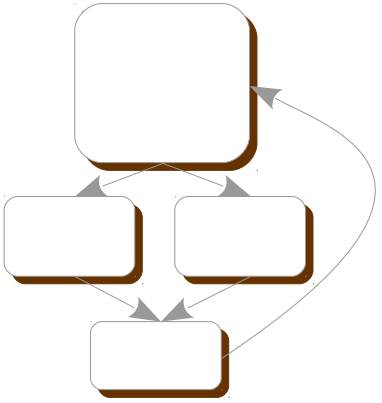
Program Analysis



Rupesh Nasre.
IIT Madras

Raisoni
August 2020

Outline

- Compilers
  - Block Diagram
- Program Analysis
  - Motivation
  - Control Flow Graph
  - Reaching Definitions Analysis
  - Live Variables Analysis
  - Analysis dimensions

Languages

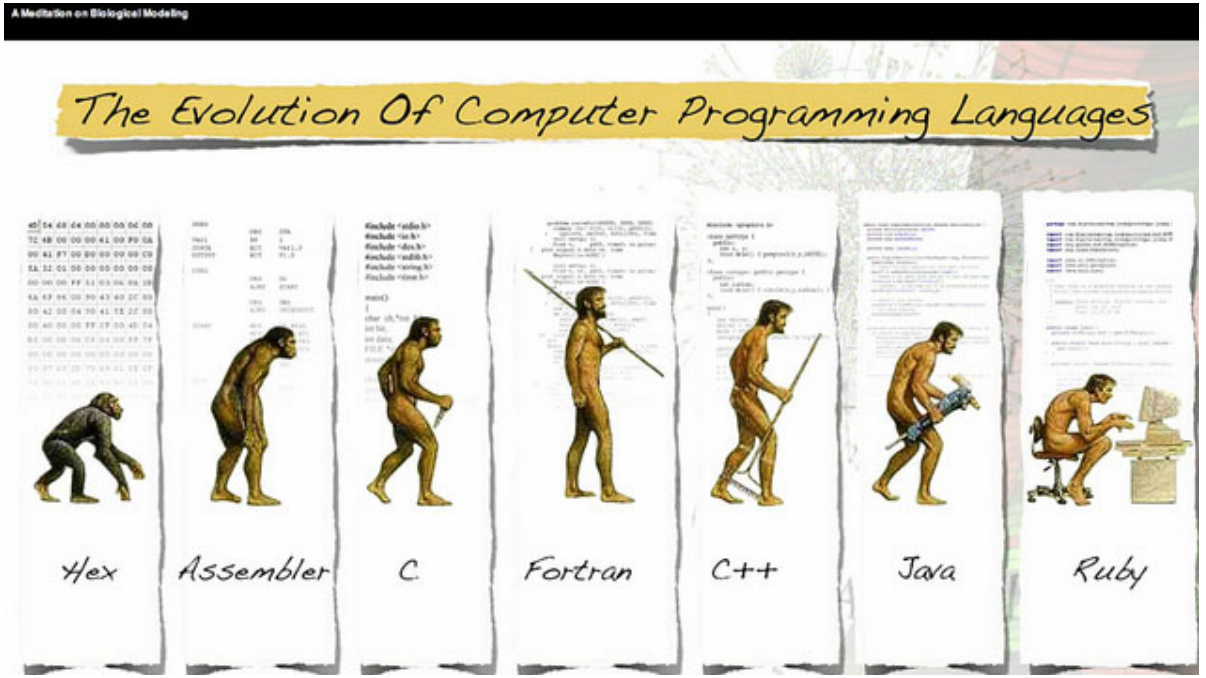
Wall of Love, Paris
Source: google images

Languages

- Why do we need languages?
  - Humans communicate
    - sign language, body language, braille
  - Birds communicate
    - mark territories, attract for mating, warn danger
  - Animals communicate
    - mark territories, convey need, preparation for attack
  - Aliens?

Programming Languages

- Why do we need programming languages?
- And why so many?
  - What is your first language?
  - Marathi. Yours?
  - C.



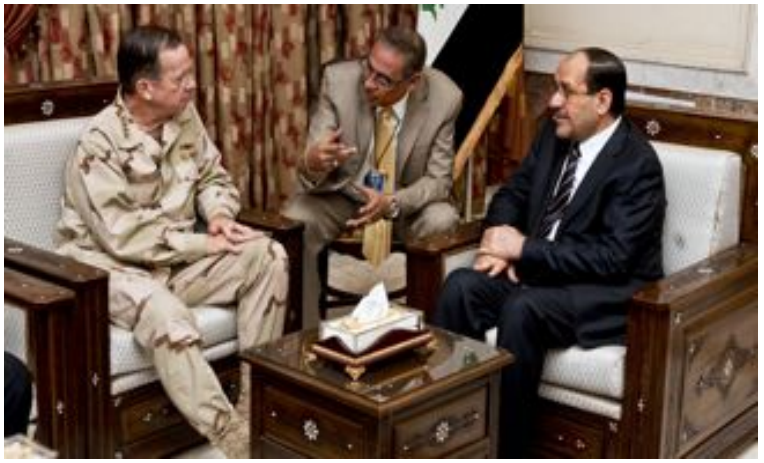
Source: google images

Programming Languages

- There are some special purpose languages
  - HTML for webpages
  - LaTeX for document formatting
  - ps for postscript files; sql, VHDL
  - Shell scripts, awk, grep, sed
  - Makefile has a language; smtp
  - How about google search?
    - filetype:pdf, link:www.cse.iitm.ac.in
  - Gmail: in:unread, in:starred
  - vi: :se ai, :wq, :se ft=c
  - What about ls -l, ls -Ri, ls --color, ls -1 dir1 dir2 ?

Compiler

- When do we need a compiler?
  - நான் தமிழ் தெரியுமா
  - मुझे हिंदी आता है
  - Ich kenne Deutsch
  - I know English



Source: google images

Jobs of a Compiler

- Translate: input language, output language
- Maintain correctness
  - पिताजी अजमेर गए।
  - Father died today.
- Be efficient
  - Why are you laughing?
  - I understood yesterday's joke.
- Generate a good language
  - I got books but more than that I got your letter.
  - मैं किताबें, लेकिन मैं अपने पत्र मिला है कि अधिक से अधिक मिला है।

Compilers work with Strings

- Characters, words / tokens, sentences, programs
- Fun with strings
  - quick brown fox jumps over the lazy dog
  - stewardesses
  - typewriter
  - skepticisms
  - quine

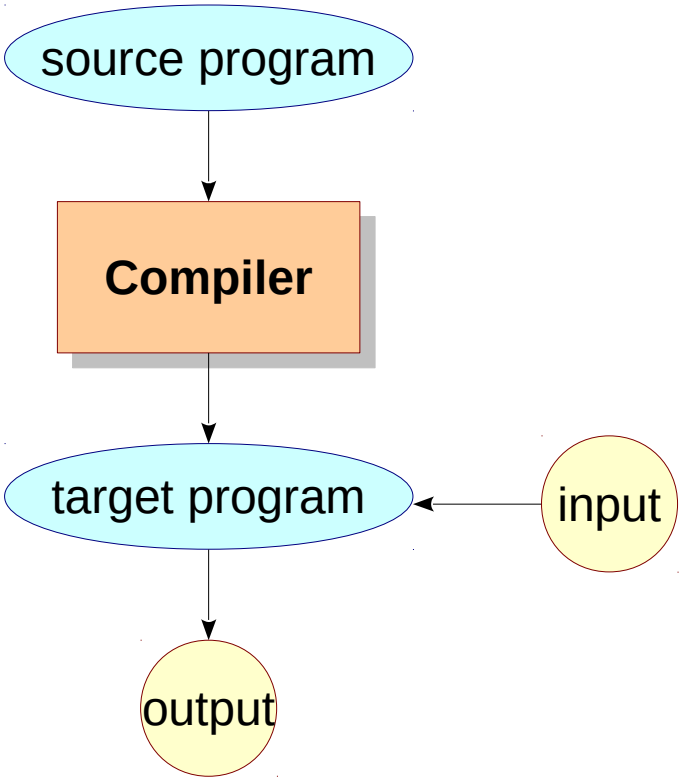
Programs as Data

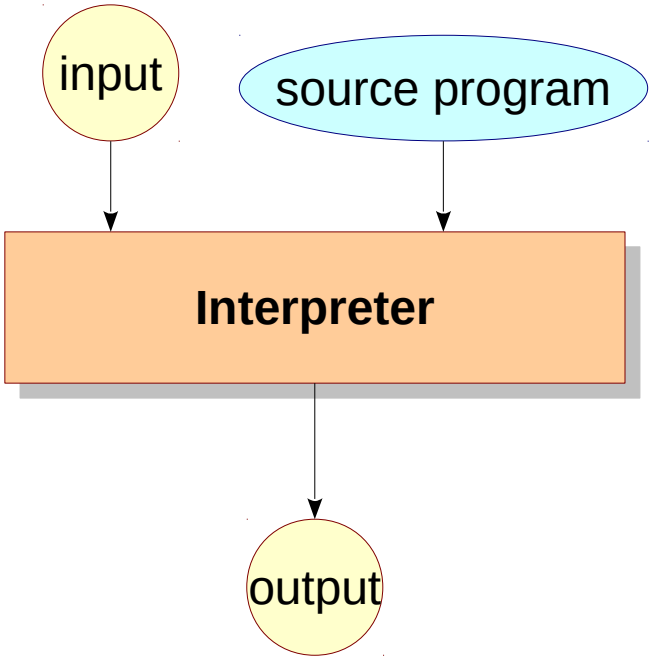
```
char*f="char*f=%c%s%c;main(){printf(f,34,f,34,10);}%c";main(){printf(f,34,f,34,10);}
```

Why should we Design a language?

- Language matters!
  - A: Would you accept a gamble that offers a 10% chance to win $95 and a 90% chance to lose $5?
  - B: Would you pay $5 to participate in a lottery that offers a 10% chance to win $100 and a 90% chance to win nothing.
- Outcomes of a treatment for lung cancer. Two descriptions were:
  - C: The one-month survival rate is 90%.
  - D: There are 10% deaths in the first month.
- B fetched many more positives. 84% physicians chose option C.

Source: Thinking Fast and Slow, Daniel Kahneman





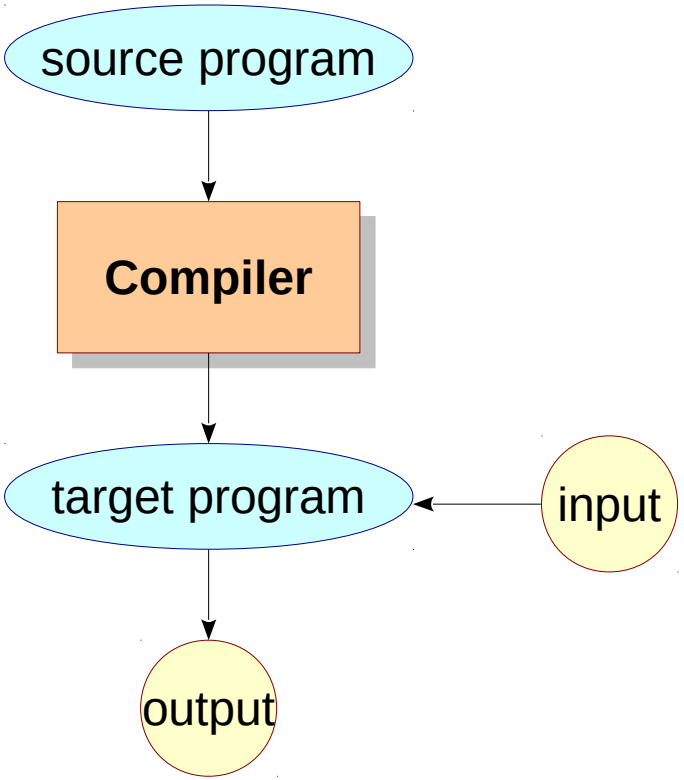
- What does this mean?
  - You may be able to do the following with interpreters.

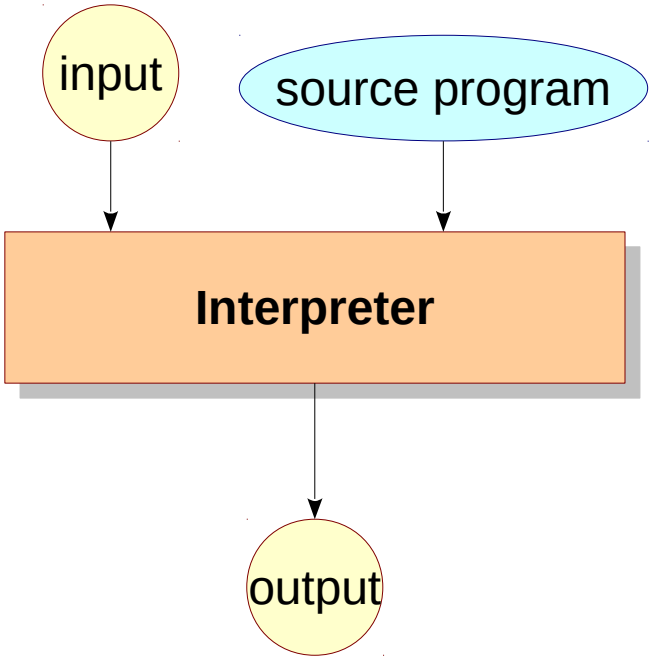
```
$x = 0; $y = 0;
echo "Enter a variable name: ";
$line = fgets(STDIN);
$line = trim($line);
${$line} = 20;
echo "x=$x, y=$y\n";
```

How about C?

```
void main() {
        int x = 0, y = 0;
#include "/dev/stdin"
        = 10;
        printf("x = %d, y = %d\n", x, y);
}
```


Everything is fair in love, war and C.



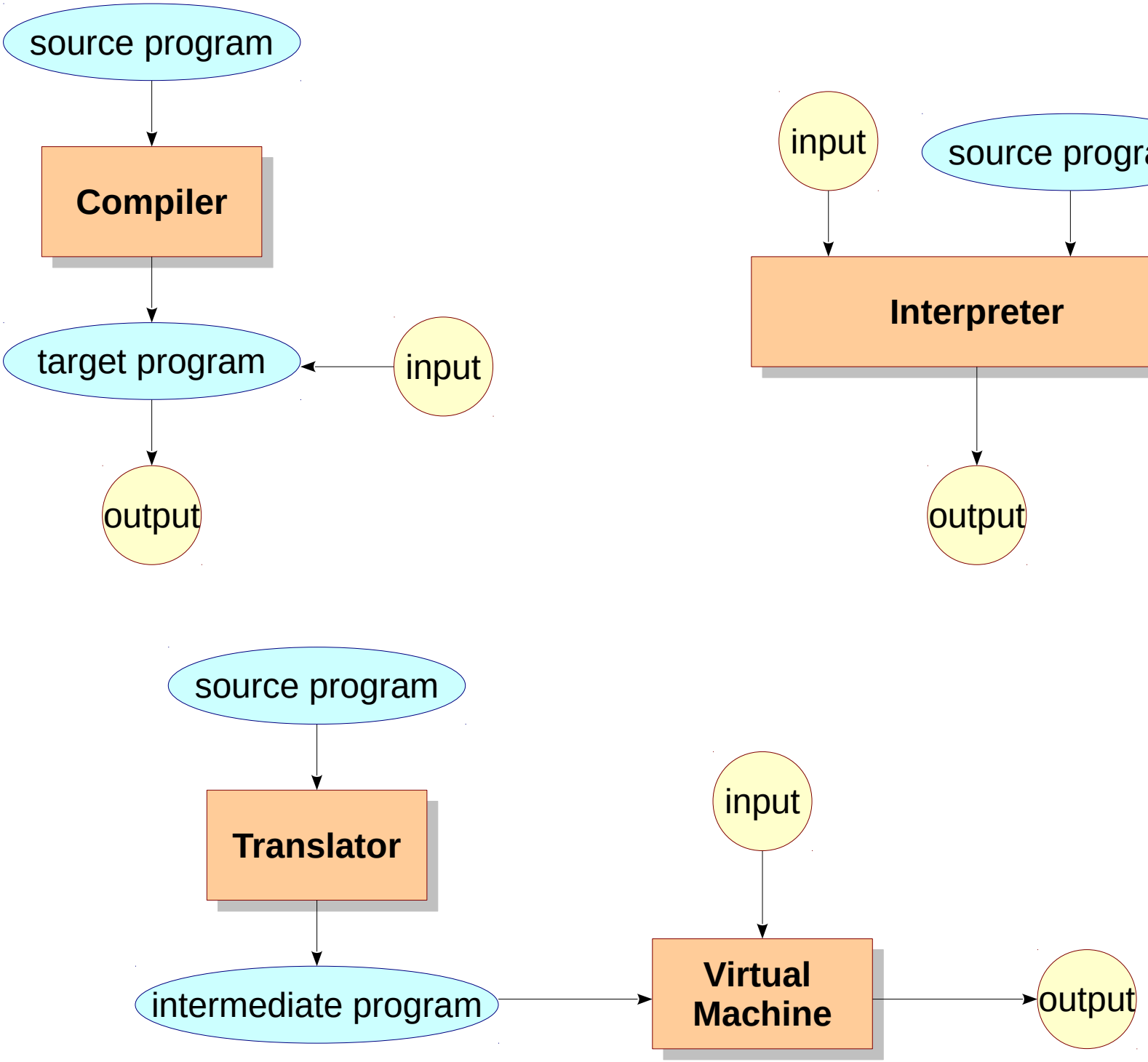


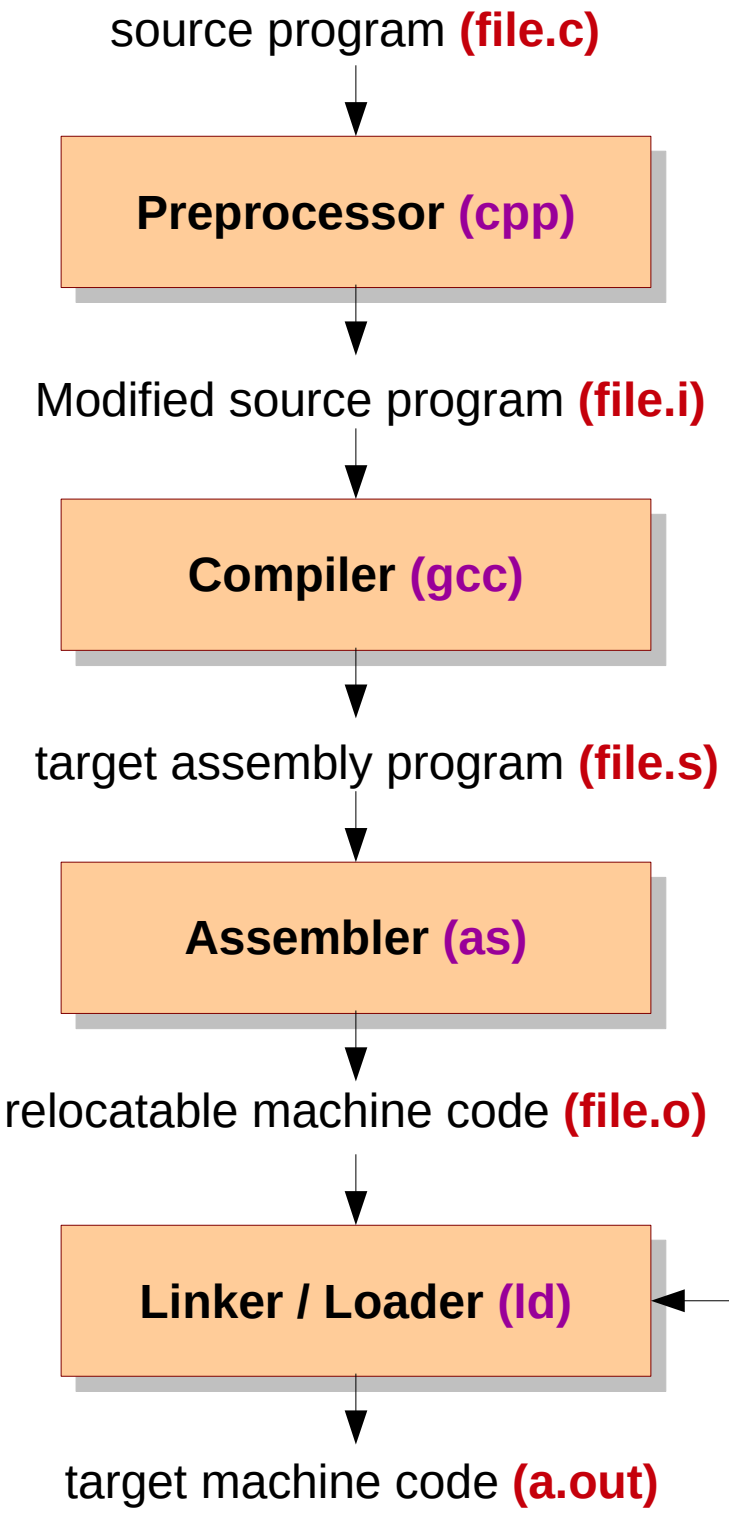
- What does this mean?
  - You may be able to do the following with compilers.

```
x += 2;
x += 2;
--x;
x += 5;
++x;
x += 9
```

is equivalent to

```
x += 18;
```


source program
Compiler
target program
input
output
input
source program
Interpreter
output
source program
Translator
intermediate program
input
Virtual Machine
output



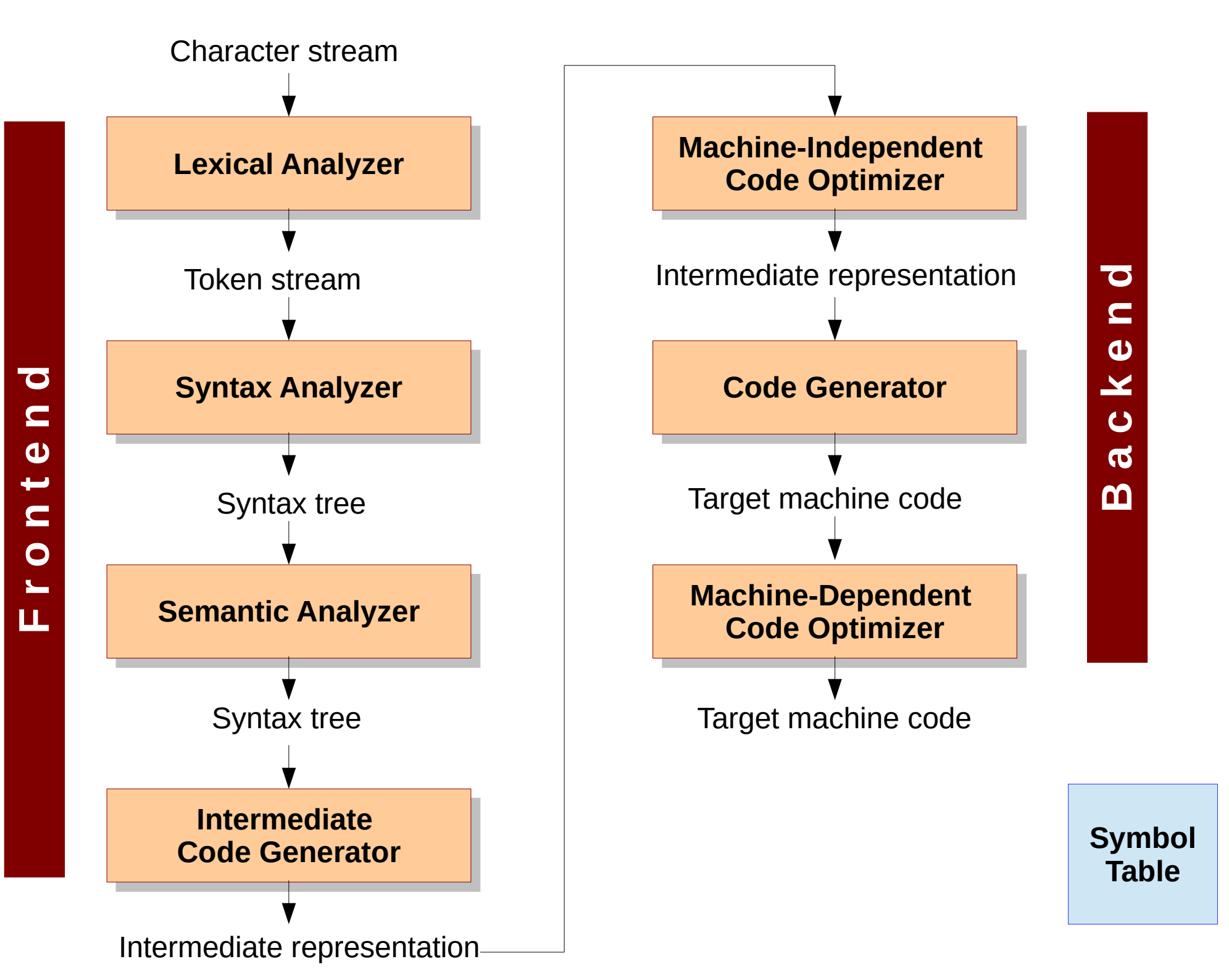
- cpp file.c >file.i
- gcc -S file.i
- as file.s -o file.o
- ld -o a.out file.o ...*libraries*...

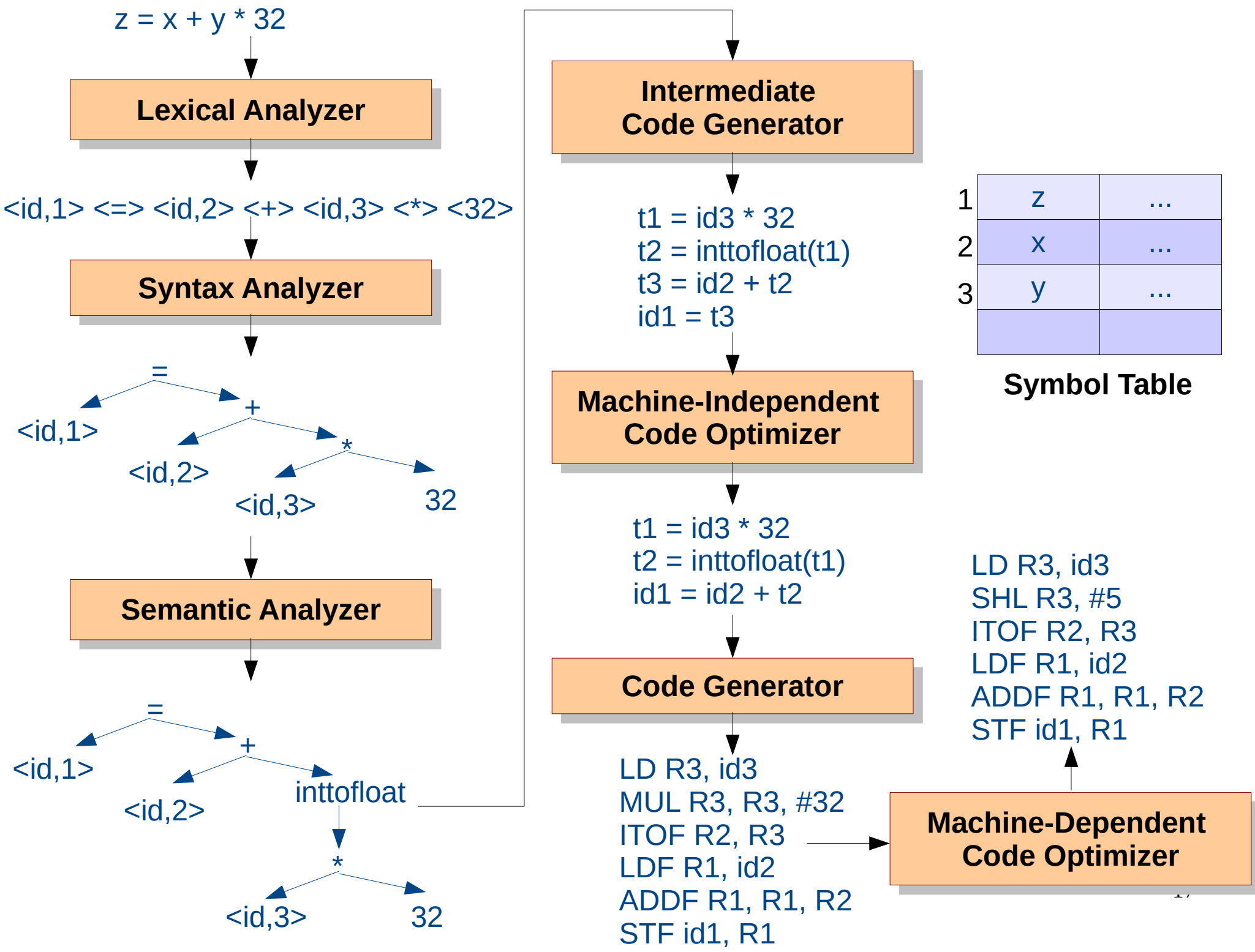
Try the following:

- gcc **-v** file.c
- gcc **-save-temps** file.c

Language Translators

- **Preprocessor**: collects source programs, expands macros.
- **Compiler**: Translates source program into a low-level assembly.
- **Assembler**: Produces (relocatable) machine code.
- **Linker**: Resolves external references **statically**, combines multiple machine codes.
- **Loader**: Loads executable codes into memory, resolves external references **dynamically**.


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


z = x + y * 32
Lexical Analyzer
<id,1> <=> <id,2> <+> <id,3> <*> <32>
Syntax Analyzer
=
<id,1>
+
<id,2>
*
<id,3>
32
Semantic Analyzer
=
<id,1>
+
<id,2>
inttofloat
*
<id,3>
32
Intermediate Code Generator
t1 = id3 * 32
t2 = inttofloat(t1)
t3 = id2 + t2
id1 = t3
Machine-Independent Code Optimizer
t1 = id3 * 32
t2 = inttofloat(t1)
id1 = id2 + t2
Code Generator
LD R3, id3
MUL R3, R3, #32
ITOF R2, R3
LDF R1, id2
ADDF R1, R1, R2
STF id1, R1
Machine-Dependent Code Optimizer
LD R3, id3
SHL R3, #5
ITOF R2, R3
LDF R1, id2
ADDF R1, R1, R2
STF id1, R1
1 z ...
2 x ...
3 y ...
Symbol Table

Outline

- Compilers
  - Block Diagram
- Program Analysis
  - Motivation
  - Control Flow Graph
  - Reaching Definitions Analysis
  - Live Variables Analysis
  - Analysis dimensions

What is Program Analysis?

For an end-goal, identify “interesting aspects” of a program's representation.

What is Program Analysis?

For an end-goal,

identify “interesting aspects”

of a program's representation.

Checking security

Array index range

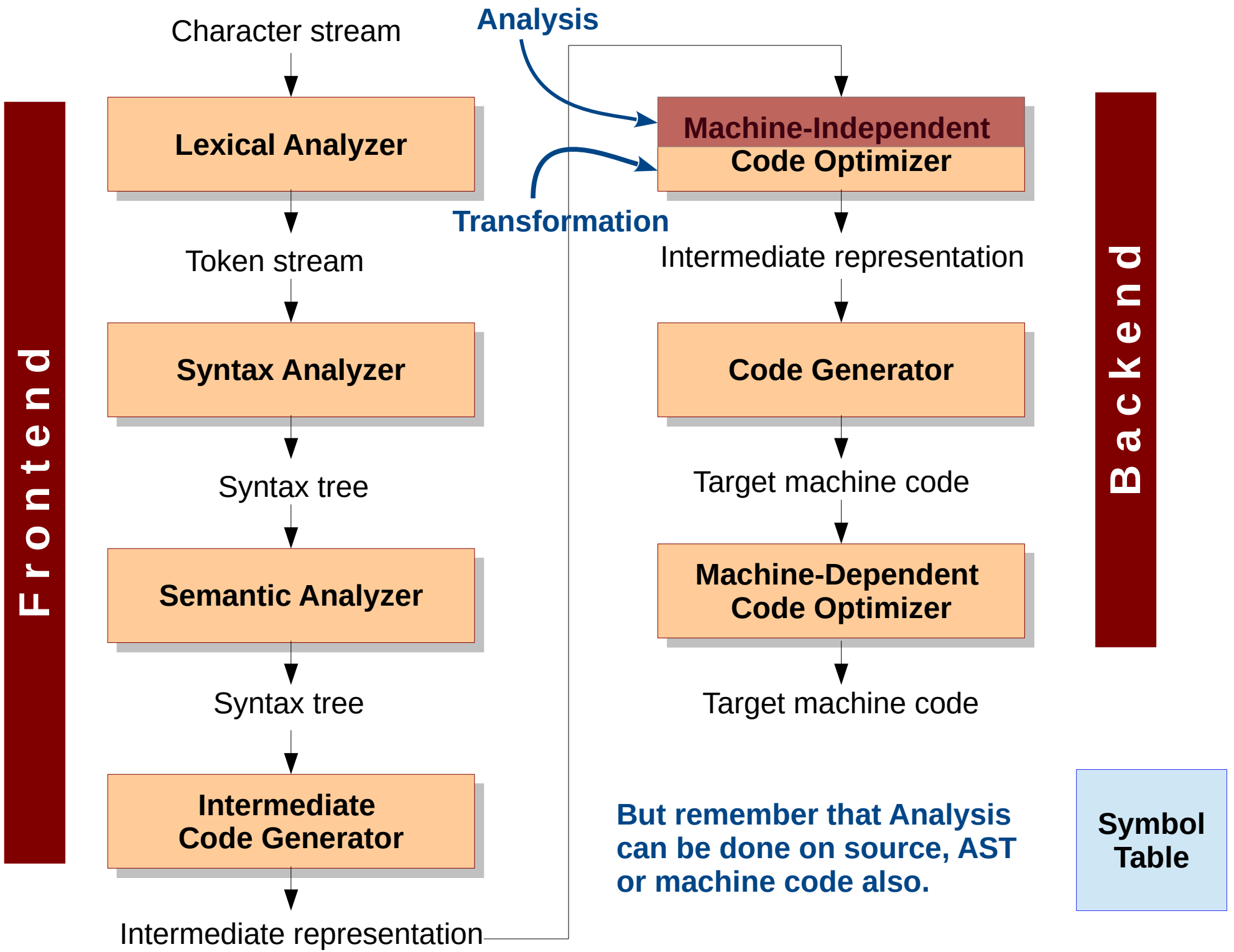
Source, AST, binary, executed instruction

Classwork: Write down two types of information you can extract from programs.

Examples

End goal	Interesting aspect
Dead code elimination	Reachability
Constant propagation	use-def
Security	Array index range, dangling pointers
Parallelization	Data dependence, SIMD opportunities
Debugging	Slice
Cache performance	Memory access pattern
Memory reduction	Live ranges
...	...

Program Analysis is often a pre-cursor to Optimization.

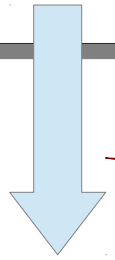

Frontend
Character stream
Lexical Analyzer
Token stream
Syntax Analyzer
Syntax tree
Semantic Analyzer
Syntax tree
Intermediate Code Generator
Intermediate representation
Analysis
Transformation
Machine-Independent Code Optimizer
Intermediate representation
Code Generator
Target machine code
Machine-Dependent Code Optimizer
Target machine code
Backend
But remember that Analysis can be done on source, AST or machine code also.
Symbol Table

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*.

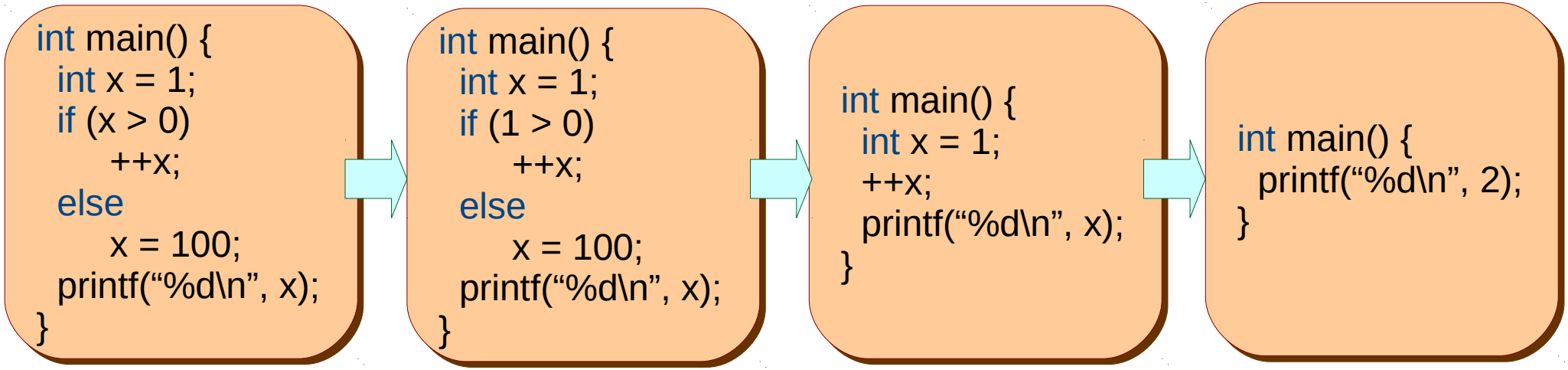
This requires another analysis called *type analysis*.

Outline

- Compilers
  - Block Diagram
- Program Analysis
  - Motivation
  - Control Flow Graph
  - Reaching Definitions Analysis
  - Live Variables Analysis
  - Analysis dimensions

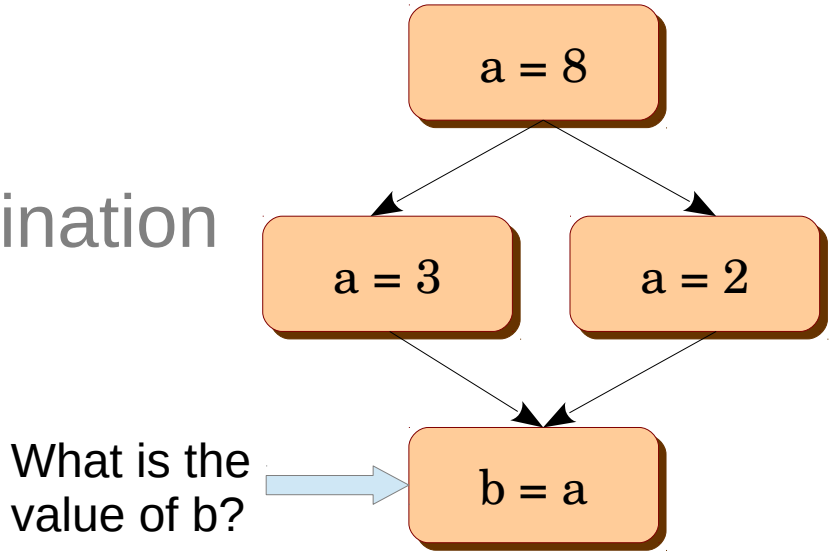
Compiler Basics

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



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

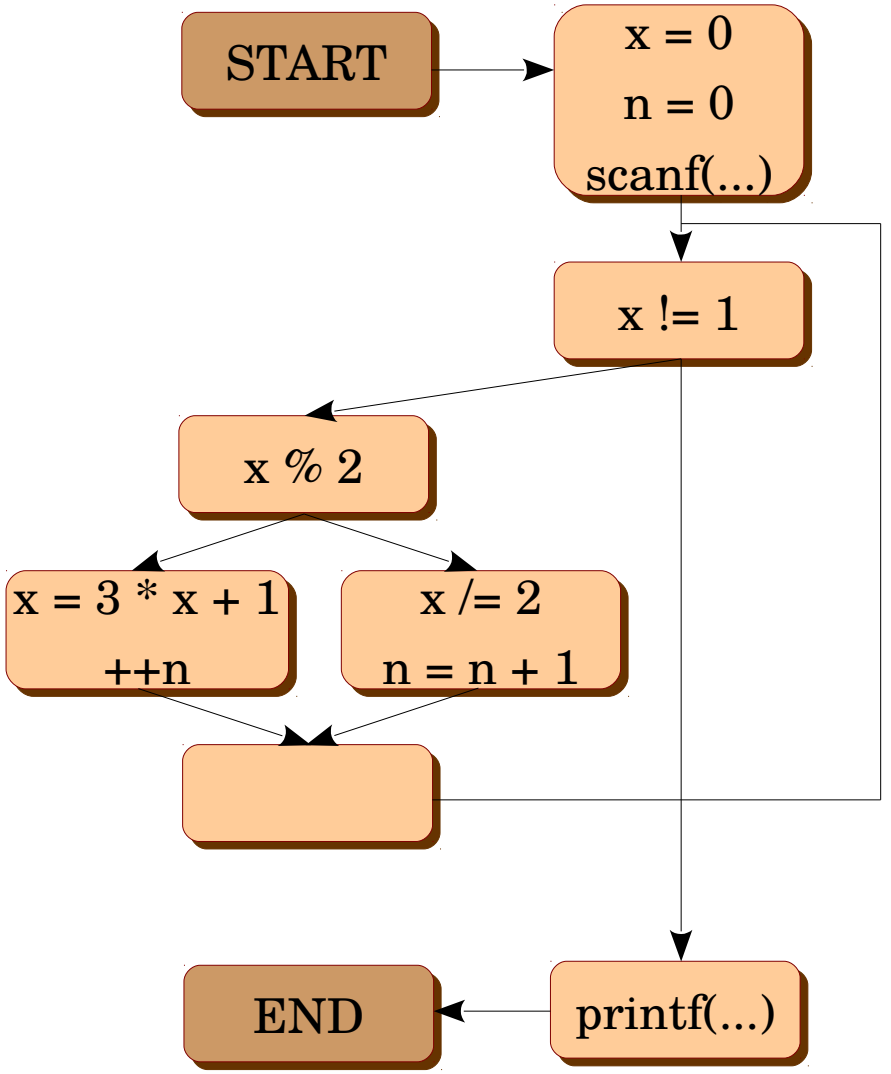


Classwork

- Draw the CFG for the following program.

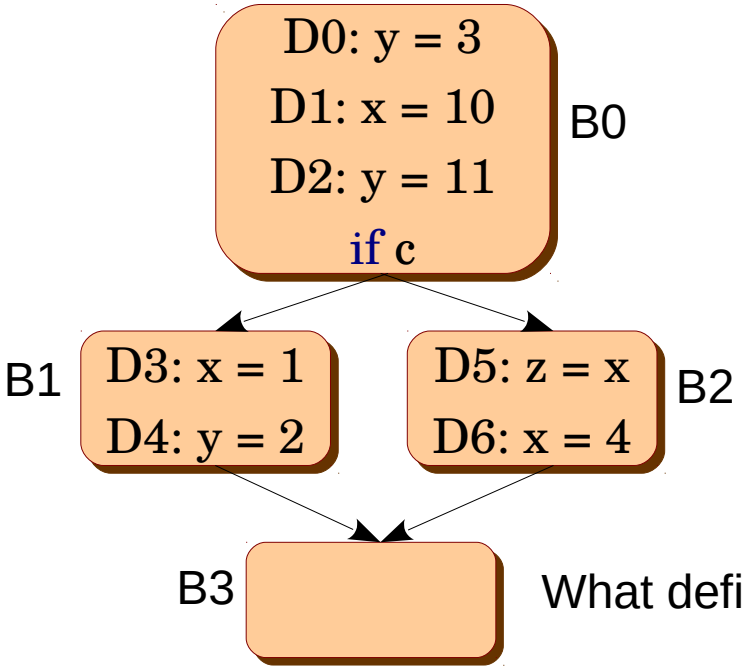
```
int main() {
  int x = 0, n = 0;
  scanf("%d", &x);

  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.

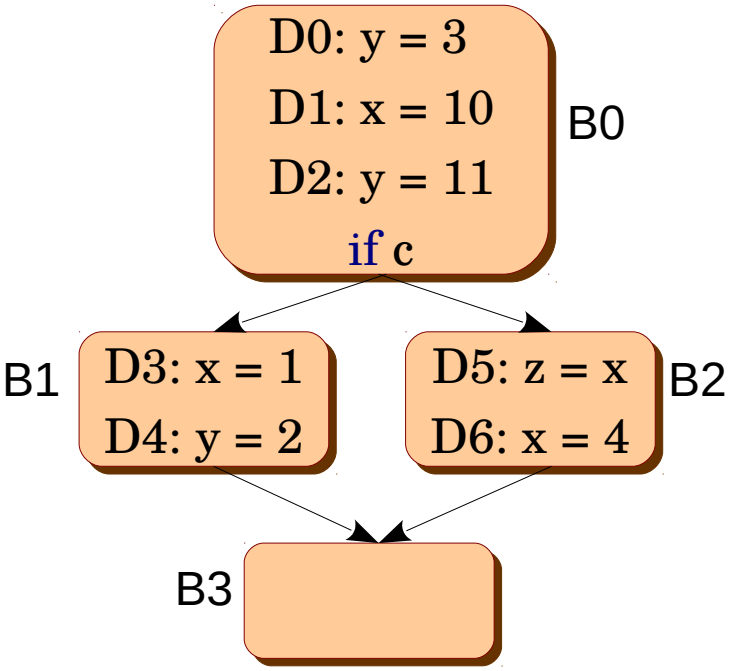


DFA Equations

- 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}

Algorithm for Reaching Definitions

for each basic block B

  compute gen(B) and kill(B)

  out(B) = {}

do {

  for each basic block B

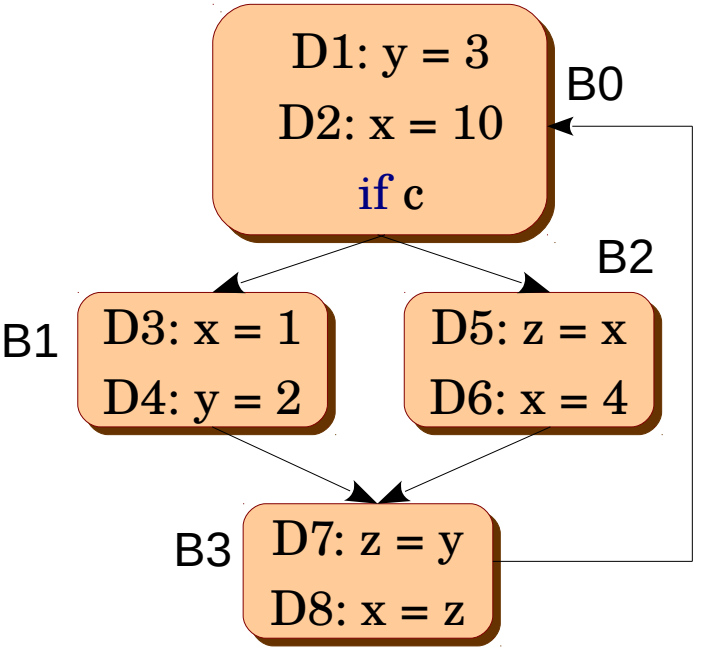
    in(B) = U out(P) where P \in pred(B)

    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) = { }



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}

DFA for Reaching Definitions

Domain	Sets of definitions
Transfer function	$\text{in}(B) = \cup\ \text{out}(P)$ $\text{out}(B) = \text{gen}(B) \cup (\text{in}(B) - \text{kill}(B))$
Direction	Forward
Meet / confluence operator	$\cup$
Initialization	$\text{out}(B) = \{\}$

Memory Optimization

- Reuse memory / register wherever possible.
- y is dead at lines 2, 3, 4.
- It is also dead at else block.
- z and y can reuse memory / register.

```
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.

DFA for Live Variables

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

Definition: 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.

How to compute live variables?

Classwork

- Write an algorithm for Live Variable Analysis

```
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
```

Algo for **reaching definitions**

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) = { }

Parameters for **live variable analysis**

Direction and Confluence

	Forward	Backward
∪	Reaching Definitions	Live Variables
∩	Available Expressions	Very Busy Expressions

An expression is available at a program point P if the expression is computed along each path to P (from START) without getting invalidated.

An expression is very busy at a program point P if along each path from P (to END) the expression is computed without getting invalidated.

Outline

- Compilers
  - Block Diagram
- Program Analysis
  - Motivation
  - Control Flow Graph
  - Reaching Definitions Analysis
  - Live Variables Analysis
  - Analysis dimensions

Analysis Dimensions

An analysis's precision and efficiency is guided by various design decisions.

- Flow-sensitivity
- Context-sensitivity
- Path-sensitivity
- Field-sensitivity

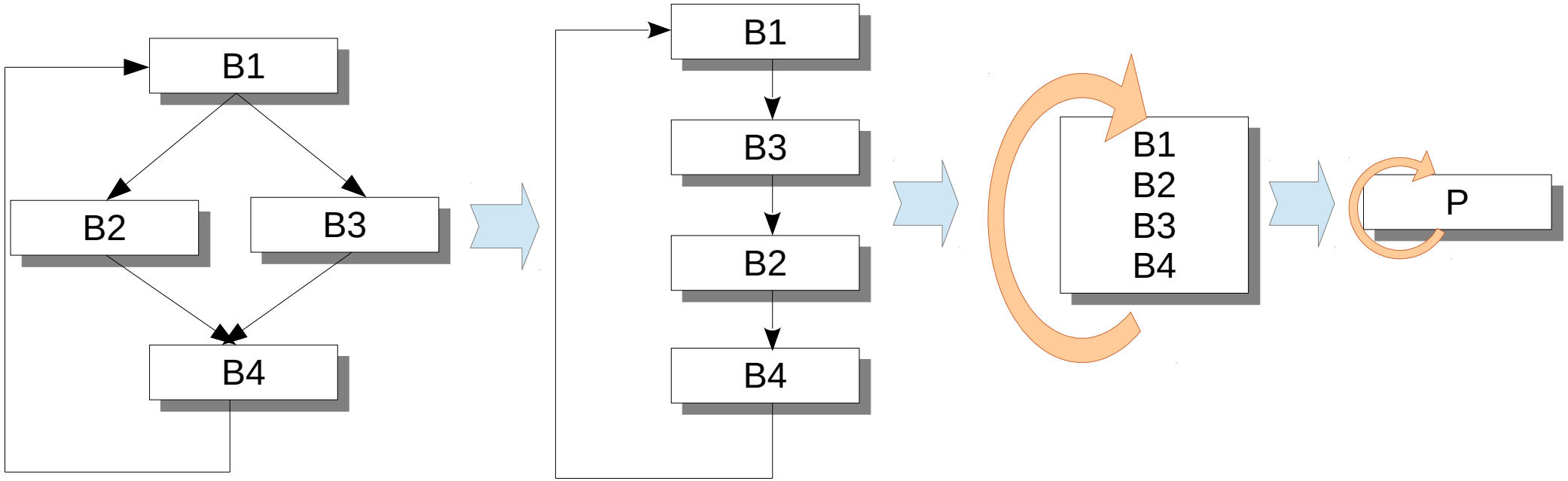
How many hands are required to know the time precisely?

Flow-sensitivity

```
L0:  a = 0;
L1:  a = 1;
L2:  ...
```

Flow-sensitive solution: *at L1 a is 0, at L2 a is 1*
Flow-insensitive solution: *in the program a is in {0, 1}*

Flow-insensitive analyses ignore the control-flow in the program.



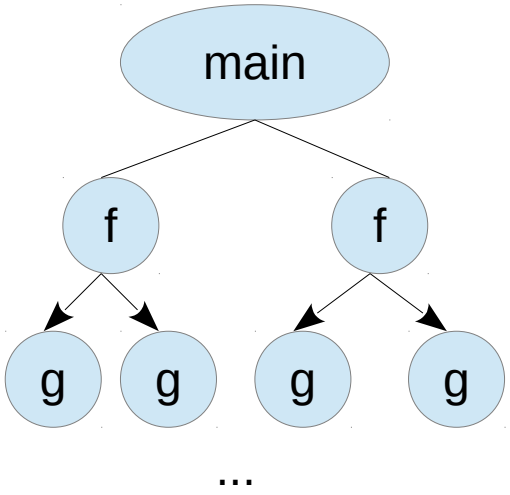
Context-sensitivity

```
main() {
      L0:  fun(0);
      L1:  fun(1);
}
```

```
fun(int x) {
      y = x;
}
```

Context-sensitive solution:
y is 0 along L0, y is 1 along L1

Context-insensitive solution:
y is in {0, 1} in the program



Exponential Number of contexts

Along main-f1-g1, …
Along main-f1-g2, …
Along main-f2-g1, …
Along main-f2-g2, ...

Exponential time requirement

Exponential storage requirement

Context-sensitivity

```
main() {
      L0:  fun(0);
      L1:  fun(1);
}
```

```
fun(int x) {
      y = x;
}
```

Context-sensitive solution:
y is 0 along L0, y is 1 along L1

Context-insensitive solution:

Inter-procedural → *y is in {0, 1} in the program*

intra-procedural → *y is in {-∞, +∞} in the program*

Path-sensitivity

```
if (a == 0)
        b = 1;
else
        b = 2;
```

Path-sensitive solution:
b is 1 when a is 0, b is 2 when a is not 0

Path-insensitive solution:
b is in {1, 2} in the program

```
if (c1)
   while (c2) {
      if (c3)
         …
      else
        for (; c4; )
           ...
    }
else
    ...
```

c1 and c2 and c3, …
c1 and c2 and !c3 and c4, …
c1 and c2 and !c3 and !c4, …
c1 and !c2, …
!c1 ...
...

Field-sensitivity

```
struct T s;

s.a = 0;
s.b = 1;
```

Field-sensitive solution:
s.a is 0, s.b is 1

Field-insensitive solution:
s is in {0, 1}

Aggregates are collapsed into a single variable.
e.g., arrays, structures, unions.

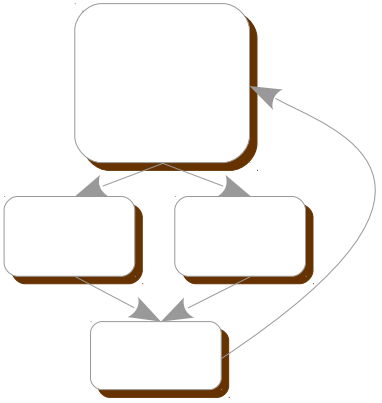
This reduces the number of variables tracked during the analysis and reduces precision.

Homework

- Find the values of variables in
  - context + flow-sensitive analysis
  - interprocedural context-insensitive but flow-sensitive analysis
  - intraprocedural flow-insensitive analysis

```
int g = 0;
void fun(int n) {
    g = n;
}
void main() {
    int a = 1;
    a = 2;
    fun(a);   // L1
    a = 3;
    fun(a);   // L2
}
```

Program Analysis



Rupesh Nasre.
IIT Madras

Raisoni
August 2020