CS251: Introduction to Language Processing

Code Generation and Optimizations

Vishwesh Jatala

Assistant Professor

Department of EECS

Indian Institute of Technology Bhilai

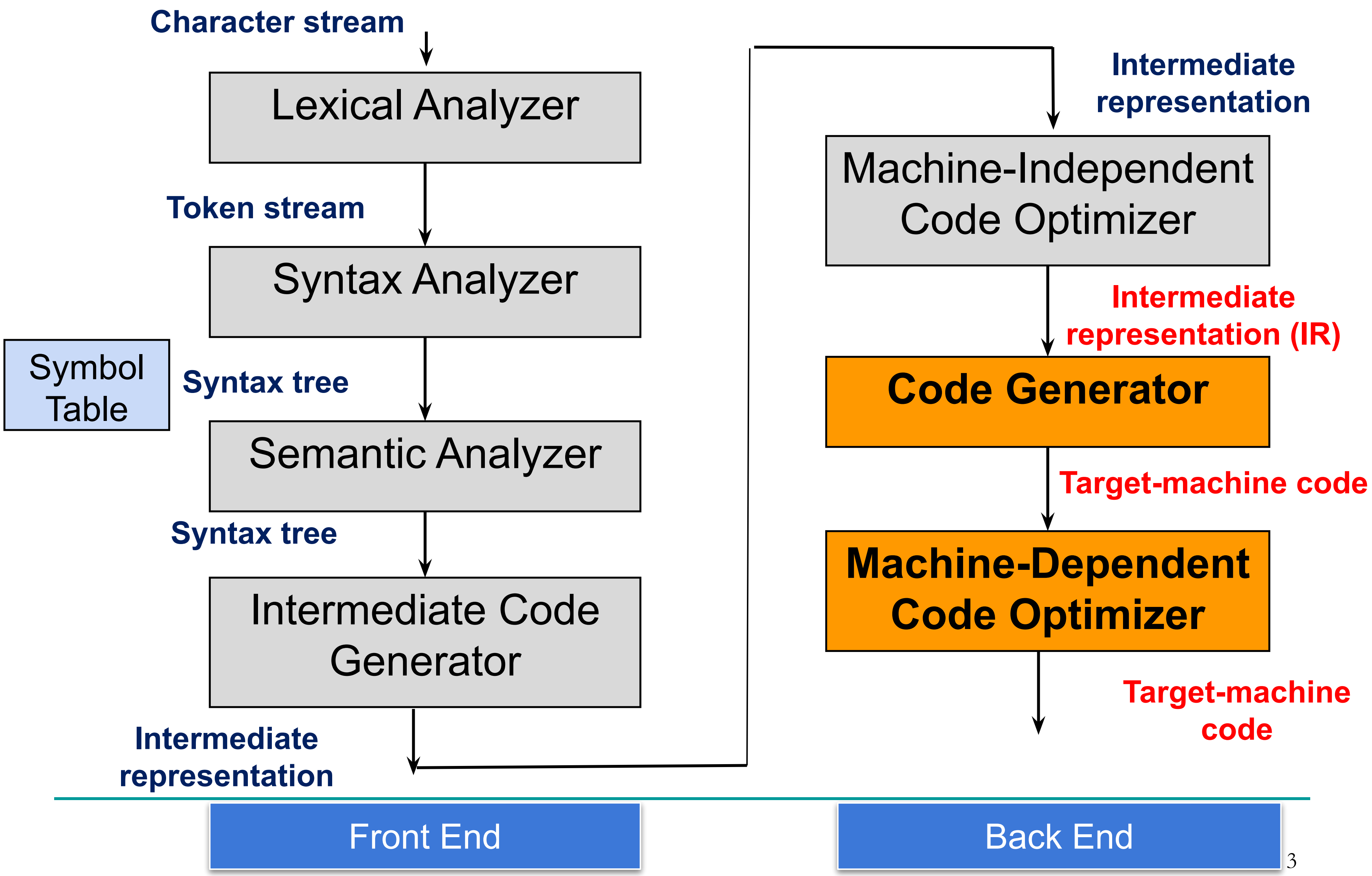
vishwesh@iitbhilai.ac.in

2020-21 Sem-2

Acknowledgement

- References for today's slides
  - *Prof. Y. N Srikant, IISc Bangalore*
    - *https://nptel.ac.in/content/storage2/courses/106108052/module4/code-gen-part-3.pdf*
  - *Course textbook*

Compiler Design


Character stream
Lexical Analyzer
Token stream
Syntax Analyzer
Syntax tree
Symbol Table
Semantic Analyzer
Syntax tree
Intermediate Code Generator
Intermediate representation
Intermediate representation
Machine-Independent Code Optimizer
Intermediate representation (IR)
Code Generator
Target-machine code
Machine-Dependent Code Optimizer
Target-machine code
Front End
Back End

Outline

- Introduction to code generation
- Challenges in code generation
- Code generation algorithm
  - Example
- Machine dependent optimizations
  - Peephole optimizations

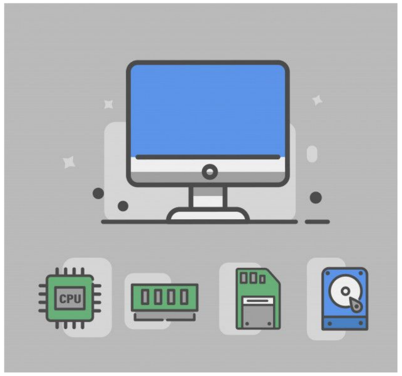
Code Generation

- *Goal: Map IR program into a code that can be executed on a target machine*
- Input: Symbol table, IR (three-address code (TAC))
- Output: Target machine code
- Assumptions:
  - No syntactic and semantic errors
  - Type checking has been done

Desired Characteristics

Correct: Preserve semantics

Fast: Less execution time



Resource efficient

Energy efficient

Reality

- Generating the optimal code is undecidable
- Several sub-problems are NP-Complete
  - Register allocation
  - Evaluation order
- Practical approaches use heuristics

Challenges in Code Generation

- Target architecture
- Instruction selection
- Register allocation
- Evaluation order

Challenges: Target Architecture

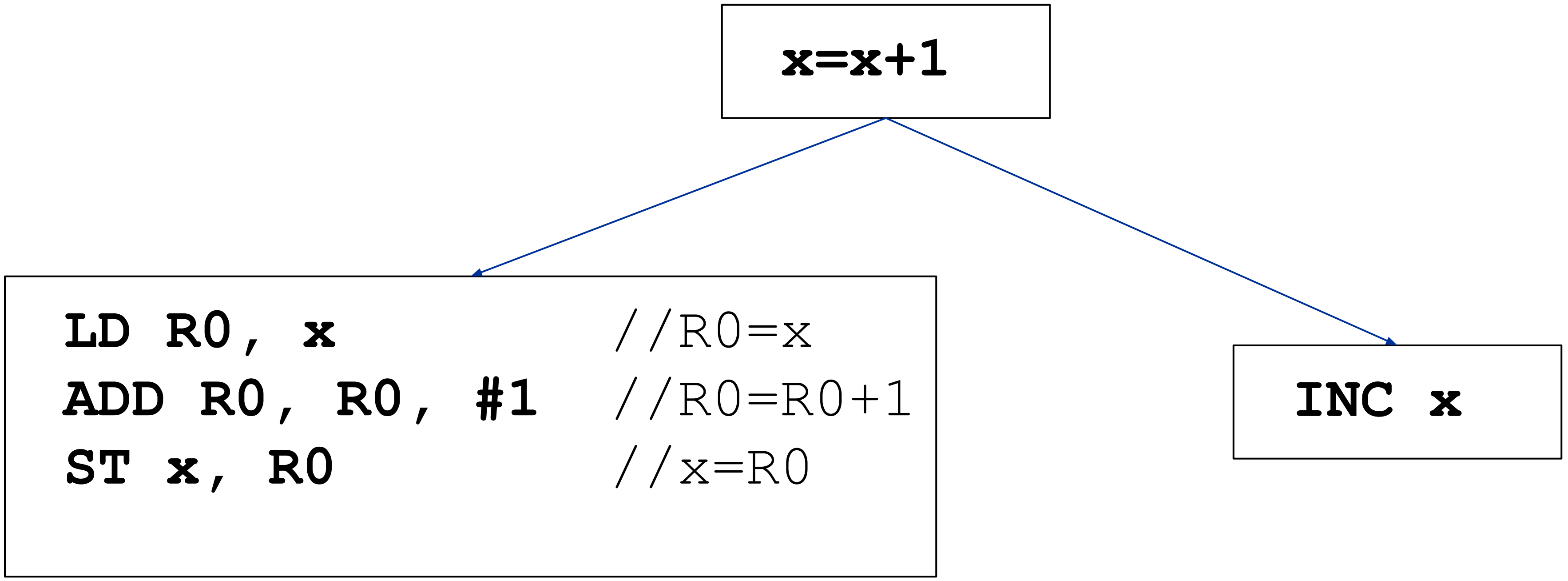
- Instruction set architecture has impact on difficulty of code generation
  - Common architectures: RISC and CISC

RISC	CISC
Many Registers	Few Registers
Simple-addressing modes and ISA	Variety of addressing modes,variable length instructions

We will use RISC in the lecture

Challenges: Instruction Selection

- Instruction selection is critical for performance



```
x=x+1
```

```
LD R0, x          //R0=x
ADD R0, R0, #1    //R0=R0+1
ST x, R0          //x=R0
```

```
INC x
```

Challenges: Instruction Selection

- Naive implementation can lead to redundant loads/stores

```
x=y+z
```

$\Rightarrow$

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

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

$\Rightarrow$

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

Challenges: Register Allocation

- Registers are fastest storage units
- Registers are few
- Efficient register utilization is critical

```
a := c + d
e := a + b
f := e - 1
```

a, **e**, and **f** can be allocated in 1 register!

Optimal register allocation is NP-Complete

Challenges: Evaluation Order

- Order in which computations are performed can affect of performance
- Some computation orders require few registers
- Picking the best order is NP-Complete

Target Language

- Assume a simple target assembly code
- Instructions:
  - Load: *LD dst, addr*
  - Store: *ST x, reg*
  - Computation operations: *OP dst, src1, src2*
    - OP can be ADD, SUB, MUL, etc.
  - Unconditional jumps: *BR L*
  - Conditional jumps: *BCond reg, L*

Exercise-1

- Manually generate the code for the following TAC according to the defined target architecture
  - Assume there is no limit on the number of registers.

```
x=y+z
```

Exercise

- Manually generate the code for the following TAC according to the defined target architecture
  - Assume there is no limit on the number of registers.

```
x=y+z
```

$\Rightarrow$

```
LD R1, y
LD R2, z
ADD R1, R1, R2
ST x, R1
```

Exercise-2

- Manually generate the code for the following TAC according to the defined target architecture
    - Assume there is no limit on the number of registers.

```
t = a-b
u = a-c
v = t+u
d = v+u
```

Peephole Optimizations

Peephole Optimizations

- Simple but effective local optimization
- Usually carried out on machine code, but intermediate code can also benefit from it
- Examines a sliding window of code (peephole), and replaces it by a shorter or faster sequence, if possible
- Each improvement provides opportunities for additional improvements
- Therefore, repeated passes over code are needed

Peephole Optimizations

- Some well known peephole optimizations
  - eliminating redundant instructions
  - eliminating unreachable code
  - eliminating jumps over jumps
  - algebraic simplifications
  - strength reduction
  - use of machine idioms

Elimination of Redundant Loads and Stores

Basic block B

```
Load X, R0
{no modifications
to R0 or X here}
Store R0, X
```

Store instruction
can be deleted

Basic block B

```
Load X, R0
{no modifications
to X or R0 here}
Load X, R0
```

Second Load instr
can be deleted

Basic block B

```
Store R0, X
{no modifications
to X or R0 here}
Load X, R0
```

Load instruction
can be deleted

Basic block B

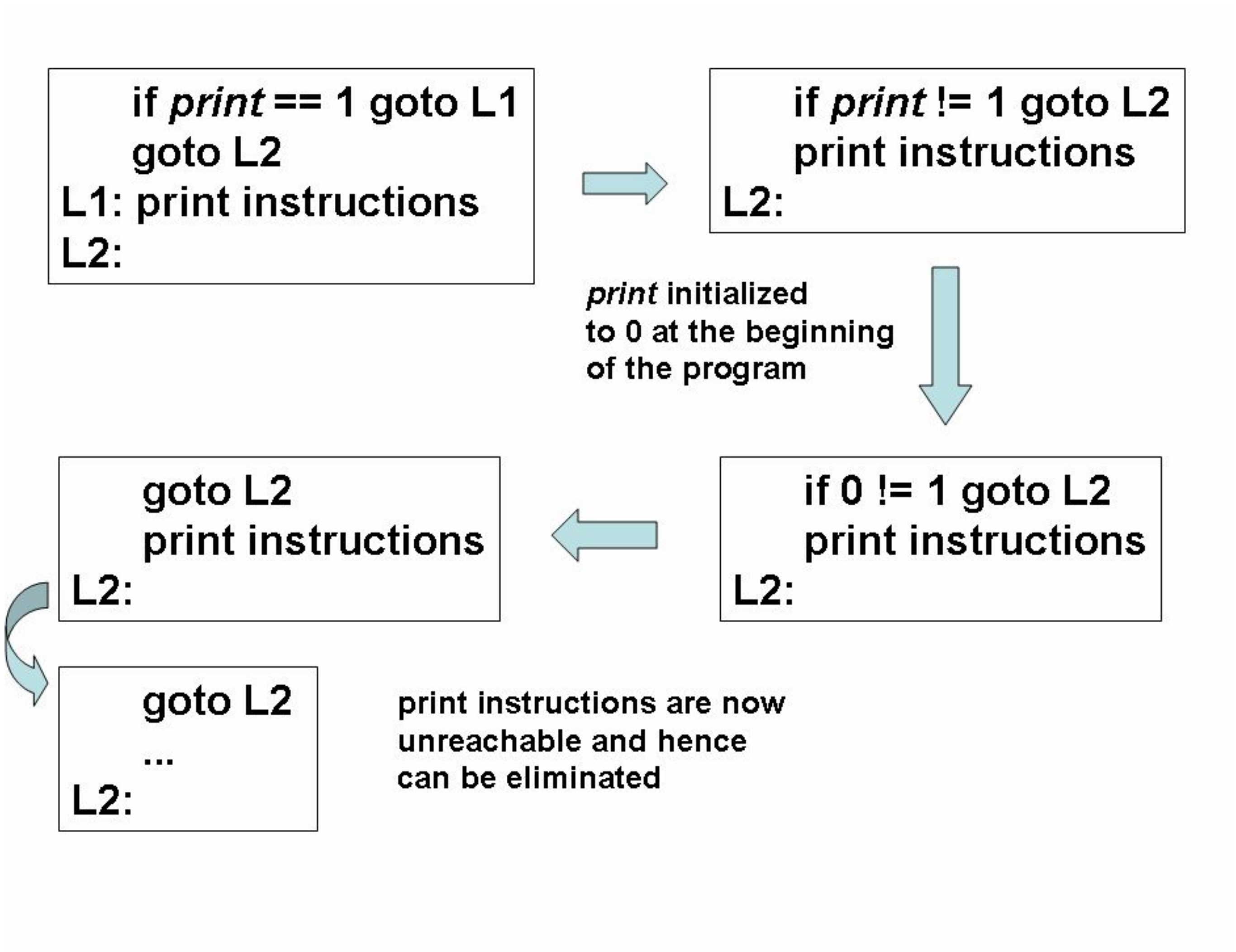
```
Store R0, X
{no modifications
to X or R0 here}
Store R0, X
```

Second Store instr
can be deleted

Eliminating Unreachable Code

- An unlabeled instruction immediately following an unconditional jump may be removed
  - May be produced due to debugging code introduced during development

Eliminating Unreachable Code



```
    if print == 1 goto L1
    goto L2
L1: print instructions
L2:
```

```
    if print != 1 goto L2
    print instructions
L2:
```

print initialized to 0 at the beginning of the program

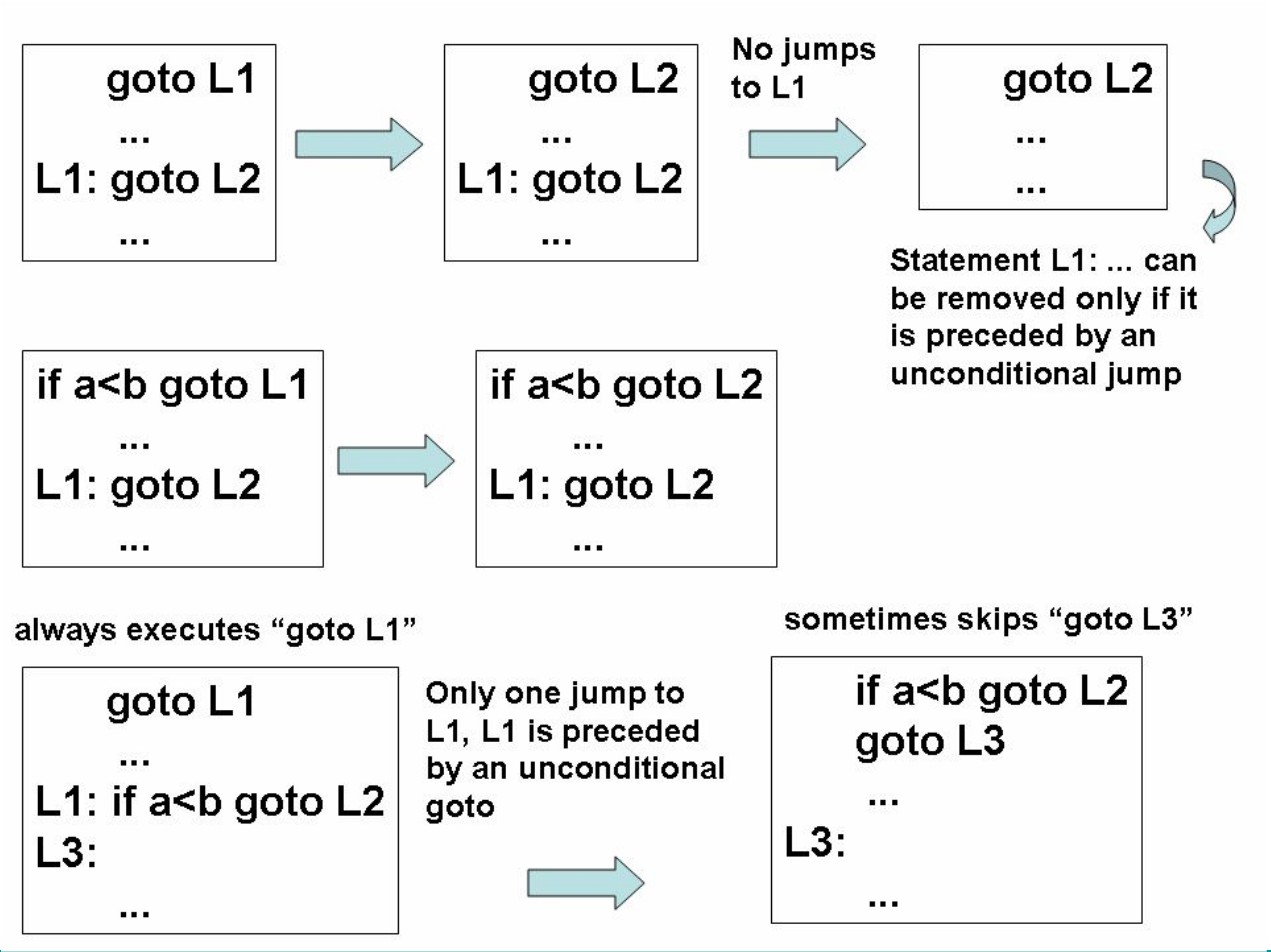
```
    if 0 != 1 goto L2
    print instructions
L2:
```

```
    goto L2
    print instructions
L2:
```

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

print instructions are now unreachable and hence can be eliminated

Flow-of-Control Optimizations



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

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

No jumps to L1

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

Statement L1: ... can be removed only if it is preceded by an unconditional jump

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

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

always executes “goto L1”

```
    goto L1
    ...
L1: if a<b goto L2
L3:
    ...
```

Only one jump to L1, L1 is preceded by an unconditional goto

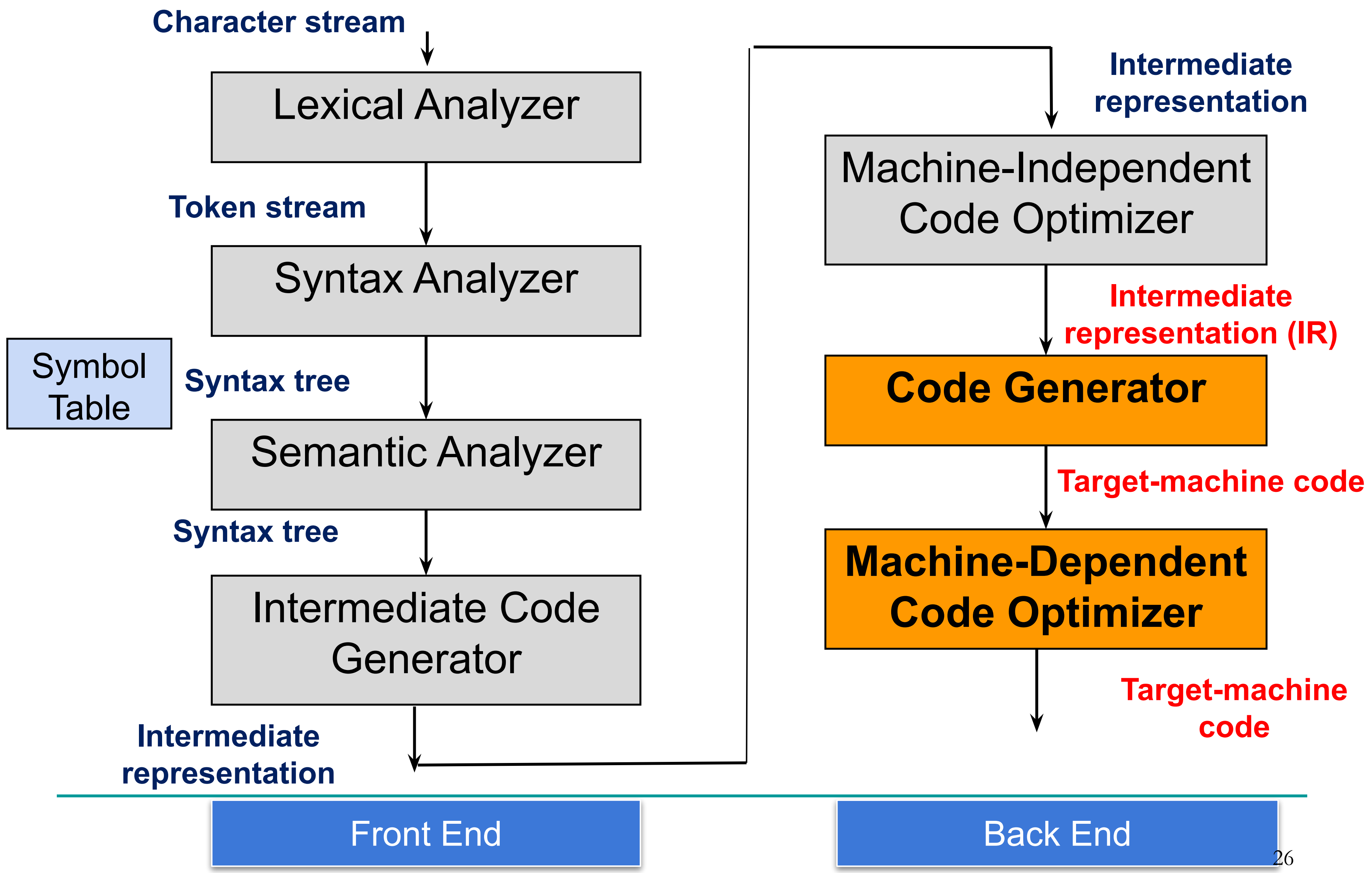
sometimes skips “goto L3”

```
    if a<b goto L2
    goto L3
    ...
L3:
    ...
```

Reduction in Strength and Use of Machine Idioms

- x^2 is cheaper to implement as x*x, than as a call to an exponentiation routine
- For integers, $x*2^3$ is cheaper to implement as x << 3 (x left-shifted by 3 bits)
- For integers, $x/2^2$ is cheaper to implement as x >> 2 (x right-shifted by 2 bits)

Compiler Design



Character stream
Lexical Analyzer
Token stream
Syntax Analyzer
Symbol Table
Syntax tree
Semantic Analyzer
Syntax tree
Intermediate Code Generator
Intermediate representation
Intermediate representation
Machine-Independent Code Optimizer
Intermediate representation (IR)
Code Generator
Target-machine code
Machine-Dependent Code Optimizer
Target-machine code
Front End
Back End