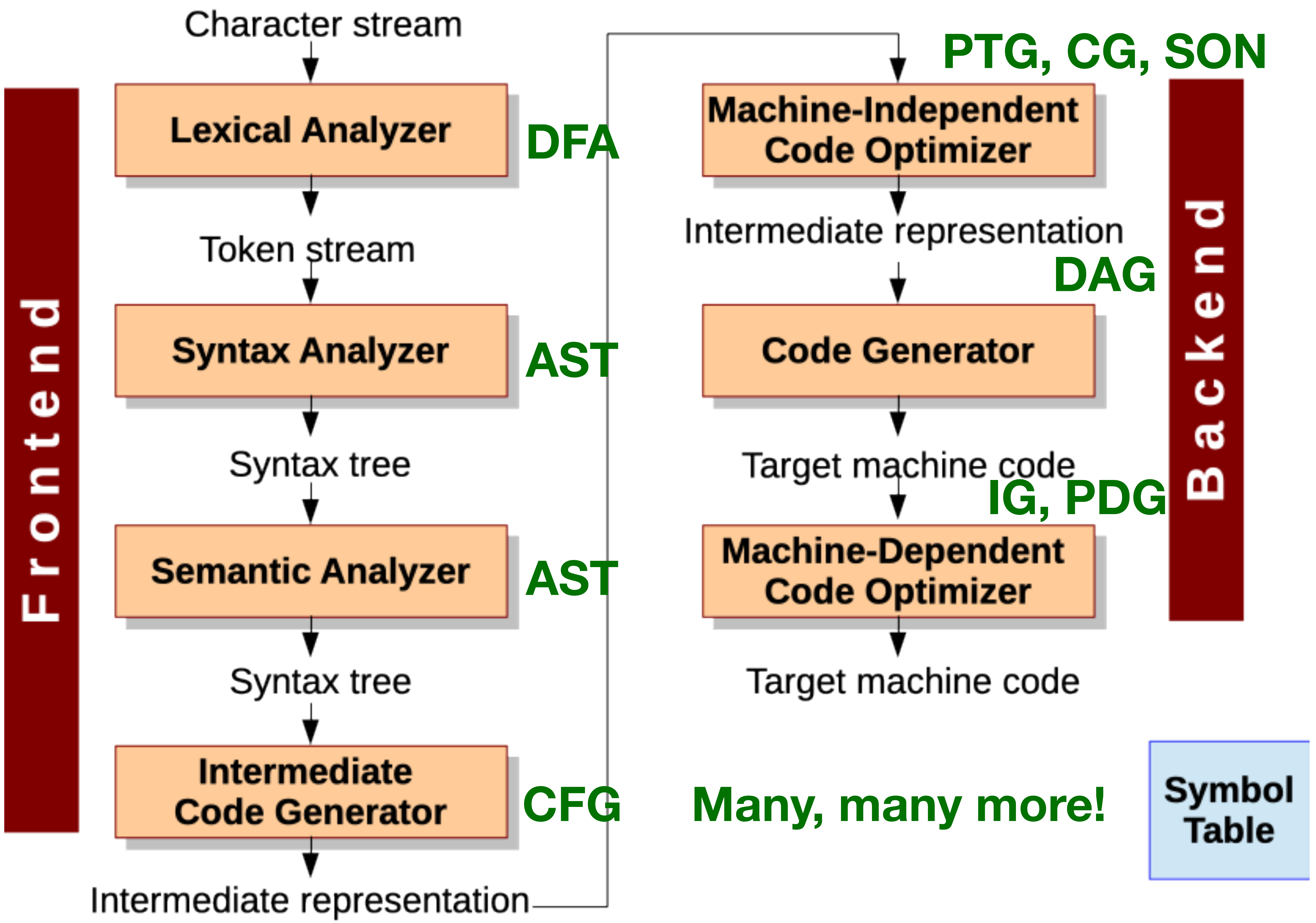
CS614: Advanced Compilers

Intro to Program Optimization

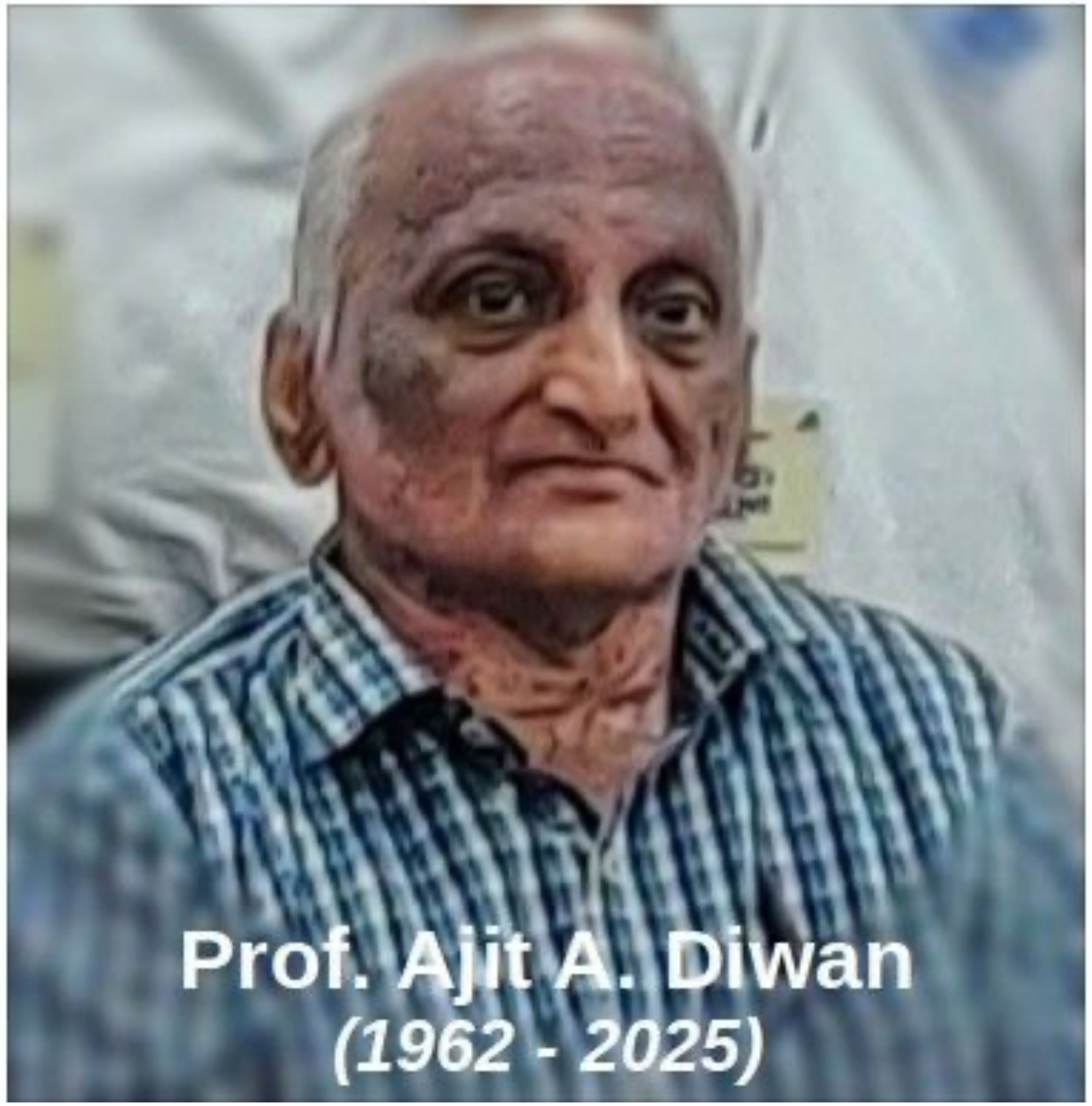
Manas Thakur
CSE, IIT Bombay

Spring 2025

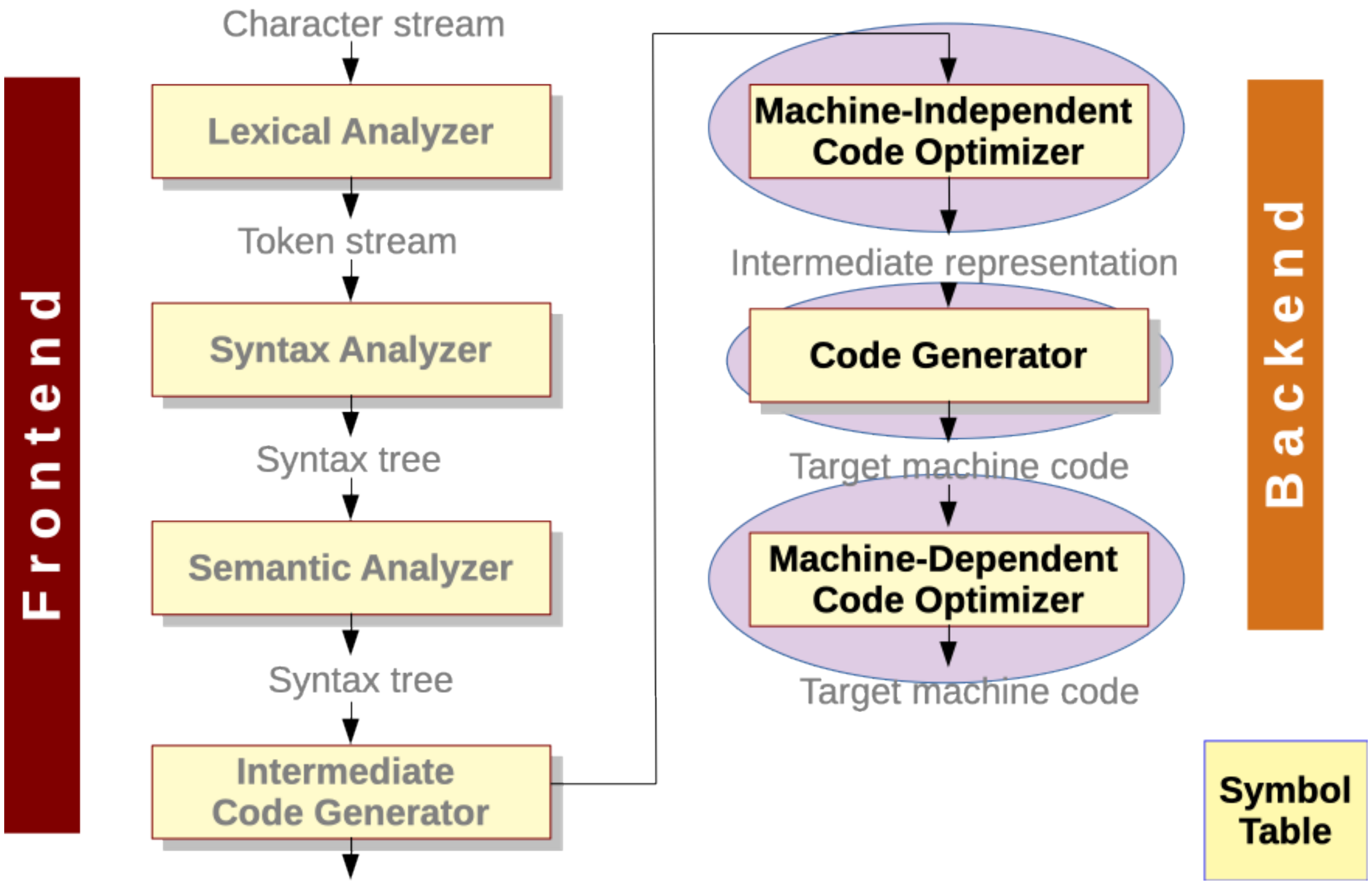
Graphs in Compilers



We will miss you profoundly…



Now we are moving to the back-end



Things we have already learnt

- **Lowering**
  - From language-level constructs to simple limited number of constructs
- At this point we could generate machine code!
  - Map from lower-level IR to actual ISA
  - Maybe some register management
  - Maybe some instruction selection
  - Pass on to assembler
  - Why not generate machine code directly?
    - *Kucch to raham karo janaab!*

But first…

- The compiler "understands" the program
    - IR captures program semantics
    - Lowering: semantics-preserving transformation
    - Why not do others?

- Compiler optimizations
    - Oh great, now my code will be optimal!
    - Sorry, it's a misnomer
    - What is an "optimization"?

Code Optimization: An Intro

- **Goal:** Generate optimized code
- **Metrics:**
  - Code size
    - Memory requirements
    - Say opcodes take 1 byte, each operand another byte
  - Number of registers
    - Along with constraints on their usage
  - Estimated cost
    - How fast is the code?
    - Say instructions with single operand take 1 cycle, with two operands take 2 cycles, and those involving memory take 4 cycles
  - Sometimes
    - power, energy, platform (in)dependence, ...

Code Optimization: An Intro

A simple one-to-one mapping:

```
1: a = 0
2: b = a + 1
3: c = c + b
4: a = b * 2
```

```
MOV  R1  0
STA  a   R1

MOV  R1  1
LDA  R2  a
ADD  R2  R1
STA  b   R2

LDA  R1  b
LDA  R2  c
ADD  R2  R1
STA  c   R2

MOV  R1  2
LDA  R2  b
MUL  R2  R1
STA  a   R2
```

Cost:
Registers: 2
Space: 42 bytes
Time: 44 cycles

Can we do better?

Code Optimization: An Intro

```
1: a = 0
2: b = a + 1
3: c = c + b
4: a = b * 2
```

Better register usage:

```
MOV R1 0
STA a  R1

MOV R1 1
LDA R2 a
ADD R2 R1
STA b  R2

LDA R1 b
LDA R2 c
ADD R2 R1
STA c  R2

MOV R1 2
LDA R2 b
MUL R2 R1
STA a  R2
```

Cost:
Registers: 2
Space: 42 bytes
Time: 44 cycles

→

```
MOV R1 0
STA a  R1

MOV R2 1
ADD R1 R2
STA b  R1

LDA R2 c
ADD R2 R1
STA c  R2

MOV R2 2
MUL R1 R2
STA a  R1
```

Cost:
Registers: 2
Space: 33 bytes
Time: 32 cycles

Can we do better?

Code Optimization: An Intro

```
1: a = 0
2: b = a + 1
3: c = c + b
4: a = b * 2
```

Remove redundant store to a:

```
MOV  R1  0
STA  a   R1

MOV  R2  1
ADD  R1  R2
STA  b   R1

LDA  R2  c
ADD  R2  R1
STA  c   R2

MOV  R2  2
MUL  R1  R2
STA  a   R1
```

Cost:
Registers: 2
Space: 33 bytes
Time: 32 cycles

→

```
MOV  R1  0

MOV  R2  1
ADD  R1  R2
STA  b   R1

LDA  R2  c
ADD  R2  R1
STA  c   R2

MOV  R2  2
MUL  R1  R2
STA  a   R1
```

Cost:
Registers: 2
Space: 30 bytes
Time: 28 cycles

Can we do better?

Code Optimization: An Intro

```
1: a = 0
2: b = a + 1
3: c = c + b
4: a = b * 2
```

Select specialized instructions:

```
MOV R1 0

MOV R2 1
ADD R1 R2
STA b  R1

LDA R2 c
ADD R2 R1
STA c  R2

MOV R2 2
MUL R1 R2
STA a  R1
```

Cost:
Registers: 2
Space: 30 bytes
Time: 28 cycles

→

```
CLR R1
INC R1
STA b  R1
LDA R2 c
ADD R2 R1
STA c  R2
SHL R1
STA a  R1
```

Cost:
Registers: 2
Space: 21 bytes
Time: 21 cycles

Can we do better?

Code Optimization: An Intro

```
1: a = 0
2: b = a + 1
3: c = c + b
4: a = b * 2
```

Propagate constant 0:

```
CLR R1
INC R1
STA b   R1
LDA R2  c
ADD R2  R1
STA c   R2
SHL R1
STA a   R1
```

Cost:
Registers: 2
Space: 21 bytes
Time: 21 cycles

→

```
MOV R1  1
STA b   R1
LDA R2  c
ADD R2  R1
STA c   R2
SHL R1
STA a   R1
```

Cost:
Registers: 2
Space: 20 bytes
Time: 21 cycles

Can we do better?

Code Optimization: An Intro

```
1: a = 0
2: b = a + 1
3: c = c + b
4: a = b * 2
```

Assuming b is not used in future,
propagate constant 1:

```
MOV  R1  1
STA  b   R1
LDA  R2  c
ADD  R2  R1
STA  c   R2
SHL  R1
STA  a   R1
```

Cost:
Registers: 2
Space: 20 bytes
Time: 21 cycles

→

```
LDA  R1  c
INC  R1
STA  c   R1
MOV  R1  2
STA  a   R1
```

Cost:
Registers: 1
Space: 14 bytes
Time: 15 cycles

Can we do better?

Code Optimization: An Intro

```
1: a = 0
2: b = a + 1
3: c = c + b
4: a = b * 2
```

Assuming the availability of a store-immediate instruction:

```
LDA  R1  c
INC  R1
STA  c   R1
MOV  R1  2
STA  a   R1
```

Cost:
Registers: 1
Space: 14 bytes
Time: 15 cycles

→

```
LDA  R1  c
INC  R1
STA  c   R1
STI  a   2
```

Cost:
Registers: 1
Space: 11 bytes
Time: 13 cycles

Assuming no other special instruction and no new knowledge about past or future instructions:

2 PCs if you can make it even better!

Hint (-1): Think about the execution at architectural level.

Code Optimization: An Intro

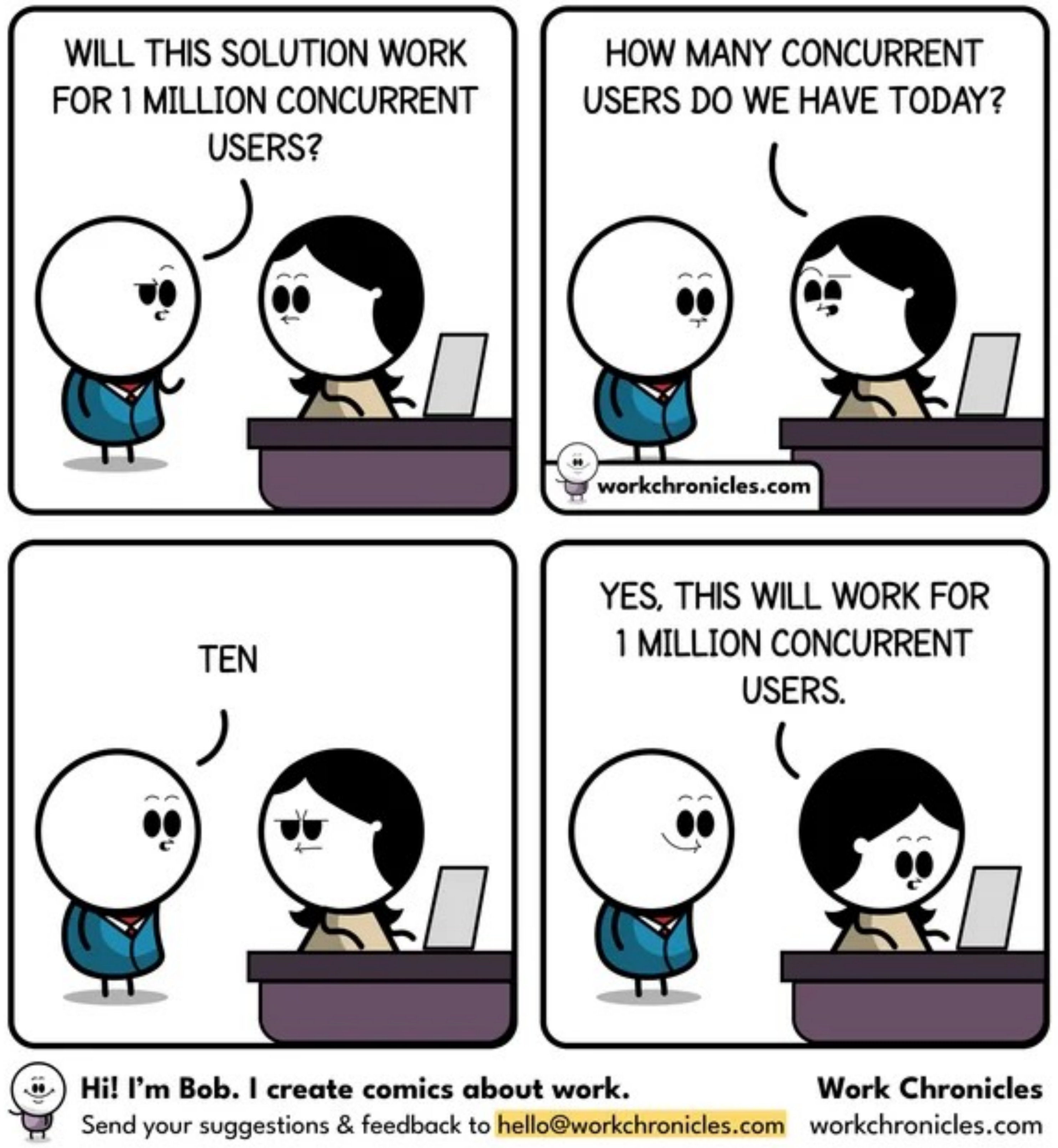
```
1: a = 0
2: b = a + 1
3: c = c + b
4: a = b * 2
```

Reordering stores to optimize memory accesses:

```
LDA R1 c
INC R1
STA c  R1
STI a  2
```

→

```
STI a  2
LDA R1 c
INC R1
STA c  R1
```



Foor for thought: Impact of multithreaded programs on possible/allowed reorderings.

This is where we are headed now!

```
MOV R1 0
STA a  R1

MOV R1 1
LDA R2 a
ADD R2 R1
STA b  R2

LDA R1 b
LDA R2 c
ADD R2 R1
STA c  R2

MOV R1 2
LDA R2 b
MUL R2 R1
STA a  R2
```

Cost:
Registers: 2
Space: 42 bytes
Time: 44 cycles

Code → Optimization

```
STI a  2
LDA R1 c
INC R1
STA c  R1
```

Cost:
Registers: 1
Space: 11 bytes
Time: 13 cycles
+Reordered

Full employment theorem for compiler writers

- **Statement:** There is no *fully optimizing* compiler.
- Assume it exists:
  - such that it transforms a program `P` to the smallest program `Opt(P)` that has the same behaviour as `P`.
  - Halting problem comes to the rescue:
    - Smallest program that never halts:

      ```
      L1: goto L1
      ```

    - Thus, a fully optimizing compiler could solve the halting problem by checking if a given program is `L1: goto L1`!
  - But HP is an undecidable problem.
  - Hence, a fully optimizing compiler can't exist!
- Therefore we talk just about an *optimizing compiler*,

  and keep working without worrying about future prospects!

How to perform optimizations?

- Analysis
  - Go over the program
  - Identify some properties
    - Potentially useful properties
- Transformation
  - Use the information computed by the analysis to transform the program
    - without affecting the semantics
- **Example:**
  - Compute liveness information
  - Delete assignments to variables that are dead

Many many optimizations

- *Constant folding, constant propagation, tail-call elimination, redundancy elimination, dead-code elimination, loop-invariant code motion, loop splitting, loop fusion, strength reduction, inlining, scalarization, synchronization elision, cloning, data prefetching, parallelization . . . etc.*

- How do they interact?
  - Optimist: we get the sum of all improvements.
  - Realist: many are in direct opposition.

- Let us *study some* of them!

Constant propagation

- **Idea:** If the value of a variable is known to be a constant at compile-time, replace the use of the variable with the constant.

```
n = 10;
c = 2;
for (i=0; i<n; ++i)
   s = s + i * c;
```

➡

```
n = 10;
c = 2;
for (i=0; i<10; ++i)
   s = s + i * 2;
```

- Usually a very helpful optimization
  - e.g., Can we now *unroll* the loop?
    - Why is it good?
    - Why could it be bad?
- When can we eliminate `n` and `c` themselves?
- Now you know how well different optimizations might interact.

Constant folding

- **Idea:** If operands are known at compile-time, evaluate expression at compile-time.

```
r = 3.141 * 10;
```

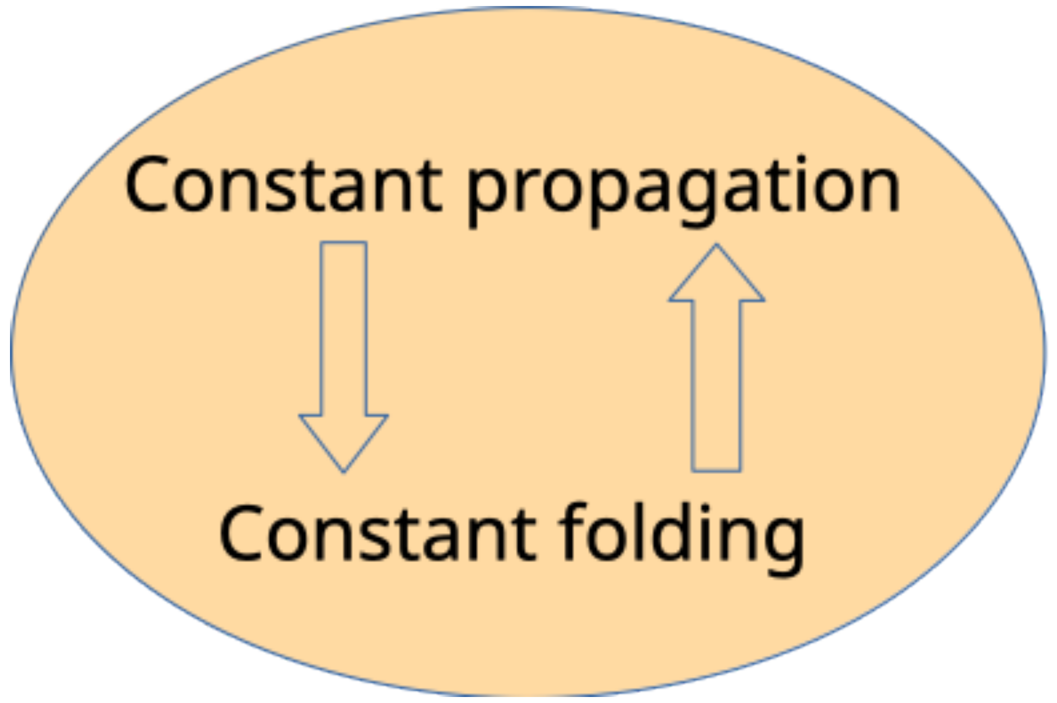
```
r = 31.41;
```

- What if the code was?

```
PI = 3.141;
r = PI * 10;
```

- And what now?

```
PI = 3.141;
r = PI * 10;
d = 2 * r;
```



Called **partial evaluation**

Common sub-expression elimination

- **Idea:** If a program computes the same value multiple times, reuse the value.

```
a = b + c;
c = b + c;
d = b + c;
```

```
t = b + c;
a = t;
c = t;
d = b + c;
```

- Subexpressions can be reused until operands are redefined.

Copy propagation

- **Idea:** After an assignment `x = y`, replace the uses of `x` with `y`.

```
x = y;
if (x > 1)
   s = x + f(x);
```

➡

```
x = y;
if (y > 1)
   s = y + f(y);
```

- Can only apply up to another assignment to `x`, or

  ... another assignment to `y`!

- What if there was an assignment `y = z` earlier?
  - Apply transitively to all assignments.

Dead-code elimination

- **Idea:** If the result of a computation is never used, remove the computation.

```
x = y + 1;
y = 1;
x = 2 * z;
```

➡

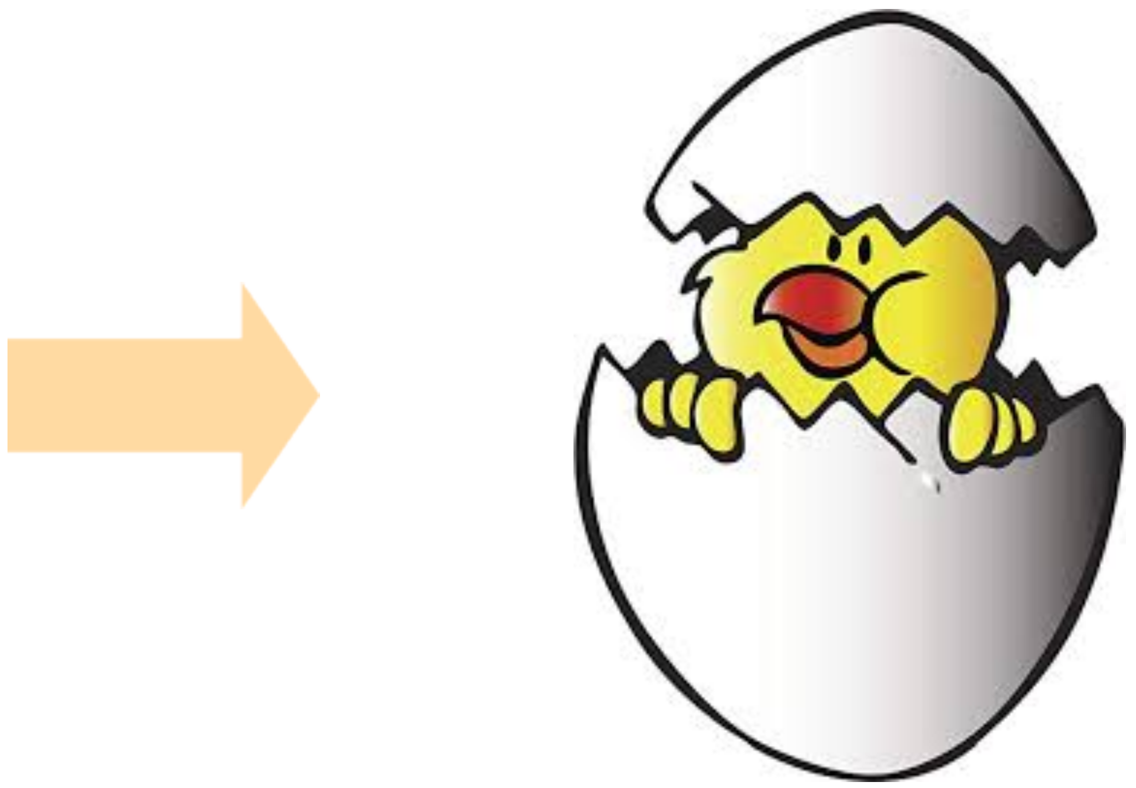
```
y = 1;
x = 2 * z;
```

- Remove code that assigns to dead variables.
  - Liveness analysis done before would help!
- This may, in turn, create more dead code.
  - Dead-code elimination usually works transitively.

Unreachable-code elimination

- **Idea:** Eliminate code that can never be executed

```
#define DEBUG 0
if (DEBUG)
   print("Current value = ", v);
```



- High-level: look for if (false) or while (false)
  - perhaps after constant propagation!
- Low-level: more difficult
  - Code is just labels and gotos
  - Traverse the program (as per its flow), marking reachable statements

Observations

- Some optimizations can be done again after doing them once.
- Some optimizations may get enabled after performing other optimizations.
- Almost all optimizations require information from some pre-performed analysis.
- Many analyses require information connecting variable uses to their definitions.
- Compilers require information from many such "dataflow analyses" to optimize code *soundly* and *precisely*.
  - Improvements in the underlying analyses may significantly impact other compilation passes.
- One of the important program representations that simplifies many tasks of a compiler is the SSA form.