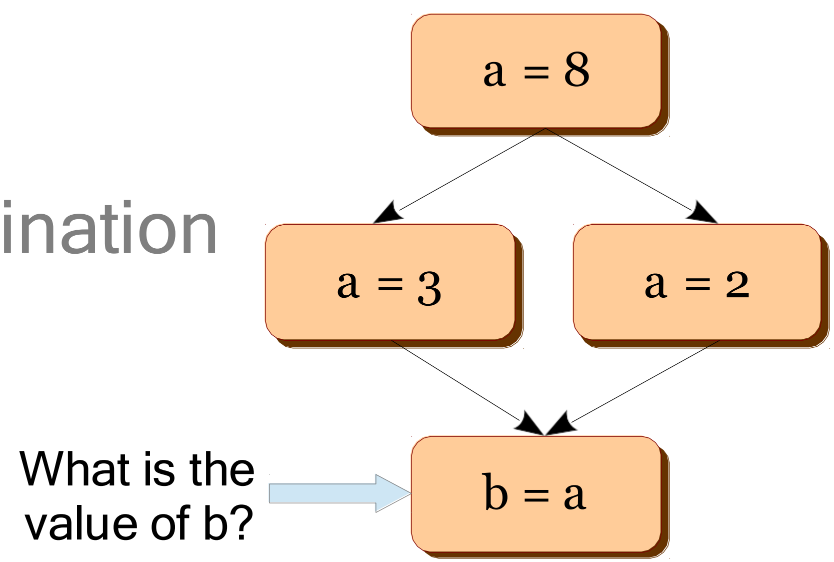
```
int main() {
  int x = 1;
  if (1 > 0)
      ++x;
  else
      x = 100;
  printf("%d\n", x);
}
```

```
int main() {
  int x = 1;
  ++x;
  printf("%d\n", x);
}
```

```
int main() {
  printf("%d\n", 2);
}
```

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

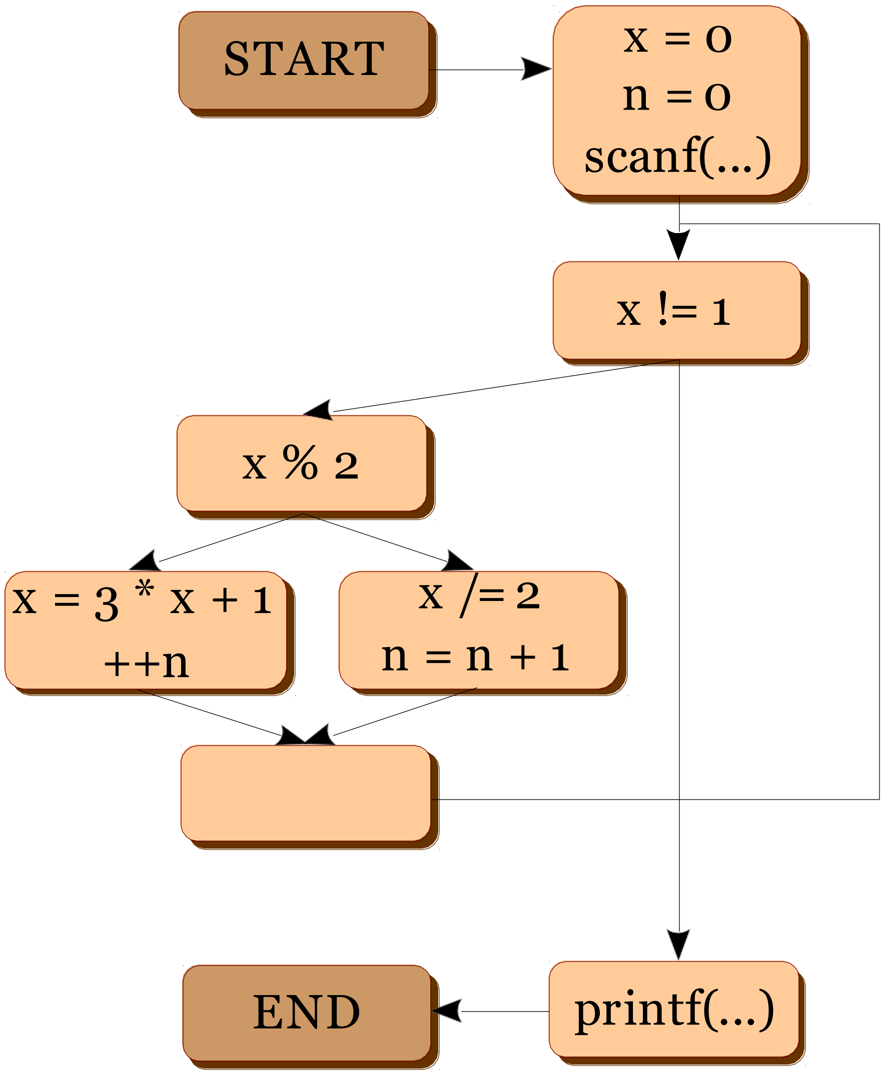


Example: Control Flow Graph (CFG)

- Draw the CFG for the following program.

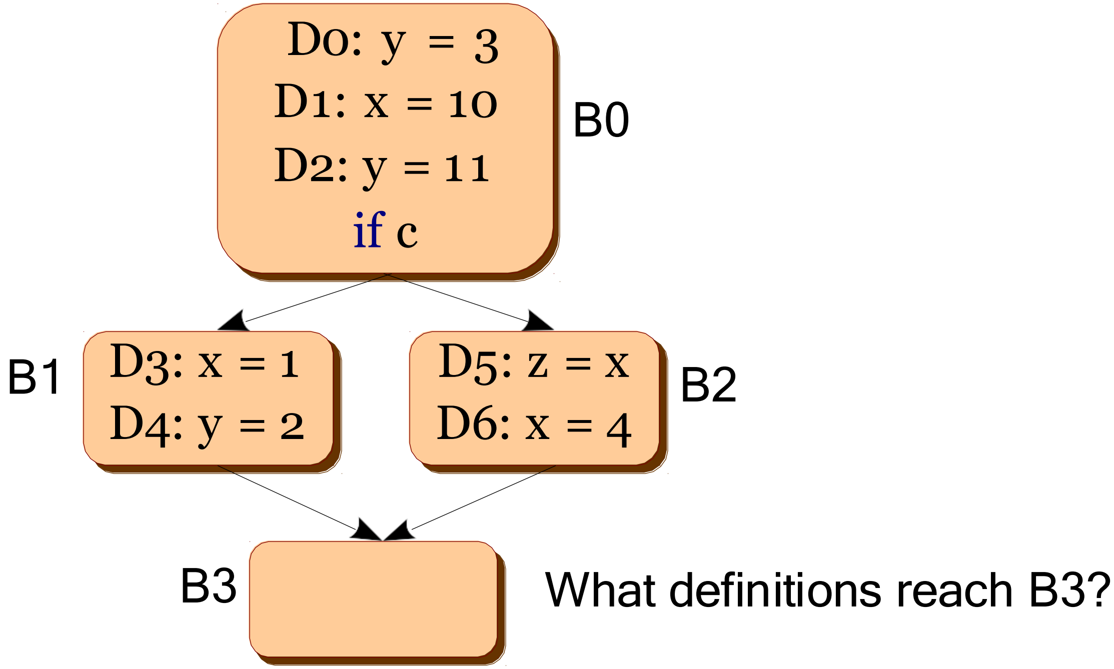
```
int main() {
  int x = 0, n = 0;
  scanf("%d", &x);

  while (x != 1) {
    if (x % 2) {
      x = 3 * x + 1;
      ++n;
    } else {
      x /= 2;
      n = n + 1;
    }
  }
  printf("%d\n", n);
}
```



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.

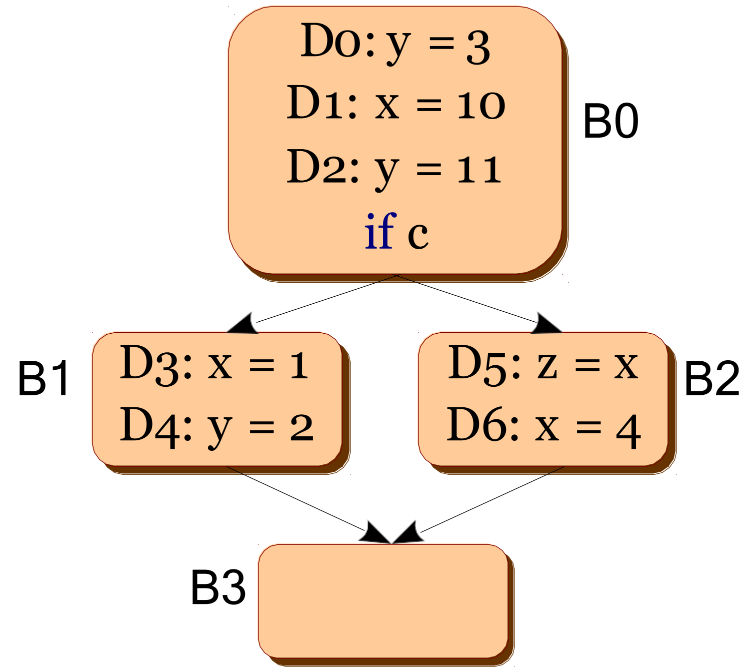


DFA Equations

- 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))



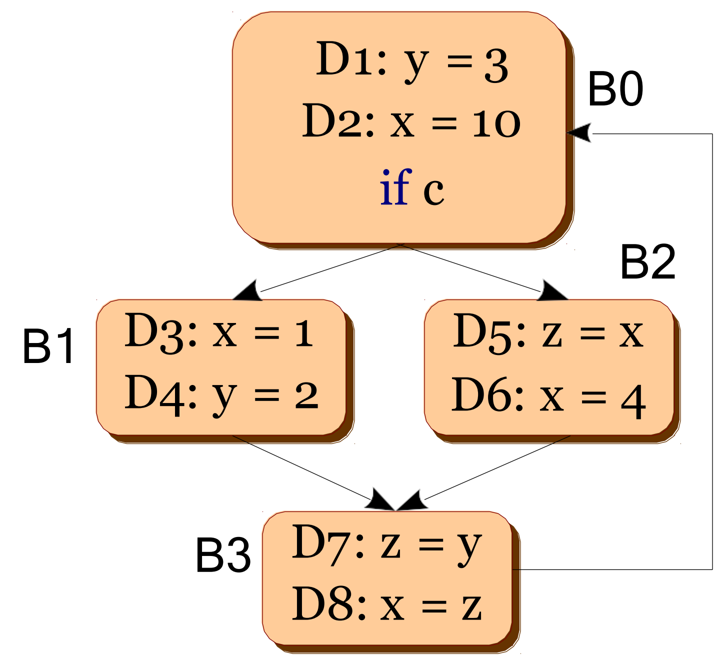
- Initially, out(B) = { }

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}

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) = { }

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, 8}	{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	{}	{D7, D8}	{D3,4,5,6}	{D4,7,8}	{D1, D3, D4, D5, D6}	{D1,4, 7, 8}	{D1,3,4,5,6,7}	{D1,4,7,8}

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 ϵ pred(B)
      out(B) = gen(B) U (in(B) -kill(B))
}  while in(B) changes for any basic block B
```

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) = { }

Memory Optimization

- Reuse memory / register wherever possible.
- z and y can reuse memory / register.

Live Ranges

x : (0,1), (3,4,5), (0,1,6,7,8)
y : (4,5,10), (7,8,9,10)
z : (0,1,2), (0,1,6,7)

```
0  int x = 2, y = 3, z = 1;
1  if (x == 2) {
2     y = z;
3     x = 9;
4     y = 7;
5     x = x – y;
6  } else {
7     y = x + z;
8     ++x;
9  }
10 printf(“%d”, y);
```

This optimization demands computation of live variables.