Theory of Compilation

Lecture 01 - Introduction

Ohad Shacham

Who?

Shachar Itzhaky

shachari@cs.technion.ac.il

Office hours should be scheduled in advanced

https://www.dropbox.com/sh/tdxc3fjavncl57p/AABXCydWo0spXNCBtuezgXjoa?dl=0

TAs:

- Hila Peleg
- Michal Badian
- Avner Elizarov

What?

- Understand:
  - What a compiler is
  - How it works
  - Proven techniques
    (most can be re-used in other settings)

How?

How?

- What will help us
  - Textbooks
    - Modern compiler design
    - Compilers: principles, techniques and tools
  - Homework assignments
    - “Dry”: deepen understanding of theory
    - “Wet”: build a compiler yourself
  - Ask questions

Exam

- 75% of the final grade
- Look at Eran's old exams from previous years
- Don't worry too much
  - If you attend lectures, you should do well in the exam, if you don't attend try to keep up with the material…
  - historical evidence – attending leads to 11pt higher grade on average

What is a Compiler?

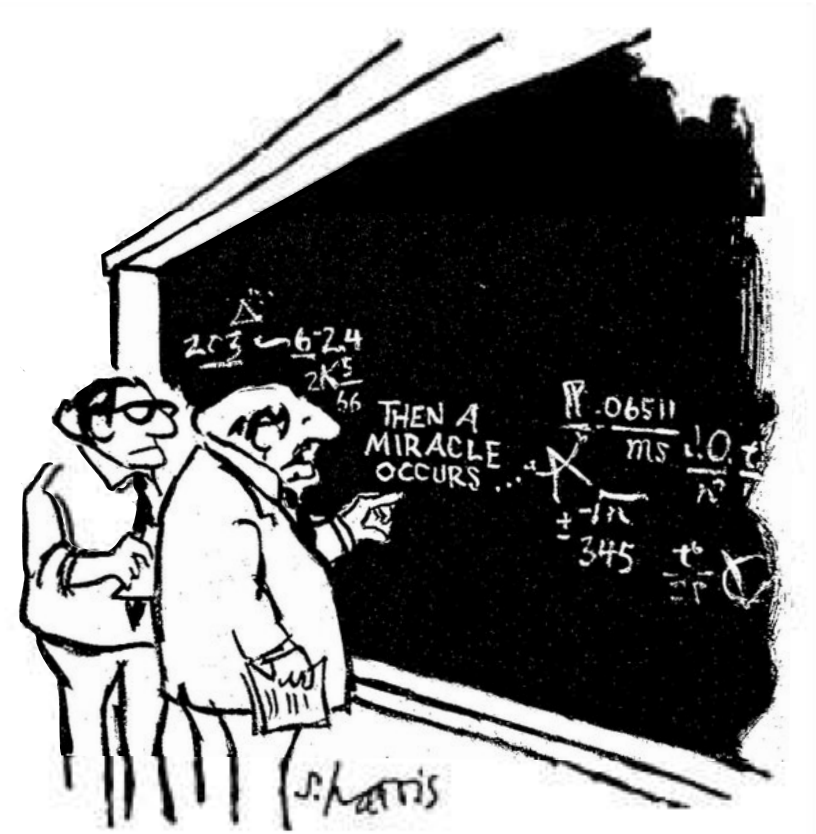
- "A compiler is a **computer program** that **transforms** source code written in a programming language (**source language**) into another language (**target language**). The most common reason for wanting to transform source code is to create an **executable program**."

-- *Wikipedia*

What is a Compiler?

source language

target language



What is a Compiler?

source language

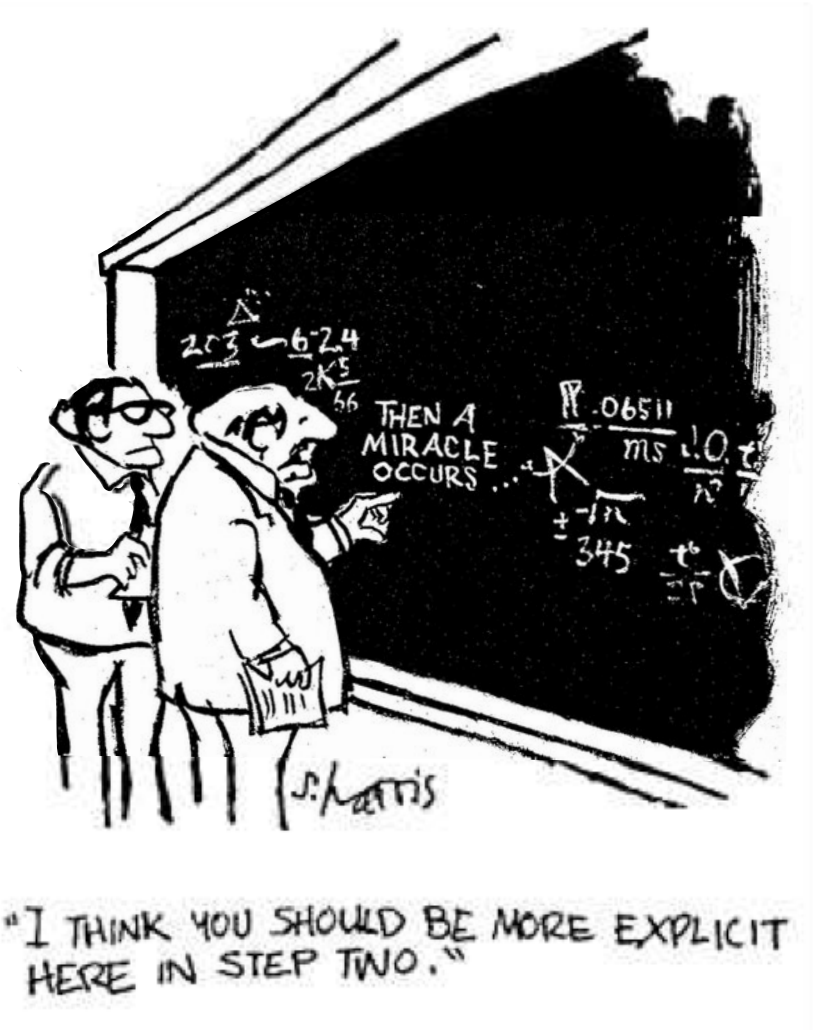
C
C++
Pascal
Java

Perl
JavaScript
Python
Ruby

Prolog

Lisp
Scheme
ML
OCaml

Postscript
TeX



target language

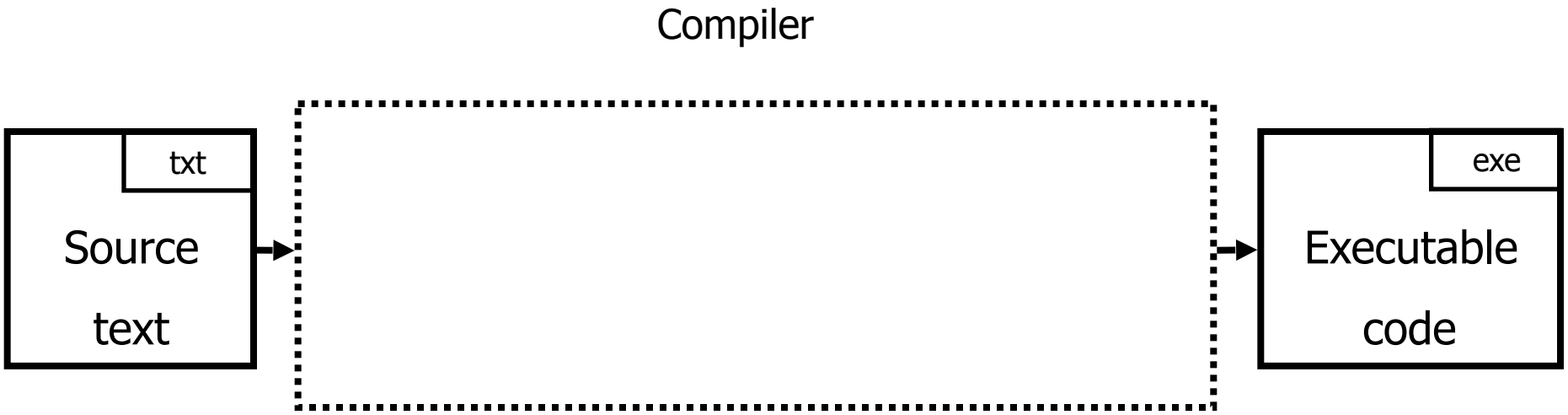
IA32
IA64
ARM
SPARC

C
C++
Pascal
Java

Java Bytecode

PDF
Bitmap
…

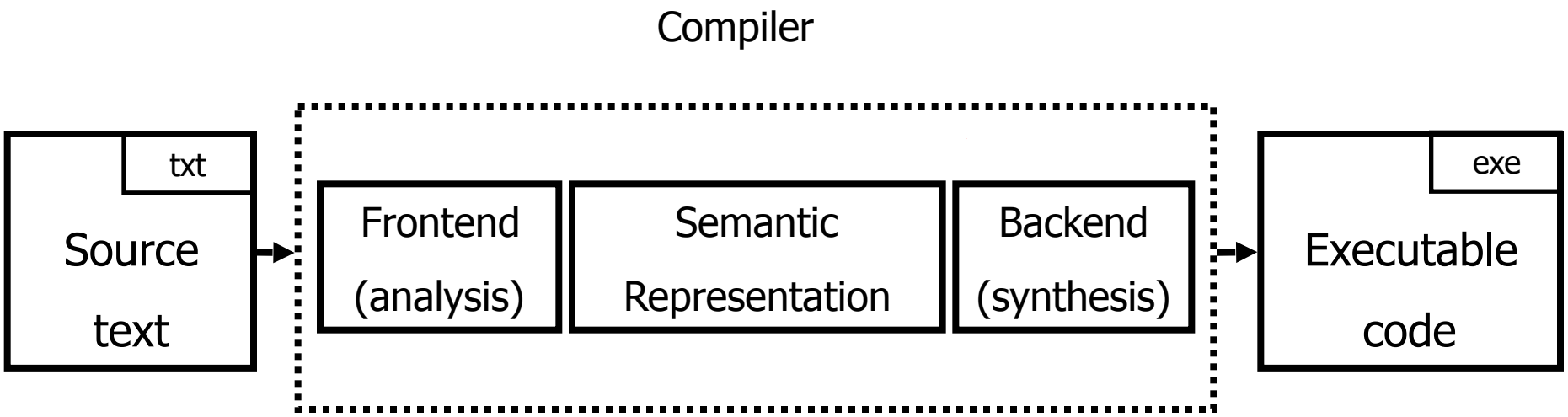
What is a Compiler?




```
int a, b;
a = 2;
b = a*2 + 1;
```

```
MOV R1,2
SAL R1
INC R1
MOV R2,R1
```

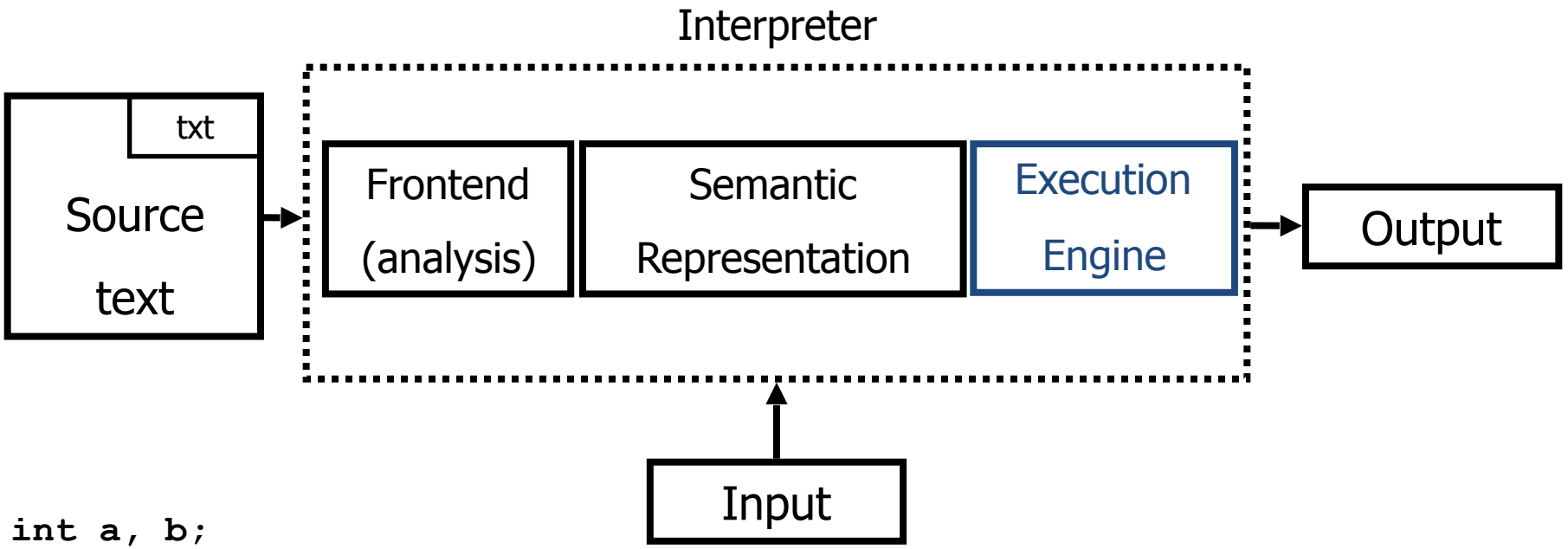
Anatomy of a Compiler




```
int a, b;
a = 2;
b = a*2 + 1;
```

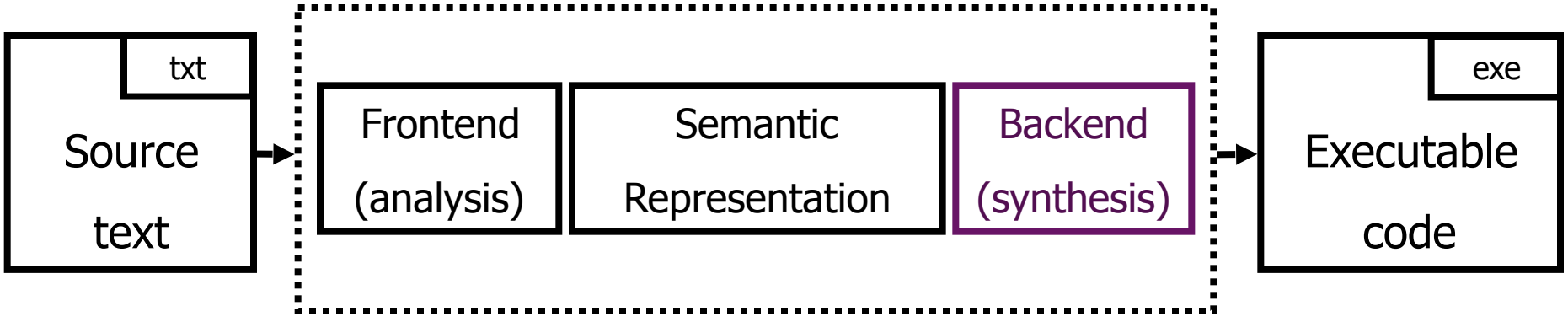
```
MOV R1,2
SAL R1
INC R1
MOV R2,R1
```

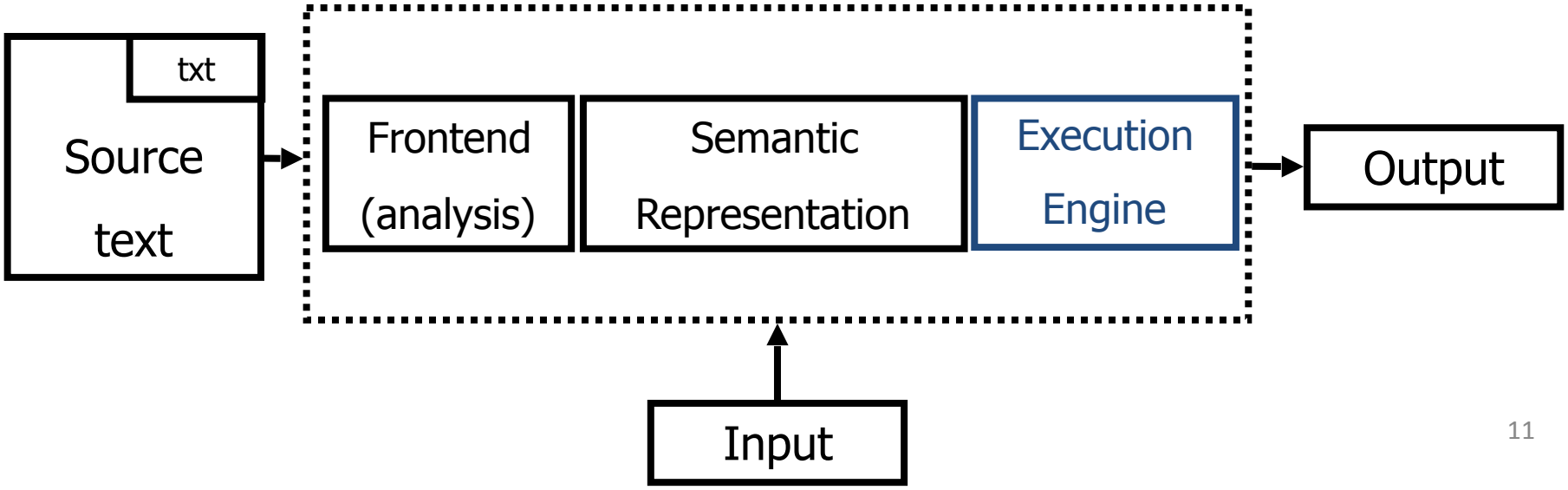
Interpreter



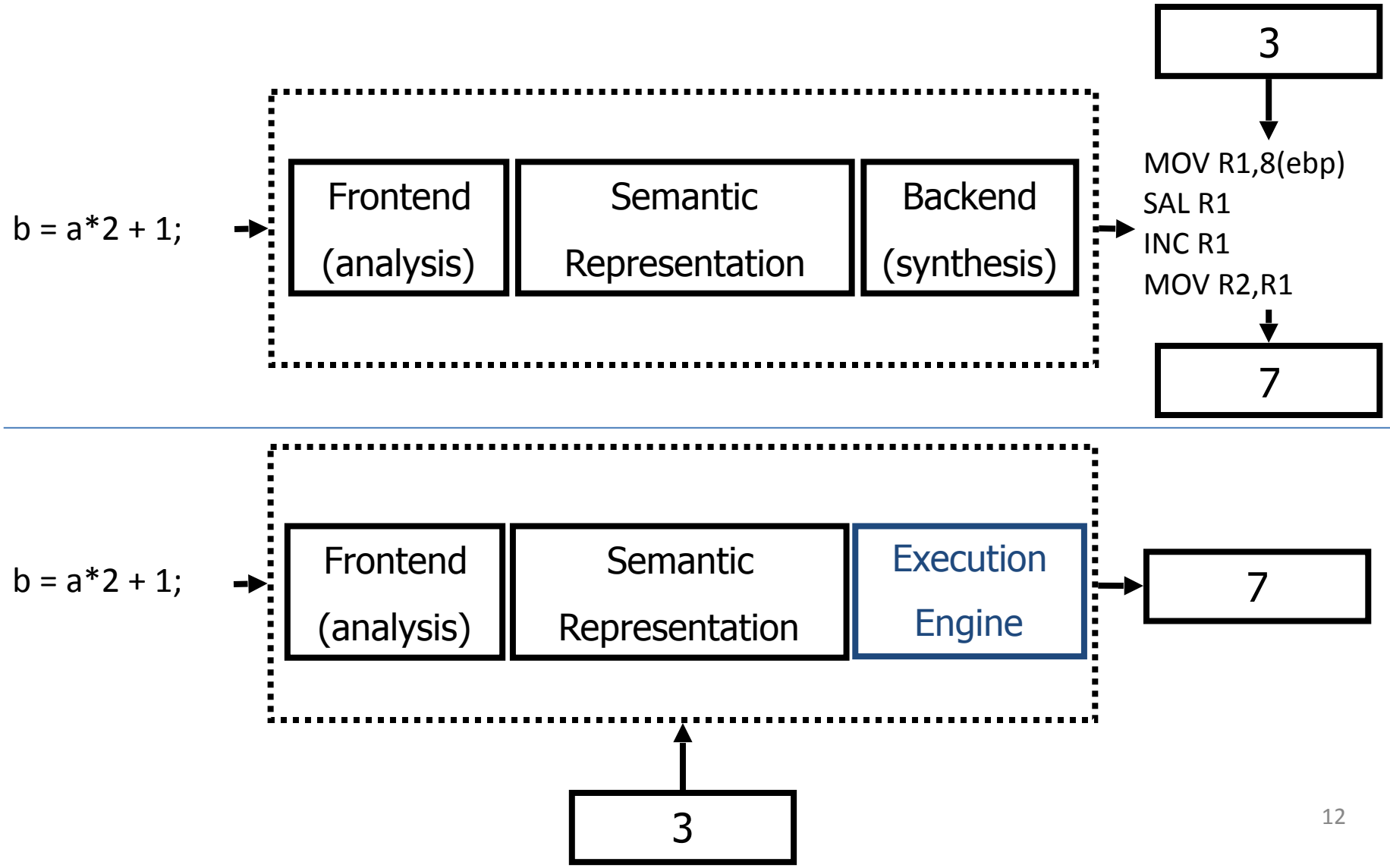

```
int a, b;
a = 2;
b = a*2 + 1;
```

Compiler vs. Interpreter



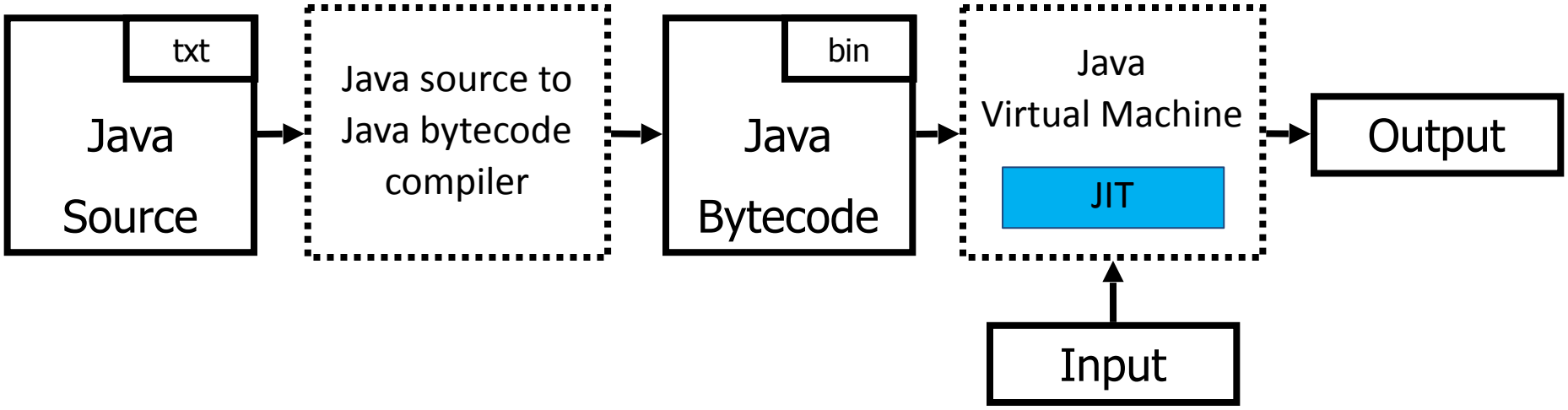


Compiler vs. Interpreter



Just-in-time Compiler

(Java example)



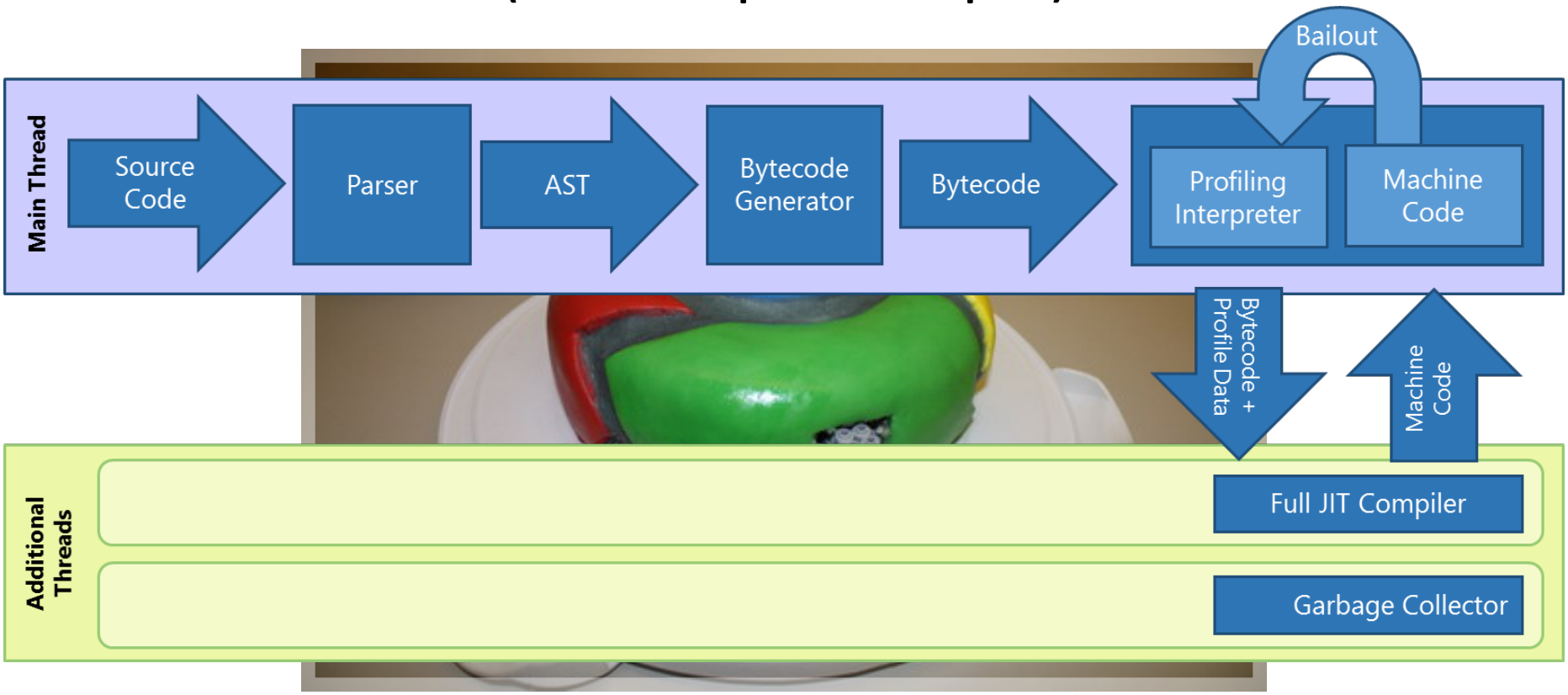
Just-in-time (JIT) compilation: bytecode interpreter (in the JVM) compiles program fragments during interpretation to avoid expensive re-interpretation.

Just-in-time Compiler

(Javascript example)

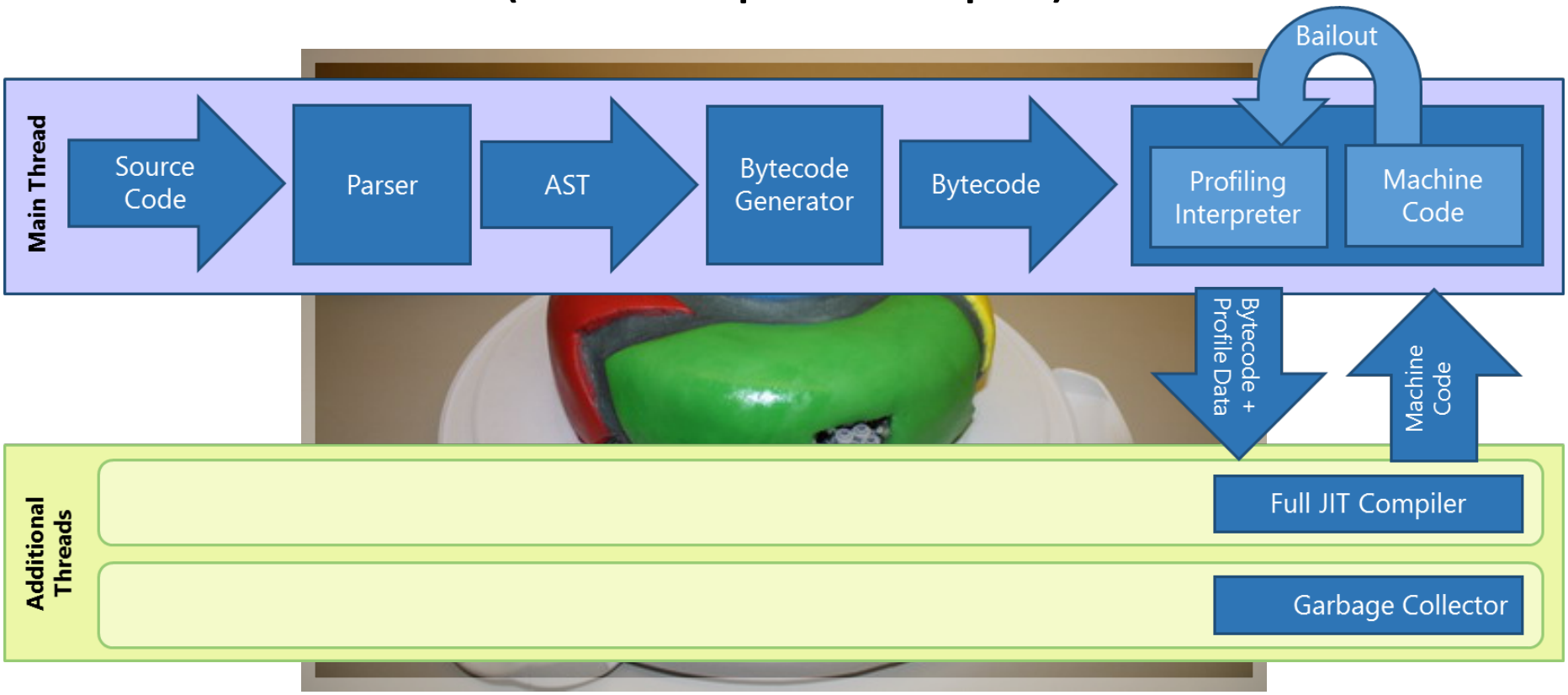
Just-in-time Compiler

(Javascript example)



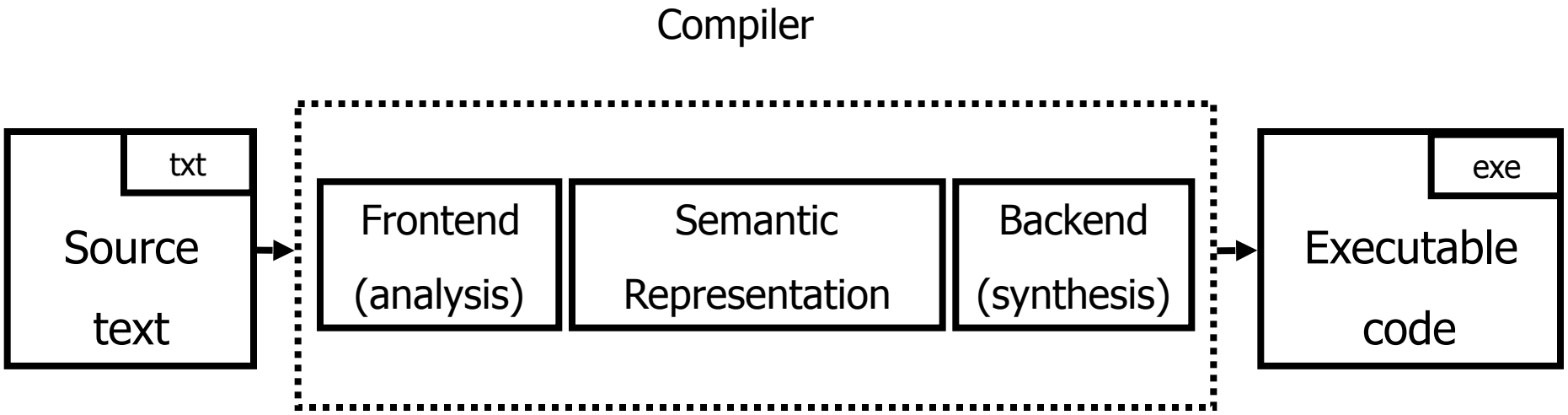
Just-in-time Compiler

(Javascript example)



- The compiled code is optimized dynamically at runtime, based on runtime behavior

Anatomy of a Compiler: Why?

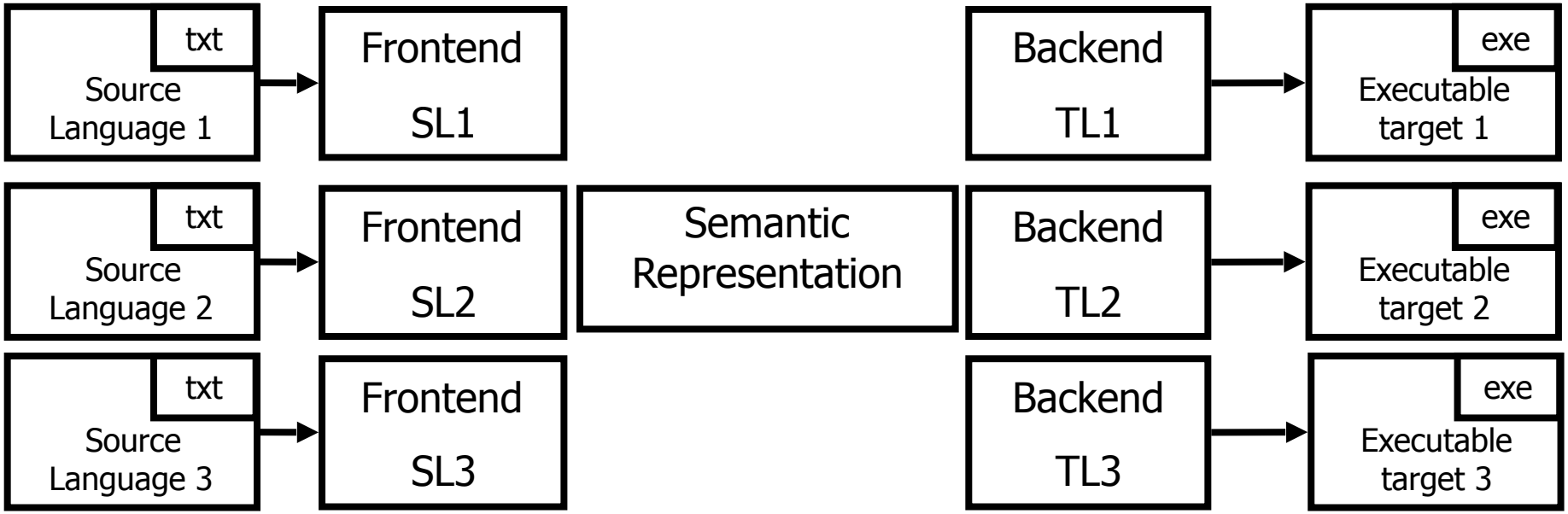



```
int a, b;
a = 2;
b = a*2 + 1;
```

```
MOV R1,2
SAL R1
INC R1
MOV R2,R1
```

Modularity

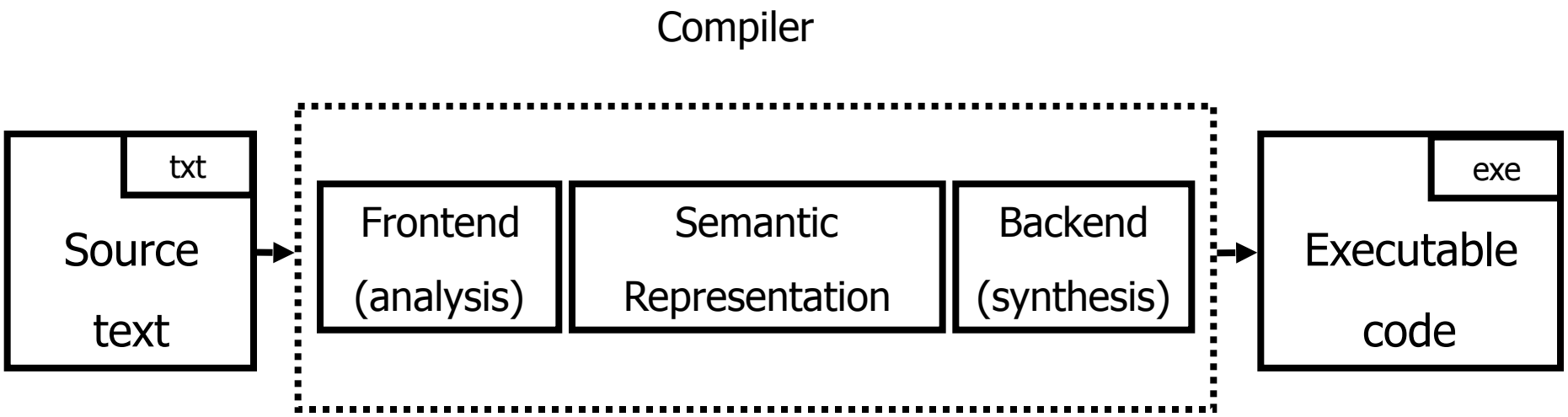
```
SET   R1,2
STORE #0,R1
SHIFT R1,1
STORE #1,R1
ADD   R1,1
STORE #2,R1
```




```
int a, b;
a = 2;
b = a*2 + 1;
```

```
MOV R1,2
SAL R1
INC R1
MOV R2,R1
```

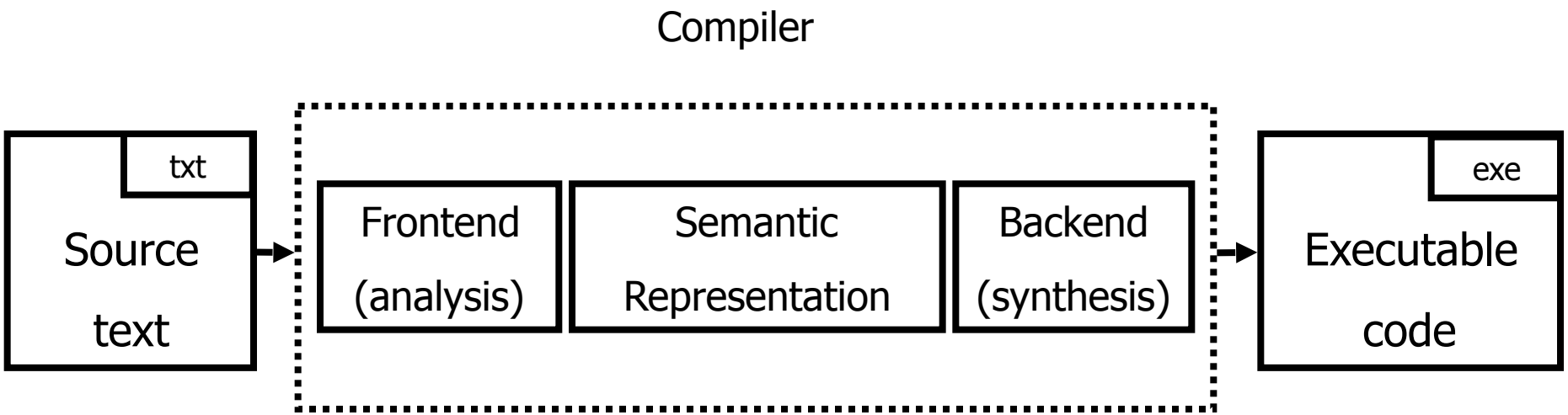
Anatomy of a Compiler



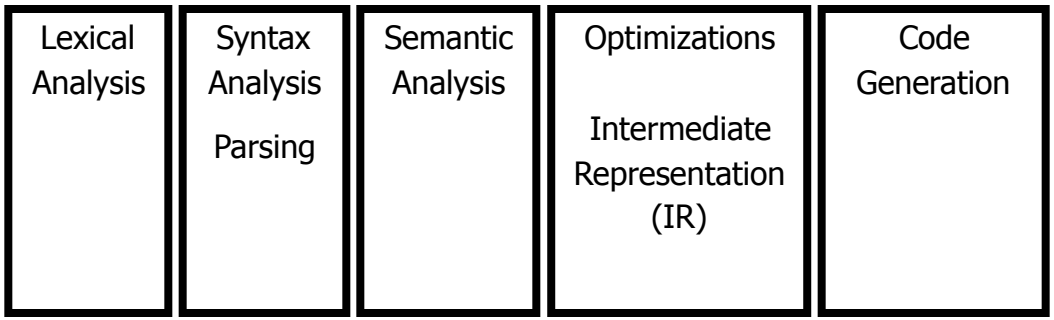

```
int a, b;
a = 2;
b = a*2 + 1;
```

```
MOV R1,2
SAL R1
INC R1
MOV R2,R1
```

Anatomy of a Compiler

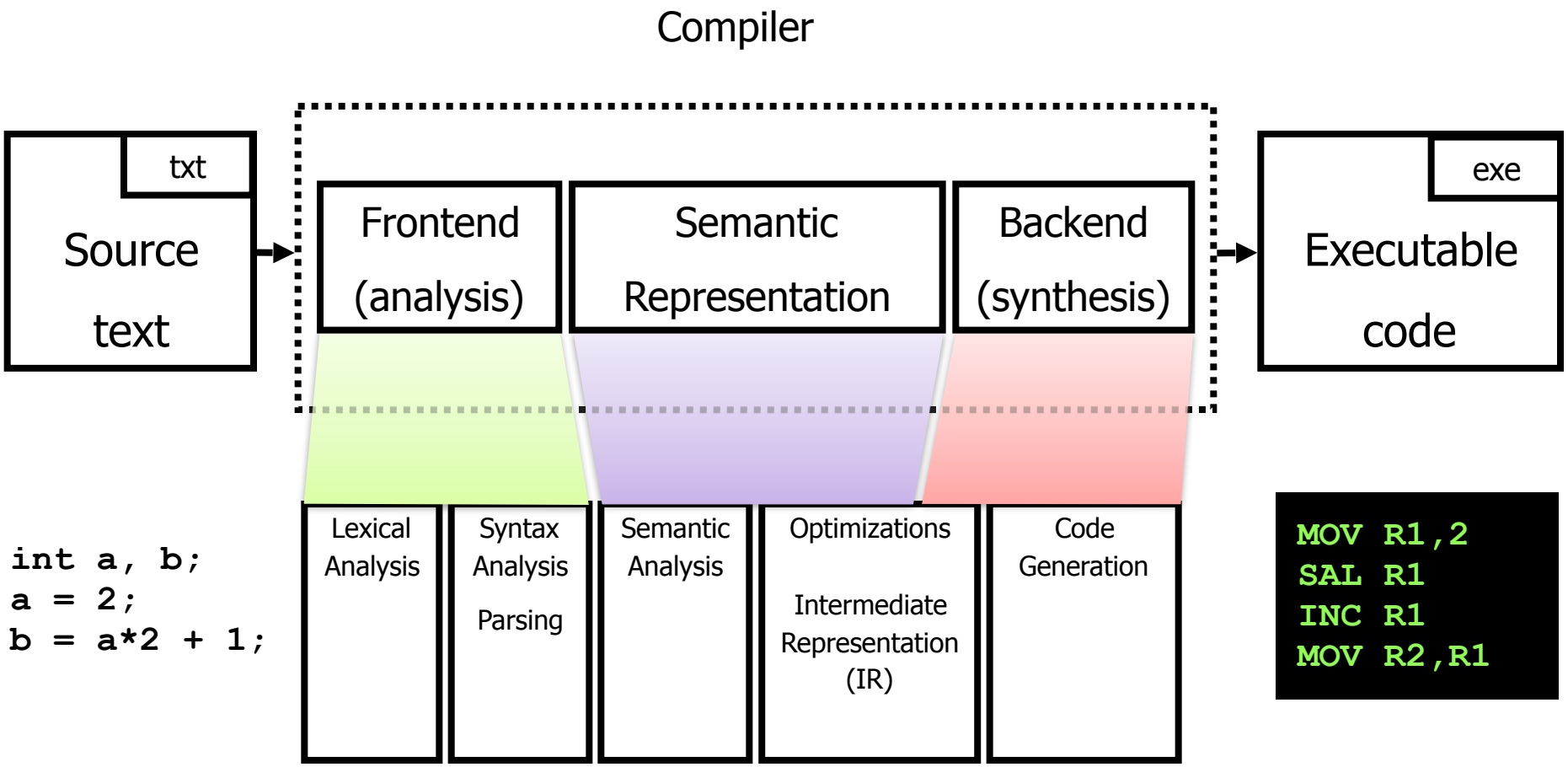



```
int a, b;
a = 2;
b = a*2 + 1;
```




```
MOV R1,2
SAL R1
INC R1
MOV R2,R1
```

Anatomy of a Compiler




```
int a, b;
a = 2;
b = a*2 + 1;
```

```
MOV R1,2
SAL R1
INC R1
MOV R2,R1
```

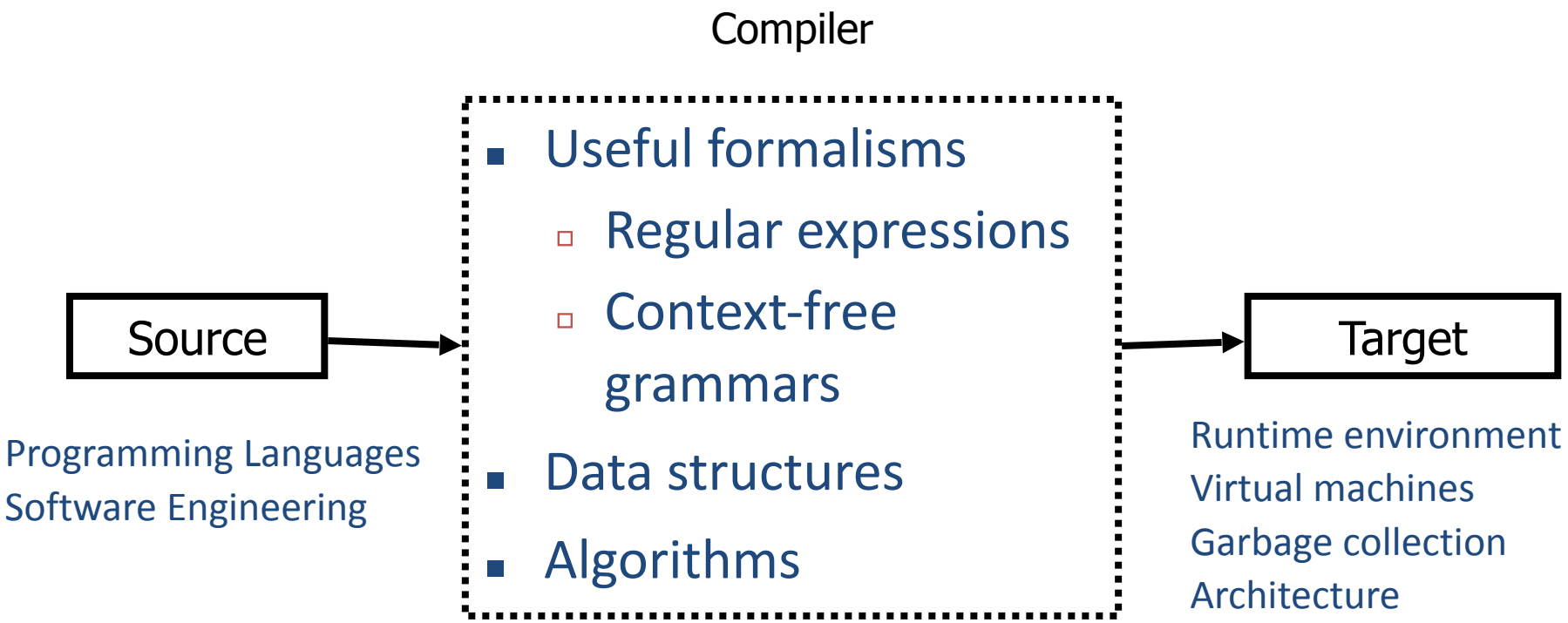
Why should you care?

- Every person in this class will build a parser some day
  - Or wish they knew how to build one…

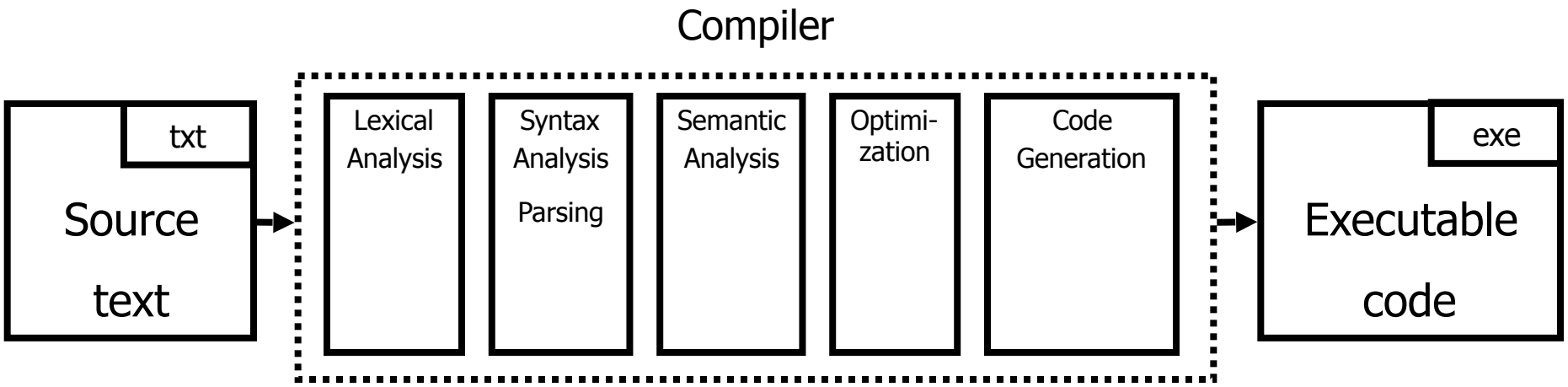
- Better understanding of programming languages
- Understand internals of compilers
- Understand (some) details of target architectures

- Useful techniques and algorithms
  - Lexical analysis / parsing
  - Semantic representation
  - …
  - Register allocation

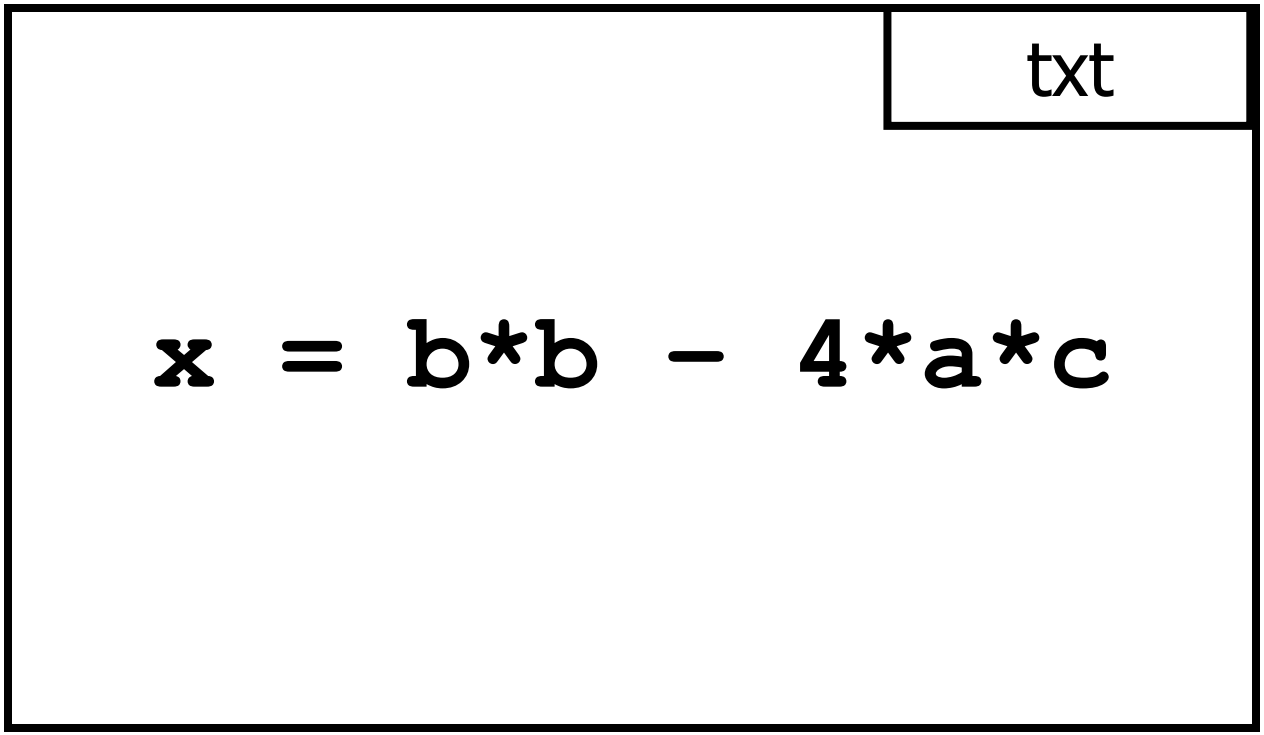
Why should you care?



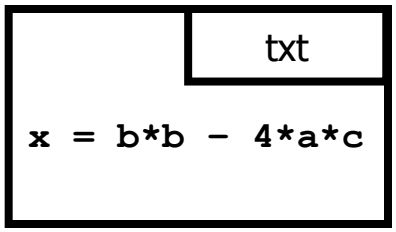
Course Overview



Journey inside a compiler



Journey inside a compiler

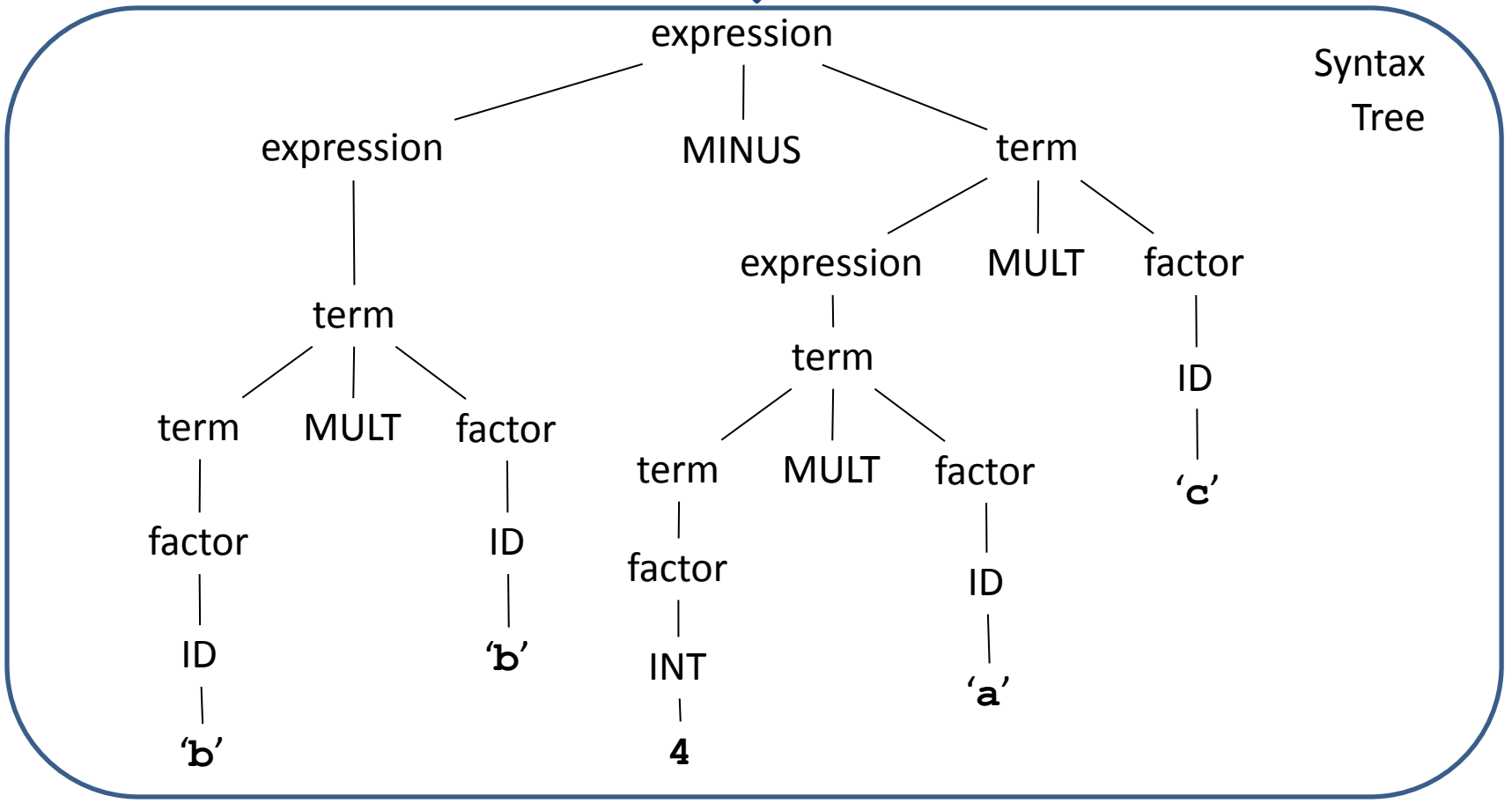


Token Stream

<ID,"**x**"> <EQ> <ID,"**b**"> <MULT> <ID,"**b**">
<MINUS> <INT,**4**> <MULT> <ID,"**a**"> <MULT> <ID,"**c**">

Journey inside a compiler

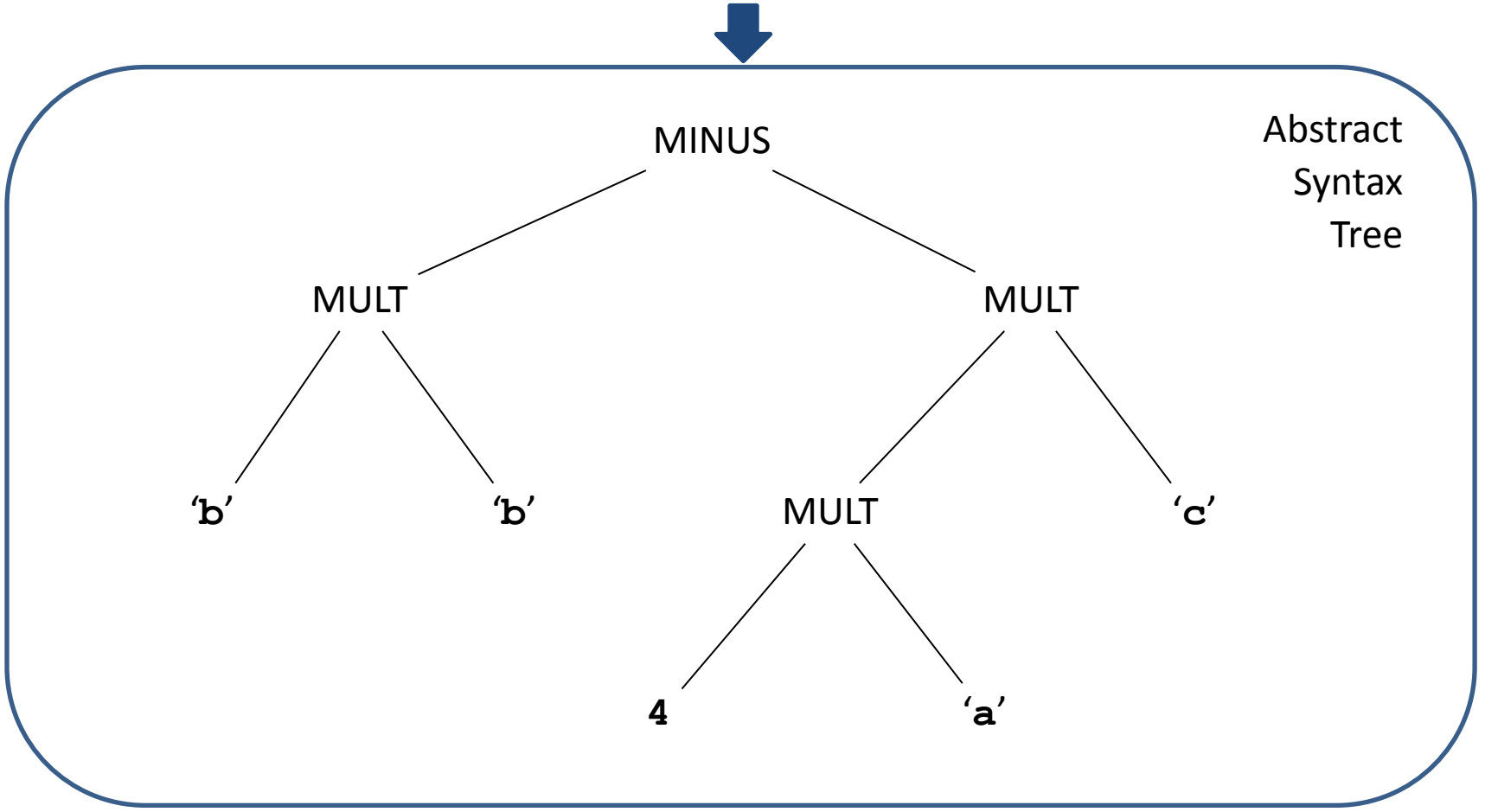
…<ID,"b"> <MULT> <ID,"b"> <MINUS> <INT,4> <MULT> <ID,"a"> <MULT> <ID,"c">



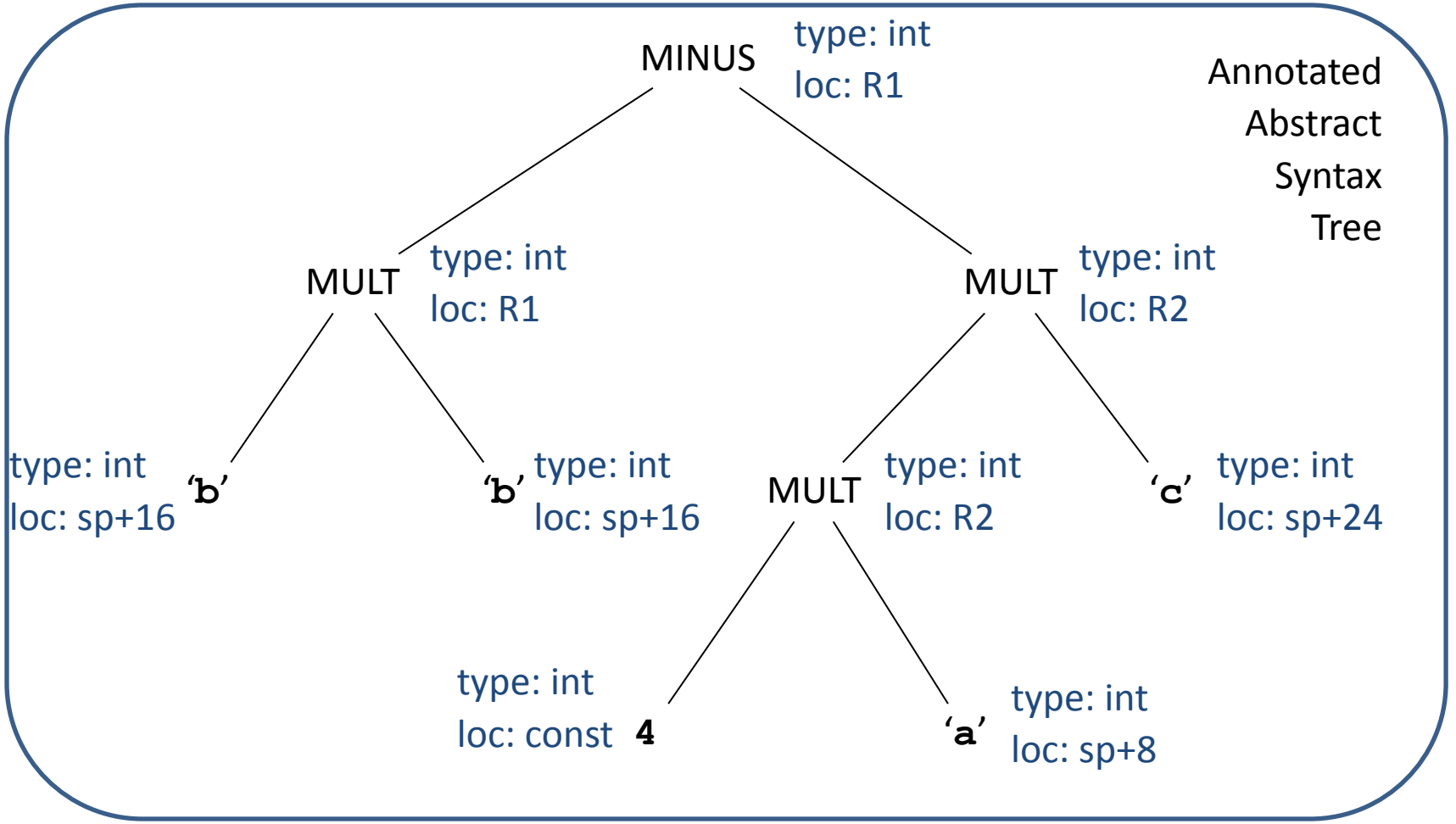
Lexical Analysis | Syntax Analysis | Sem. Analysis | Inter. Rep. | Code Gen.

Journey inside a compiler

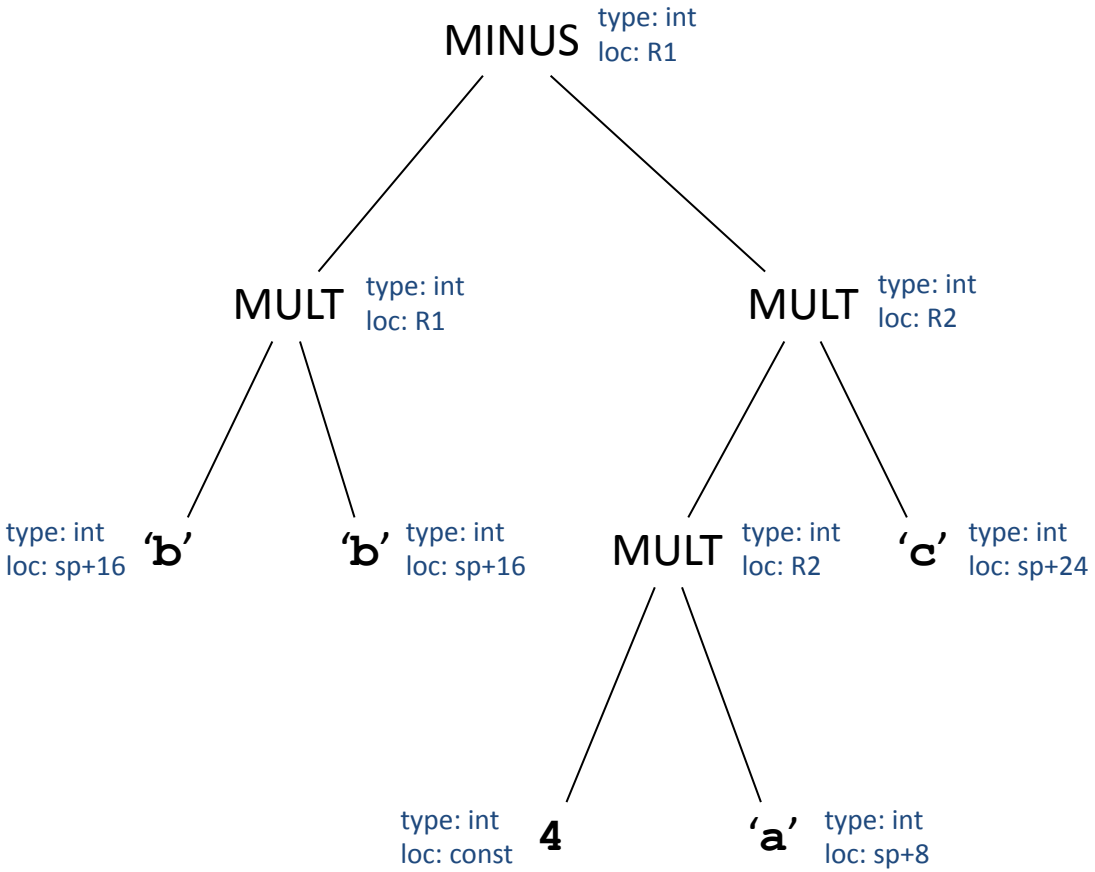
…<ID,"**b**"> <MULT> <ID,"**b**"> <MINUS> <INT,**4**> <MULT> <ID,"**a**"> <MULT> <ID,"**c**">



Journey inside a compiler



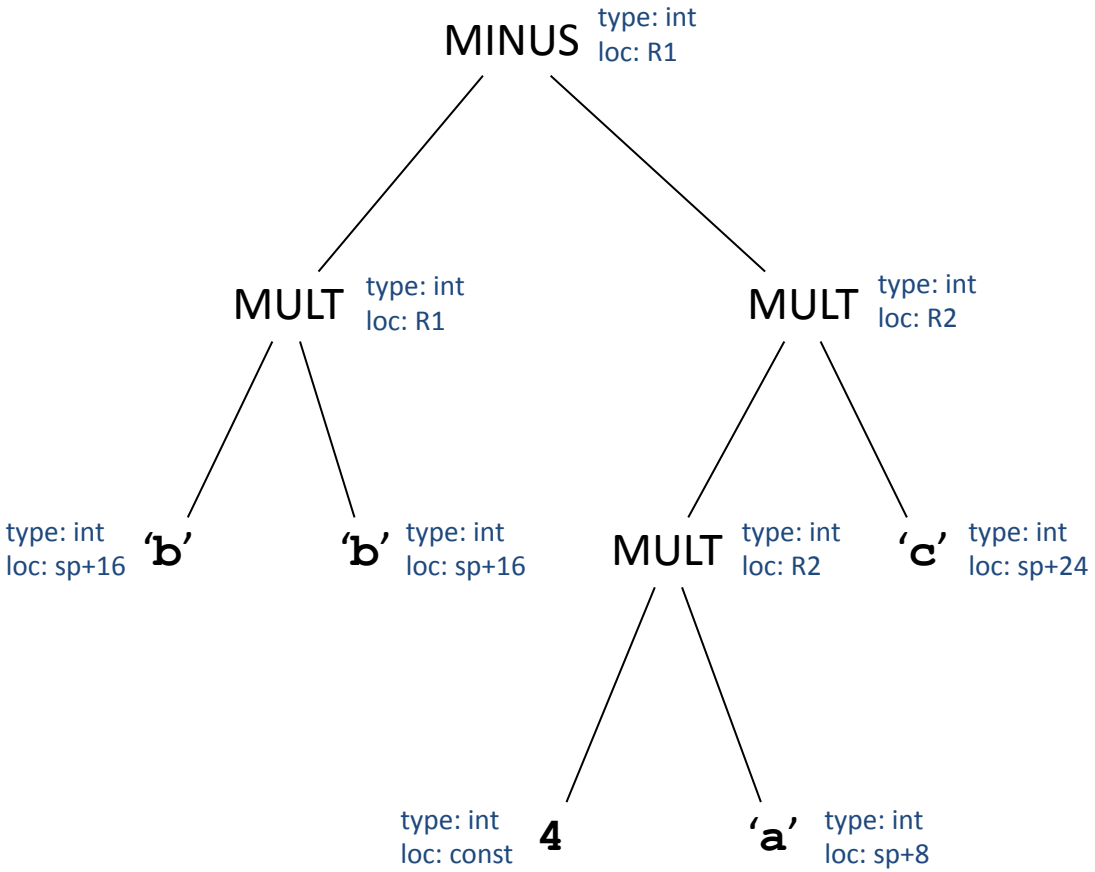
Journey inside a compiler



Intermediate Representation

R2 = 4 * a
R1 = b * b
R2 = R2 * c
R1 = R1 – R2

Journey inside a compiler



Intermediate Representation

R2 = 4 * a
R1 = b * b
R2 = R2 * c
R1 = R1 – R2

Assembly Code

MOV R2,(sp+8)
SAL R2,2
MOV R1,(sp+16)
MUL R1,(sp+16)
MUL R2,(sp+24)
SUB R1,R2

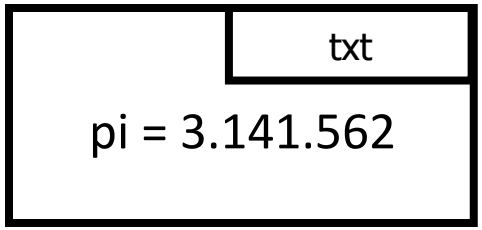
Error Checking

- In every stage…

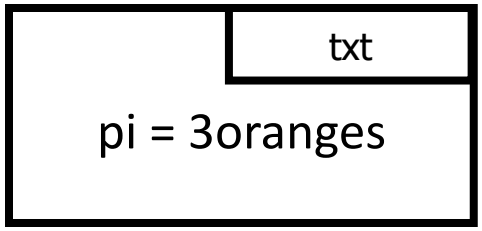
- Lexical analysis: illegal tokens
- Syntax analysis: illegal syntax
- Semantic analysis: incompatible types, undefined variables, …

- Every phase tries to recover and proceed with compilation

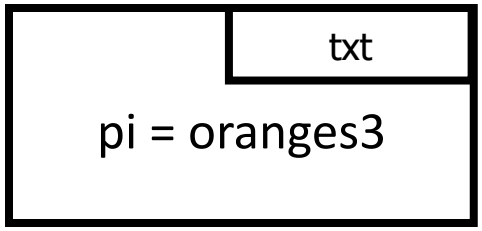
Errors in lexical analysis



Illegal token

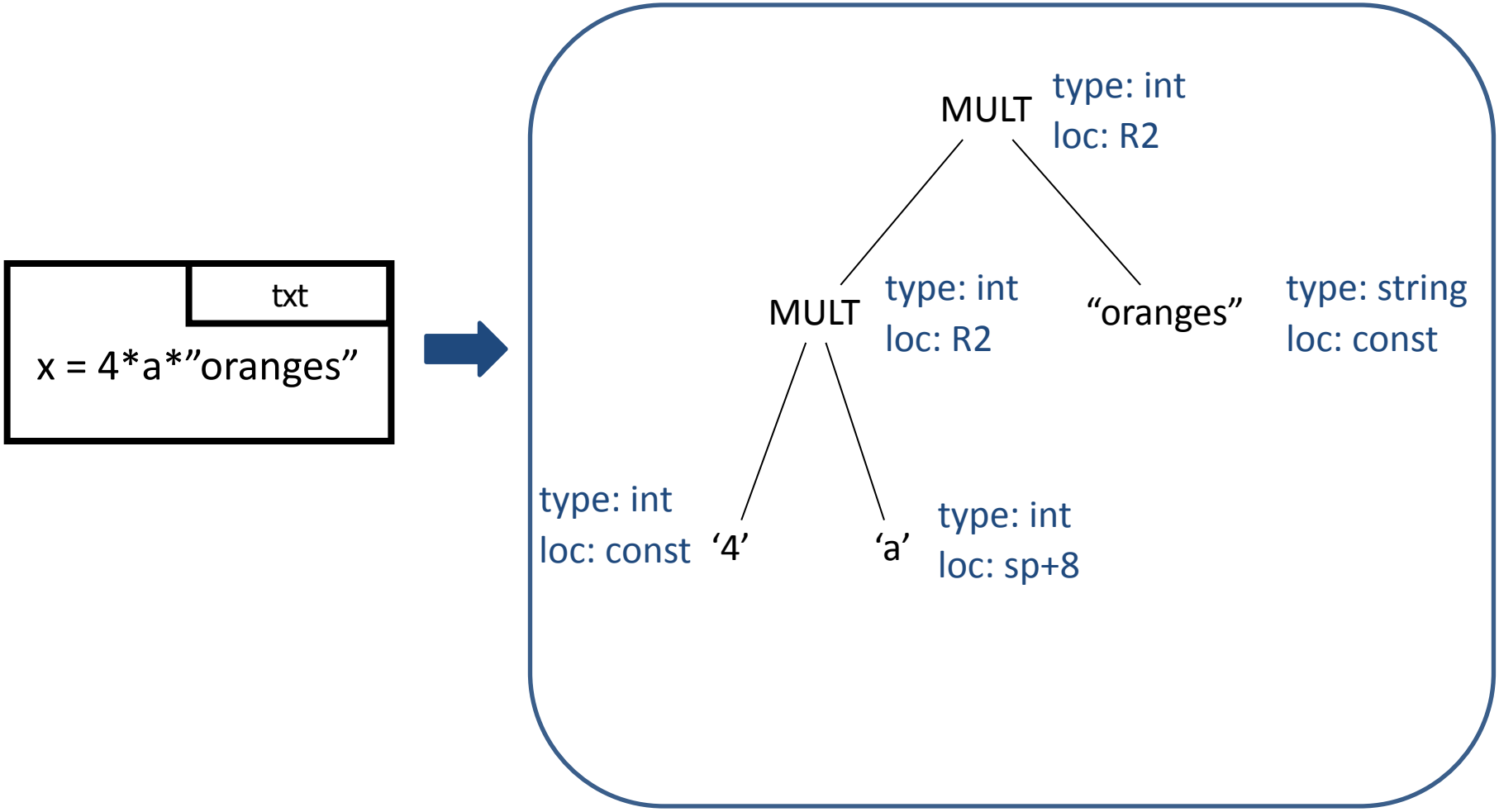


Illegal token

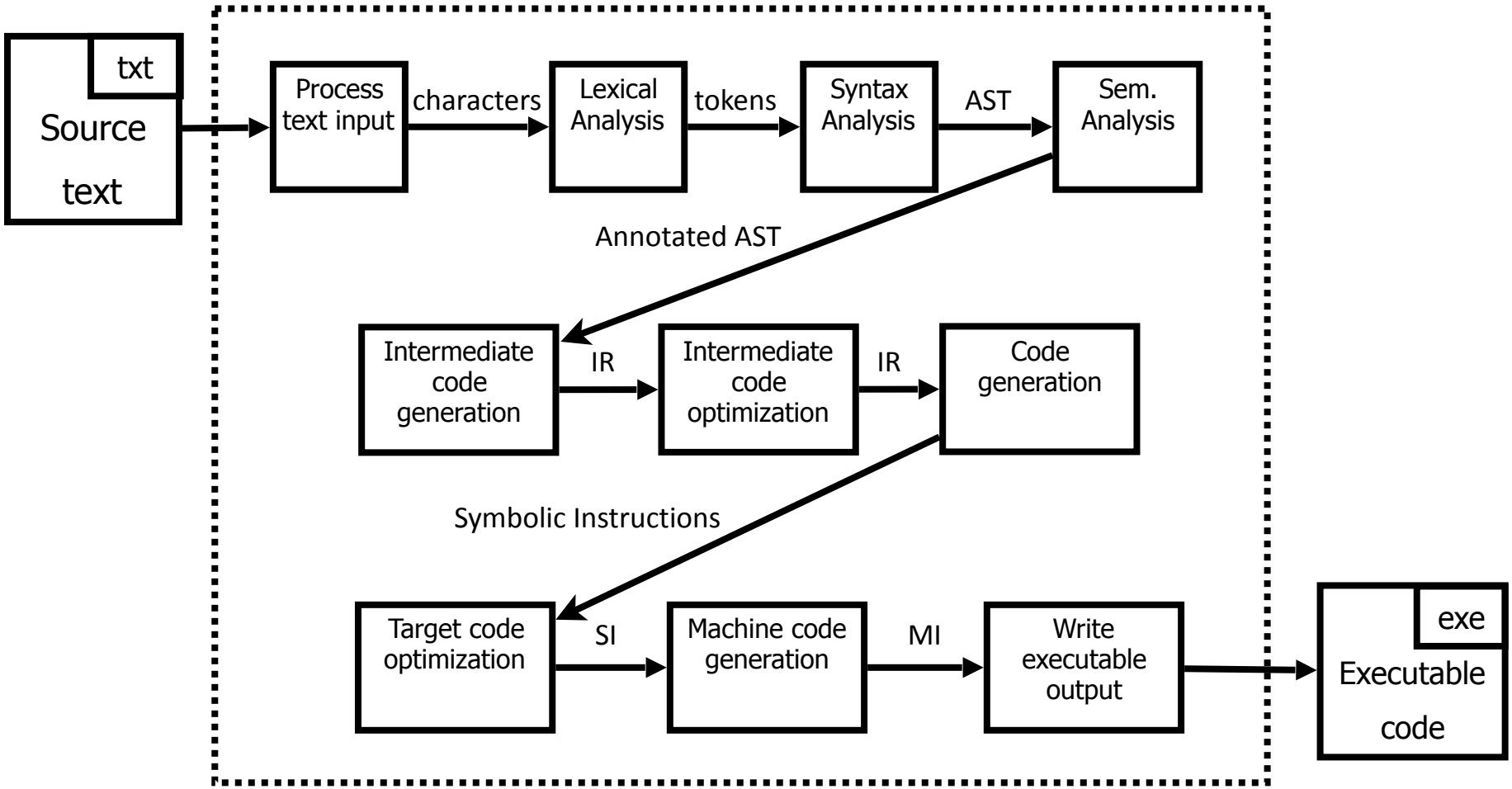


<ID,"pi">, <EQ>, <ID,"oranges3">

Error detection: type checking



The Real Anatomy of a Compiler



Optimizations

- "Optimal code" is out of reach
  - many problems are undecidable or too expensive (NP-complete)
  - Use approximation and/or heuristics
  - Must preserve correctness, should (mostly) improve code

- Many optimization heuristics
  - Loop optimizations: hoisting, unrolling, …
  - Peephole optimizations: constant folding, strength reduction, …
  - Constant propagation
    - Leverage compile-time information to save work at runtime (pre-computation)
  - Dead code elimination

- Majority of compilation time is spent in the optimization phase

Loop hoisting

Loop hoisting

```
void foo(int x, int y) {
```

Loop hoisting

```
void foo(int x, int y) {
```

Loop hoisting

```
void foo(int x, int y) {
        for (int i=0; i < 100; ++i) {
```

Loop hoisting

```
void foo(int x, int y) {
        for (int i=0; i < 100; ++i) {
                array[i] = x + y;
```

Loop hoisting

```
void foo(int x, int y) {
        for (int i=0; i < 100; ++i) {
                array[i] = x + y;
        }
```

Loop hoisting

```
void foo(int x, int y) {
        for (int i=0; i < 100; ++i) {
                array[i] = x + y;
        }
}
```

Loop hoisting

```
void foo(int x, int y) {
        for (int i=0; i < 100; ++i) {
                array[i] = x + y;
        }
}
```

Loop hoisting

```
void foo(int x, int y) {
        for (int i=0; i < 100; ++i) {
                array[i] = x + y;
        }
}
```

Loop hoisting

```
void foo(int x, int y) {
        for (int i=0; i < 100; ++i) {
                array[i] = x + y;
        }
}
```

```
void foo(int x, int y) {
```

Loop hoisting

```
void foo(int x, int y) {
        for (int i=0; i < 100; ++i) {
                array[i] = x + y;
        }
}
```

```
void foo(int x, int y) {
        int t = x + y;
```

Loop hoisting

```
void foo(int x, int y) {
        for (int i=0; i < 100; ++i) {
                array[i] = x + y;
        }
}
```

```
void foo(int x, int y) {
        int t = x + y;
        for (int i=0; i < 100; ++i) {
```

Loop hoisting

```
void foo(int x, int y) {
        for (int i=0; i < 100; ++i) {
                array[i] = x + y;
        }
}
```

```
void foo(int x, int y) {
        int t = x + y;
        for (int i=0; i < 100; ++i) {
                array[i] = t;
```

Loop hoisting

```
void foo(int x, int y) {
        for (int i=0; i < 100; ++i) {
                array[i] = x + y;
        }
}
```

```
void foo(int x, int y) {
        int t = x + y;
        for (int i=0; i < 100; ++i) {
                array[i] = t;
        }
```

Loop hoisting

```
void foo(int x, int y) {
        for (int i=0; i < 100; ++i) {
                array[i] = x + y;
        }
}
```

```
void foo(int x, int y) {
        int t = x + y;
        for (int i=0; i < 100; ++i) {
                array[i] = t;
        }
}
```

Loop unrolling

Loop unrolling

```
for (int i=0; i < 100; ++i) {
```

Loop unrolling

```
for (int i=0; i < 100; ++i) {
        delete array[i];
}
```

```
for (int i=0; i < 100; i += 5) {
        delete array[i];
        delete array[i+1];
        delete array[i+2];
        delete array[i+3];
        delete array[i+4];

}
```

Machine code generation

- Register allocation
  - Optimal register assignment is NP-Complete
  - In practice, known heuristics perform well
- assign variables to memory locations
- Instruction selection
  - Convert IR to actual machine instructions

- Modern architectures
  - Multicores
  - Challenging memory hierarchies
  - SIMD instructions

Compiler Construction Toolset

- Lexical analysis generators
  - lex
- Parser generators
  - yacc

Summary

- Compiler is a program that translates code from source language to target language
- Compilers play a critical role
  - Bridge from programming languages to the machine
  - Many useful techniques and algorithms
  - Many useful tools (e.g., lexer/parser generators)
- Compiler constructed from modular phases
  - Reusable
  - Different front/back ends

Coming up next

- Lexical analysis