

Compilation

0368-3133

Lecture 1:
Introduction

Noam Rinetzky

**Please
Be Sure
Your
Cell
Phone
Is Off**



Admin

- Lecturer: Noam Rinetzky
 - *maon@tau.ac.il*
 - <http://www.cs.tau.ac.il/~maon>
- T.A.: Oren Ish Shalom
- Textbooks:
 - Modern Compiler Design
 - Compilers: principles, techniques and tools

Admin

- Compiler Project 50%
 - 4-5 practical exercises
 - Groups of 3
- Final exam 50%
 - must pass

Course Goals

- What is a compiler
- How does it work
- (Reusable) techniques & tools

Course Goals

- What is a compiler
- How does it work
- (Reusable) techniques & tools
- Programming language implementation
 - runtime systems
- Execution environments
 - Assembly, linkers, loaders, OS

Introduction

Compilers: principles, techniques and tools

What is a Compiler?



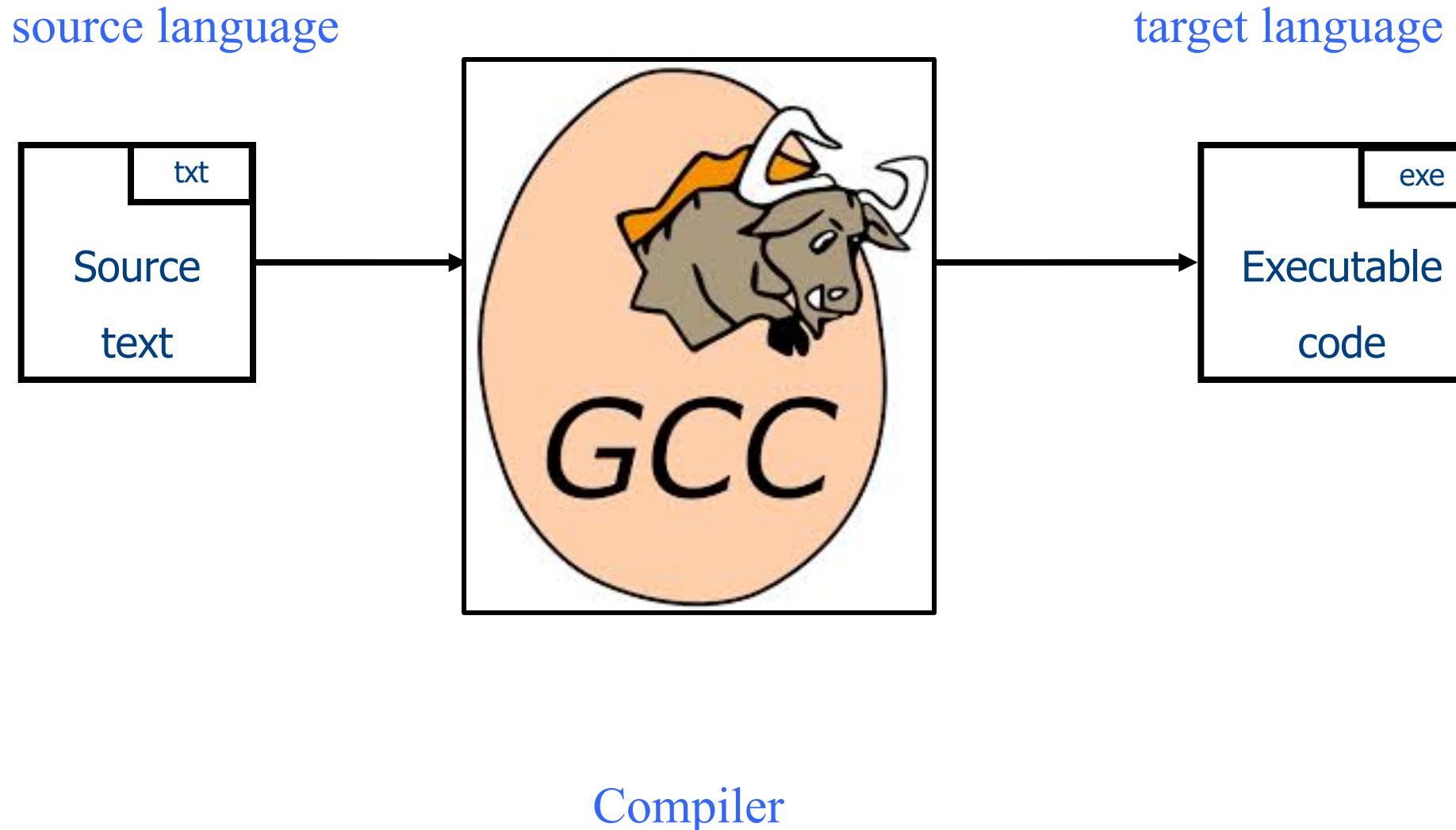
What is a Compiler?

“A compiler is a **computer program** that **transforms** source code written in a programming language (**source language**) into another language (**target language**).

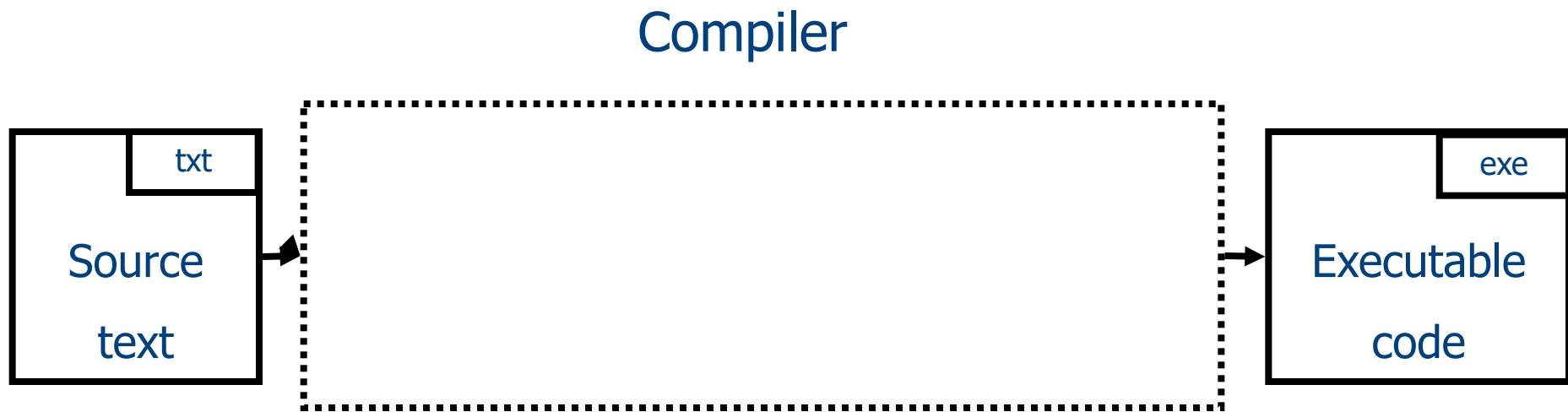
The most common reason for wanting to transform source code is to create an **executable program**. ”

--*Wikipedia*

What is a Compiler?



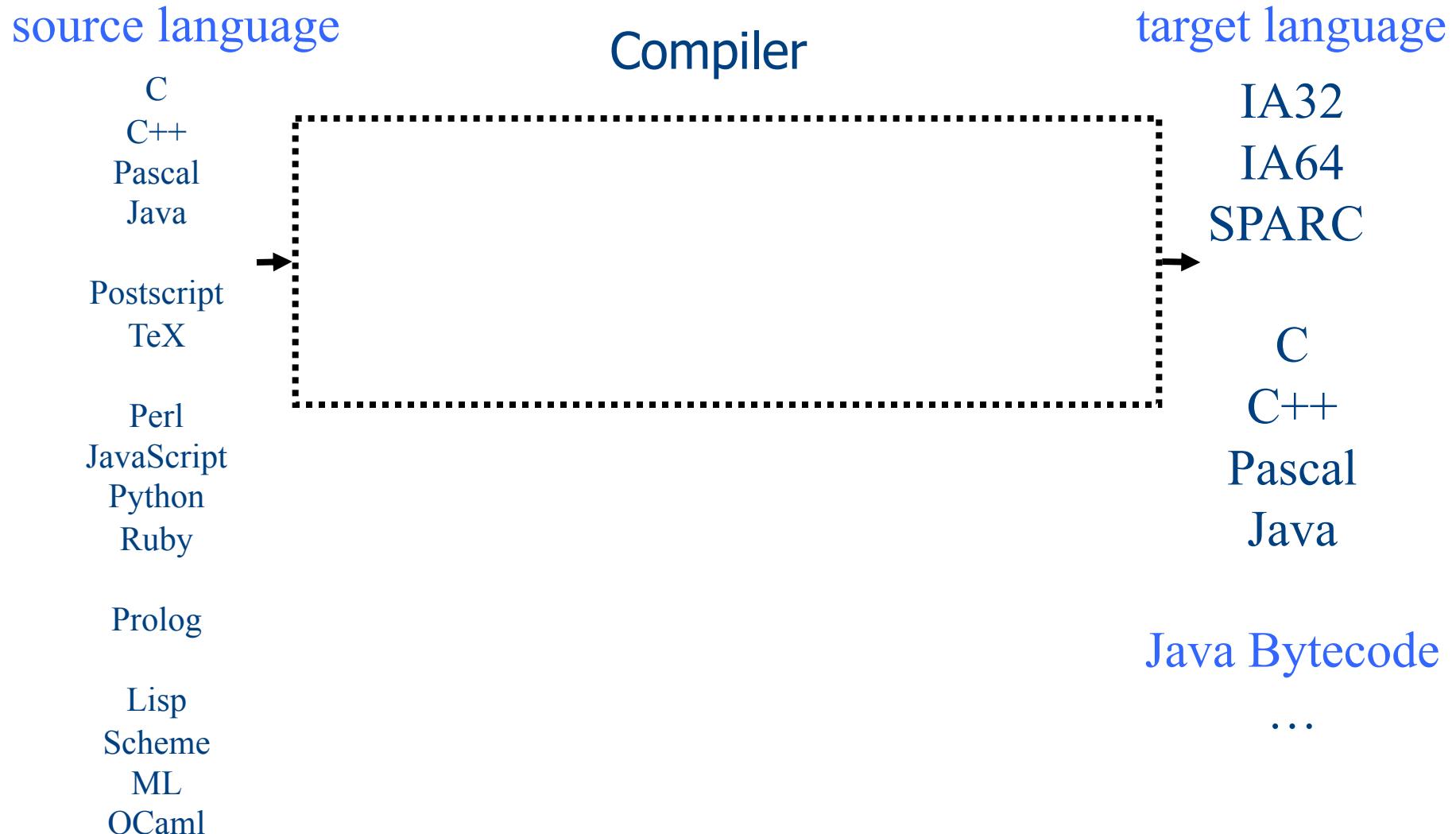
What is a Compiler?



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

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

What is a Compiler?



High Level Programming Languages

- **Imperative** Algol, PL1, Fortran, Pascal, Ada, Modula, C
 - Closely related to “von Neumann” Computers
- **Object-oriented** Simula, Smalltalk, Modula3, C++, Java, C#, Python
 - Data abstraction and ‘evolutionary’ form of program development
 - Class an implementation of an abstract data type (data+code)
 - Objects Instances of a class
 - Inheritance + generics
- **Functional** Lisp, Scheme, ML, Miranda, Hope, Haskell, OCaml, F#
- **Logic Programming** Prolog

More Languages

- **Hardware description languages** VHDL
 - The program describes Hardware components
 - The compiler generates hardware layouts
- **Graphics and Text processing** TeX, LaTeX, postscript
 - The compiler generates page layouts
- **Scripting languages** Shell, C-shell, Perl
 - Include primitives constructs from the current software environment
- **Web/Internet** HTML, Telescript, JAVA, Javascript
- **Intermediate-languages** Java bytecode, IDL

High Level Prog. Lang., Why?

High Level Prog. Lang., Why?

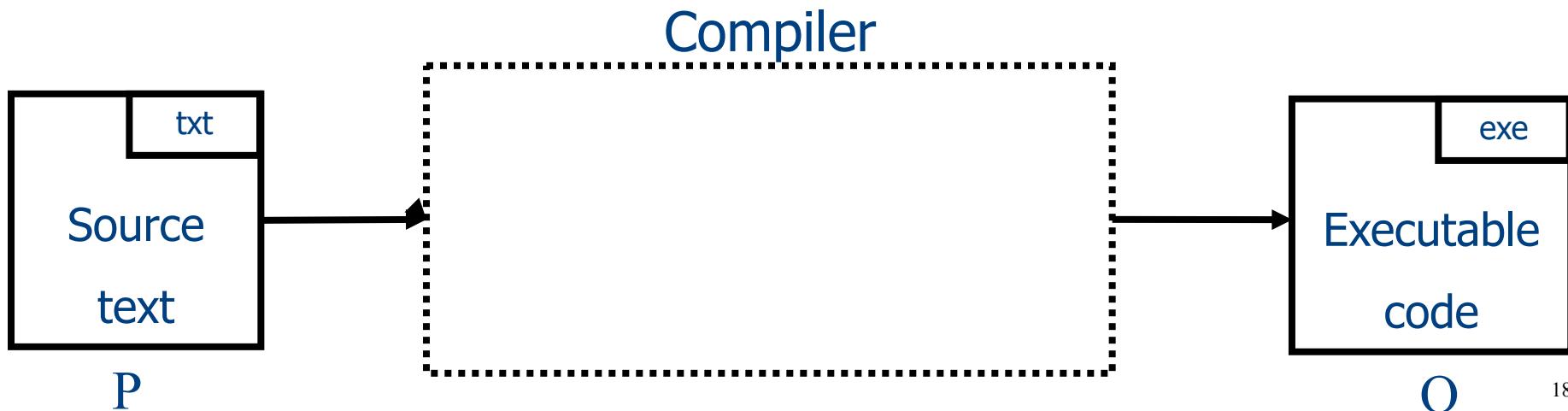


Compiler vs. Interpreter

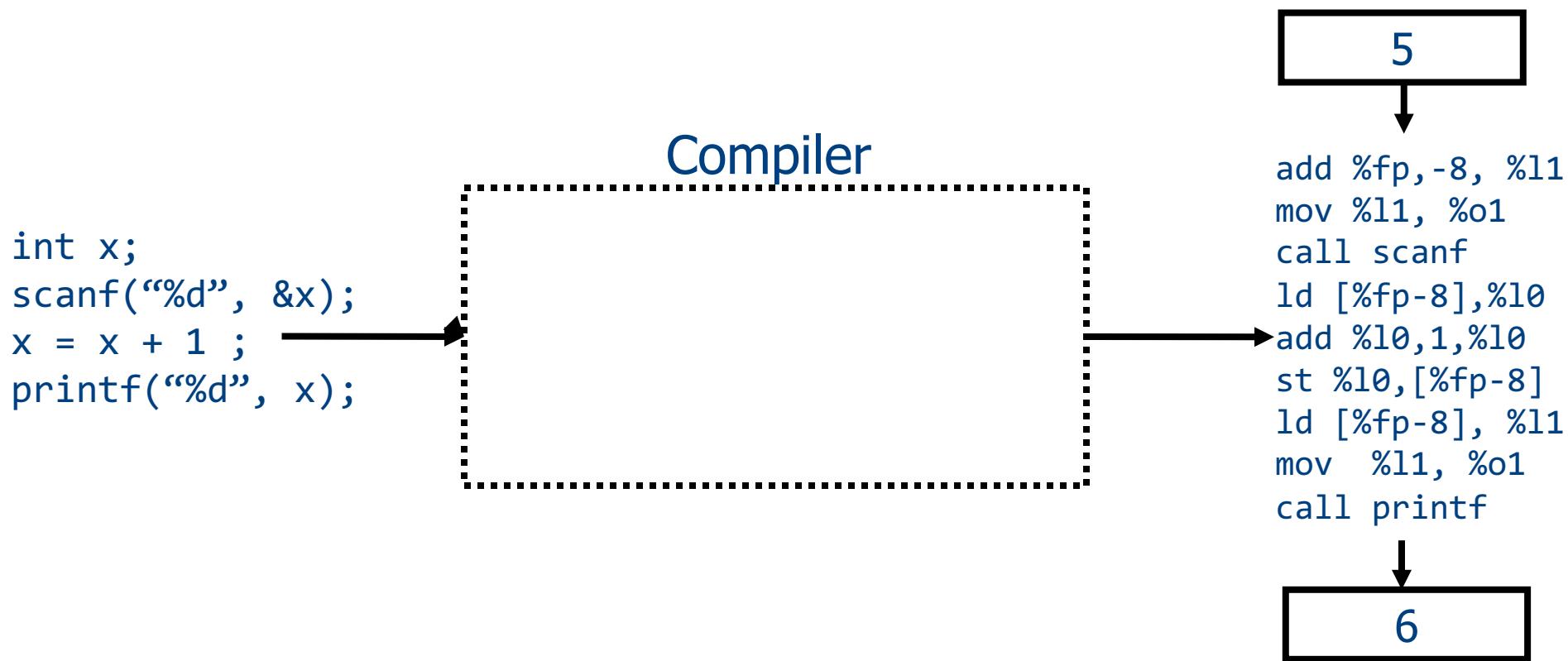


Compiler

- A program which **transforms** programs
- Input a program (P)
- Output an object program (O)
 - For any x , “ $O(x)$ ” “=” “ $P(x)$ ”

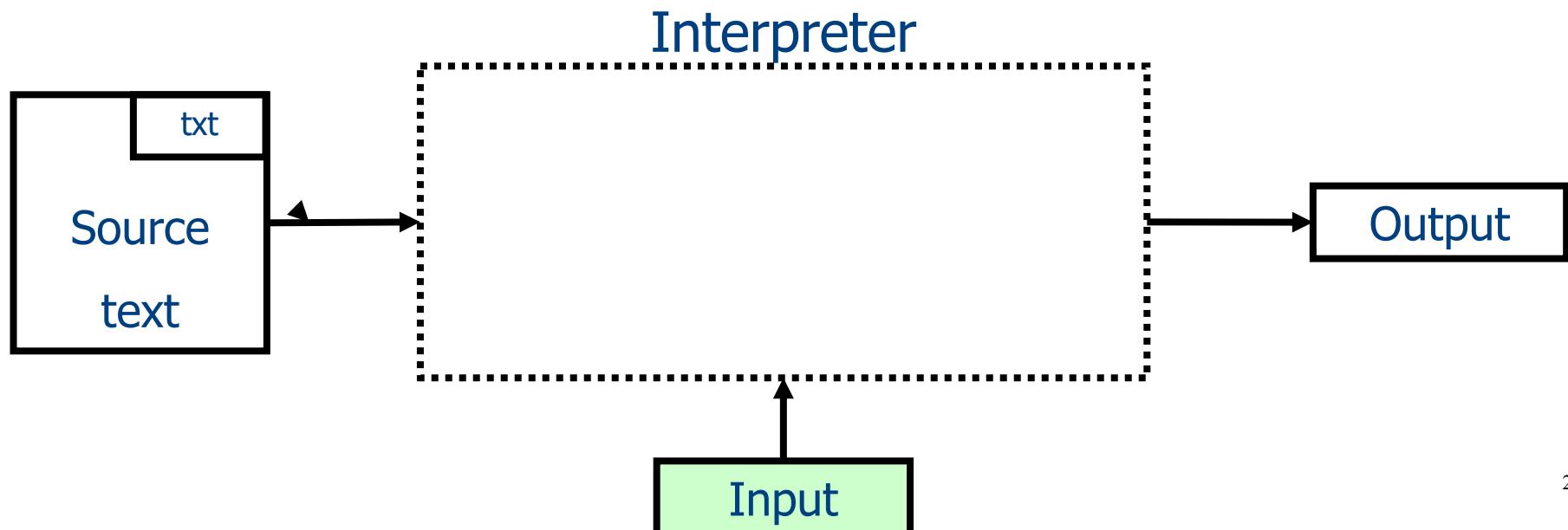


Compiling C to Assembly



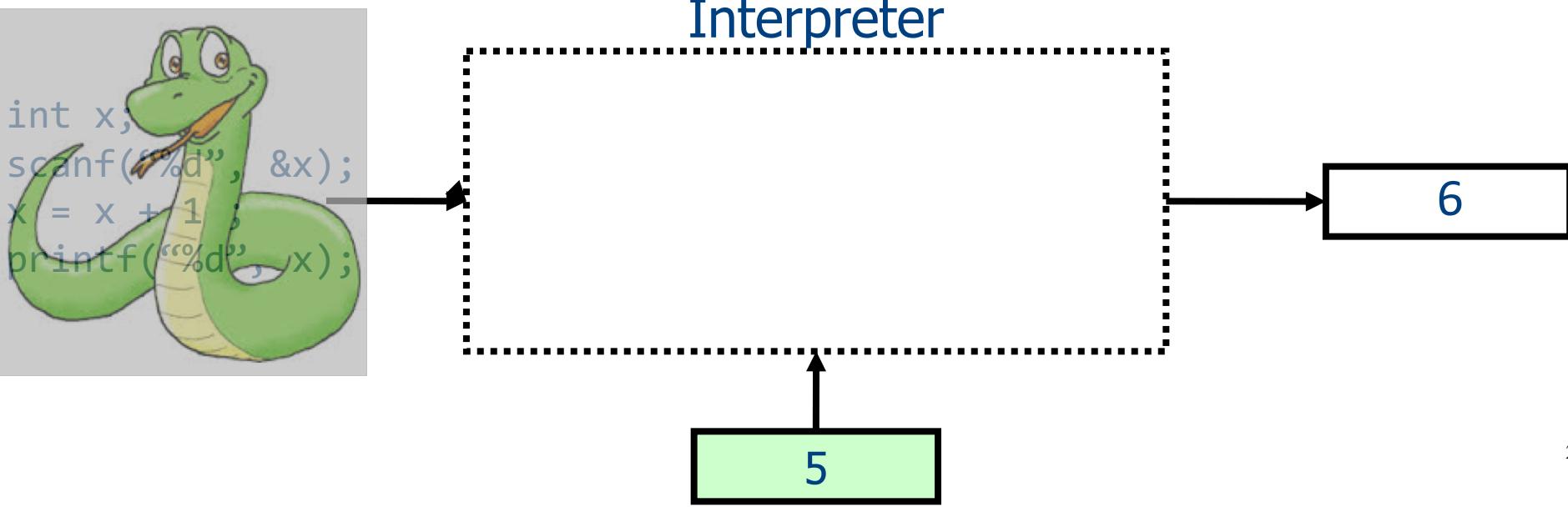
Interpreter

- A program which **executes** a program
- **Input** a program (P) + its input (x)
- **Output** the computed output ($P(x)$)

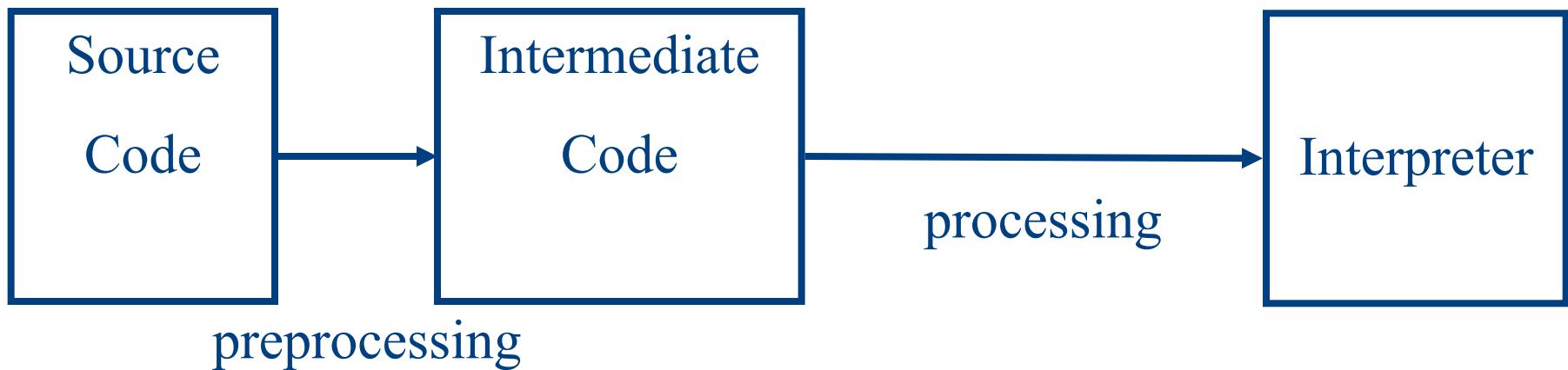
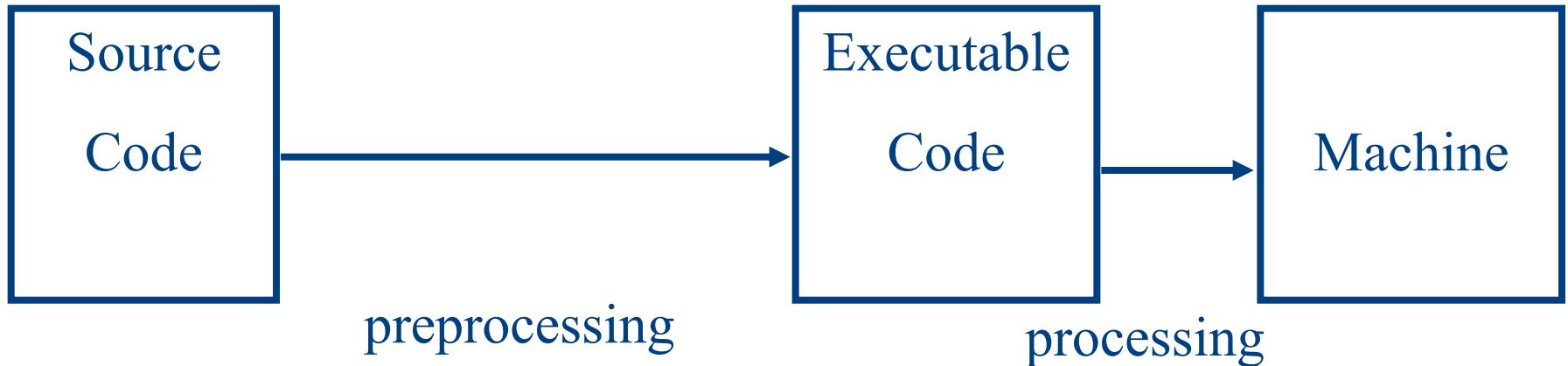


Interpreting (running) .py programs

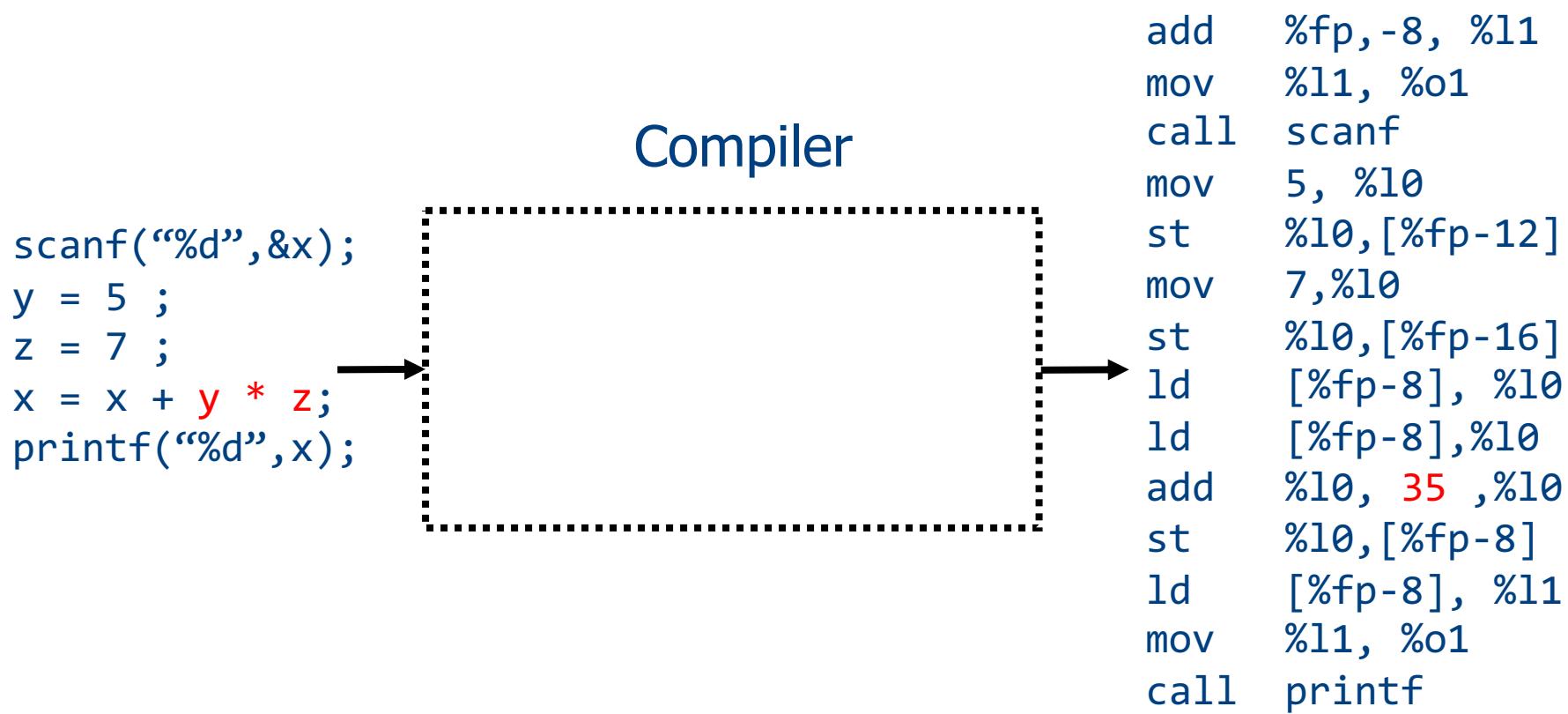
- A program which **executes** a program
- **Input** a program (P) + its input (x)
- **Output** the computed output ("P(x)")



Compiler vs. Interpreter



Compiled programs are usually more efficient than interpreted



Compilers report input-independent **possible** errors

- Input-program

```
scanf("%d", &y);
if (y < 0)
    x = 5;

...
if (y <= 0)
    z = x + 1;
```

- Compiler-Output

- “line 88: x may be used before set”

Interpreters report input-specific **definite** errors

- Input-program

```
scanf("%d", &y);
if (y < 0)
    x = 5;
...
if (y <= 0)
    z = 1 / (x + 5 * y);
```

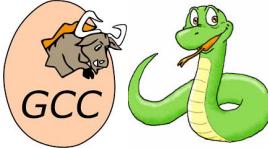
- Input data

- y = -1
 - y = 0

Interpreter vs. Compiler

- Conceptually simpler
 - “define” the prog. lang.
 - Can provide more specific error report
 - Easier to port
 - Faster response time
 - [More secure]
- 
- How do we know the translation is correct?
 - Can report errors before input is given
 - More efficient code
 - Compilation can be expensive
 - move computations to compile-time
 - *compile-time + execution-time < interpretation-time* is possible

Concluding Remarks

- Both compilers and interpreters are programs written in high level language
- Compilers and interpreters share functionality
- In this course we focus on **compilers**

Everything should be built top-down, except
the first time

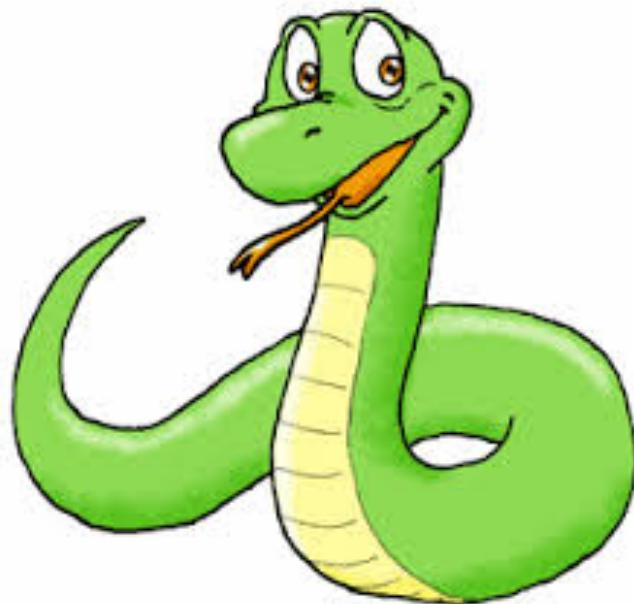
-- Alan J. Perlis



[Alan Jay Perlis](#) (1922 – 1990)

a pioneer in the field of programming
language and compilers, first recipient
of the Turing Award

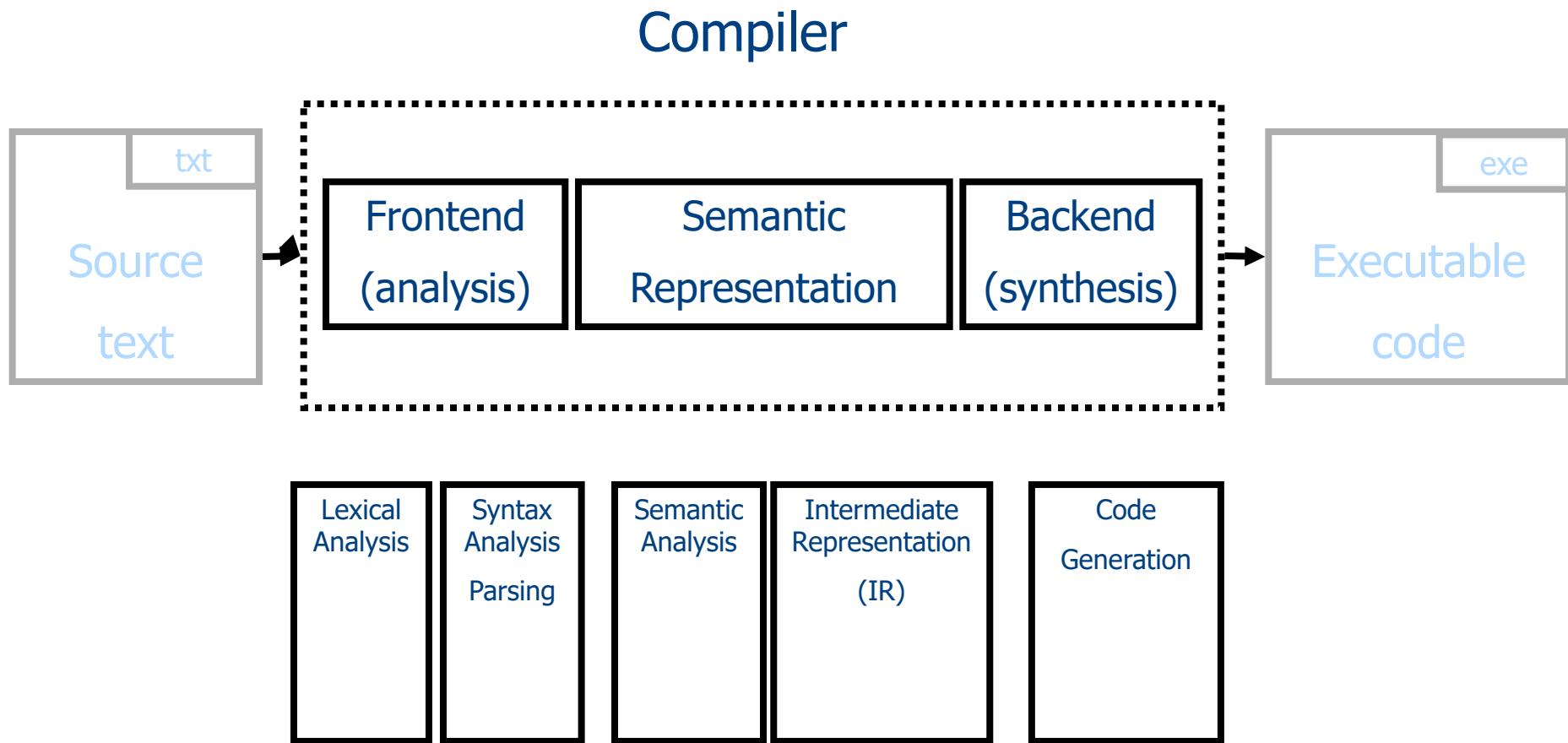
Ex 0: A Simple Interpreter



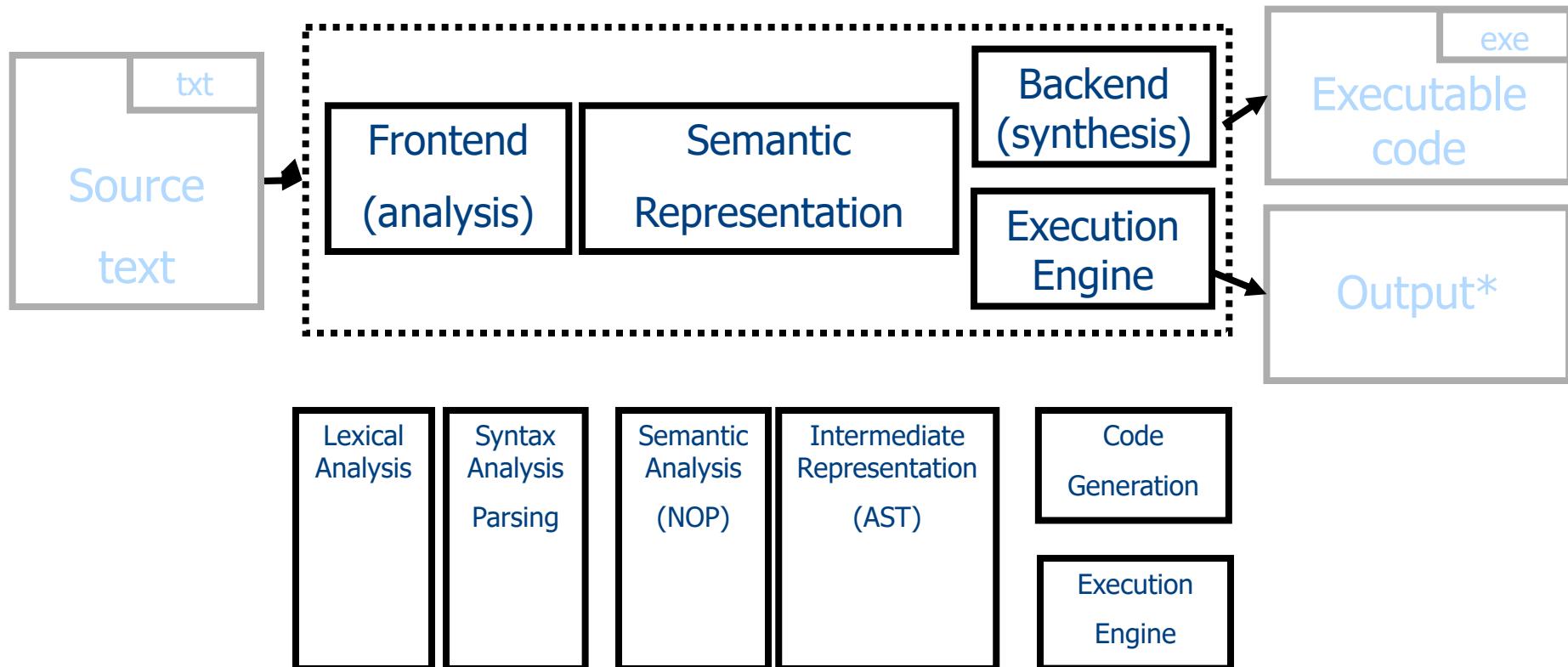
Toy compiler/interpreter

- Trivial programming language
- Stack machine
- Compiler/interpreter written in C
- Demonstrate the basic steps
- Textbook: Modern Compiler Design 1.2

Conceptual Structure of a Compiler



Structure of toy Compiler / interpreter



* Programs in our PL do not take input

Source Language

- Fully parameterized expressions
- Arguments can be a single digit

✓ $(4 + (3 * 9))$

X $3 + 4 + 5$

X $(12 + 3)$

expression \rightarrow digit | ‘(‘ expression operator expression ‘)’

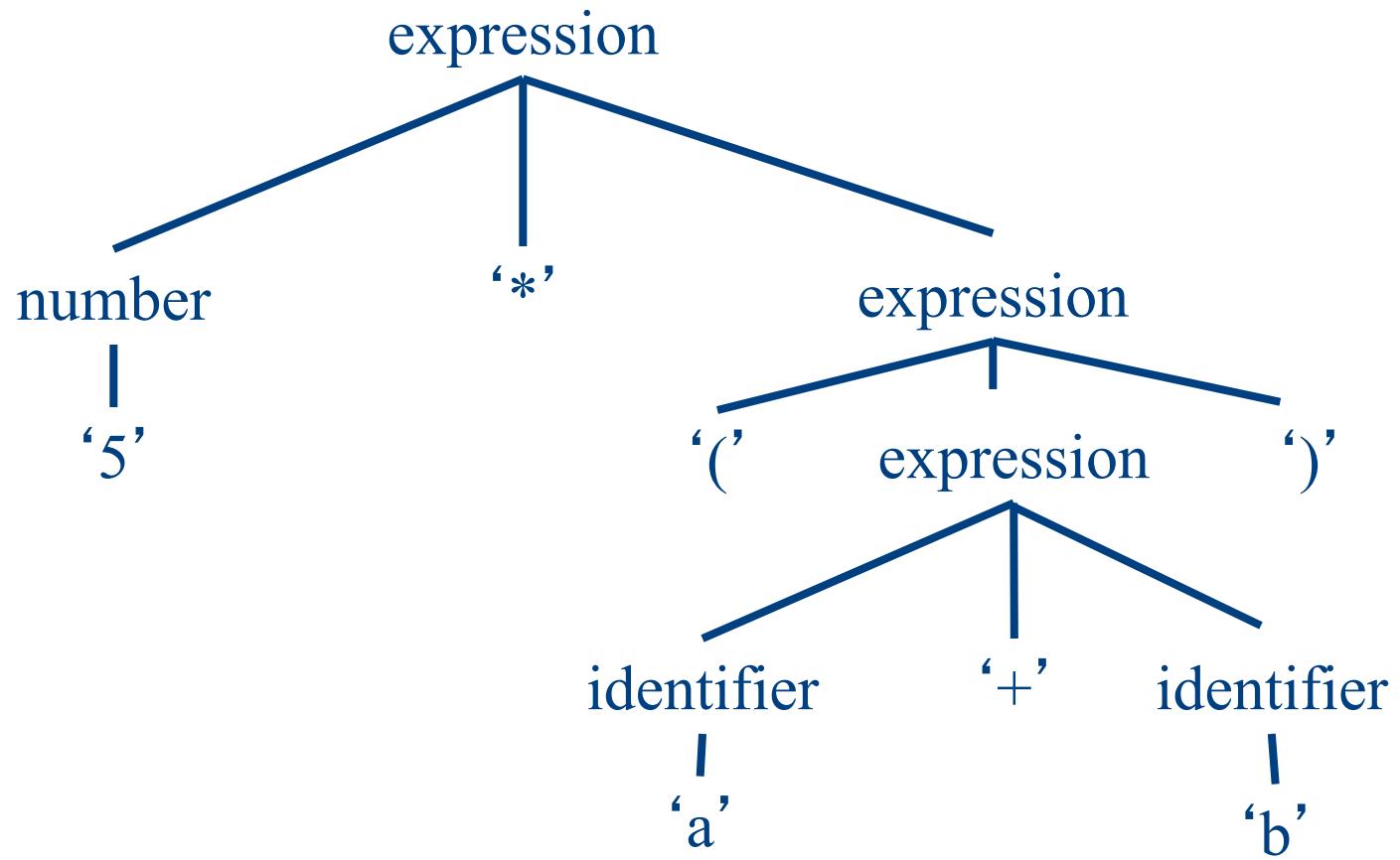
operator \rightarrow ‘+’ | ‘*’

digit \rightarrow ‘0’ | ‘1’ | ‘2’ | ‘3’ | ‘4’ | ‘5’ | ‘6’ | ‘7’ | ‘8’ | ‘9’

The abstract syntax tree (AST)

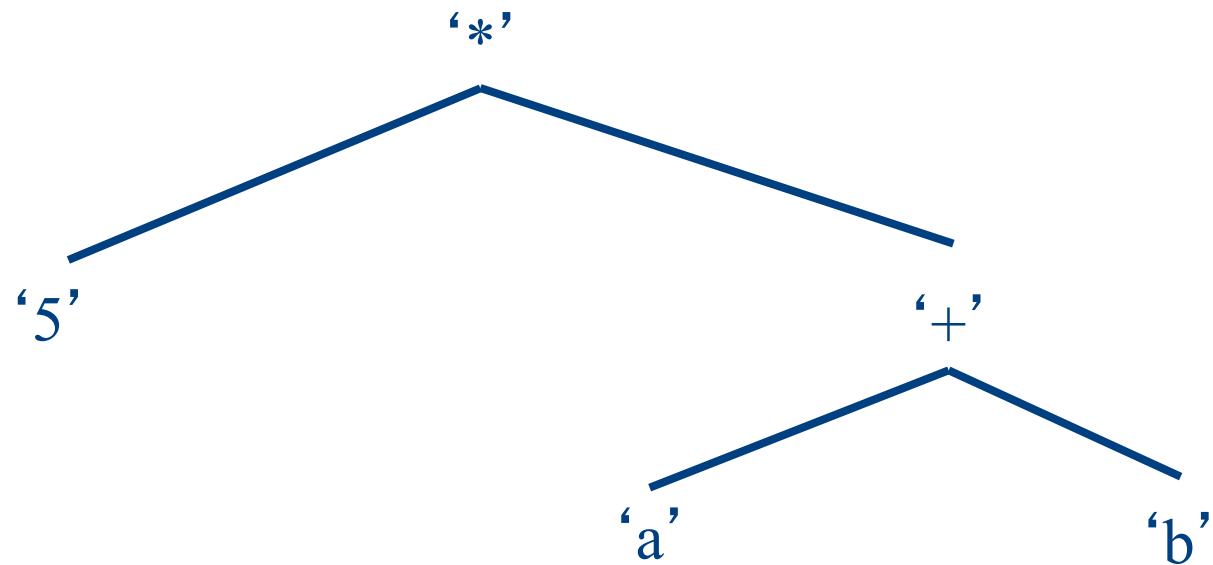
- Intermediate program representation
- Defines a tree
 - Preserves program hierarchy
- Generated by the **parser**
- Keywords and punctuation symbols are not stored
 - Not relevant once the tree exists

Concrete syntax tree[#] for $5*(a+b)$

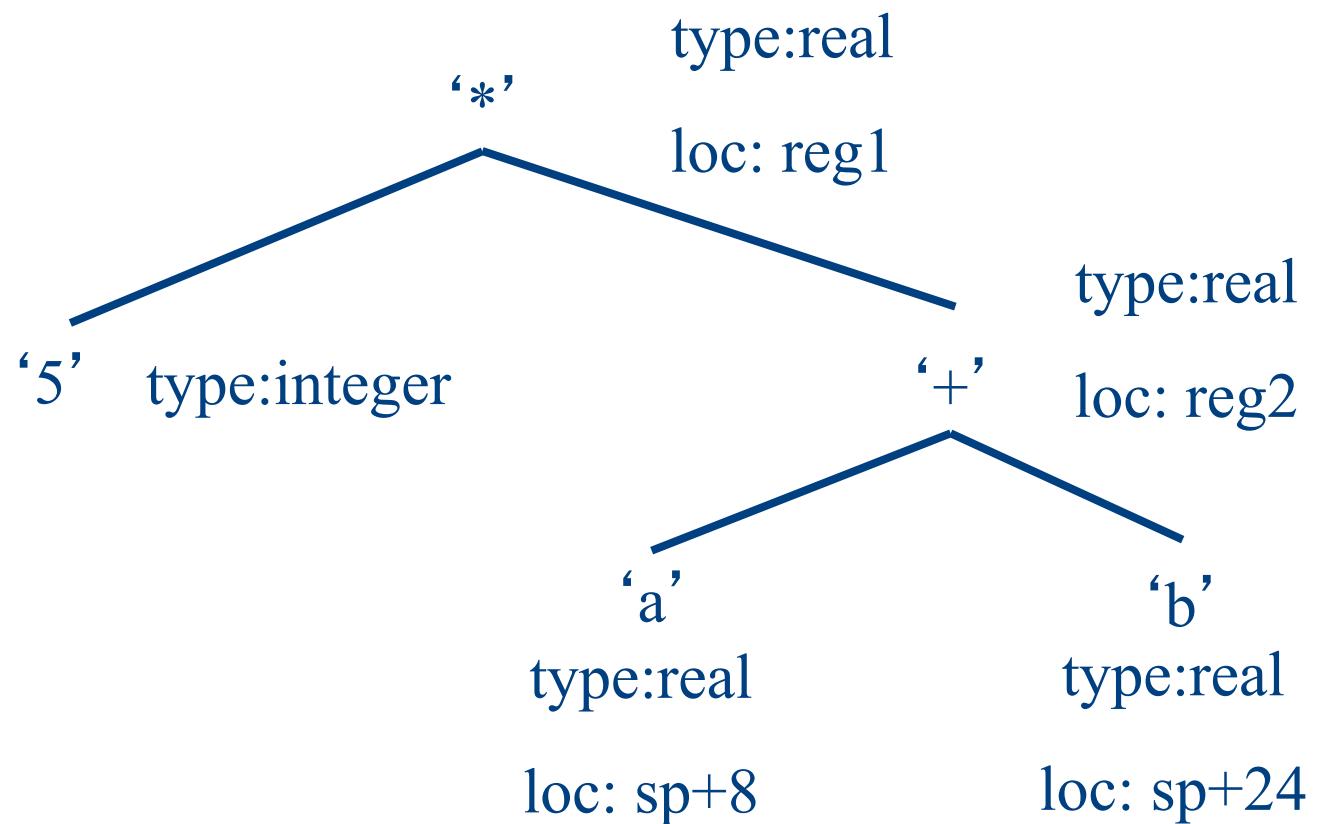


#Parse tree

Abstract Syntax tree for $5*(a+b)$



Annotated Abstract Syntax tree



Driver for the toy compiler/interpreter

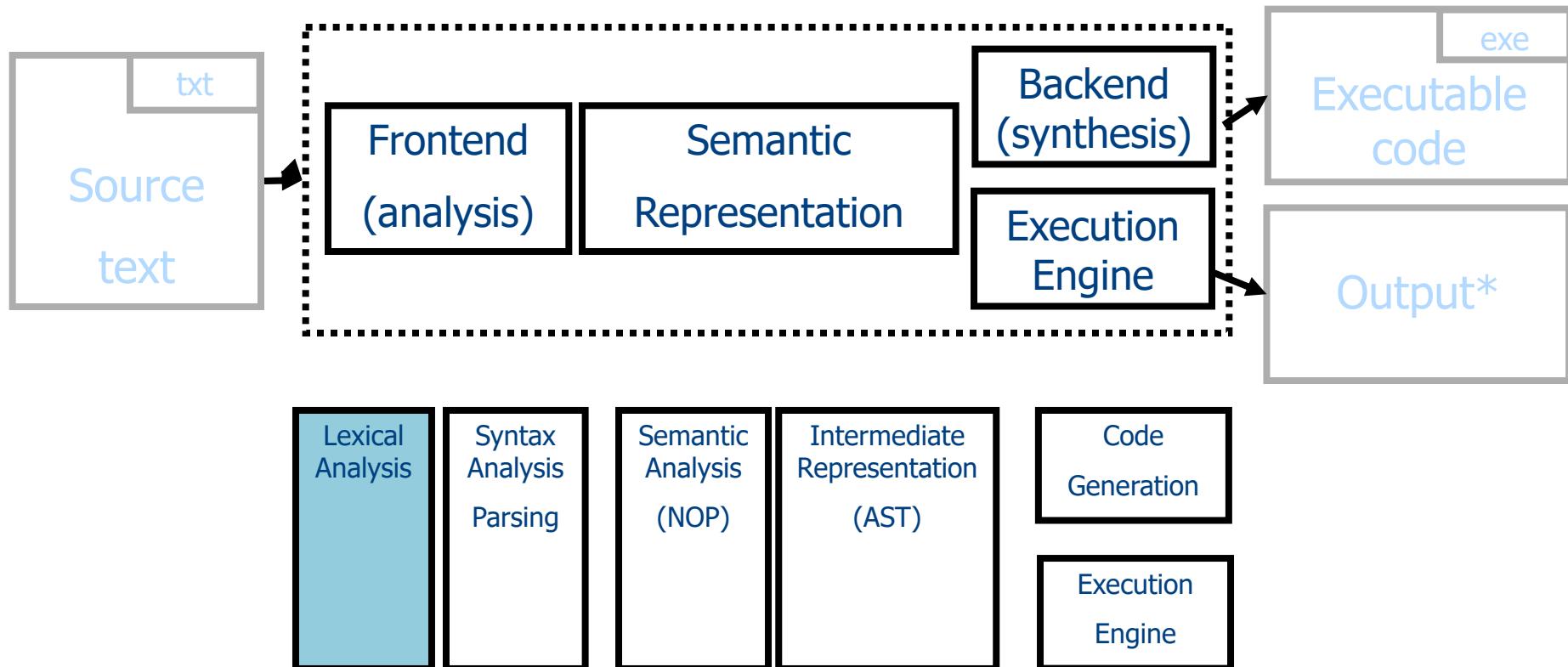
```
#include    "parser.h"      /* for type AST_node */
#include    "backend.h"     /* for Process() */
#include    "error.h"       /* for Error() */

int main(void) {
    AST_node *icode;

    if (!Parse_program(&icode)) Error("No top-level expression");
    Process(icode);

    return 0;
}
```

Structure of toy Compiler / interpreter



* Programs in our PL do not take input

Lexical Analysis

- Partitions the inputs into tokens
 - DIGIT
 - EOF
 - ‘*’
 - ‘+’
 - ‘(’
 - ‘)’
- Each token has its representation
- Ignores whitespaces

lex.h: Header File for Lexical Analysis

```
/* Define class constants */  
/* Values 0-255 are reserved for ASCII characters */  
  
#define EoF      256  
#define DIGIT    257  
  
typedef struct {  
    int class;  
    char repr;} Token_type;  
  
  
extern Token_type Token;  
extern void get_next_token(void);
```

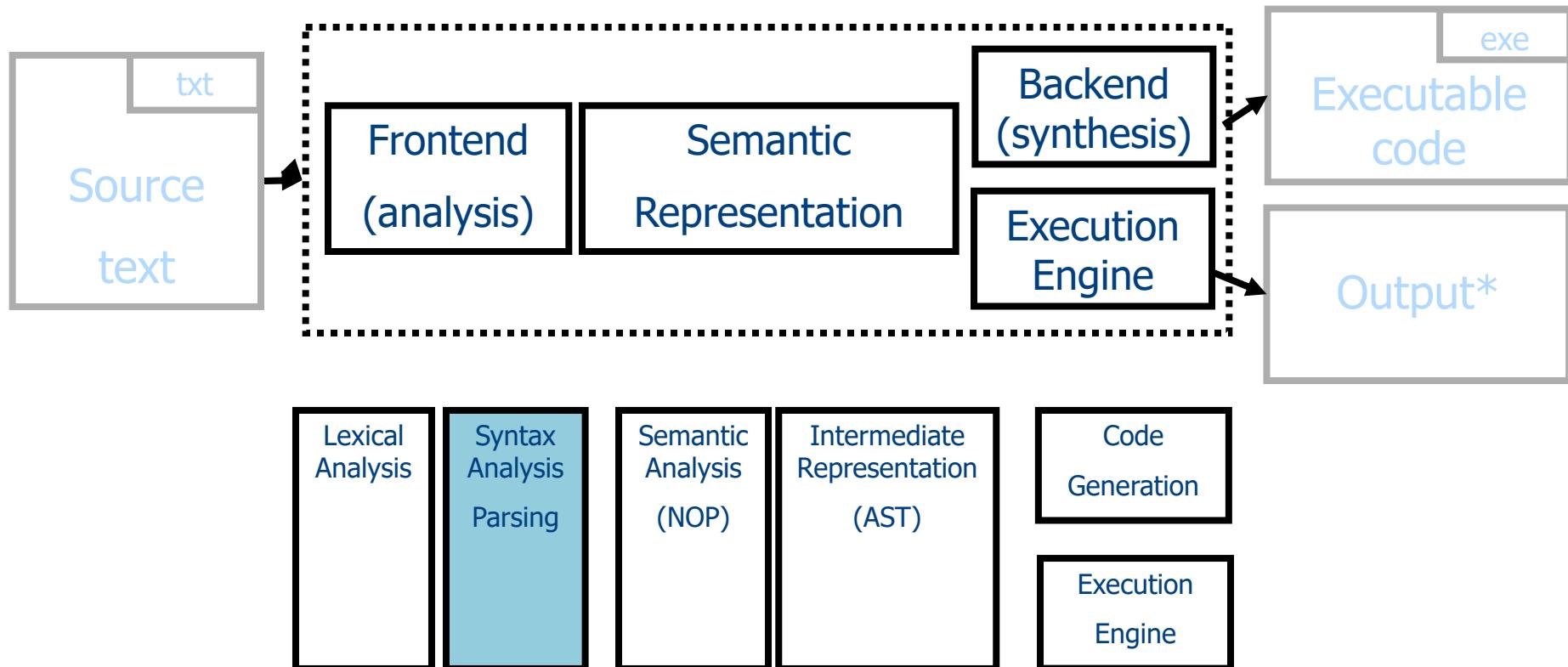
Lexical Analyzer

```
#include "lex.h"
token_type Token; // Global variable

void get_next_token(void) {
    int ch;
    do {
        ch = getchar();
        if (ch < 0) {
            Token.class = EoF; Token.repr = '#';
            return;
        }
    } while (Layout_char(ch));
    if ('0' <= ch && ch <= '9') {Token.class = DIGIT;}
    else {Token.class = ch;}
    Token.repr = ch;
}

static int Layout_char(int ch) {
    switch (ch) {
        case ' ': case '\t': case '\n': return 1;
        default: return 0;
    }
}
```

Structure of toy Compiler / interpreter



* Programs in our PL do not take input

Parser

- Invokes lexical analyzer
- Reports syntax errors
- Constructs AST

Source Language

- Fully parenthesized expressions
- Arguments can be a single digit

✓ $(4 + (3 * 9))$

X $3 + 4 + 5$

X $(12 + 3)$

expression \rightarrow digit | ‘(‘ expression operator expression ‘)’

operator \rightarrow ‘+’ | ‘*’

digit \rightarrow ‘0’ | ‘1’ | ‘2’ | ‘3’ | ‘4’ | ‘5’ | ‘6’ | ‘7’ | ‘8’ | ‘9’

Driver for the Toy Compiler

```
#include    "parser.h"      /* for type AST_node */
#include    "backend.h"     /* for Process() */
#include    "error.h"       /* for Error() */

int main(void) {
    AST_node *icode;

    if (!Parse_program(&icode)) Error("No top-level expression");
    Process(icode);

    return 0;
}
```

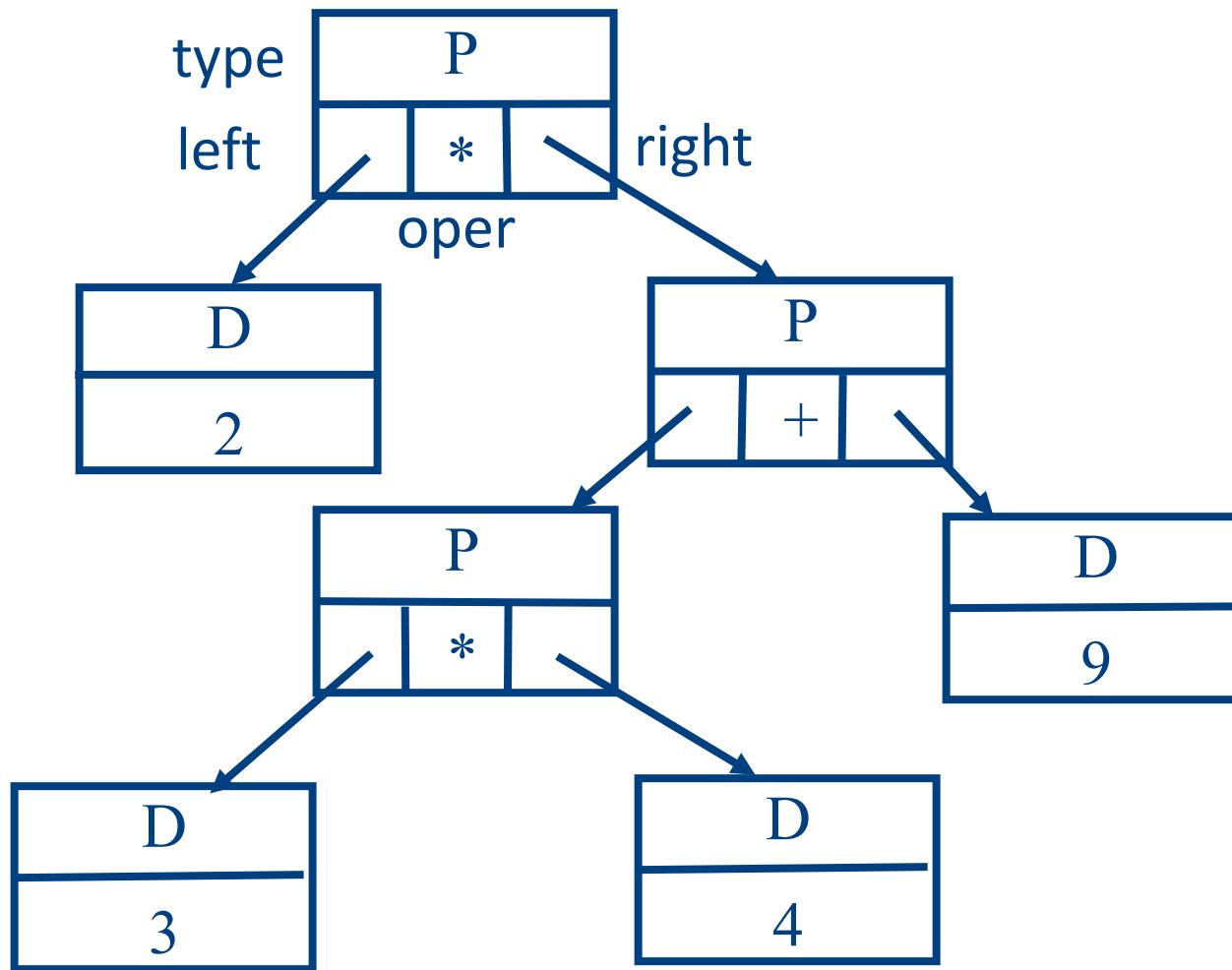
Parser Header File

```
typedef int Operator;

typedef struct _expression {
    char type;                      /* 'D' or 'P' */
    int value;                      /* for 'D' type expression */
    struct _expression *left, *right; /* for 'P' type expression */
    Operator oper;                  /* for 'P' type expression */
} Expression;

typedef Expression AST_node;          /* the top node is an Expression */
extern int Parse_program(AST_node **);
```

AST for $(2 * ((3 * 4) + 9))$



lex.h: Header File for Lexical Analysis

```
/* Define class constants */
/* Integers are used to encode characters + special codes */
/* Values 0-255 are reserved for ASCII characters */
#define EoF      256
#define DIGIT    257
typedef struct {
    int class;
    char repr;} Token_type;

extern Token_type Token;
extern void get_next_token(void);
```

Parser Environment

```
#include "lex.h", "error.h", "parser.h"

static Expression *new_expression(void) {
    return (Expression *)malloc(sizeof(Expression));
}

static int Parse_operator(Operator *oper_p);
static int Parse_expression(Expression **expr_p);
int Parse_program(AST_node **icode_p) {
    Expression *expr;
    get_next_token();           /* start the lexical analyzer */
    if (Parse_expression(&expr)) {
        if (Token.class != EoF) {
            Error("Garbage after end of program");
        }
        *icode_p = expr;
        return 1;
    }
    return 0;
}
```

Top-Down Parsing

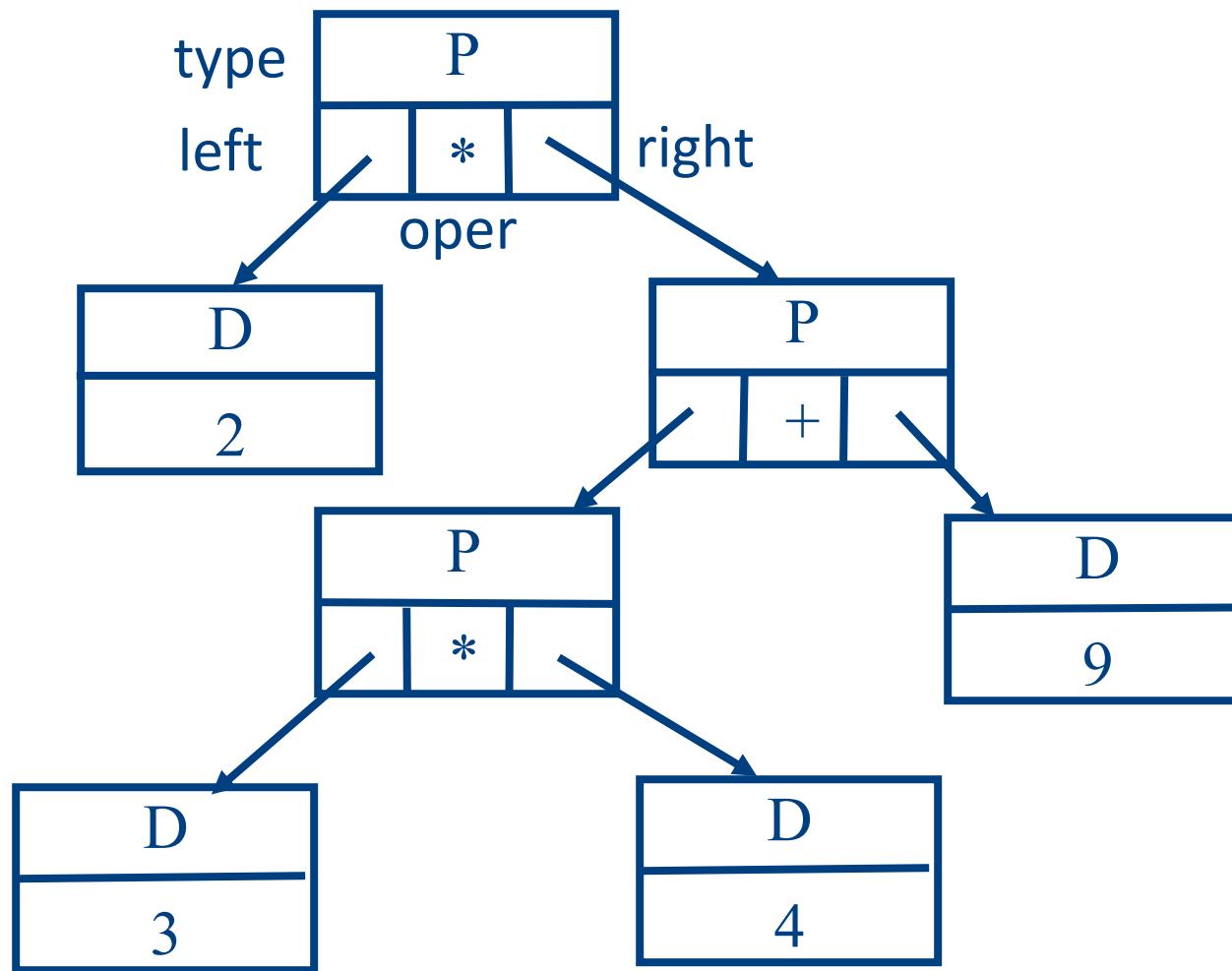
- Optimistically build the tree from the root to leaves
- For every $P \rightarrow A_1 A_2 \dots A_n \mid B_1 B_2 \dots B_m$
 - If A_1 succeeds
 - If A_2 succeeds & A_3 succeeds & ...
 - Else fail
 - Else if B_1 succeeds
 - If B_2 succeeds & B_3 succeeds & ..
 - Else fail
 - Else fail
- Recursive descent parsing
 - Simplified: no backtracking
- Can be applied for certain grammars

Parser

```
static int Parse_expression(Expression **expr_p) {
    Expression *expr = *expr_p = new_expression();
    if (Token.class == DIGIT) {
        expr->type = 'D'; expr->value = Token.repr - '0';
        get_next_token(); return 1;
    }
    if (Token.class == '(') {
        expr->type = 'P'; get_next_token();
        if (!Parse_expression(&expr->left)) { Error("Missing expression"); }
        if (!Parse_operator(&expr->oper)) { Error("Missing operator"); }
        if (!Parse_expression(&expr->right)) { Error("Missing expression"); }
        if (Token.class != ')') { Error("Missing )"); }
        get_next_token();
        return 1;
    }
    /* failed on both attempts */
    free_expression(expr); return 0;
}

static int Parse_operator(Operator *oper) {
    if (Token.class == '+') {
        *oper = '+'; get_next_token(); return 1;
    }
    if (Token.class == '*') {
        *oper = '*'; get_next_token(); return 1;
    }
    return 0;
}
```

AST for $(2 * ((3 * 4) + 9))$



Errora

Parser

- $(Z + 3)$
- $((9 + 3)$
- $(9 + 3))$
- 33

Lexer

- ? $(Z + 3)$
- ? $((9 + 3)$
- ? $(9 + 3))$
- ? 33

Errora

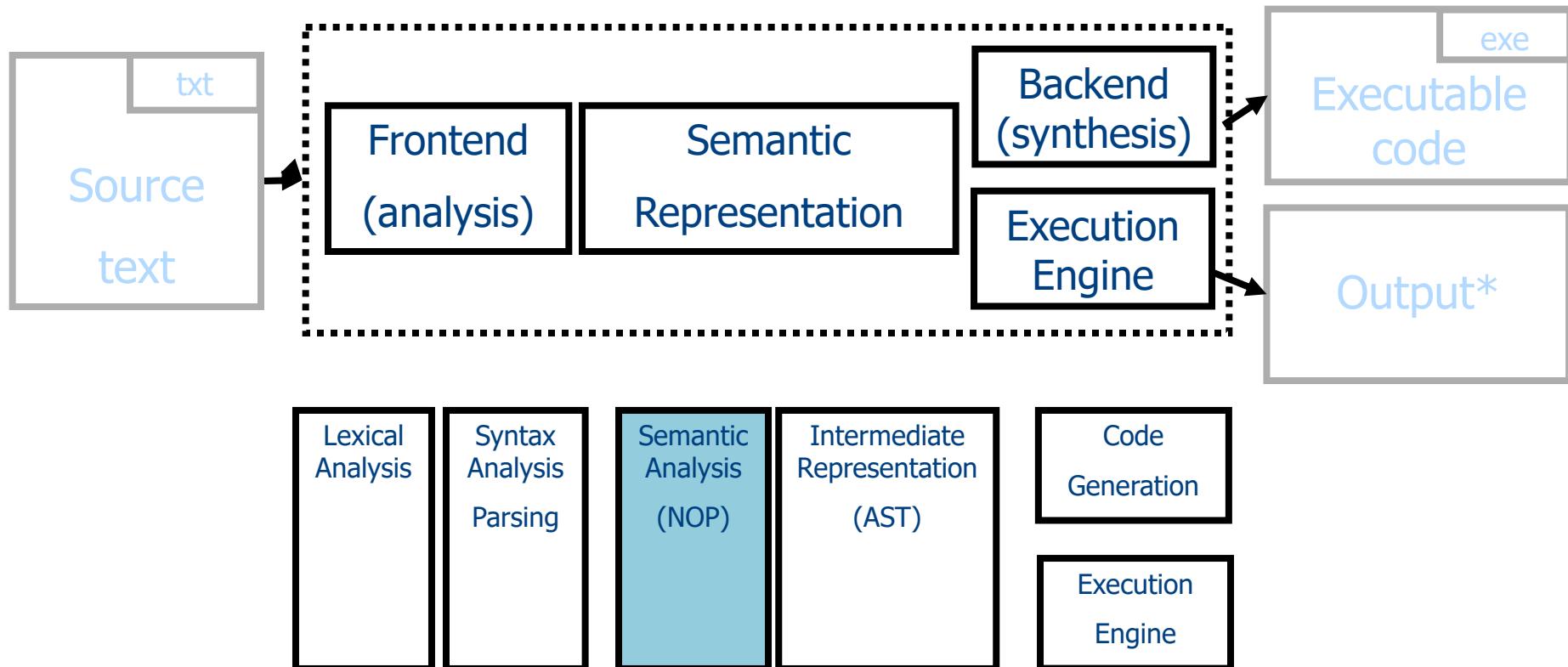
Parser

- $(Z + 3)$
- $((9 + 3)$
- $(9 + 3))$
- 33

Lexer

$(Z + 3)$
 $\times ((9 + 3)$
 $\times (9 + 3))$
 ~ 33

Structure of toy Compiler / interpreter

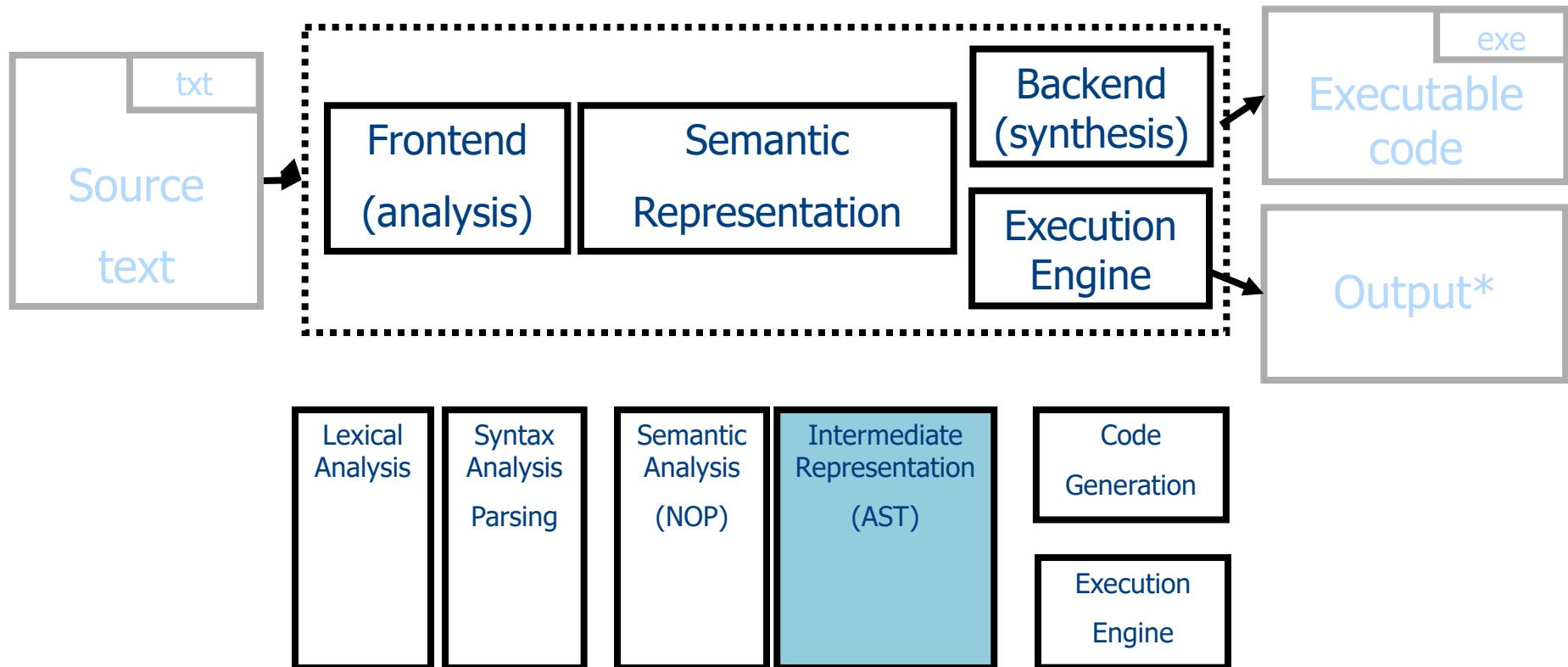


* Programs in our PL do not take input

Semantic Analysis

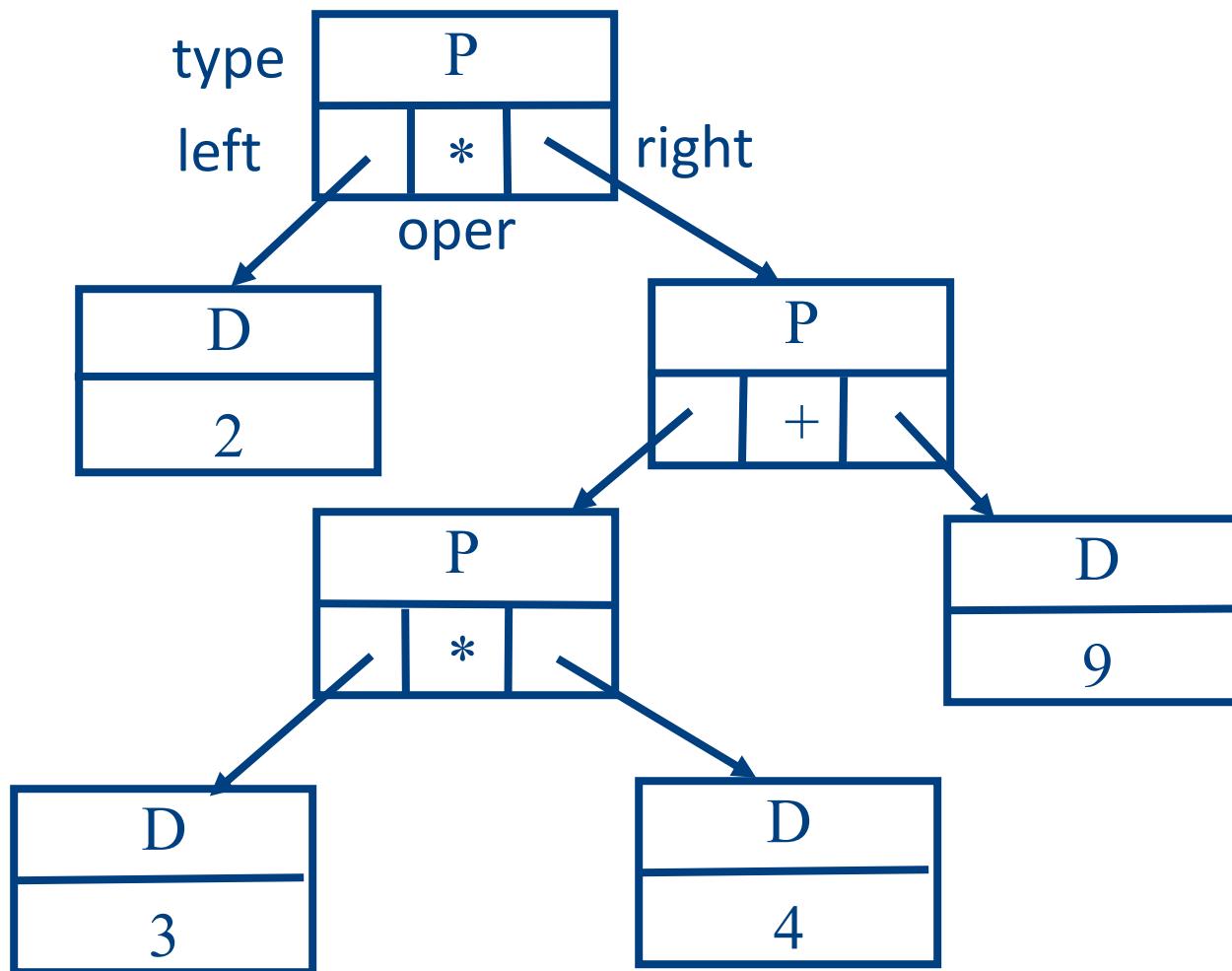
- Trivial in our case
- No identifiers
- No procedure / functions
- A single type for all expressions

Structure of toy Compiler / interpreter



* Programs in our PL do not take input

Intermediate Representation

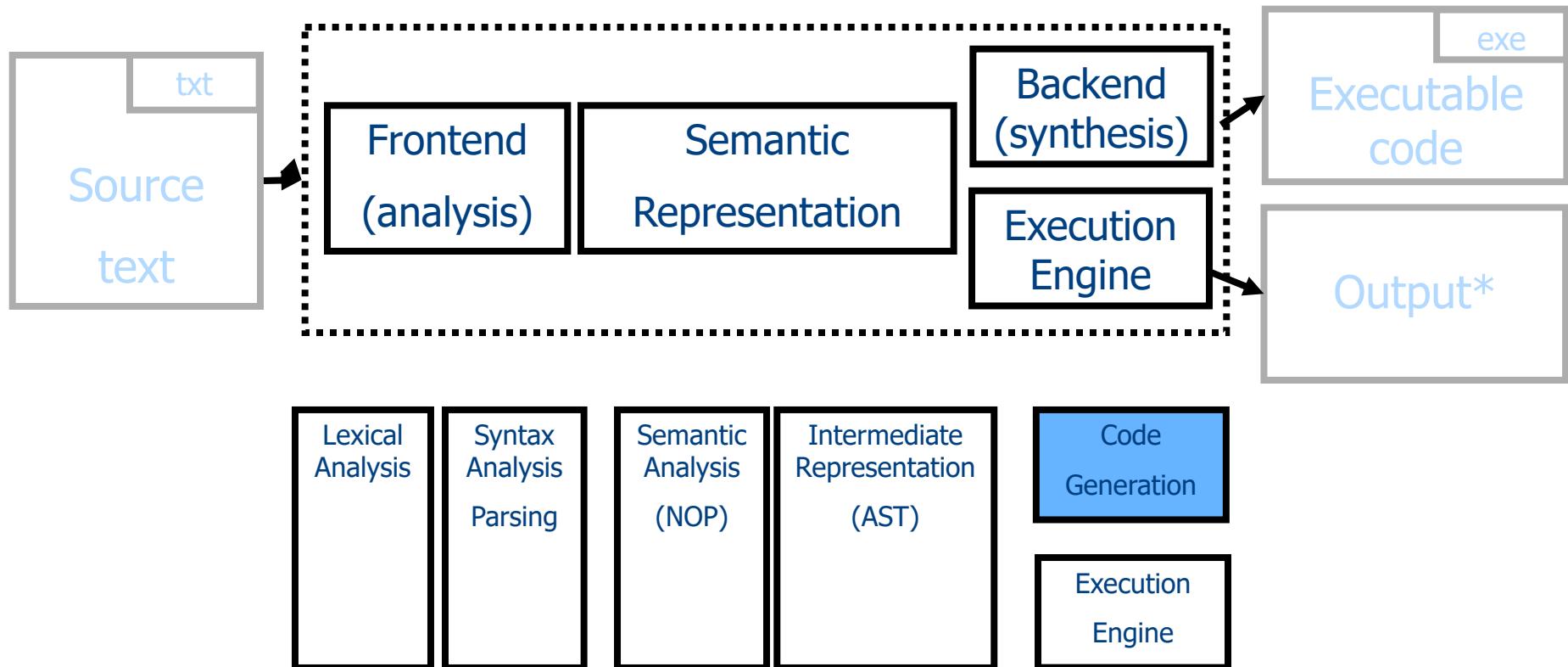


Alternative IR: 3-Address Code

```
L1:  
    _t0=a  
    _t1=b  
    _t2=_t0*_t1  
    _t3=d  
    _t4=_t2-_t3  
    GOTO L1
```

“Simple Basic-like programming language”

Structure of toy Compiler / interpreter



* Programs in our PL do not take input

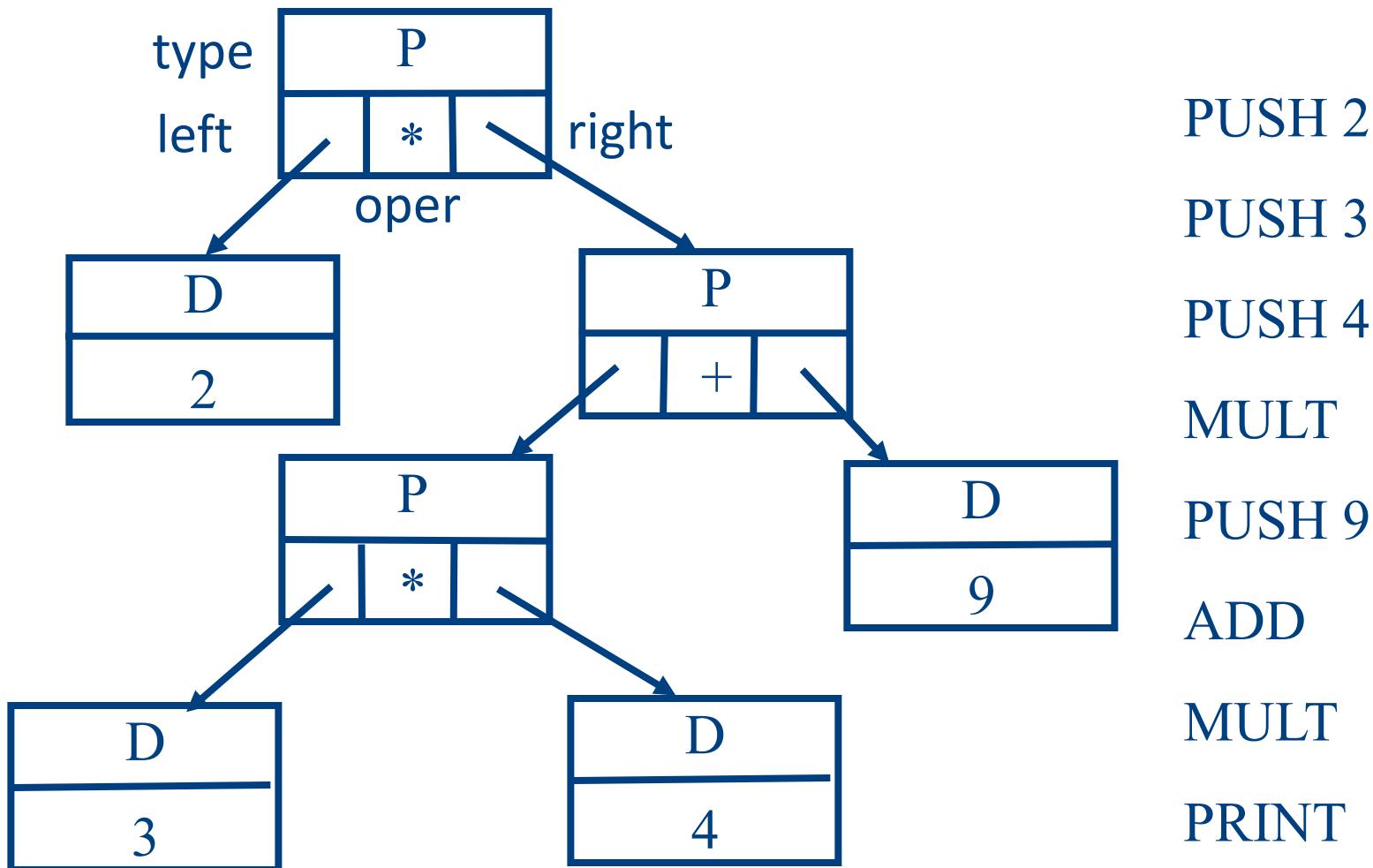
Code generation

- Stack based machine
- Four instructions
 - PUSH n
 - ADD
 - MULT
 - PRINT

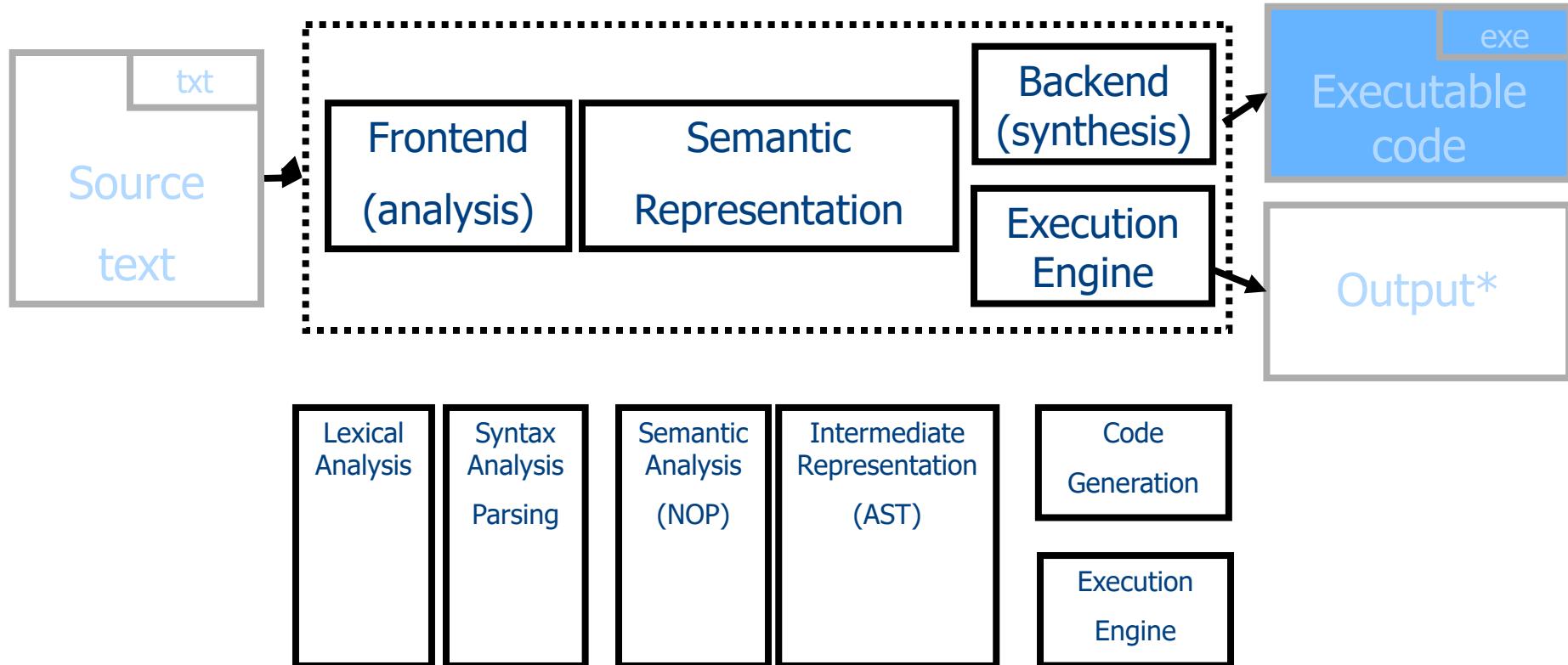
Code generation

```
#include    "parser.h"
#include    "backend.h"
static void Code_gen_expression(Expression *expr) {
    switch (expr->type) {
    case 'D':
        printf("PUSH %d\n", expr->value);
        break;
    case 'P':
        Code_gen_expression(expr->left);
        Code_gen_expression(expr->right);
        switch (expr->oper) {
        case '+': printf("ADD\n"); break;
        case '*': printf("MULT\n"); break;
        }
        break;
    }
}
void Process(AST_node *icode) {
    Code_gen_expression(icode); printf("PRINT\n");
}
```

Compiling $(2*((3*4)+9))$



Executing Compiled Program



* Programs in our PL do not take input

Generated Code Execution



PUSH 2

Stack

Stack'

PUSH 3

2

PUSH 4

MULT

PUSH 9

ADD

MULT

PRINT

Generated Code Execution

	Stack	Stack'
PUSH 2		
PUSH 3	2	3
PUSH 4		2
MULT		
PUSH 9		
ADD		
MULT		
PRINT		

Generated Code Execution

	Stack	Stack'
PUSH 2		
PUSH 3	3	4
PUSH 4	2	3
MULT		2
PUSH 9		
ADD		
MULT		
PRINT		



Generated Code Execution

	Stack	Stack'
PUSH 2		
PUSH 3	4	12
PUSH 4	3	2
→ MULT	2	
PUSH 9		
ADD		
MULT		
PRINT		

Generated Code Execution

	Stack	Stack'
PUSH 2		
PUSH 3	12	9
PUSH 4	2	12
MULT		2
 PUSH 9		
ADD		
MULT		
PRINT		

Generated Code Execution

	Stack	Stack'
PUSH 2		
PUSH 3	9	21
PUSH 4	12	2
MULT	2	
PUSH 9		
→ ADD		
MULT		
PRINT		

Generated Code Execution

	Stack	Stack'
PUSH 2		
PUSH 3	21	42
PUSH 4	2	
MULT		
PUSH 9		
ADD		
→ MULT		
PRINT		

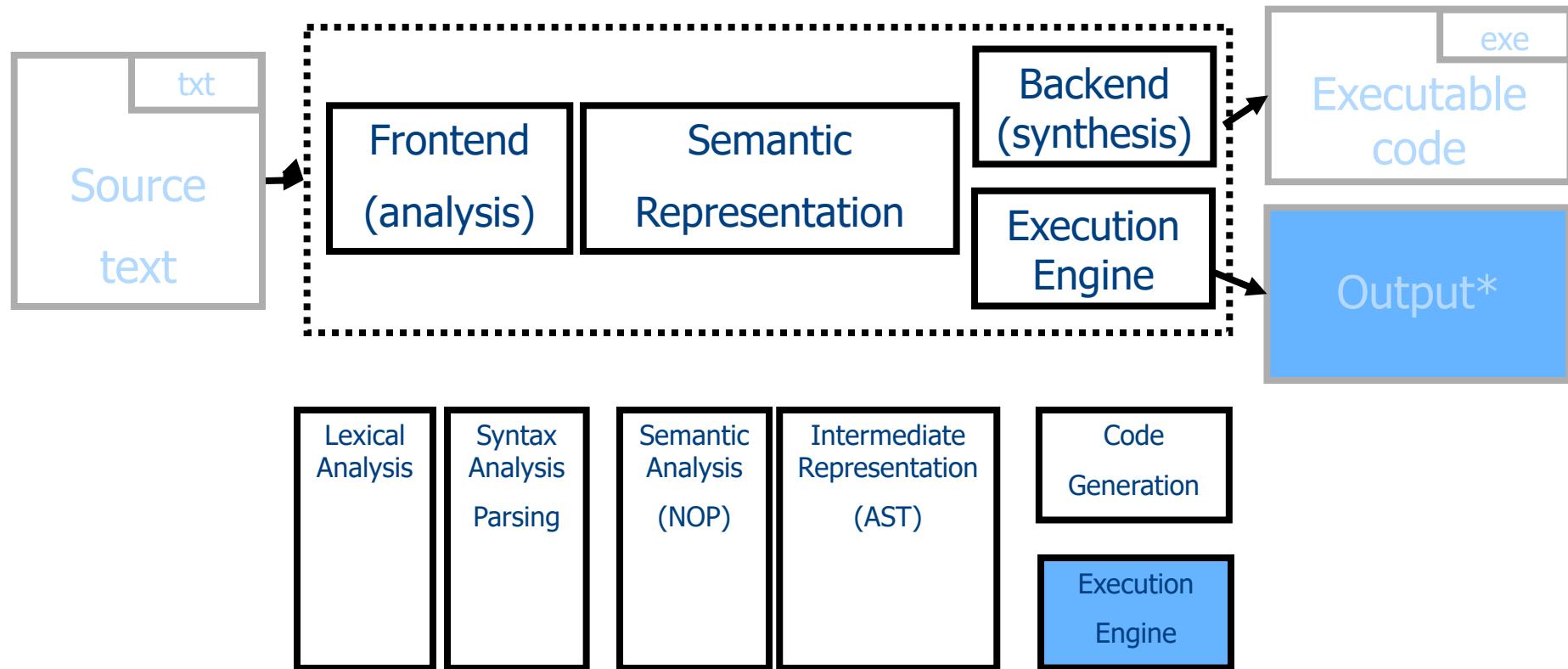
Generated Code Execution

	Stack	Stack'
PUSH 2		
PUSH 3	42	
PUSH 4		
MULT		
PUSH 9		
ADD		
MULT		
PRINT		

Shortcuts

- Avoid generating machine code
 - Use assembler
- Generate C code

Structure of toy Compiler / interpreter



* Programs in our PL do not take input

Interpretation

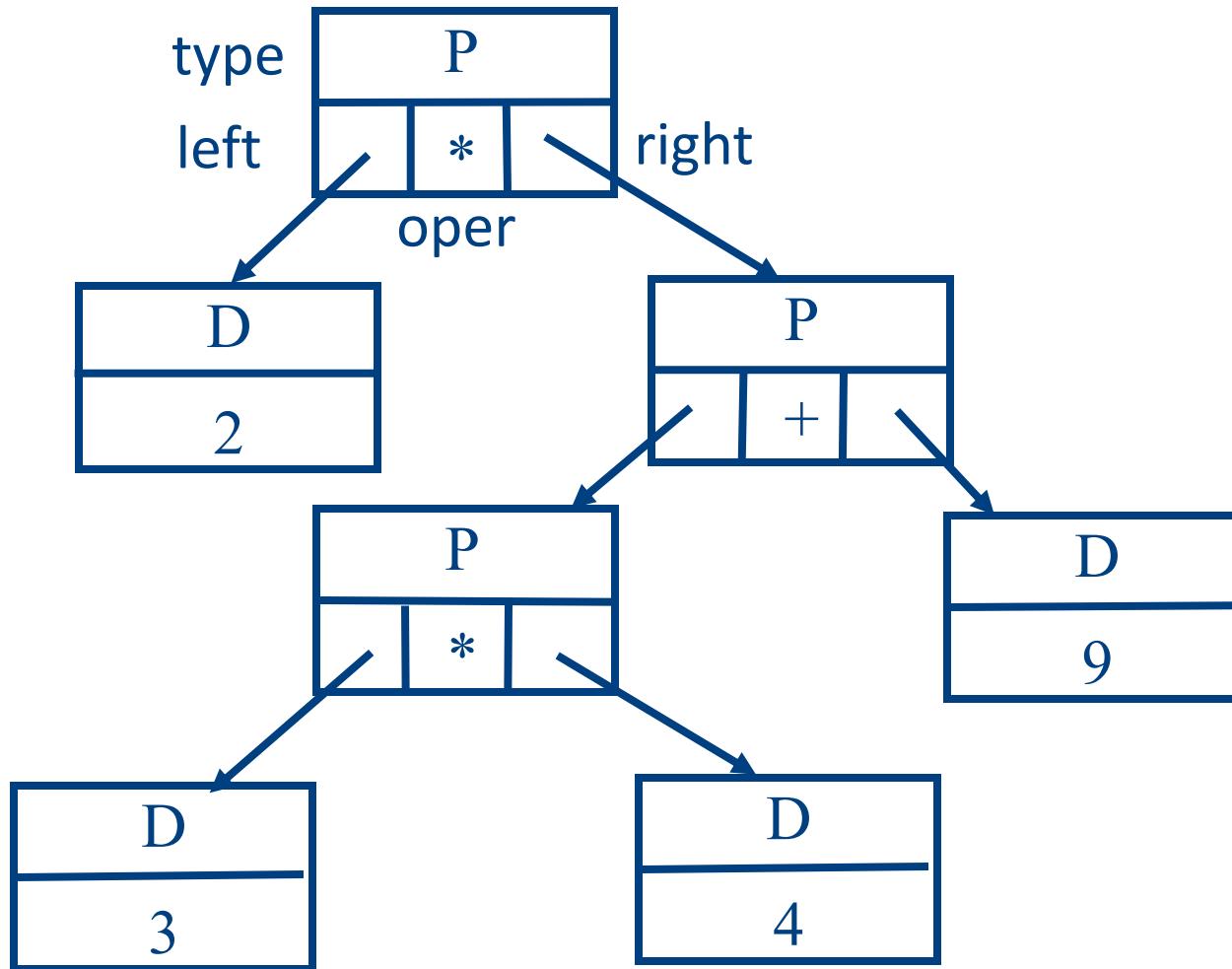
- Bottom-up evaluation of expressions
- The same interface of the compiler

```
#include "parser.h"
#include "backend.h"

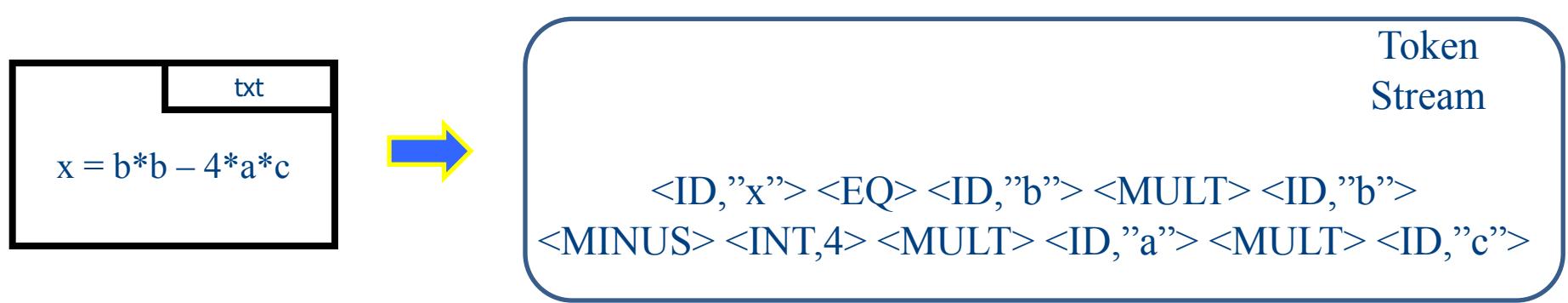
static int Interpret_expression(Expression *expr) {
    switch (expr->type) {
        case 'D':
            return expr->value;
            break;
        case 'P':
            int e_left = Interpret_expression(expr->left);
            int e_right = Interpret_expression(expr->right);
            switch (expr->oper) {
                case '+': return e_left + e_right;
                case '*': return e_left * e_right;
                break;
            }
    }
}

void Process(AST_node *icode) {
    printf("%d\n", Interpret_expression(icode));
}
```

Interpreting $(2*((3*4)+9))$

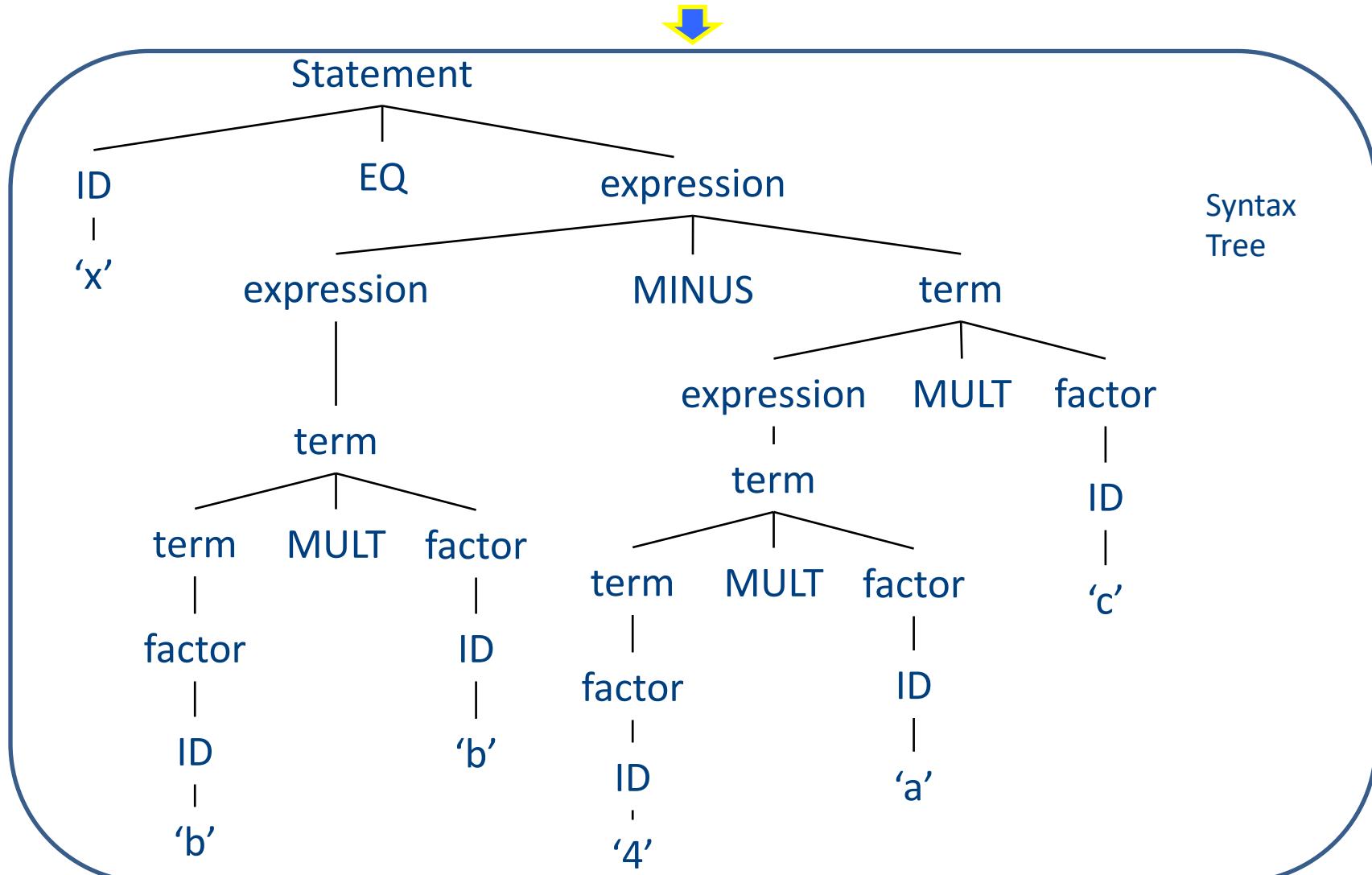


Summary: Journey inside a compiler



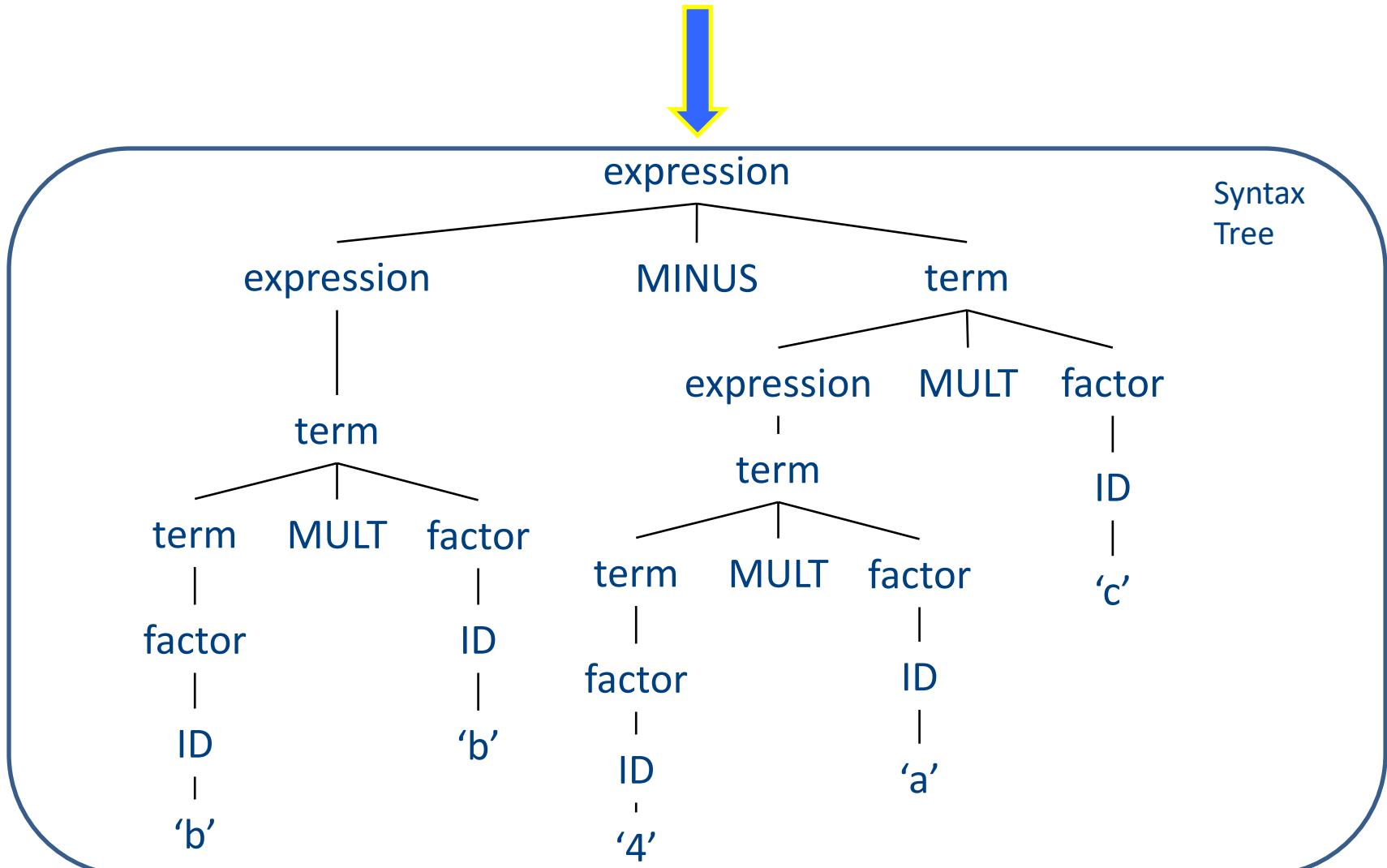
Summary: Journey inside a compiler

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

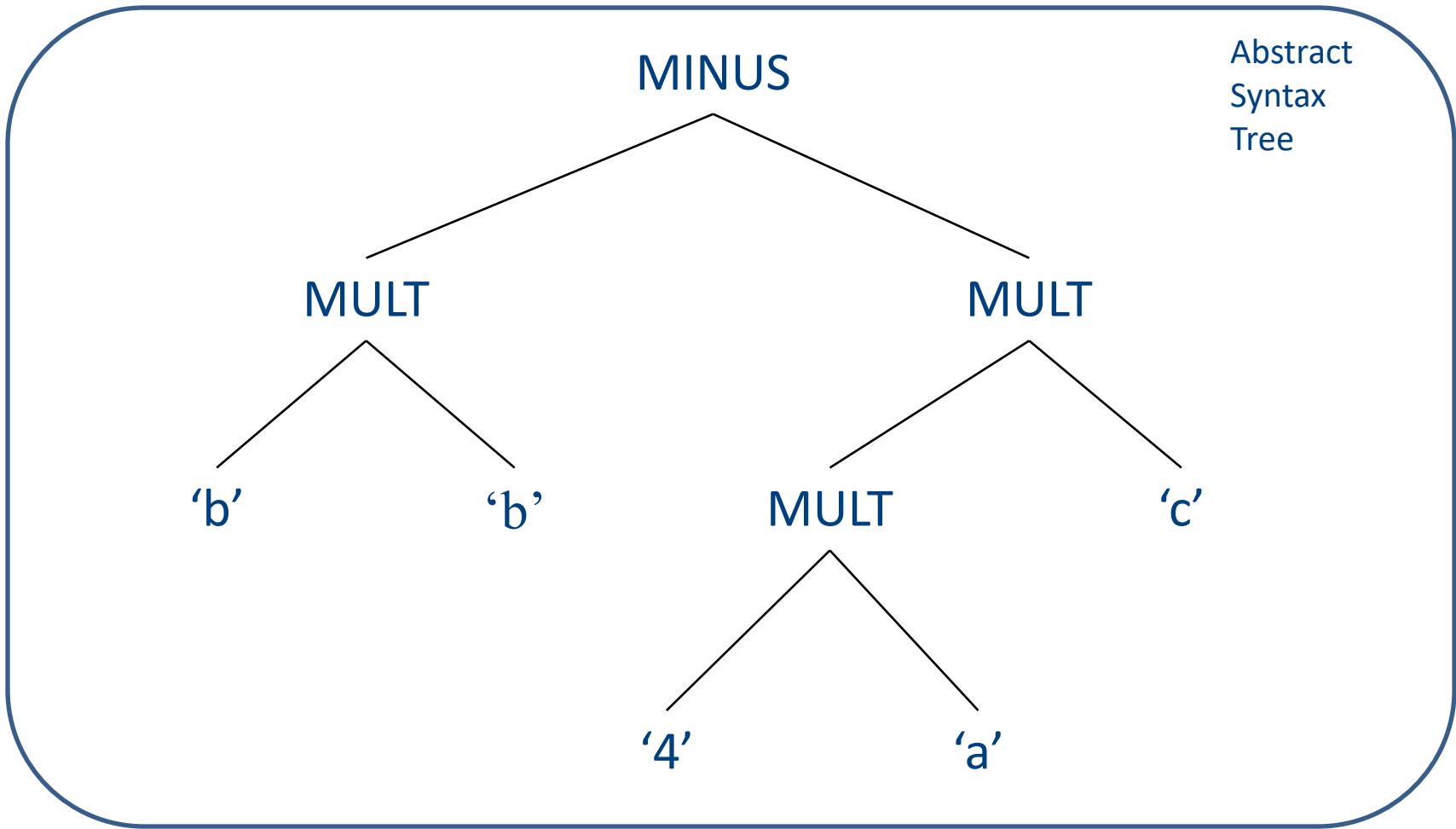


Summary: Journey inside a compiler

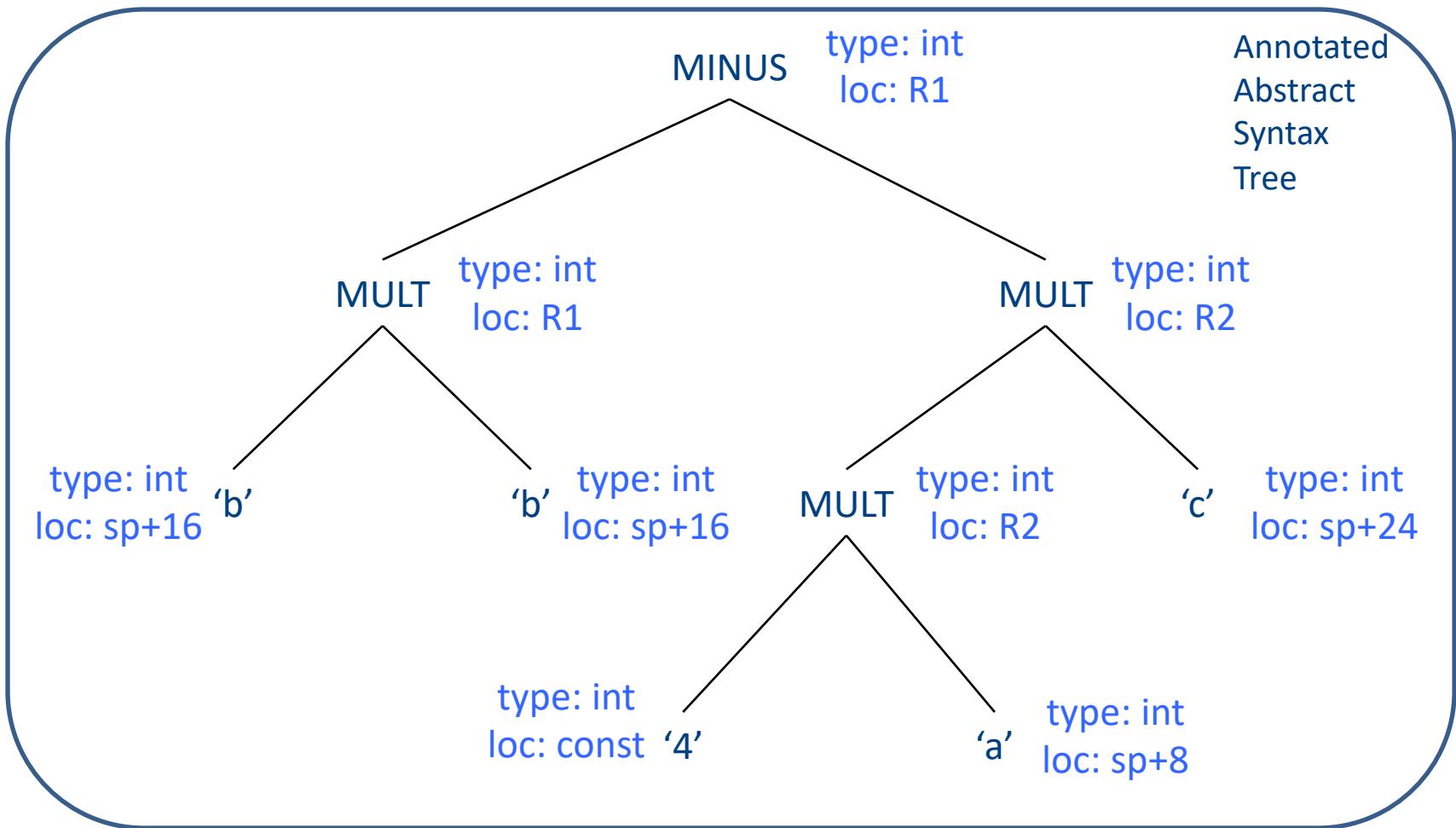
<ID,"x"> <EQ> <ID,"b"> <MULT> <ID,"b"> <MINUS> <INT,4> <MULT> <ID,"a"> <MULT> <ID,"c">



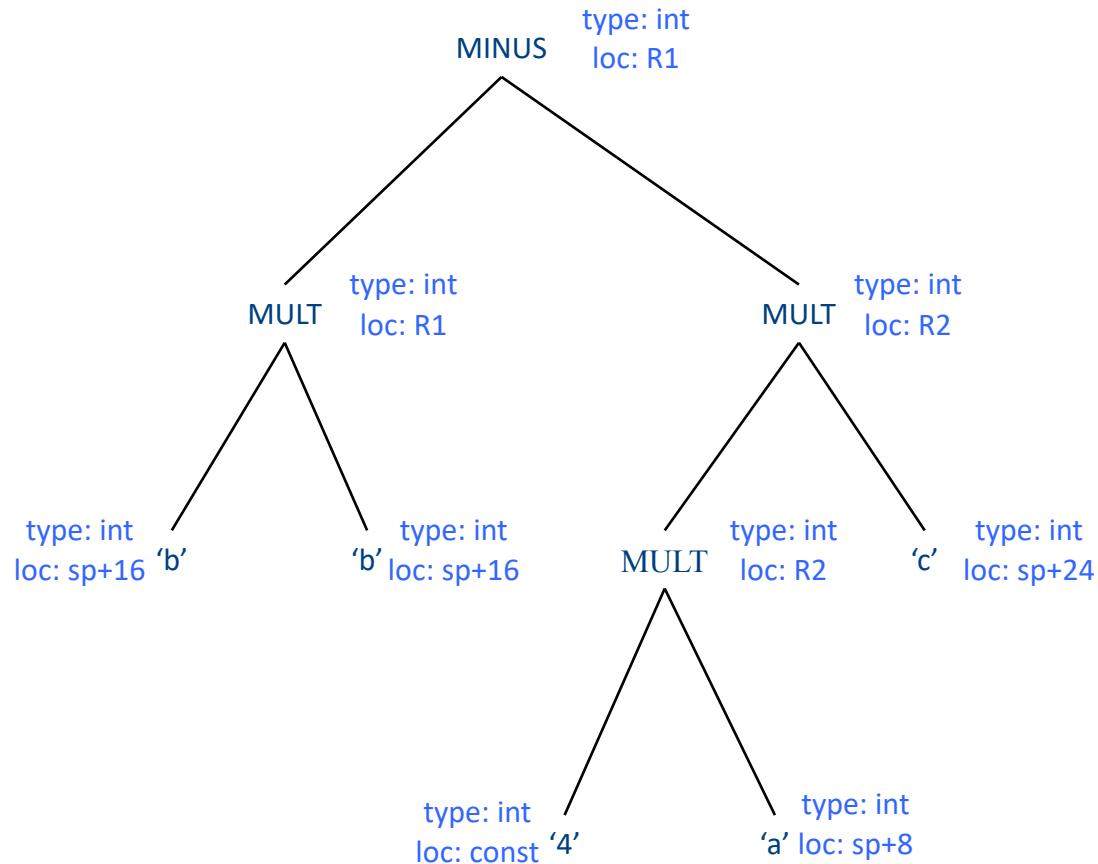
Summary: Journey inside a compiler



Summary: Journey inside a compiler



Journey inside a compiler



Intermediate Representation

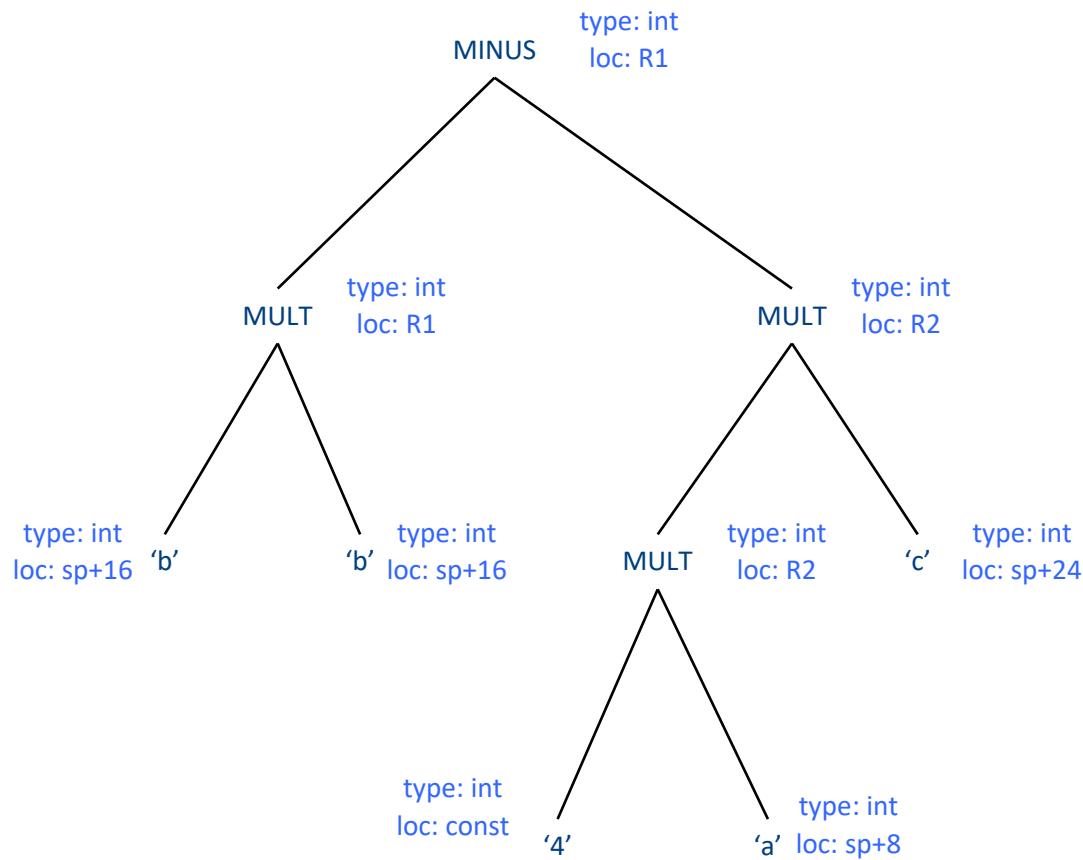
$$R2 = 4*a$$

$$R1=b*b$$

$$R2= R2*c$$

$$R1=R1-R2$$

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)`

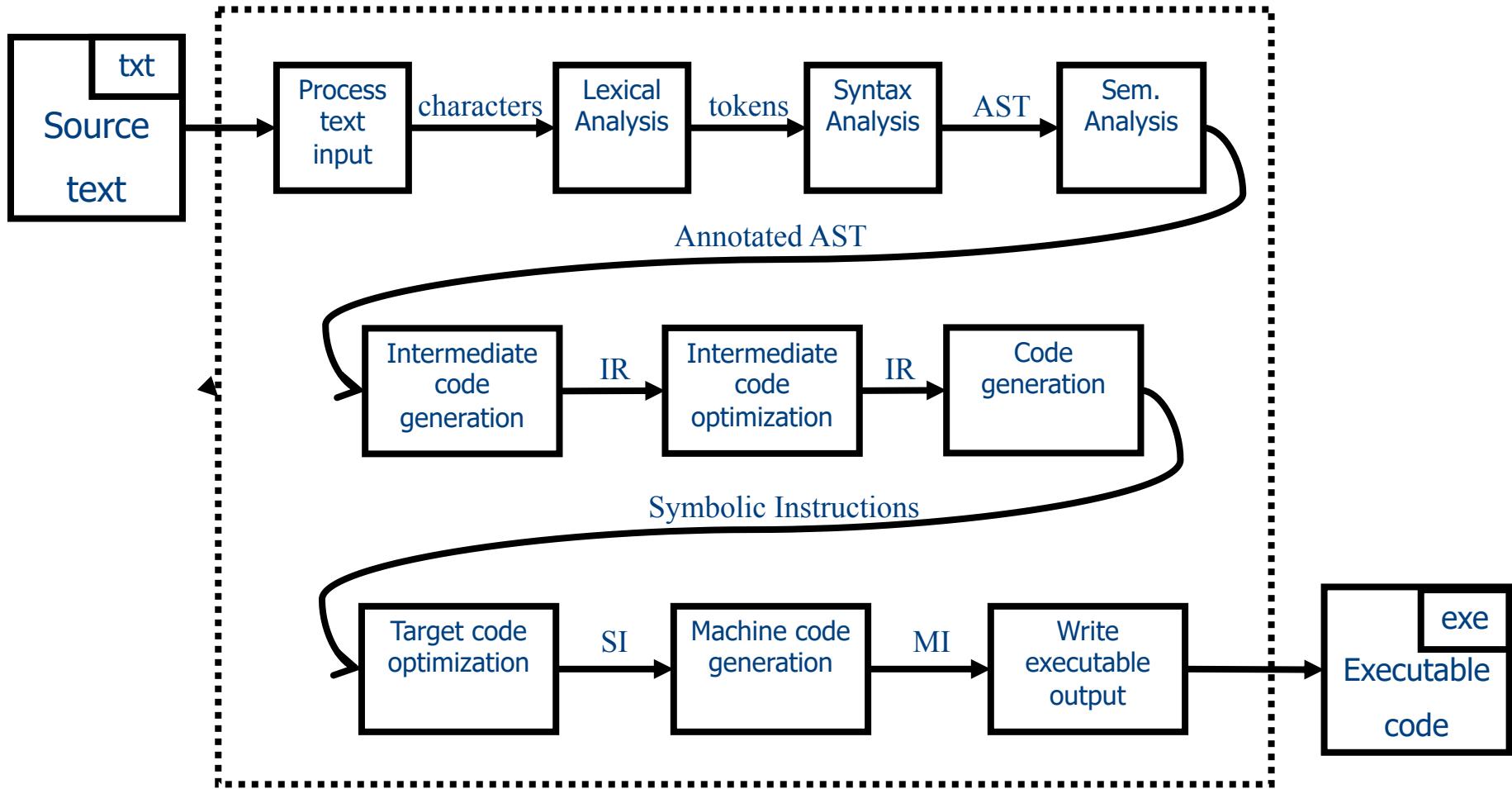
`MUL R2,(sp+24)`

`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 (why?)
 - Divergence is a challenge

The Real Anatomy of a Compiler



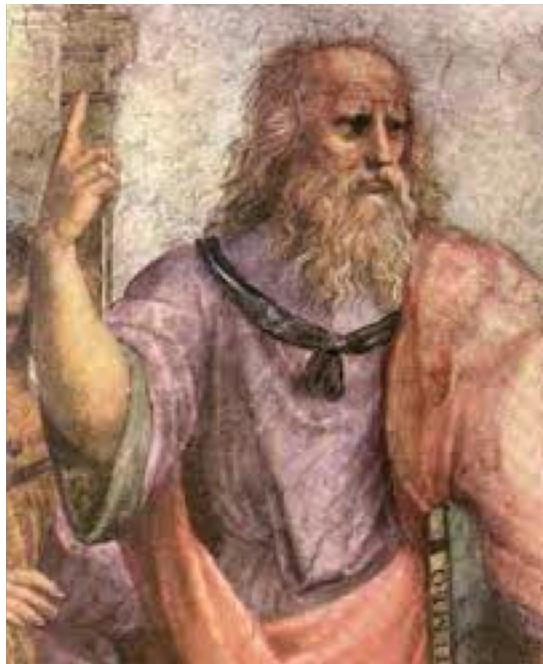
Optimizations

- “Optimal code” is out of reach
 - many problems are undecidable or too expensive (NP-complete)
 - Use approximation and/or heuristics
- Loop optimizations: hoisting, unrolling, ...
- Peephole optimizations
- Constant propagation
 - Leverage compile-time information to save work at runtime (pre-computation)
- Dead code elimination
 - space
- ...

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

And on a More General Note



Course Goals

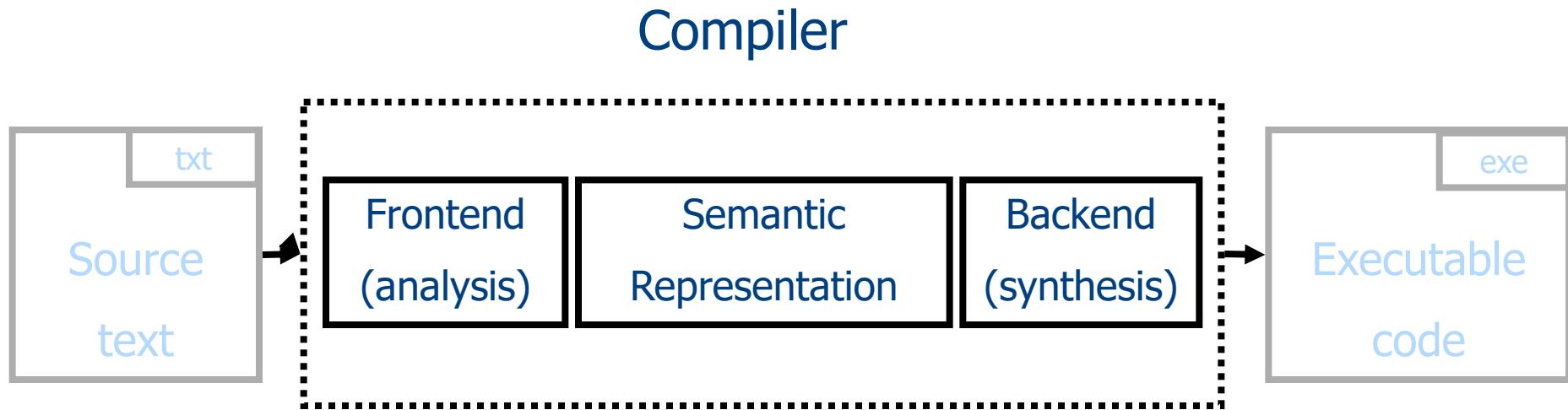
- What is a compiler
- How does it work
- (Reusable) techniques & tools
- Programming language implementation
 - runtime systems
- Execution environments
 - Assembly, linkers, loaders, OS

To Compilers, and Beyond ...

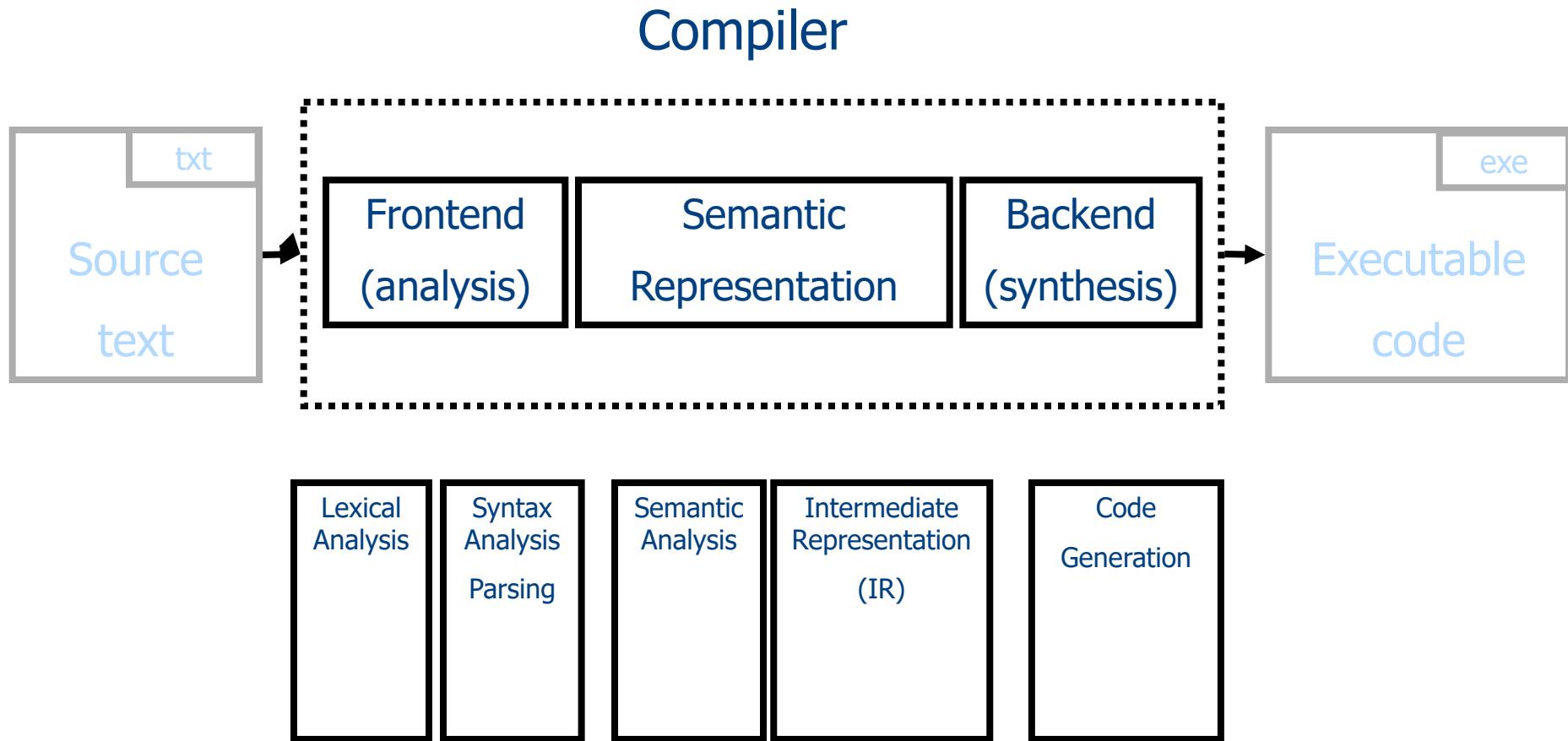
- Compiler construction is successful
 - Clear problem
 - Proper structure of the solution
 - Judicious use of formalisms
- Wider application
 - Many conversions can be viewed as compilation
- Useful algorithms



Conceptual Structure of a Compiler

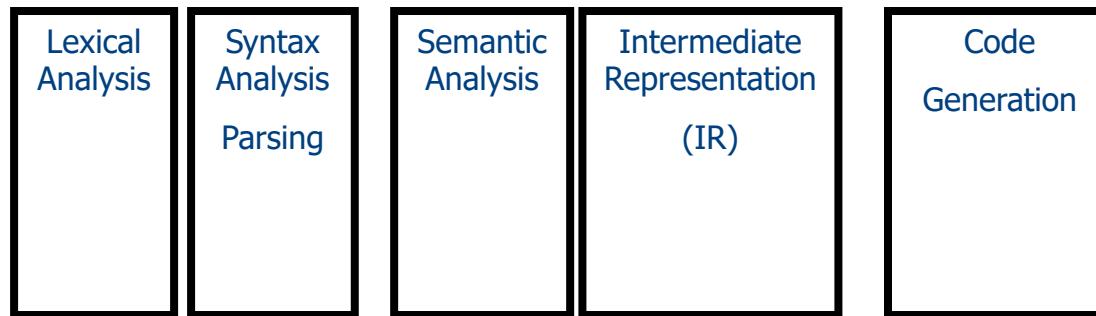


Conceptual Structure of a Compiler



Judicious use of formalisms

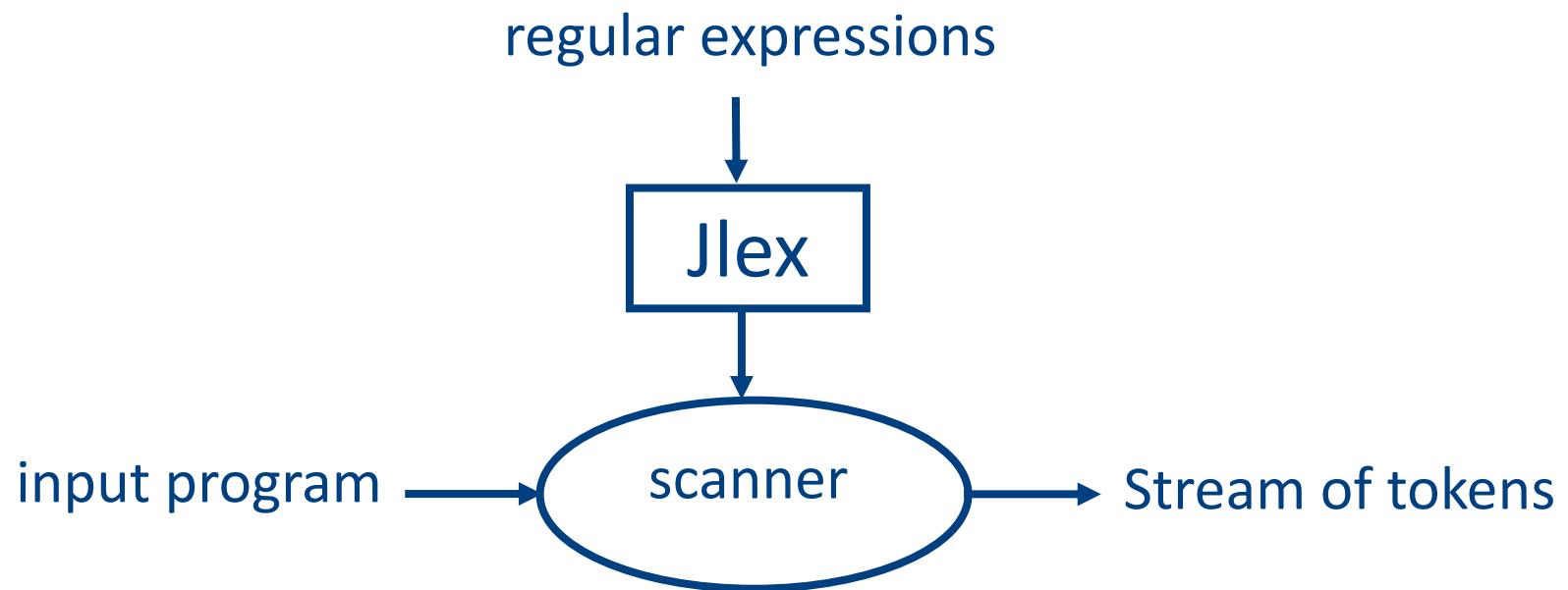
- Regular expressions (lexical analysis)
- Context-free grammars (syntactic analysis)
- Attribute grammars (context analysis)
- Code generator generators (dynamic programming)



- But also some nitty-gritty programming

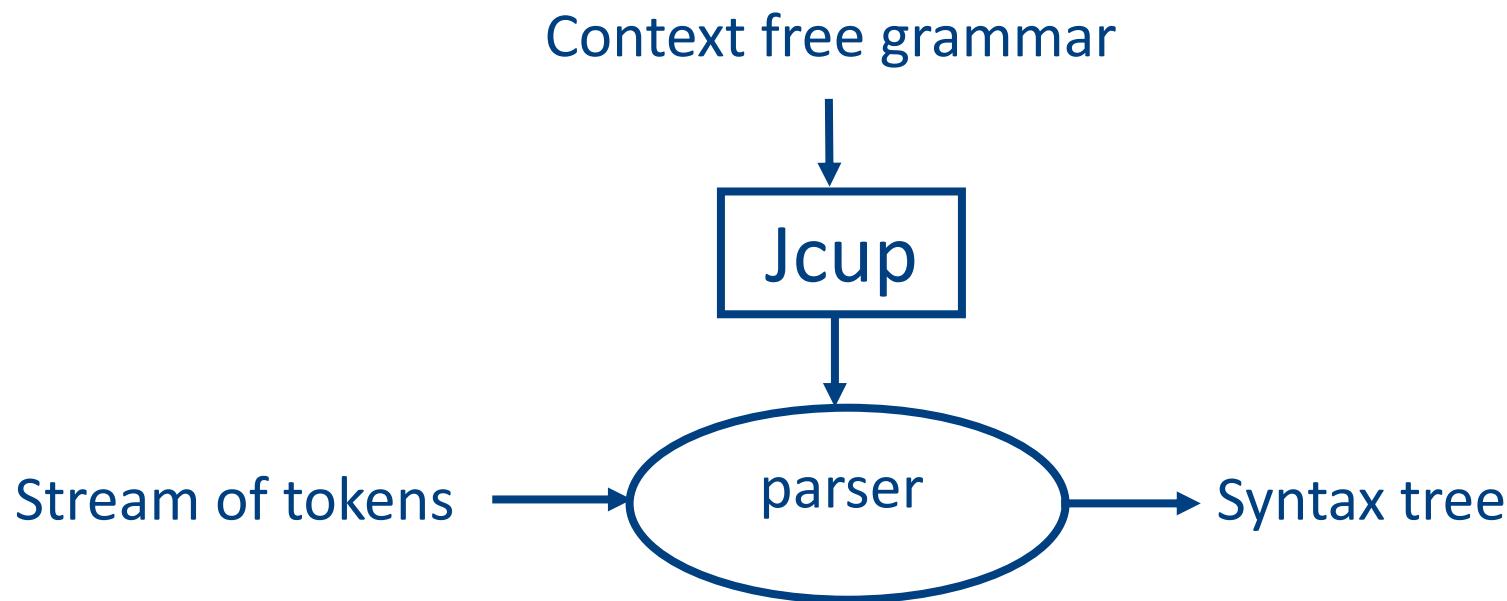
Use of program-generating tools

- Parts of the compiler are automatically generated from specification



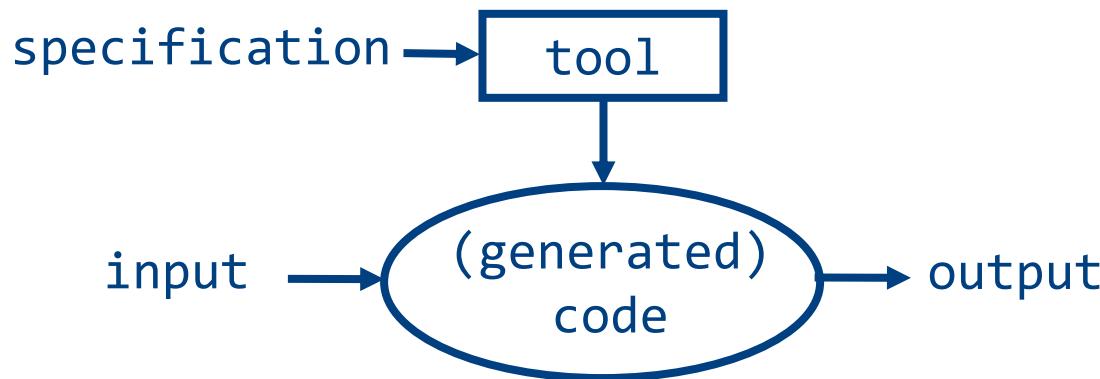
Use of program-generating tools

- Parts of the compiler are automatically generated from specification



Use of program-generating tools

- Simpler compiler construction
 - Less error prone
 - More flexible
- Use of pre-canned tailored code
 - Use of dirty program tricks
- Reuse of specification



Compiler Construction Toolset

- Lexical analysis generators
 - Lex, JLex
- Parser generators
 - Yacc, Jcup
- Syntax-directed translators
- Dataflow analysis engines

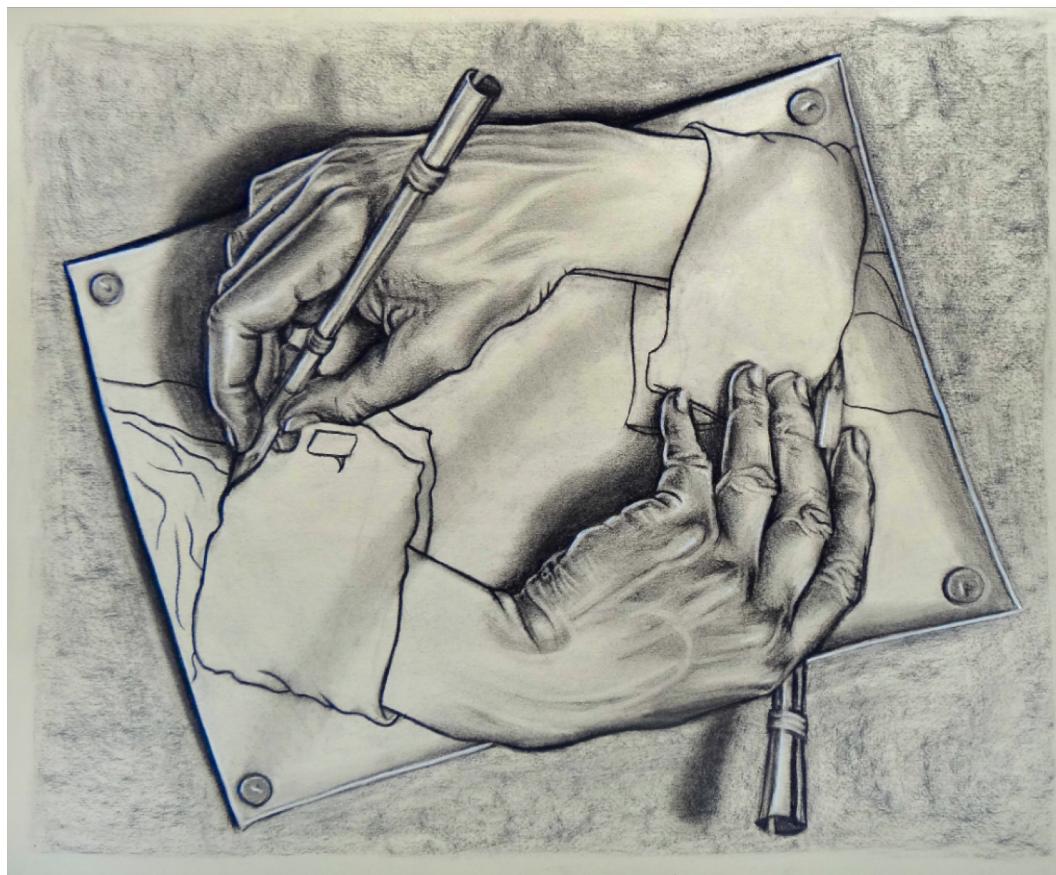
Wide applicability

- Structured data can be expressed using context free grammars
 - HTML files
 - Postscript
 - Tex/dvi files
 - ...

Generally useful algorithms

- Parser generators
- Garbage collection
- Dynamic programming
- Graph coloring

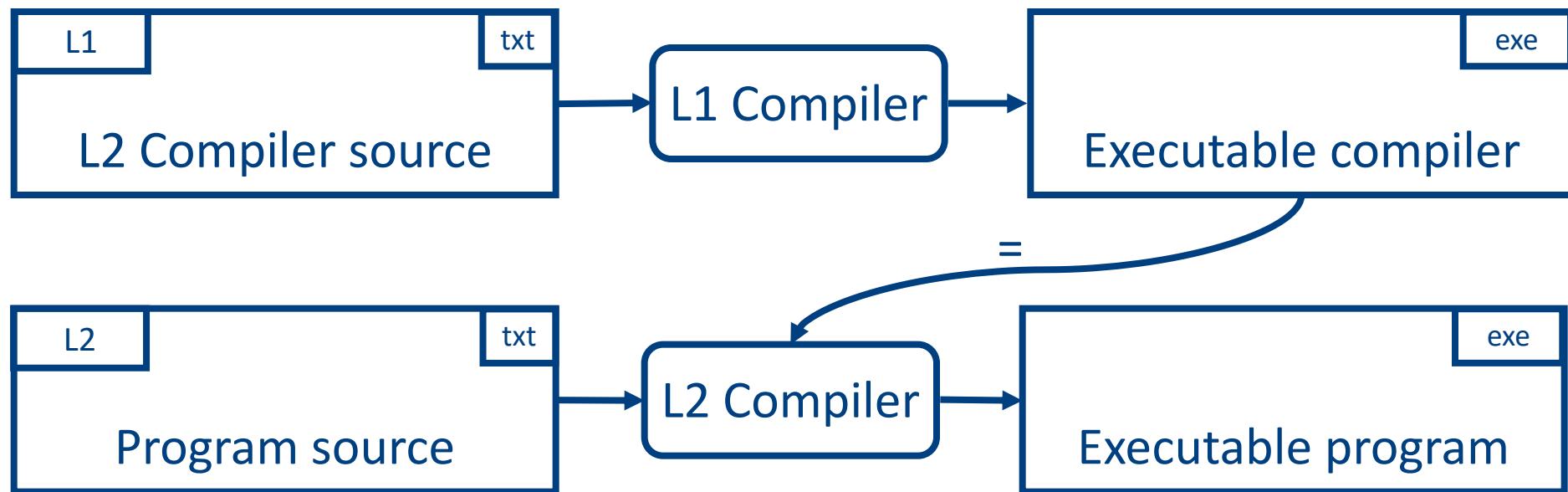
How to write a compiler?



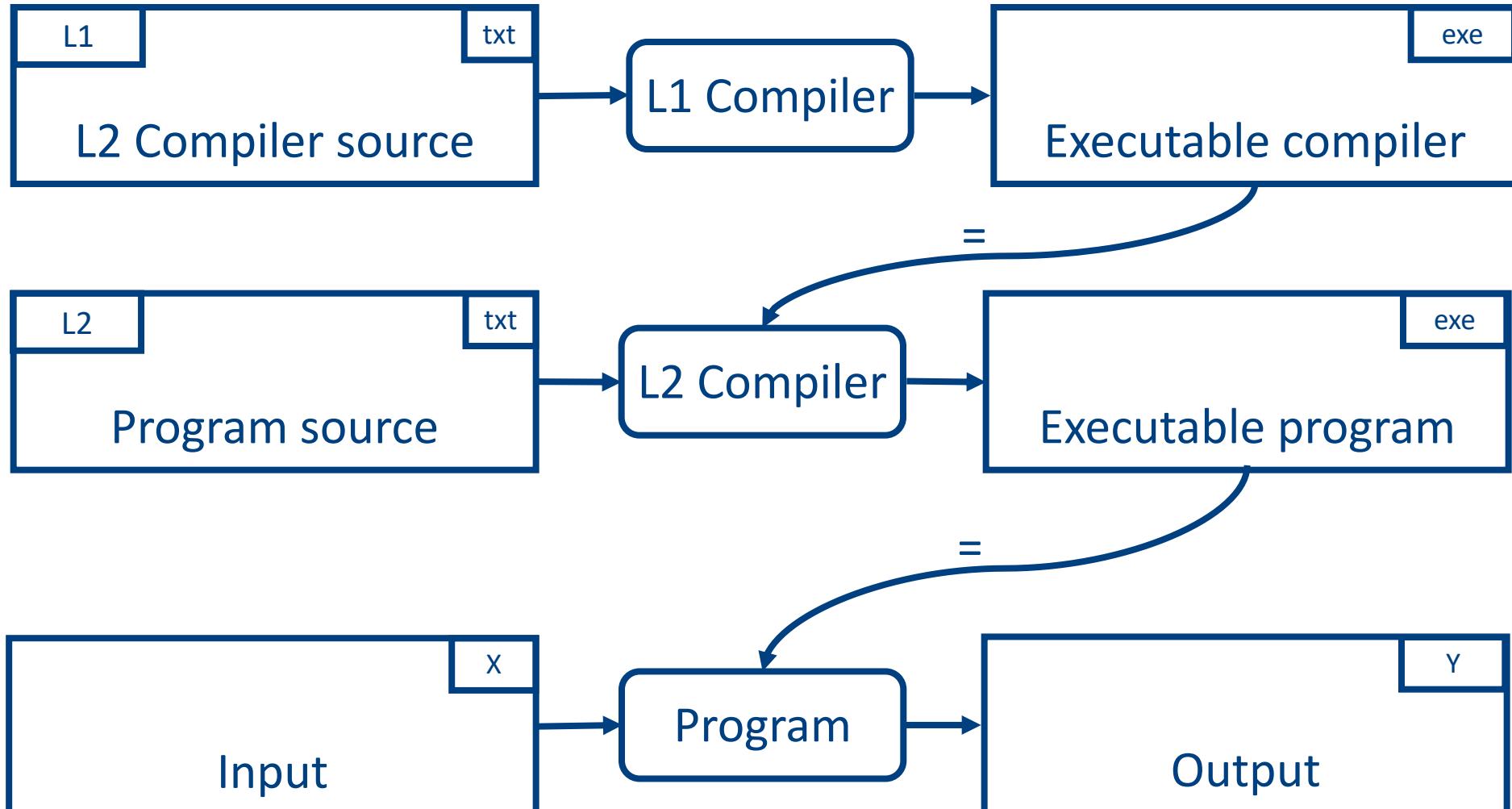
How to write a compiler?



How to write a compiler?



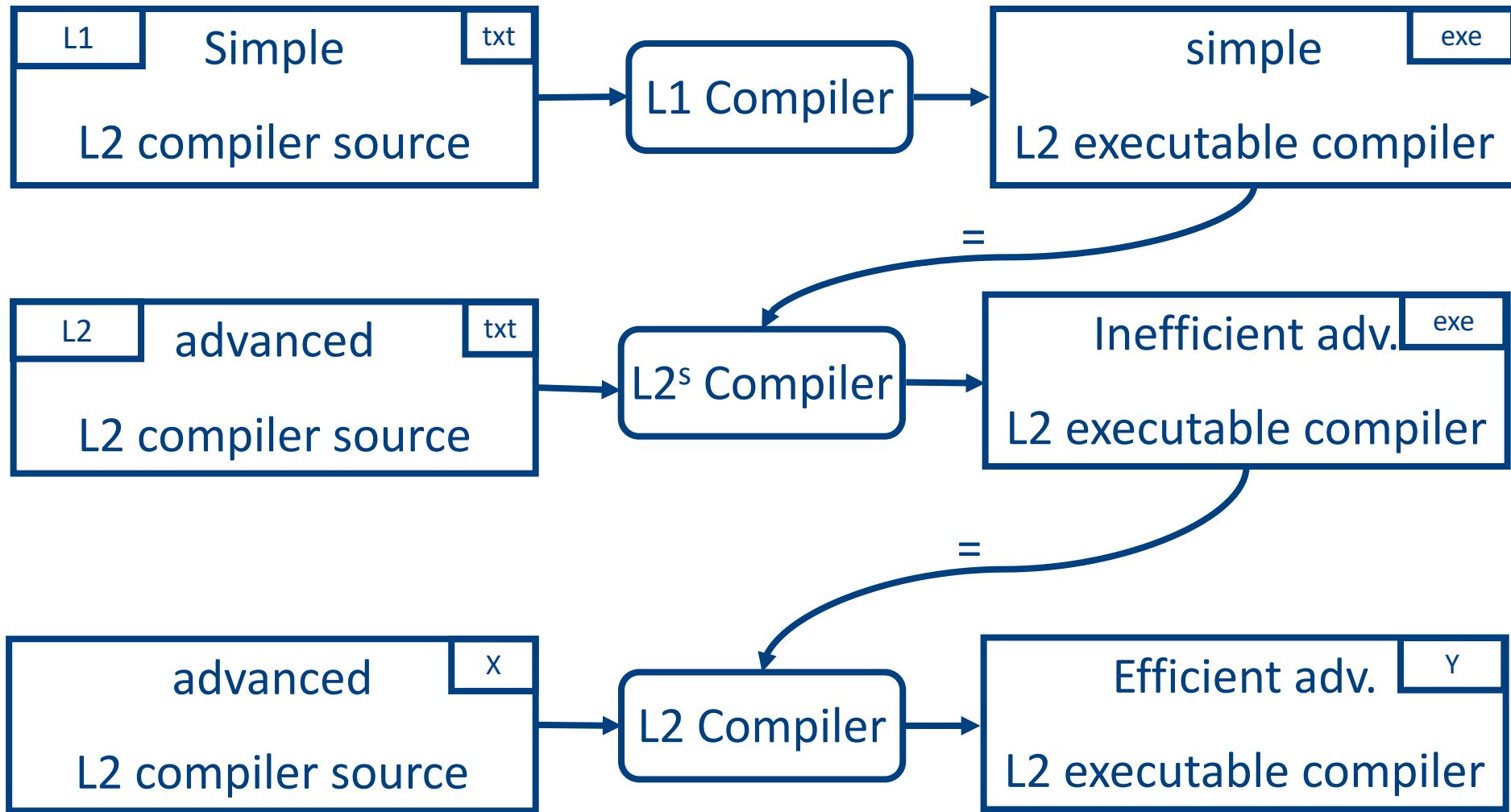
How to write a compiler?



Bootstrapping a compiler

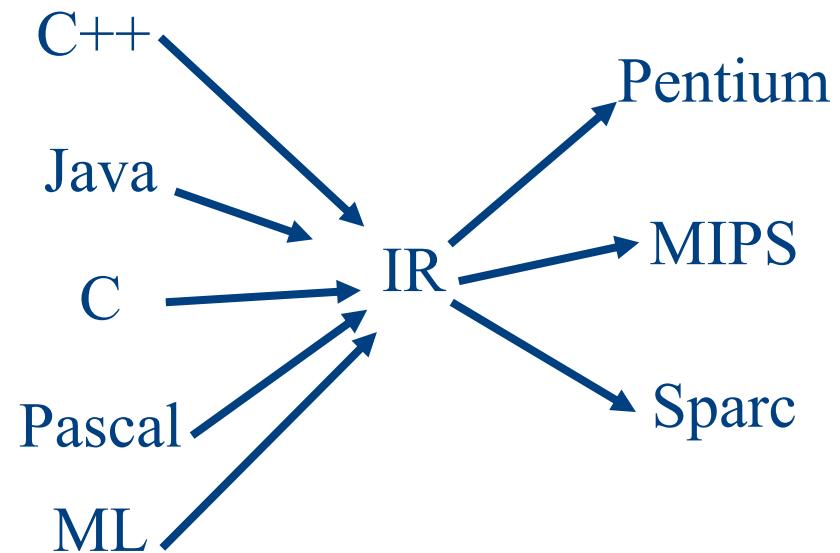
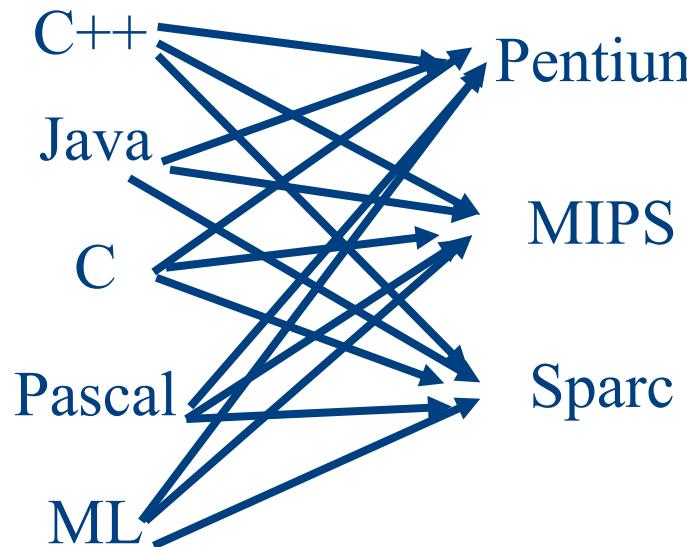


Bootstrapping a compiler



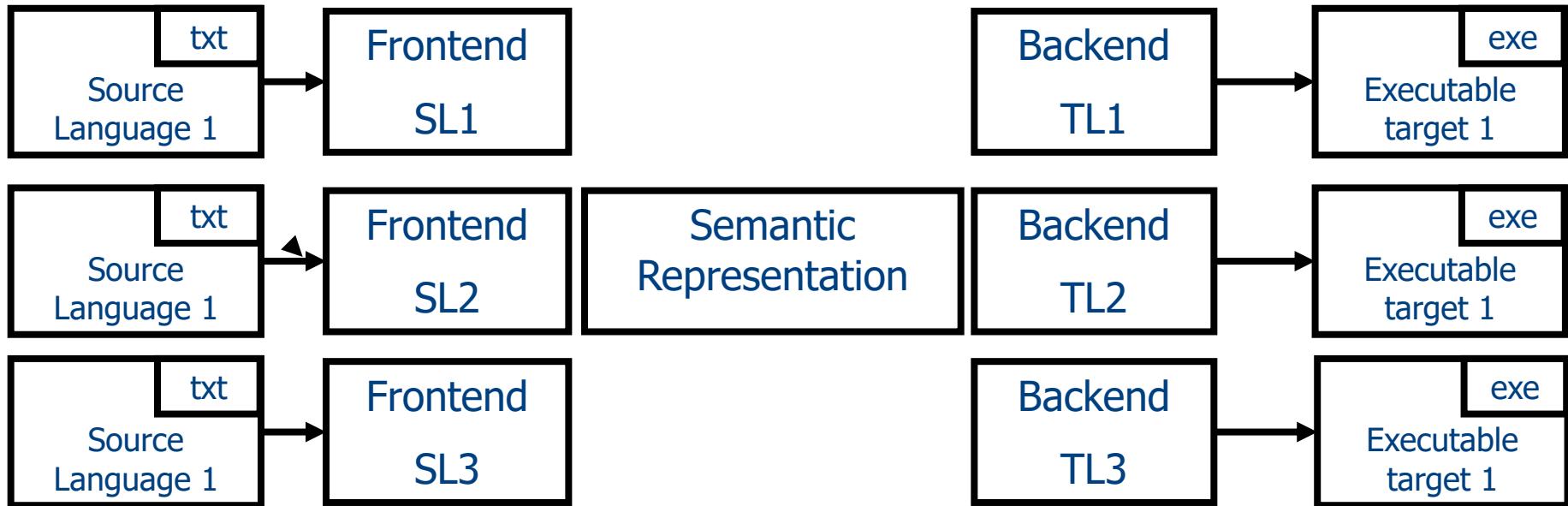
Proper Design

- Simplify the compilation phase
 - Portability of the compiler frontend
 - Reusability of the compiler backend
- Professional compilers are integrated



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
```



GRAHAM CHAPMAN · JOHN CLEES · TERRY GILLIAM · KEN JONES · TERRY JONES · MICHAEL PALIN

MONTY PYTHON'S

AND NOW FOR
SOMETHING
COMPLETELY
DIFFERENT

モントイ・パイン

アンド・ナウ



THE BEST OF MONTY PYTHON'S FLYING CIRCUS