Compilation

0368-3133

Lecture 1:

Introduction

Lexical Analysis

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.: Orr Tamir
- Textbooks:
  - Modern Compiler Design
  - Compilers: principles, techniques and tools

Admin

- Compiler Project 40%
  - 4.5 practical exercises
  - Groups of 3
- 1 theoretical exercise 10%
  - Groups of 1
- 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).

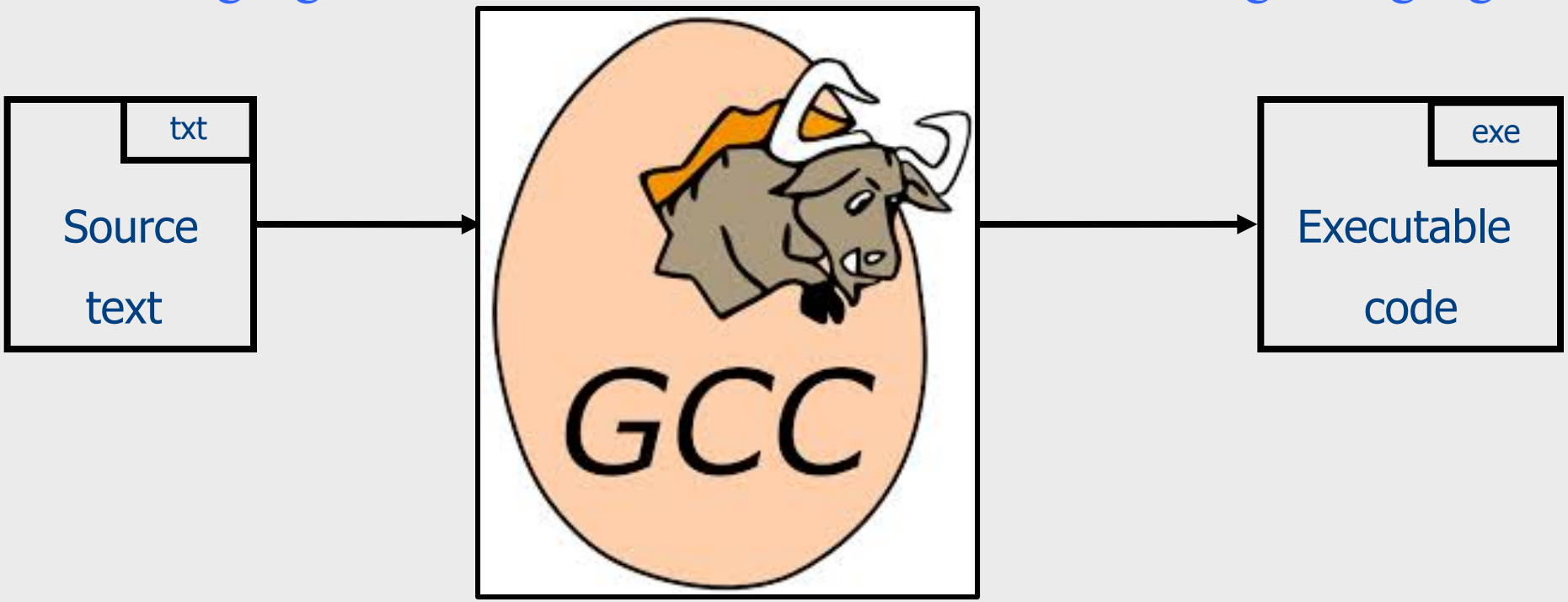
The most common reason for wanting to transform source code is to create an executable program."

--*Wikipedia*

What is a Compiler?

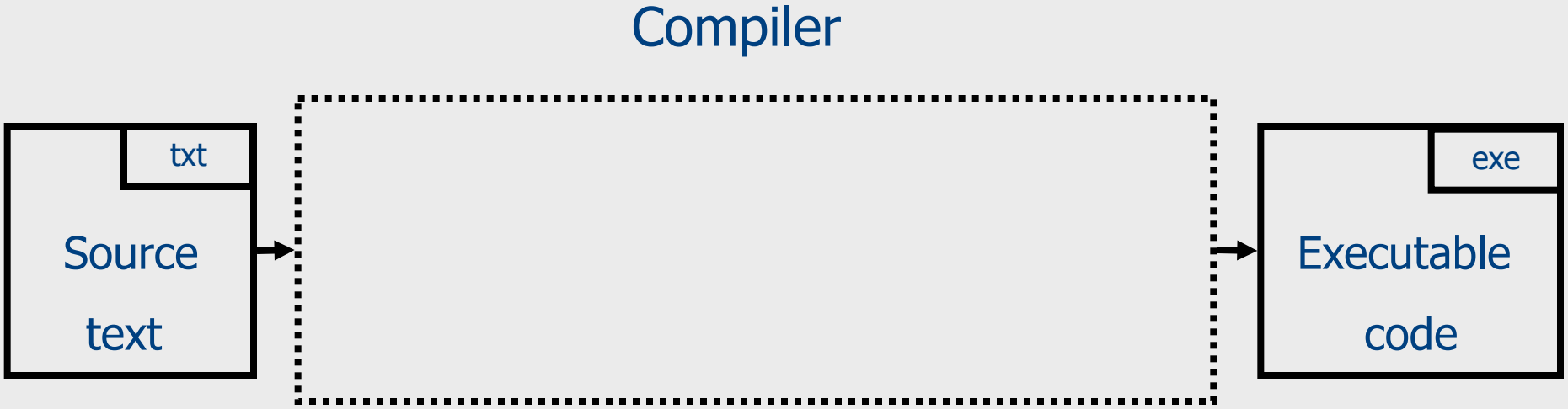
source language

target language



Compiler

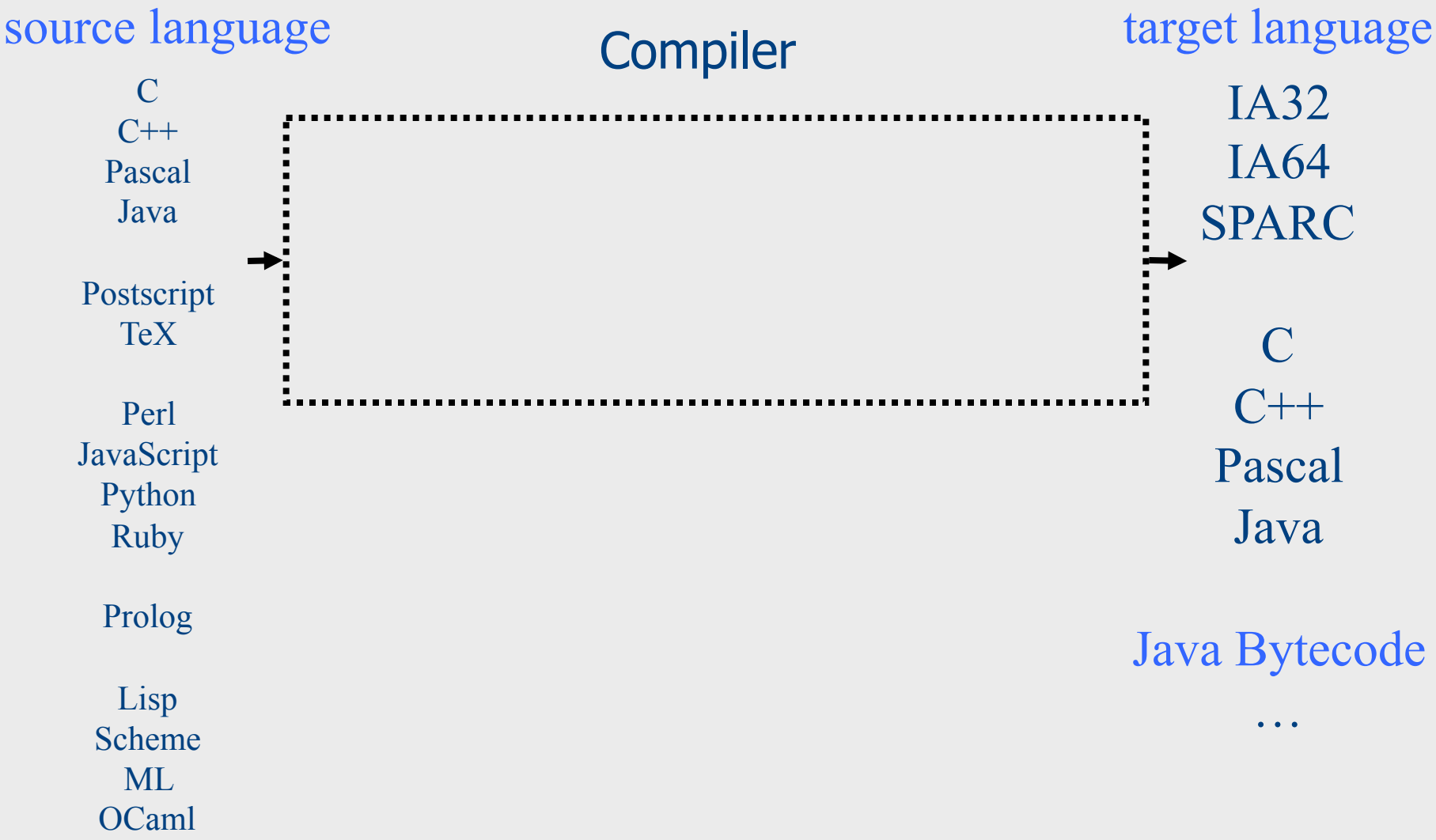
What is a Compiler?




```
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

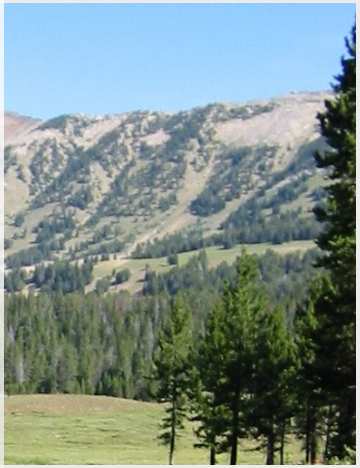
- Imperative Algol, PL1, Fortran, Pascal, Ada, Modula, C
  - Closely related to “von Neumann” Computers
- Object-oriented Simula, Smalltalk, Modula3, C++, Java, C#, Python
  - Data abstraction and ‘evolutionary’ form of program development
    - Class an implementation of an abstract data type (data+code)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

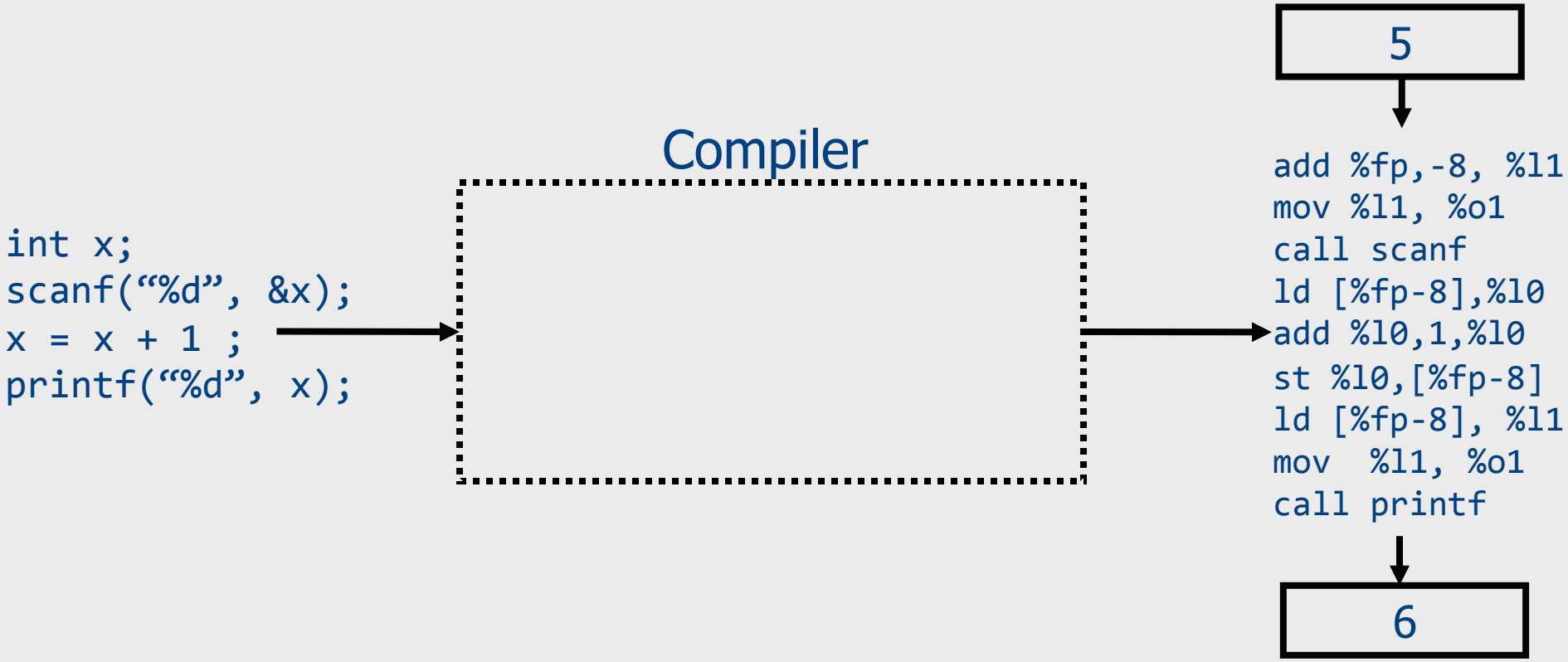


Compiler vs. Interpreter

Compiler

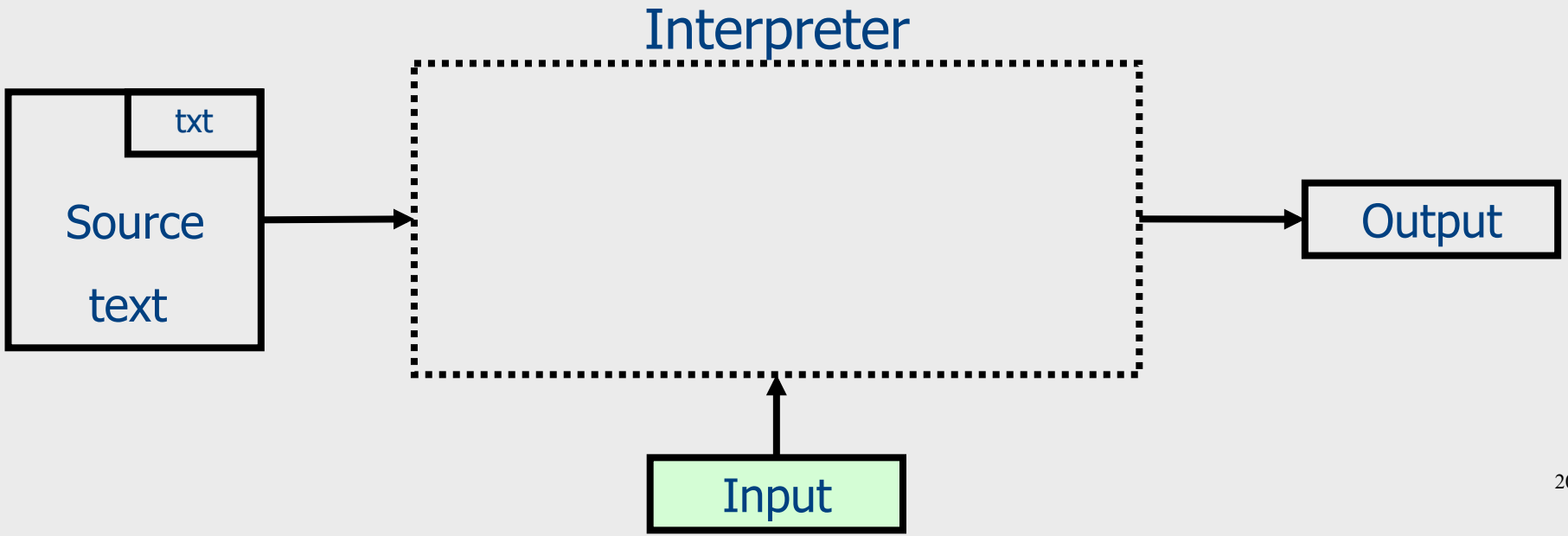
- A program which transforms programs
- Input a program (P)
- Output an object program (O)
  - For any x, “O(x)” “=“ “P(x)”

Compiling C to Assembly



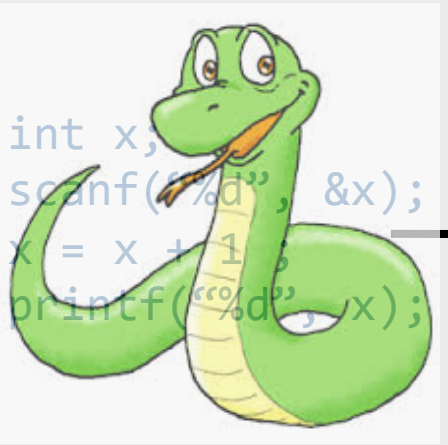
Interpreter

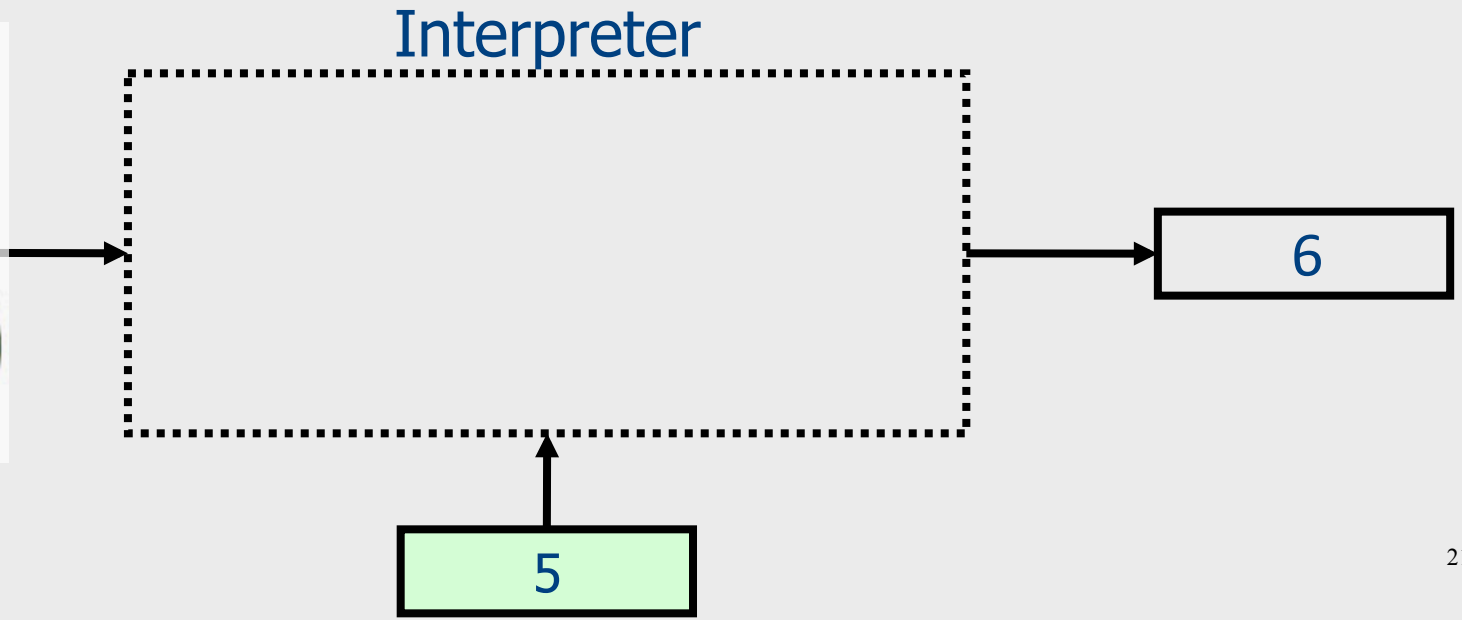
- A program which executes a program
- Input a program (P) + its input (x)
- Output the computed output (P(x))



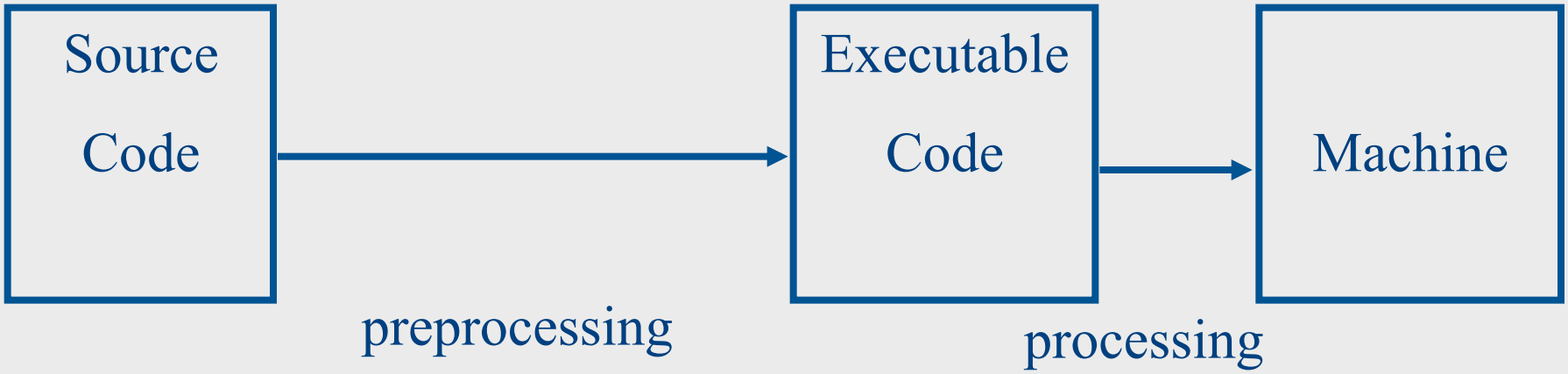
Interpreting (running) .py programs

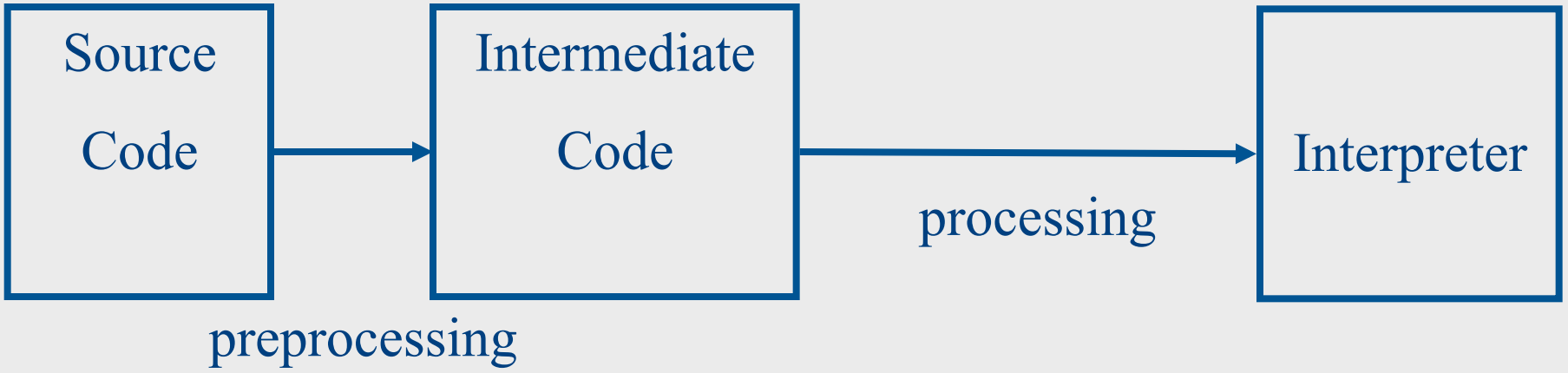
- A program which executes a program
- Input a program (P) + its input (x)
- Output the computed output ("P(x)")



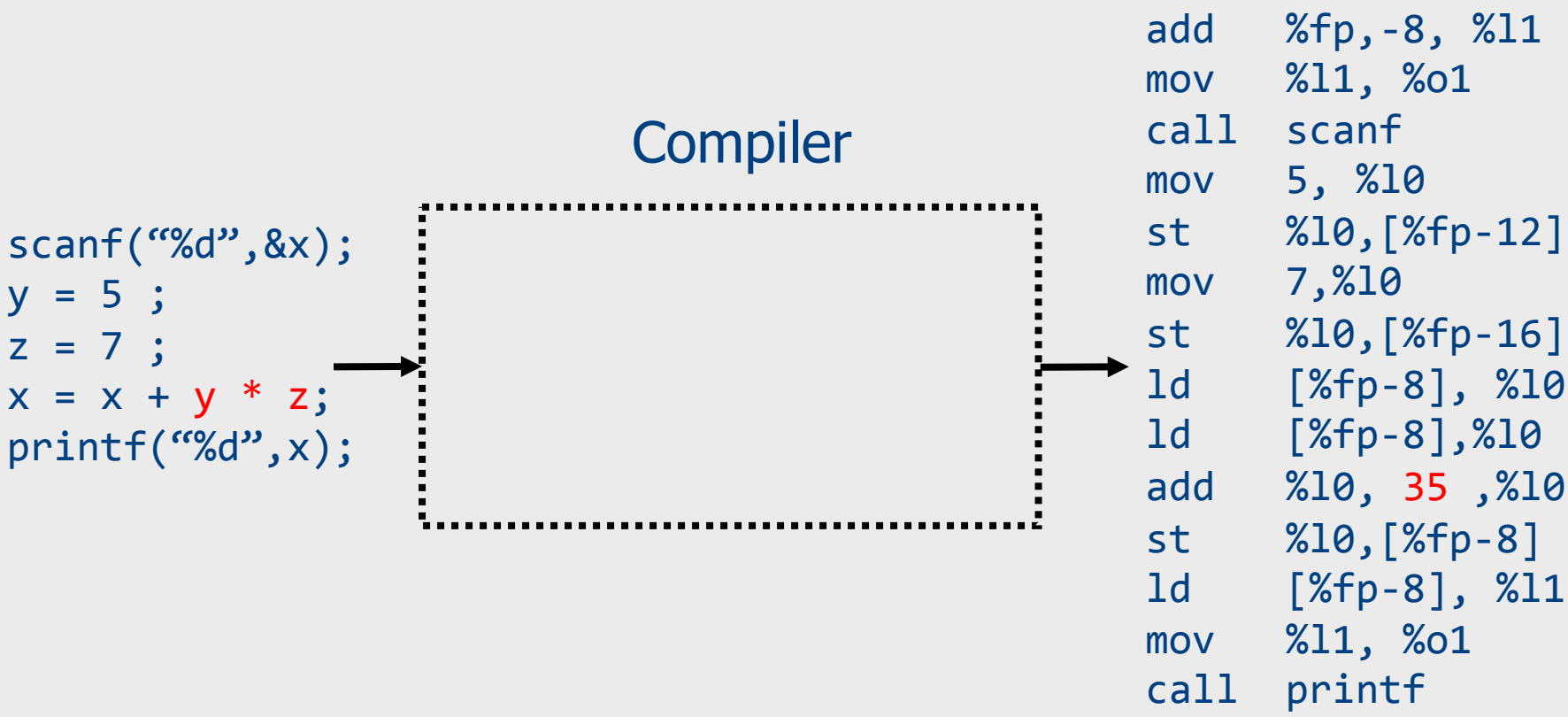


Compiler vs. Interpreter





Compiled programs are usually more efficient than



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 = x + 1;
```

- Input data
  - y = -1
  - y = 0

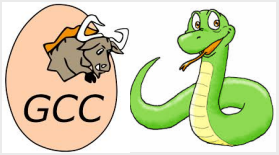
Interpreter vs. Compiler

- Conceptually simpler
  - "define" the prog. lang.
- Can provide more specific error report
- Easier to port
- Faster response time
- [More secure]

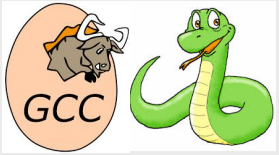
- How do we know the translation is correct?
- Can report errors before input is given
- More efficient code
  - Compilation can be expensive
  - move computations to compile-time
- *compile-time + execution-time < interpretation-time* is possible

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**

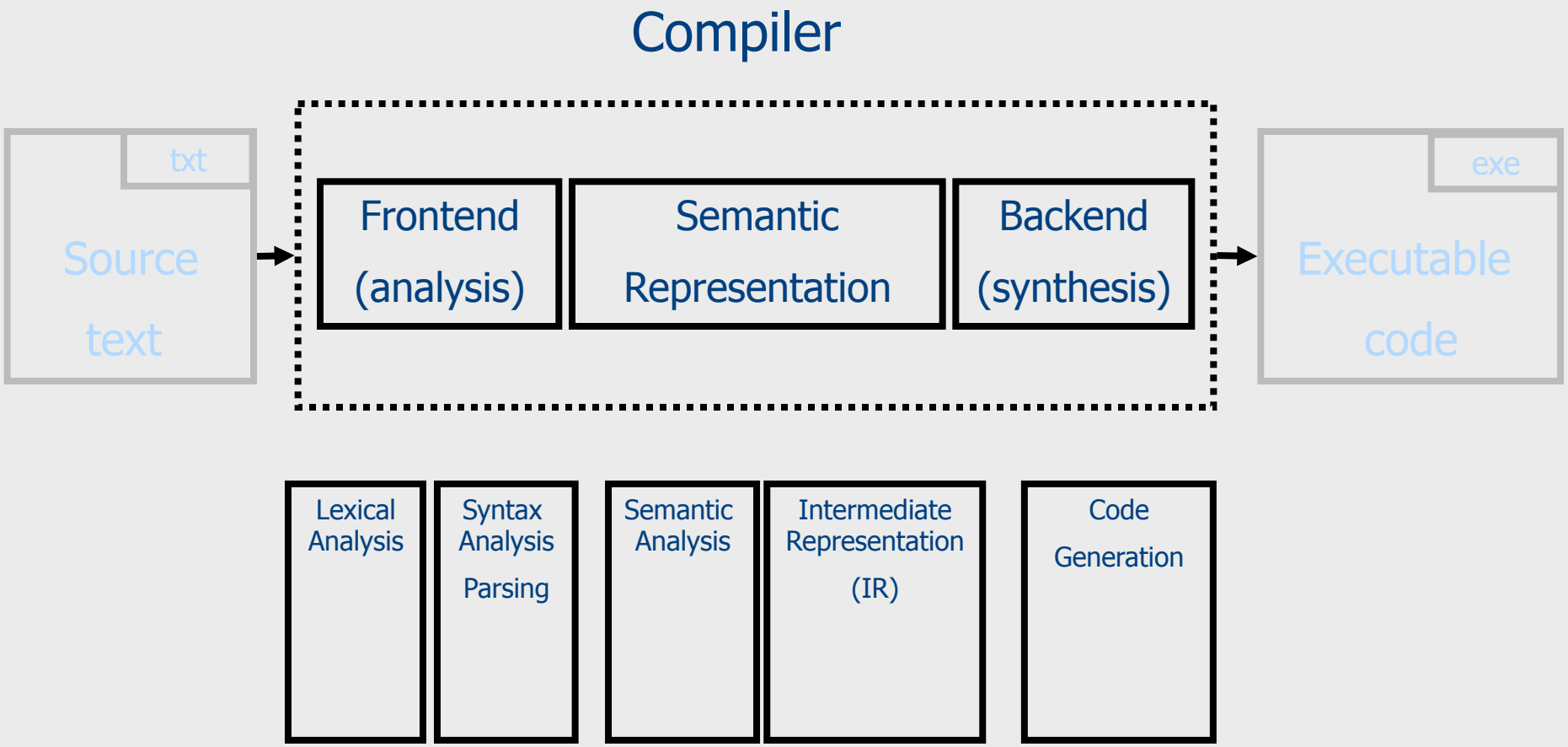
Ex 0: A Simple Interpreter

Toy compiler/interpreter

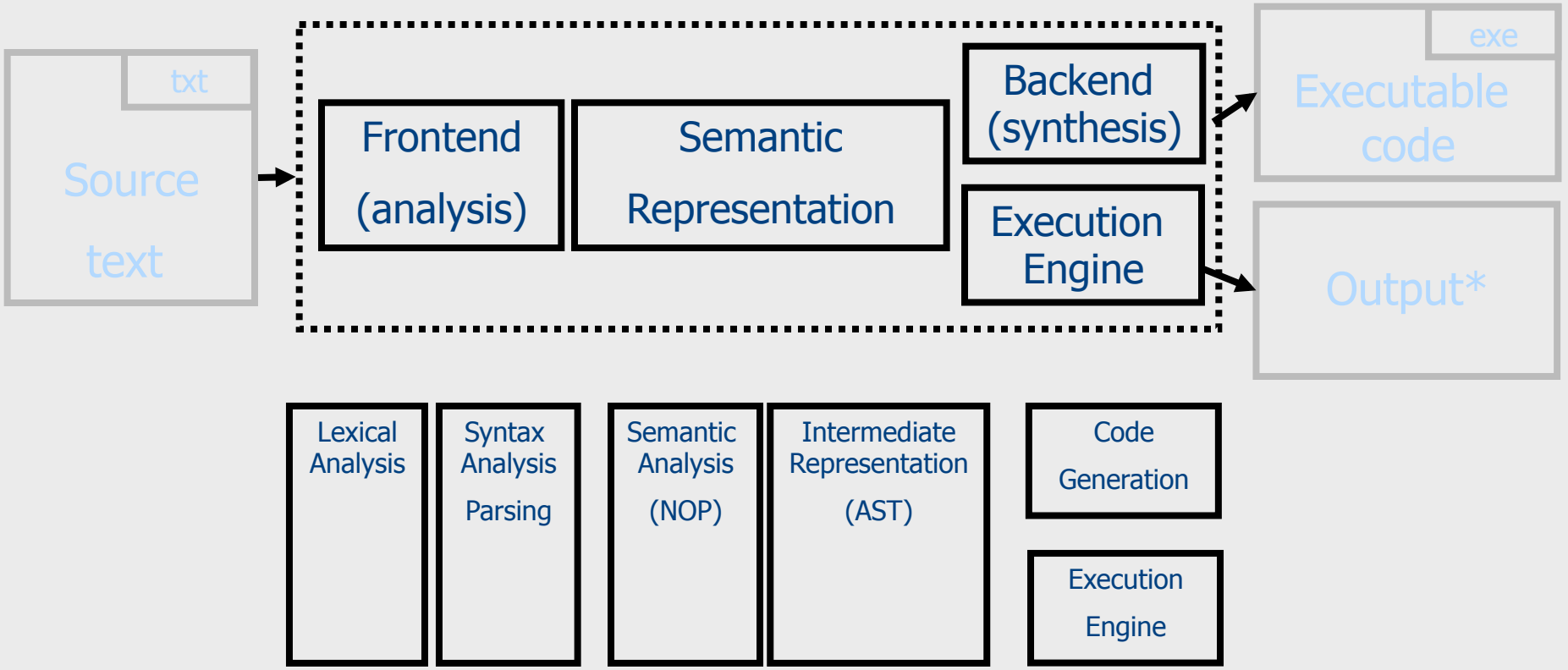
- Trivial programming language
- Stack machine
- Compiler/interpreter written in C
- Demonstrate the basic steps

- 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))
  - ✗ 3 + 4 + 5
  - ✗ (12 + 3)

expression → digit | ‘(‘ expression operator expression ‘)’

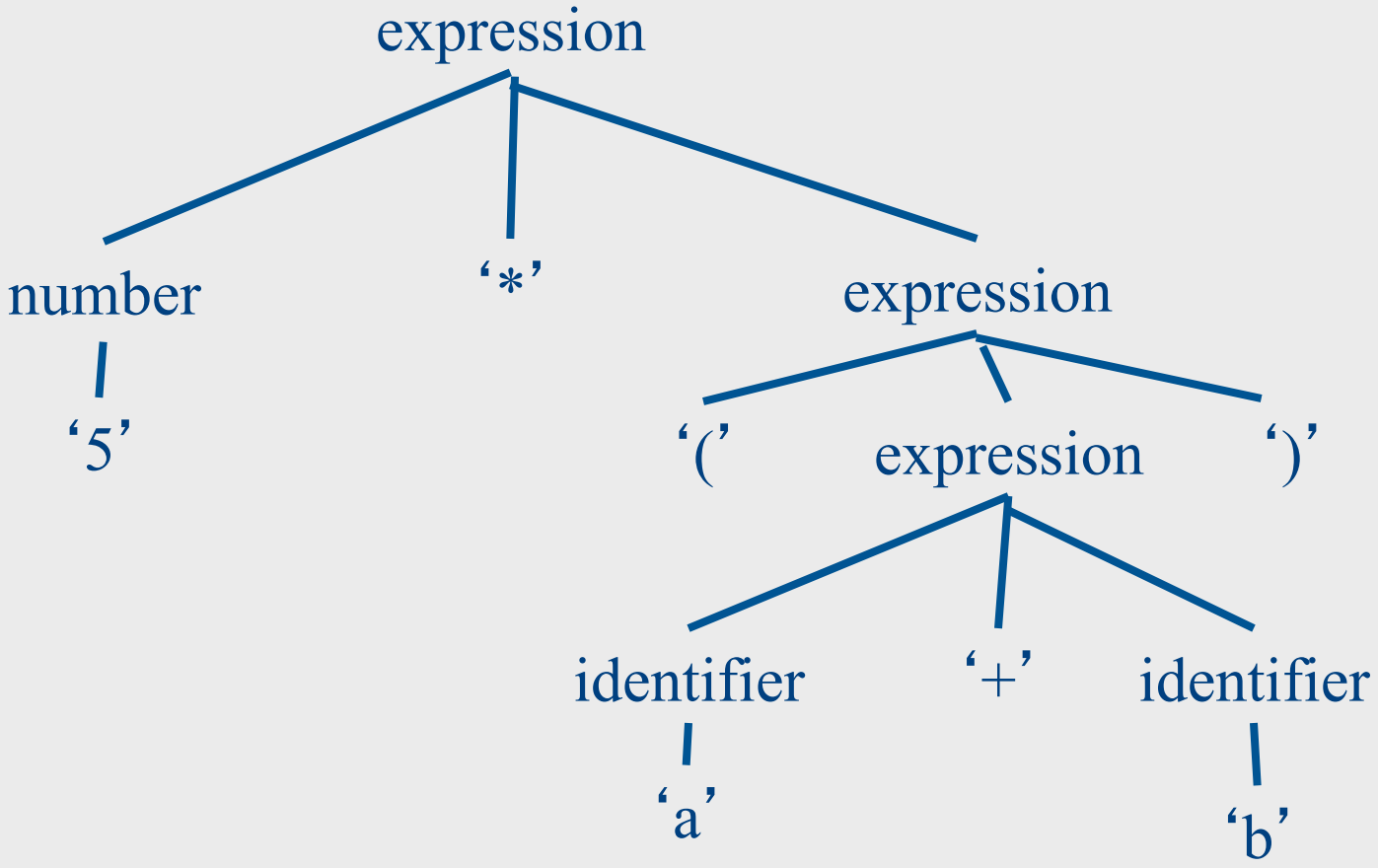
operator → ‘+’ | ‘*’

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

The abstract syntax tree (AST)

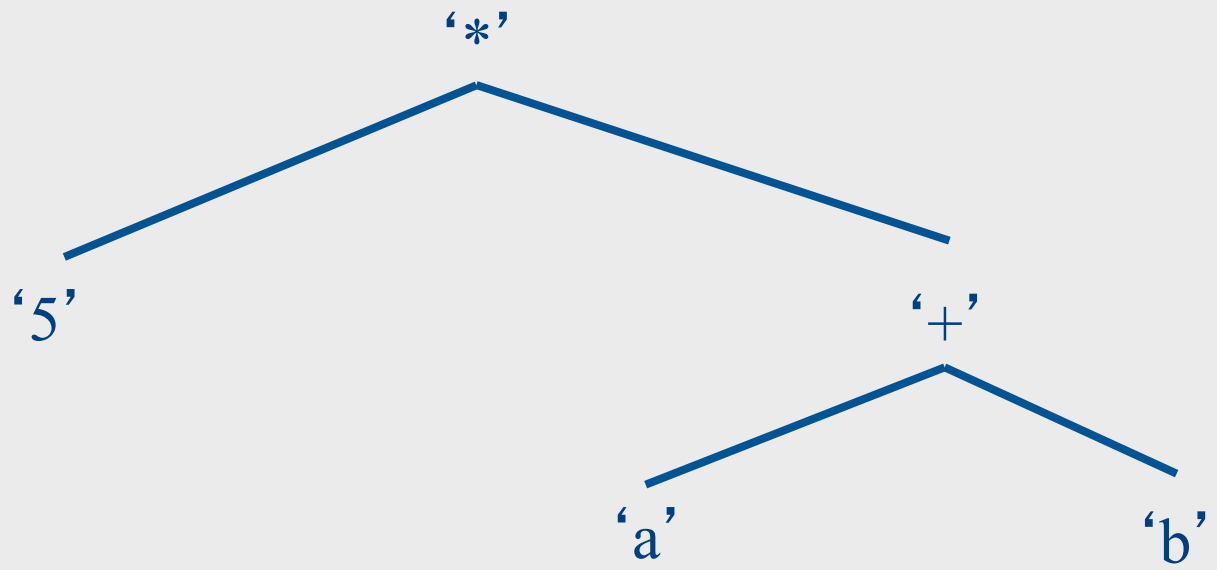
- Intermediate program representation
- Defines a tree
  - Preserves program hierarchy
- Generated by the parser
- Keywords and punctuation symbols are not stored
  - Not relevant once the tree exists

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

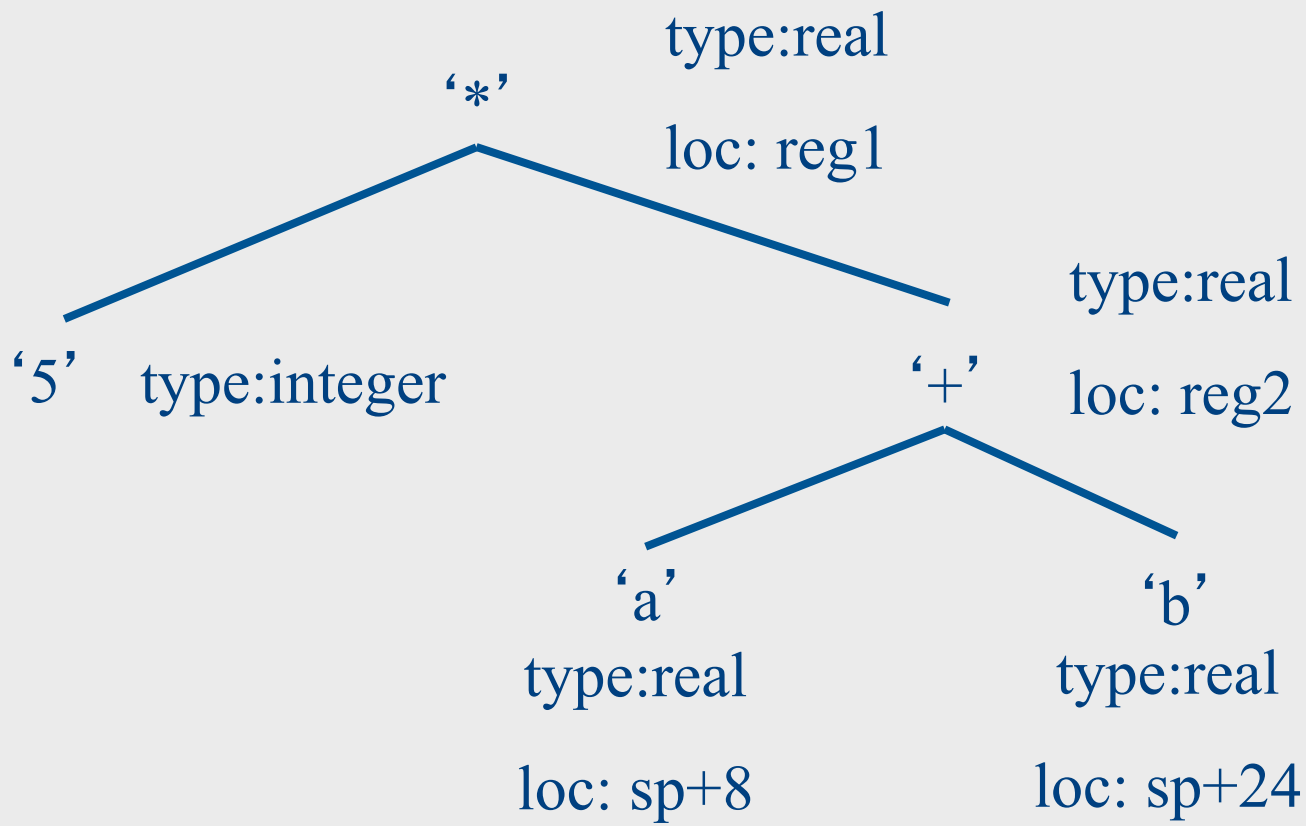


[#]Parse tree

Abstract Syntax tree for 5*(a+b)



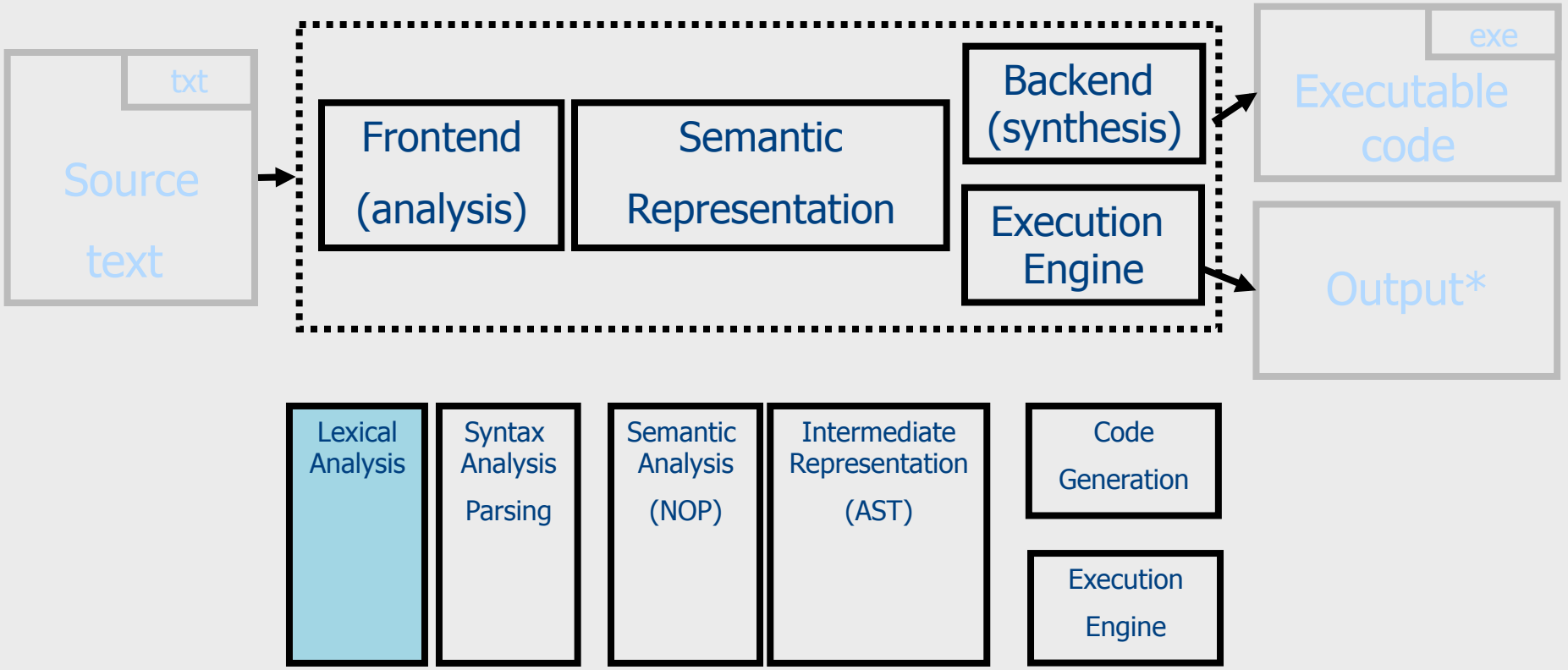
Annotated Abstract Syntax tree



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

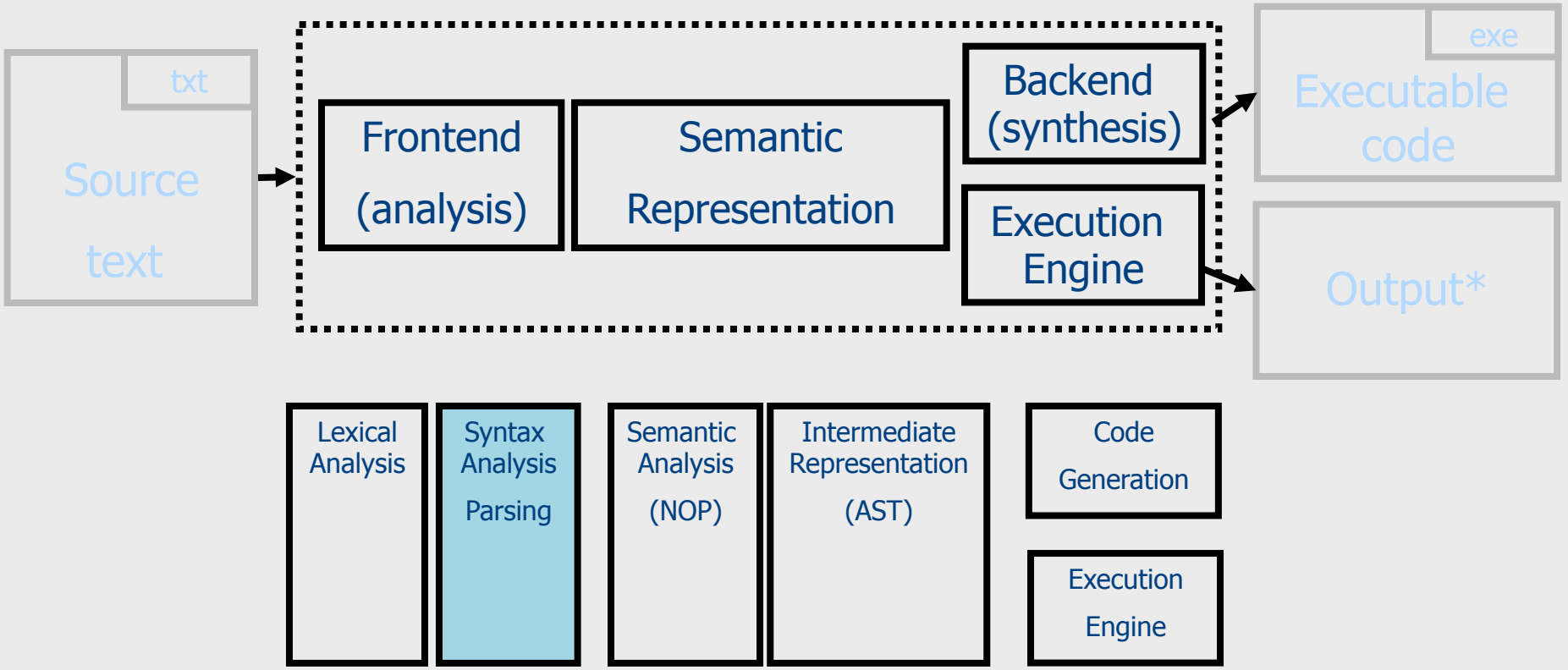
lex.h: Header File for Lexical Analysis

```
/* 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

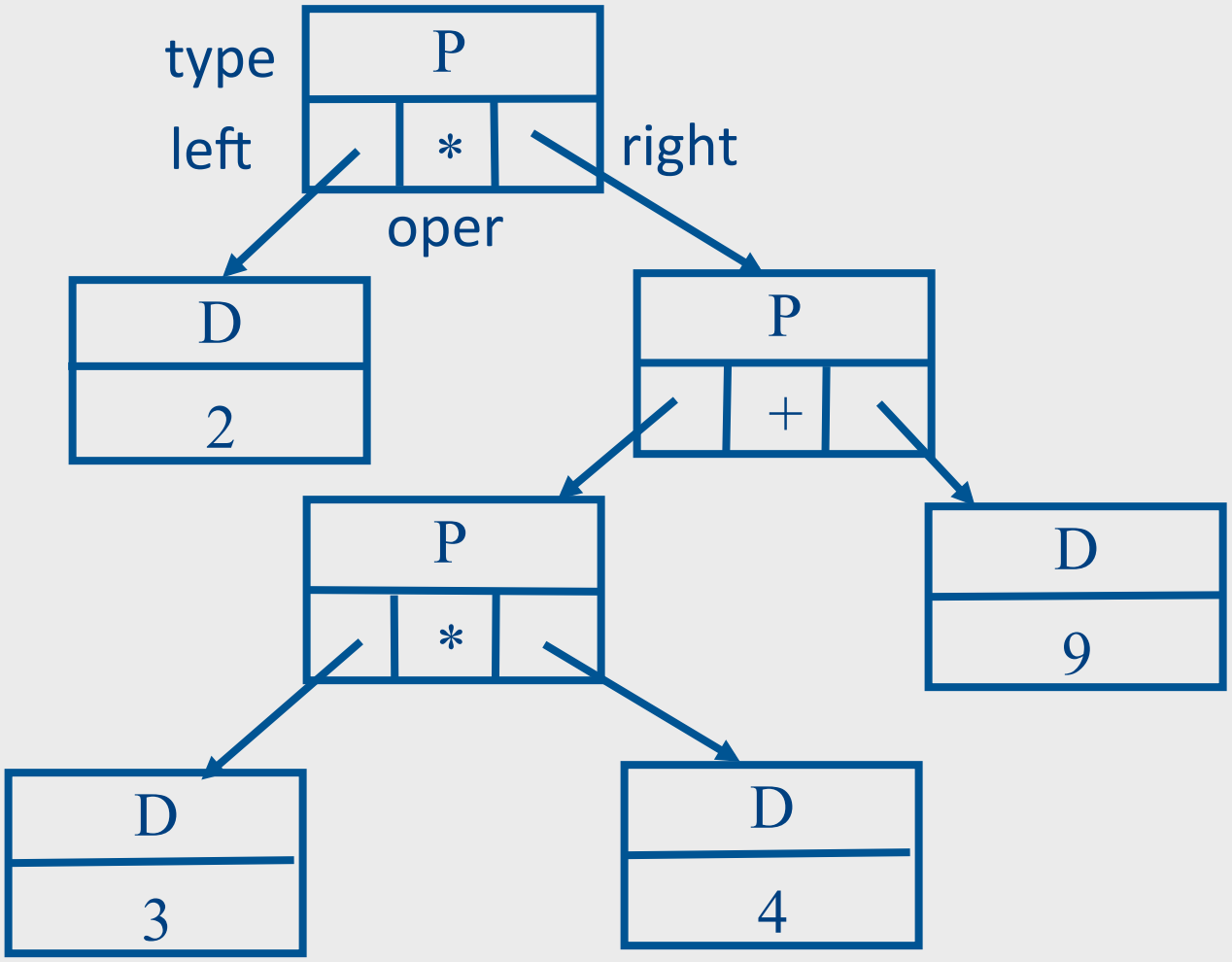
Parser

- Invokes lexical analyzer
- Reports syntax errors
- Constructs AST

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 **);
```

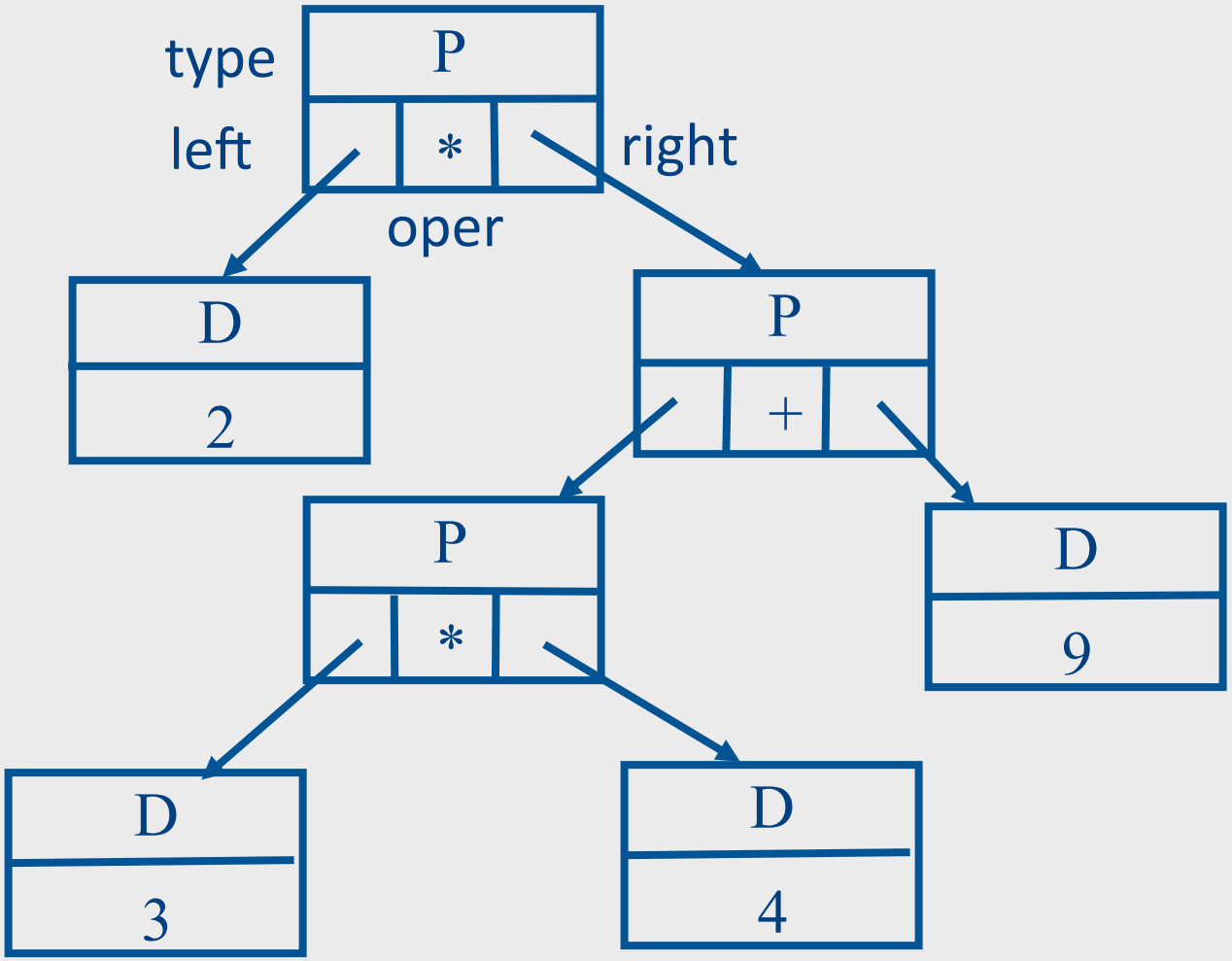
AST for (2 * ((3*4)+9))



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;
}
```

AST for (2 * ((3*4)+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;
}
```

Source Language

- Fully parenthesized expressions
- Arguments can be a single digit
  - ✓ (4 + (3 * 9))
  - ✗ 3 + 4 + 5
  - ✗ (12 + 3)

expression → digit | ‘(‘ expression operator expression ‘)’

operator → ‘+’ | ‘*’

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

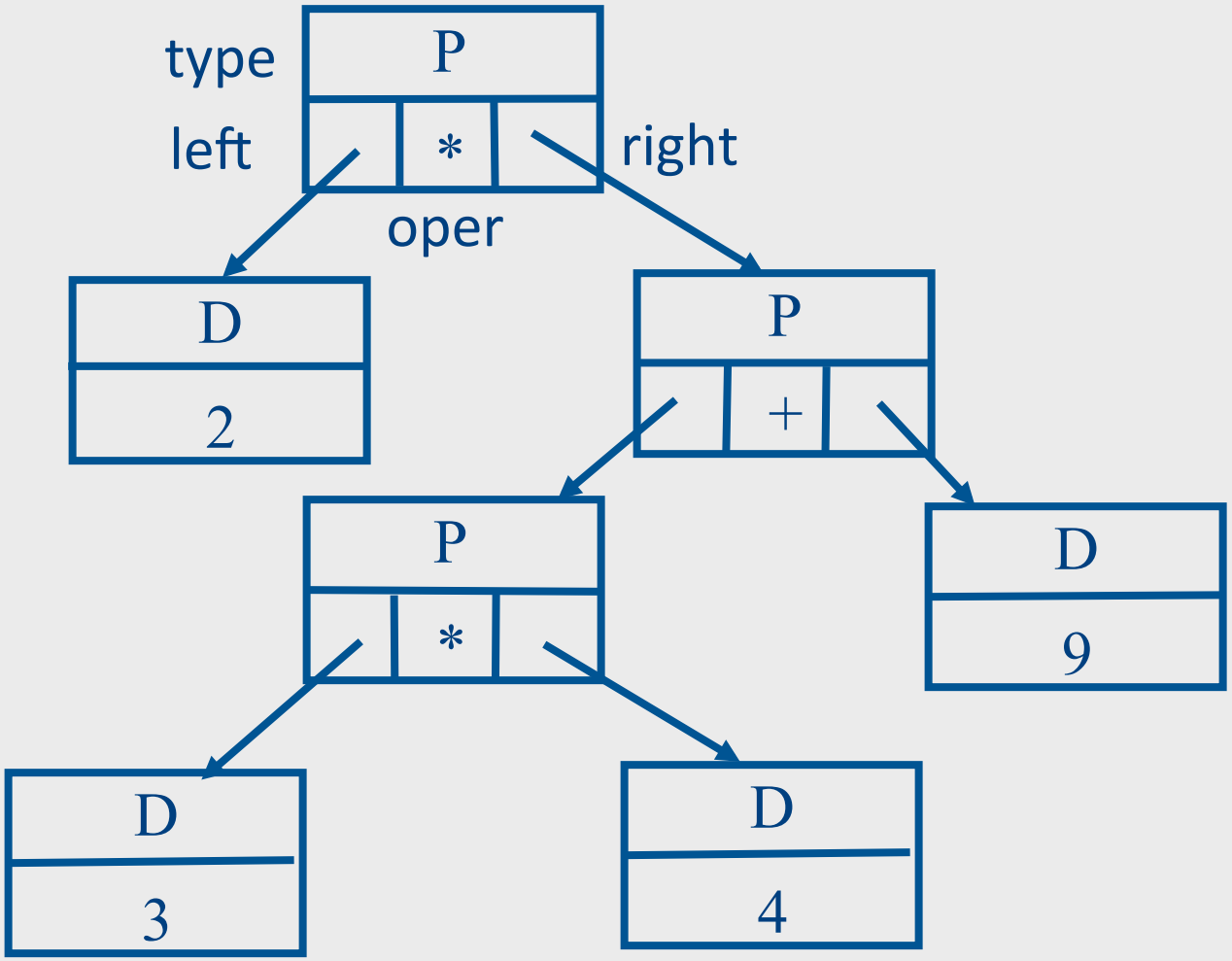
lex.h: Header File for Lexical Analysis

```
/* 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);
```

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;
    }
}
```

AST for (2 * ((3*4)+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 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;
}
```

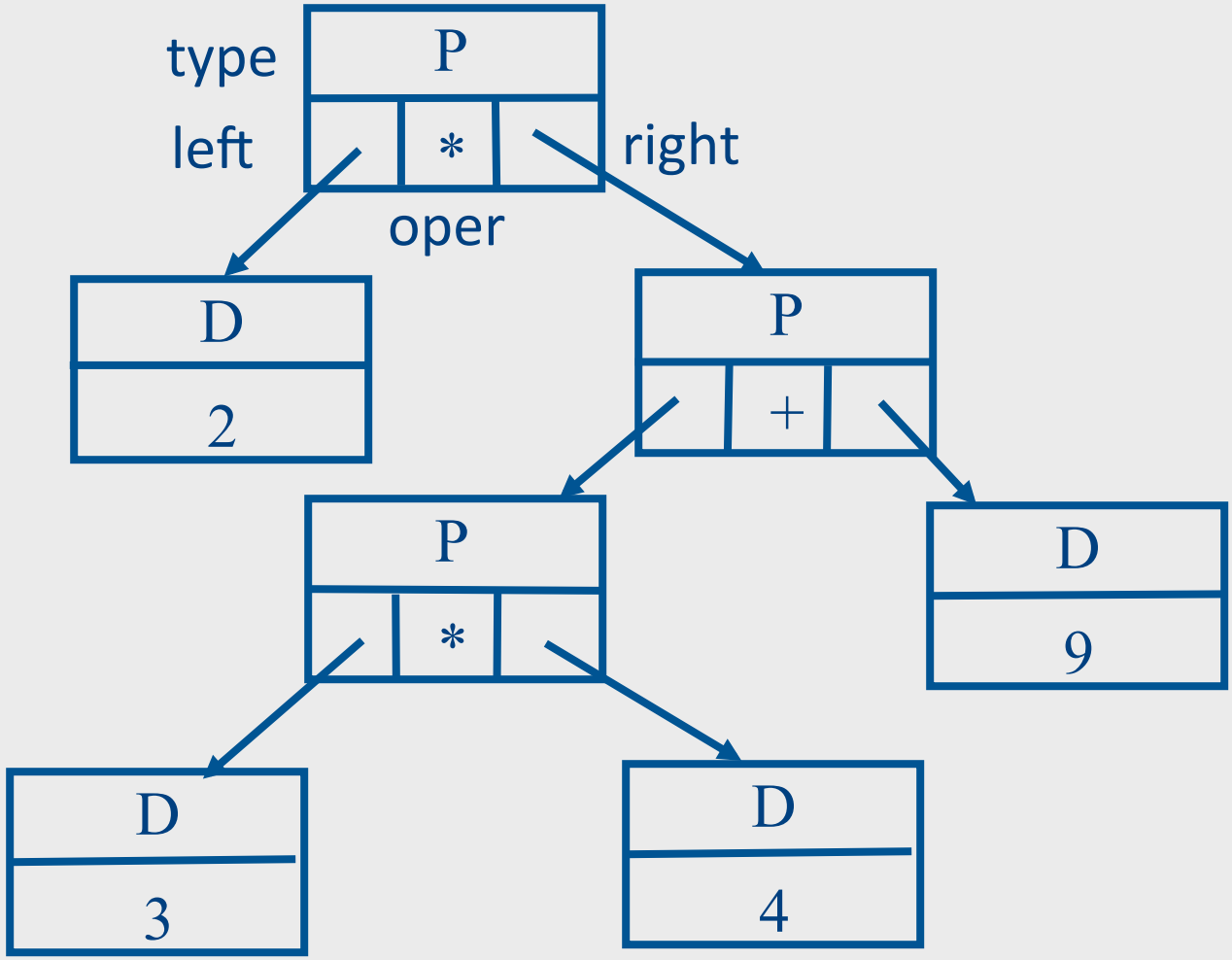
Top-Down Parsing

- Optimistically build the tree from the root to leaves
- For every P → A1 A2 … An | B1 B2 … Bm
  - If A1 succeeds
    - If A2 succeeds & A3 succeeds & …
    - Else fail
  - Else if B1 succeeds
    - If B2 succeeds & B3 succeeds & ..
    - Else fail
  - Else fail
- Recursive descent parsing
  - Simplified: no backtracking
- Can be applied for certain grammars

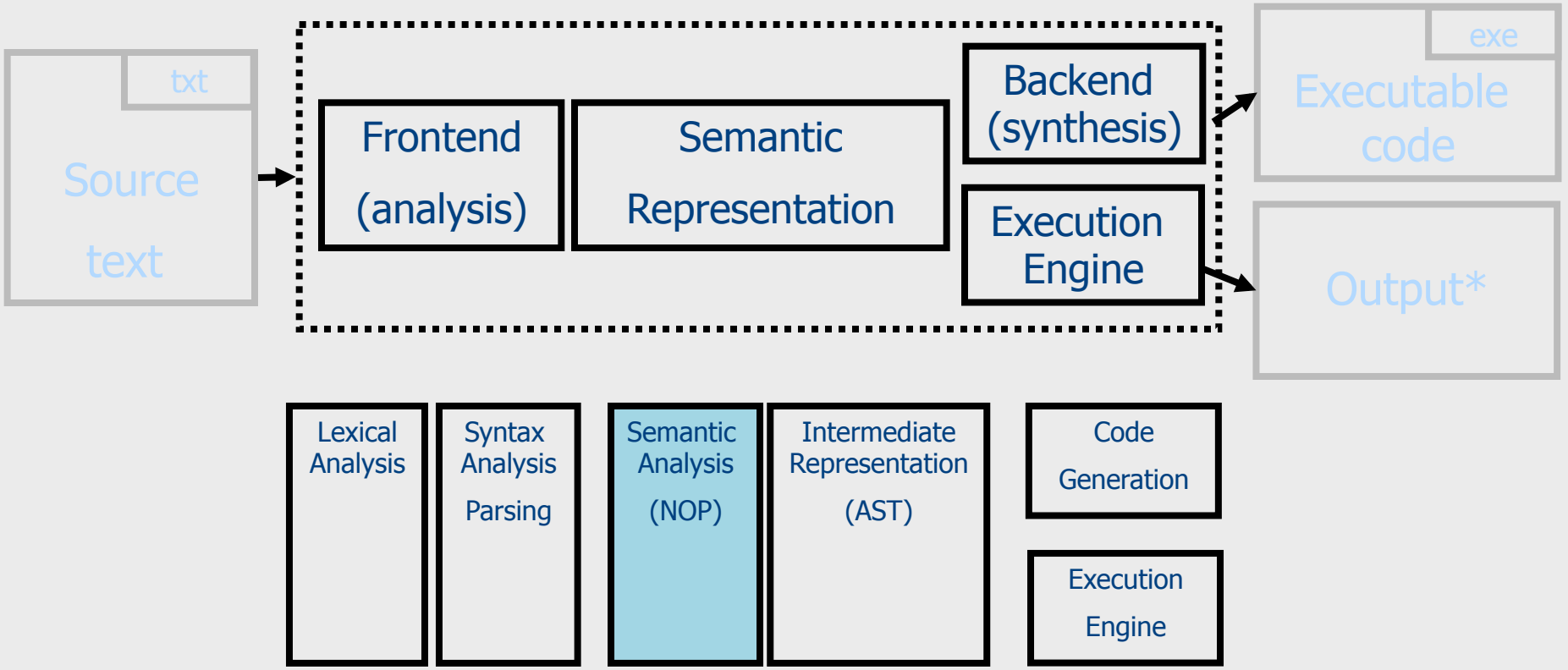
Parser

```
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))



Structure of toy Compiler / interpreter

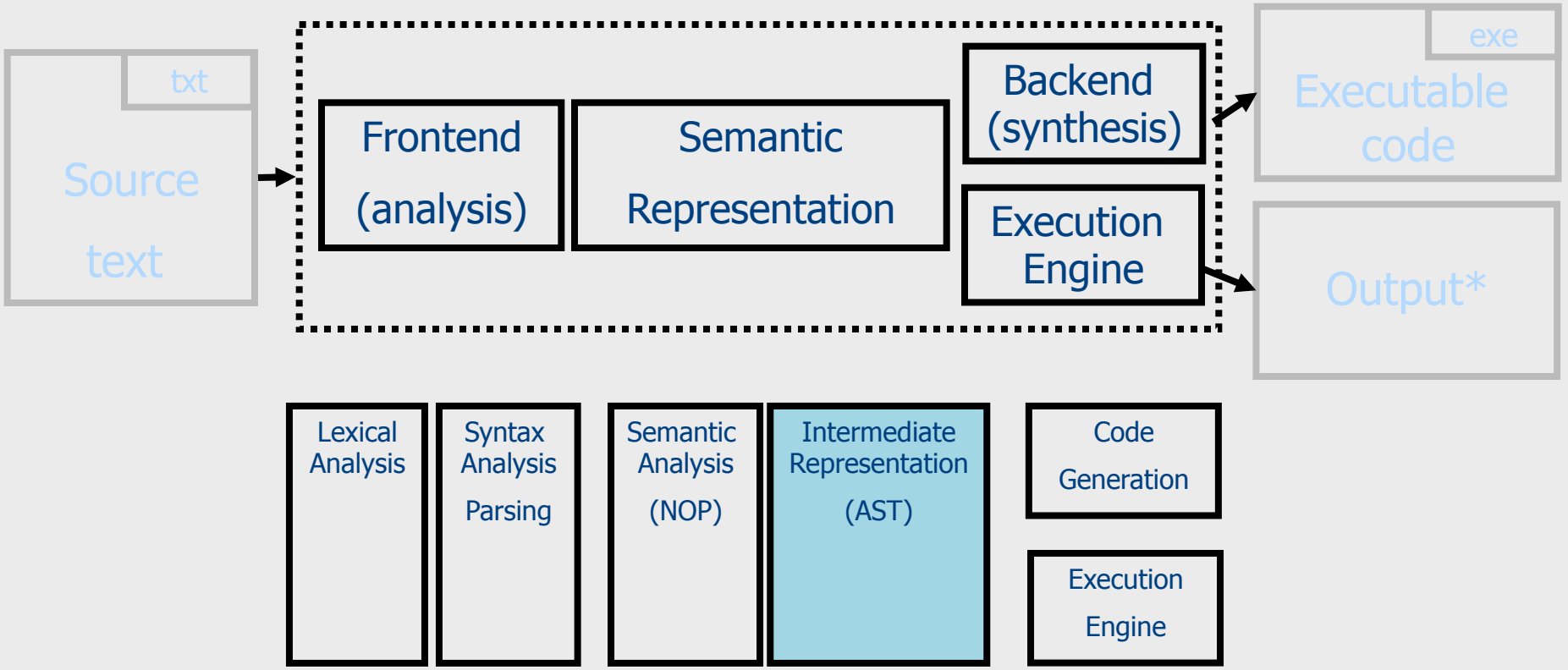


* Programs in our PL do not take input

Semantic Analysis

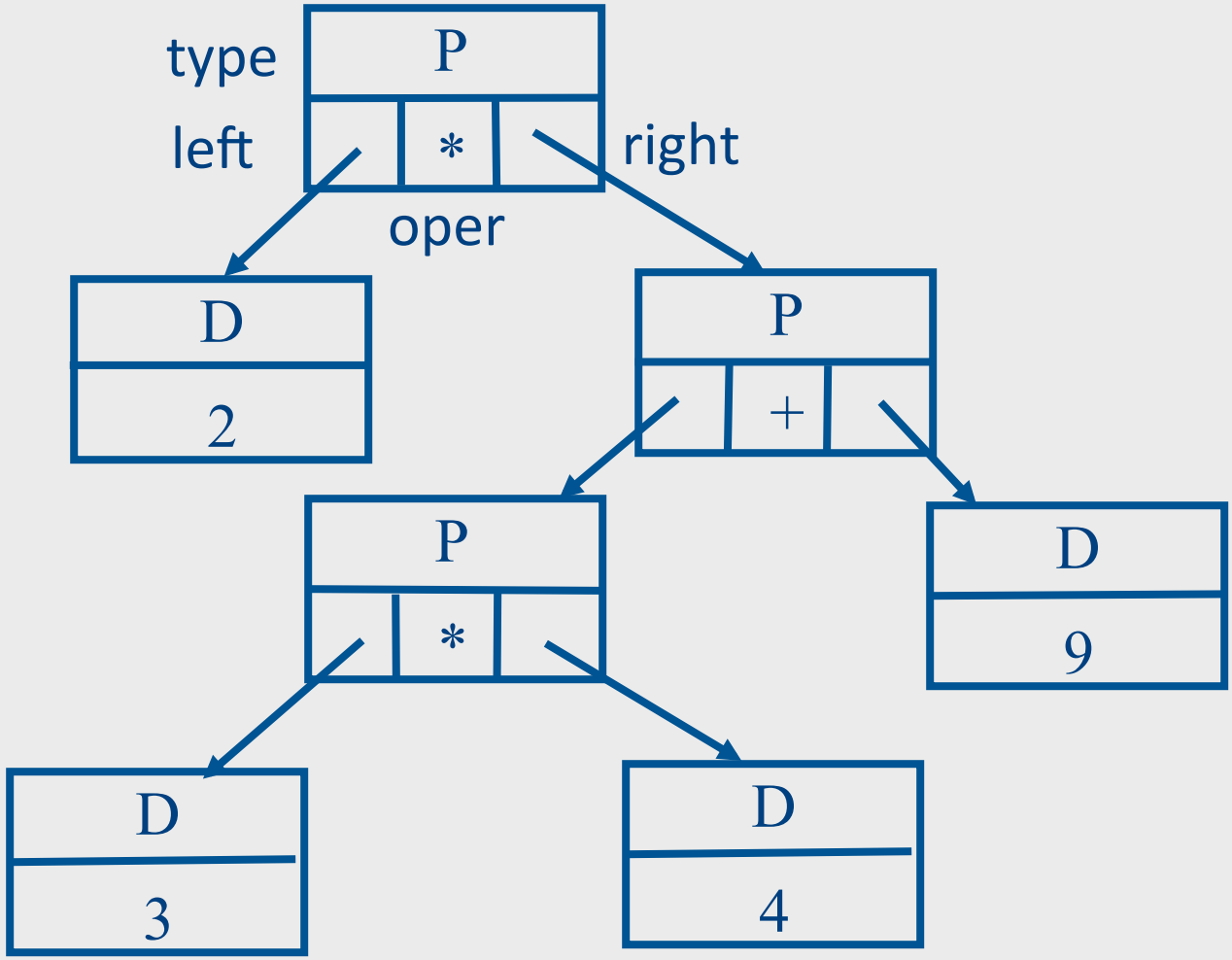
- Trivial in our case
- No identifiers
- No procedure / functions
- A single type for all expressions

Structure of toy Compiler / interpreter



* Programs in our PL do not take input

Intermediate Representation

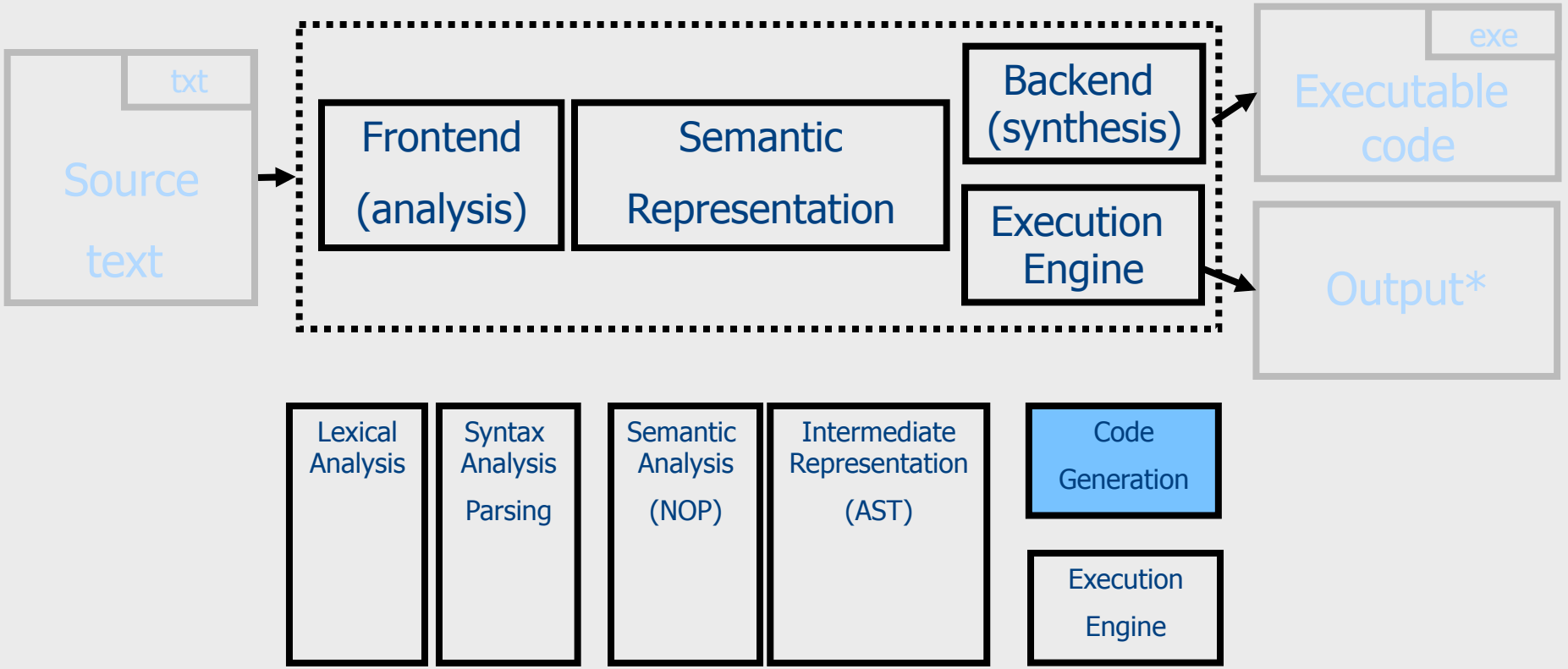


Alternative IR: 3-Address Code

```
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

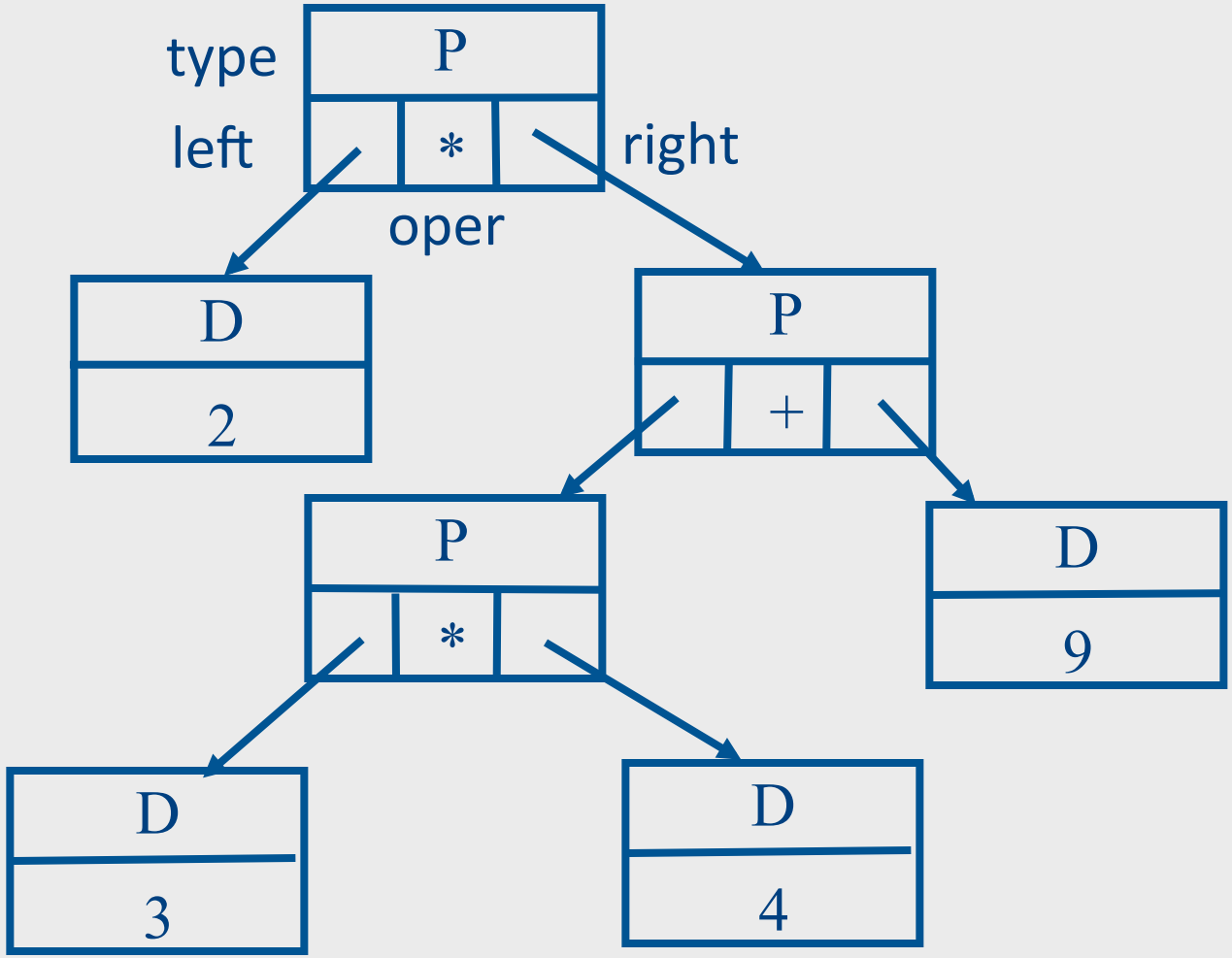
Code generation

- 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))



PUSH 2

PUSH 3

PUSH 4

MULT

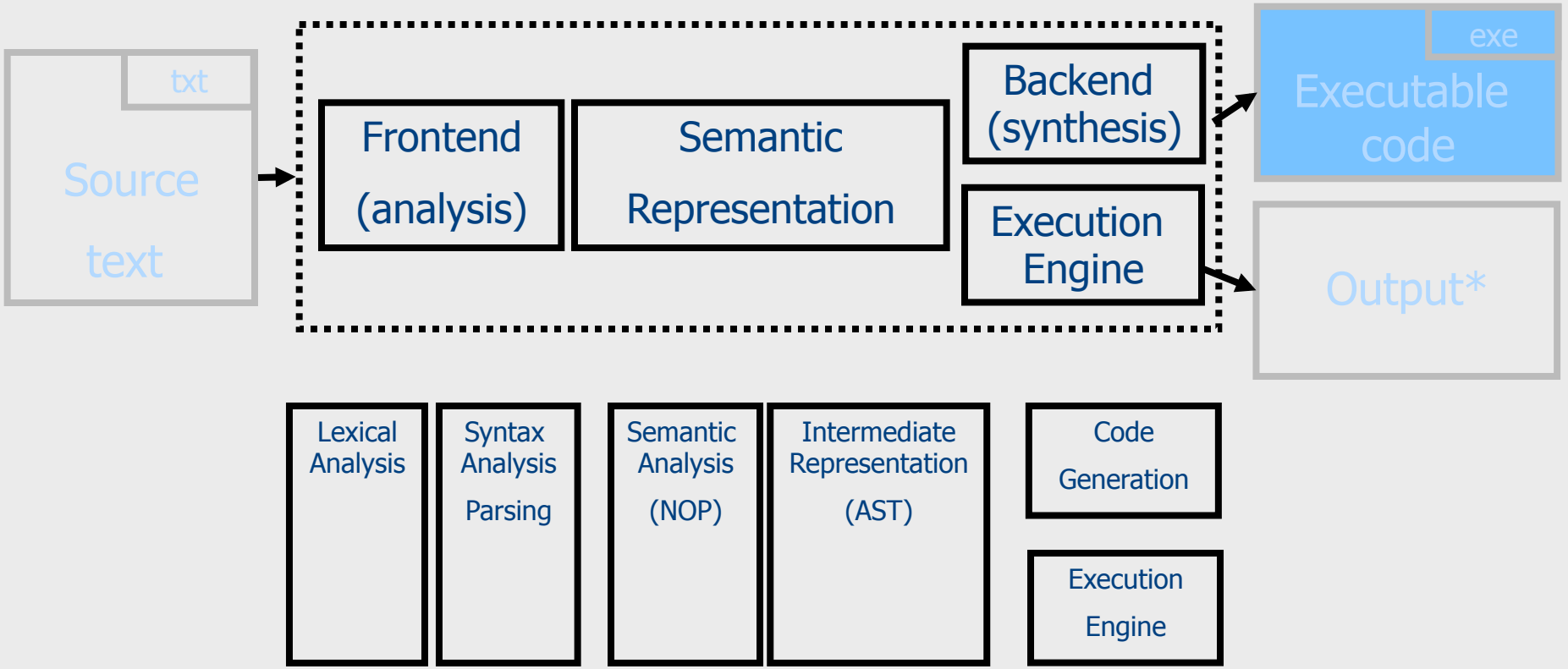
PUSH 9

ADD

MULT

PRINT

Executing Compiled Program



* Programs in our PL do not take input

Generated Code Execution

	Stack	Stack’
PUSH 2		
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		

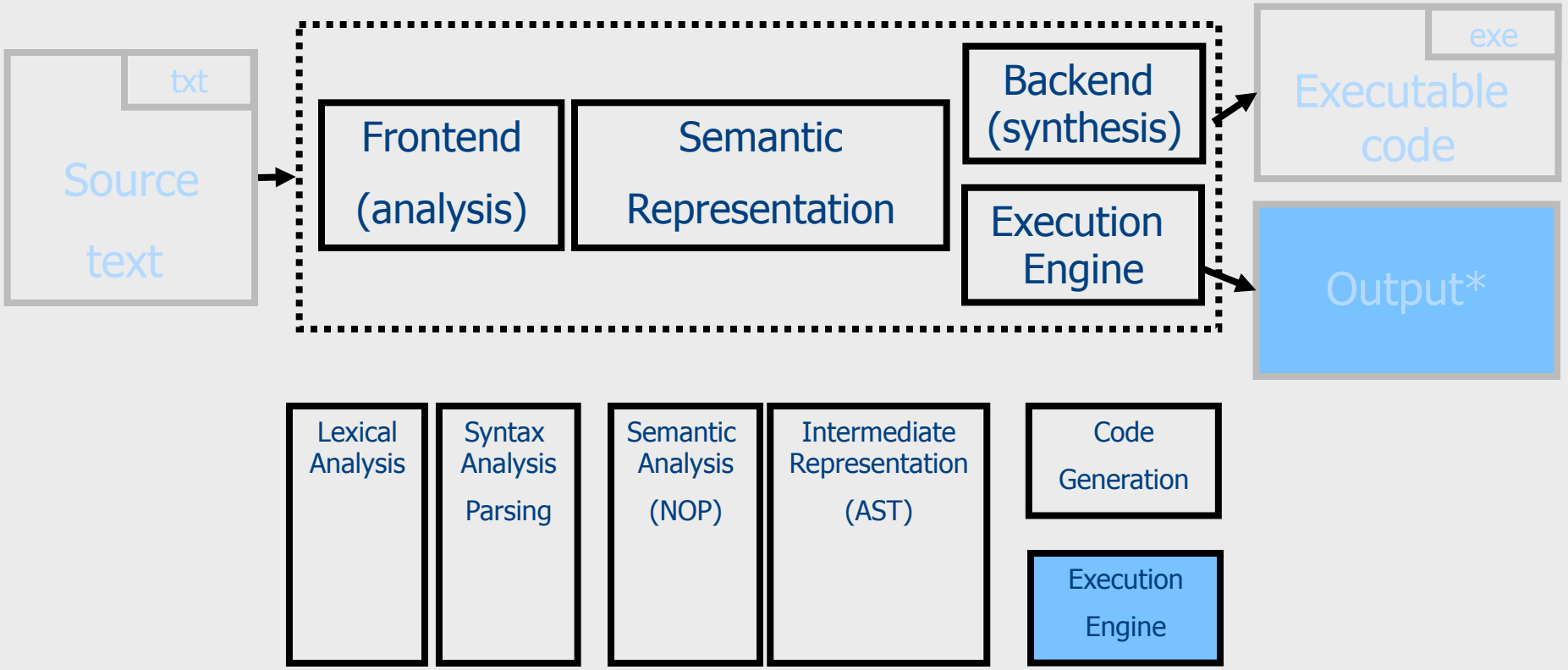
Generated Code Execution

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

Shortcuts

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

Structure of toy Compiler / interpreter



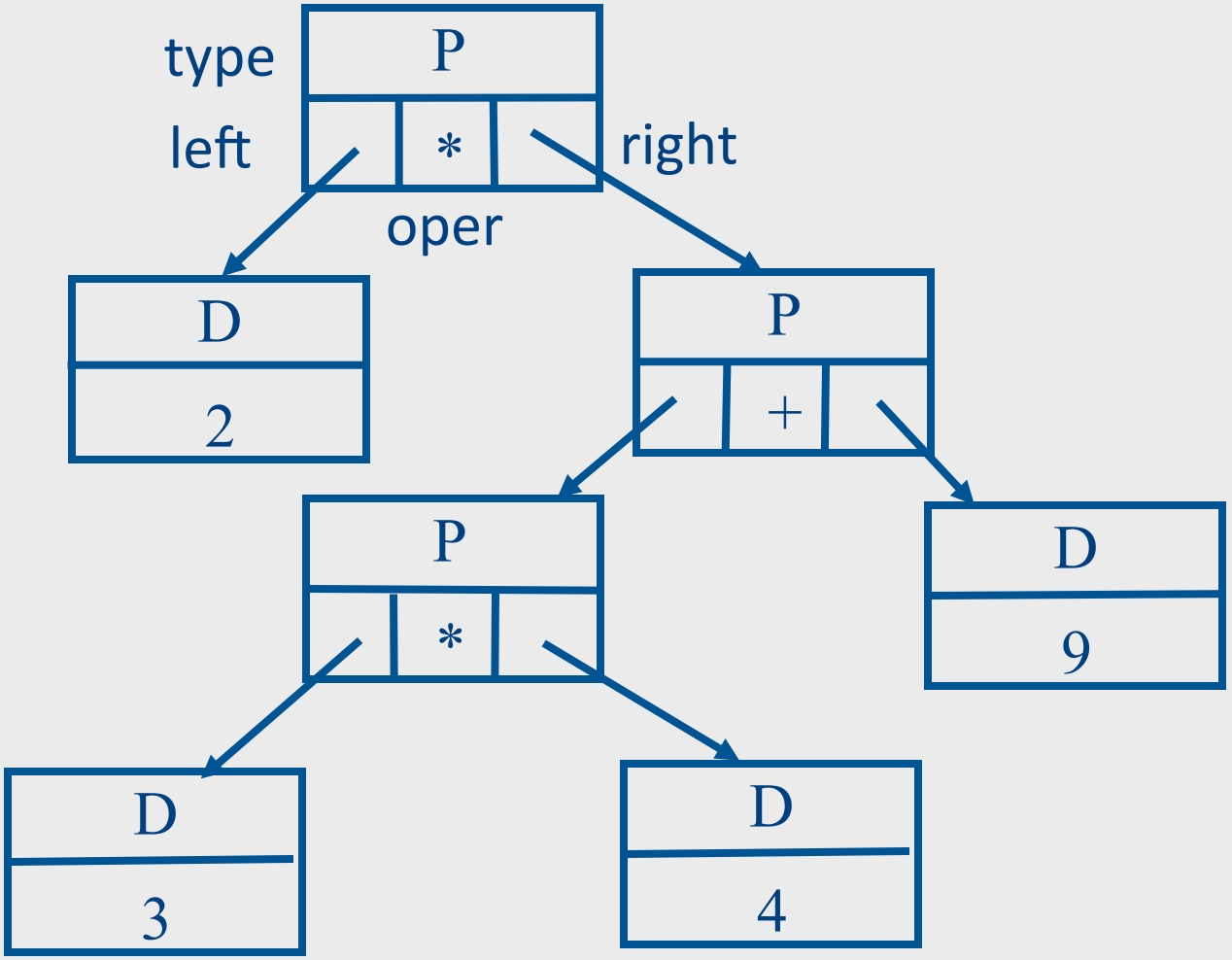
* Programs in our PL do not take input

Interpretation

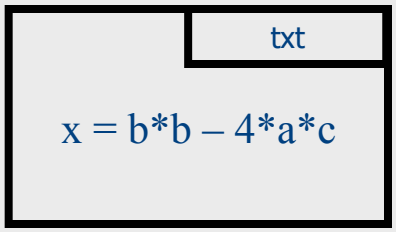
- Bottom-up evaluation of expressions
- The same interface of the compiler

```
#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

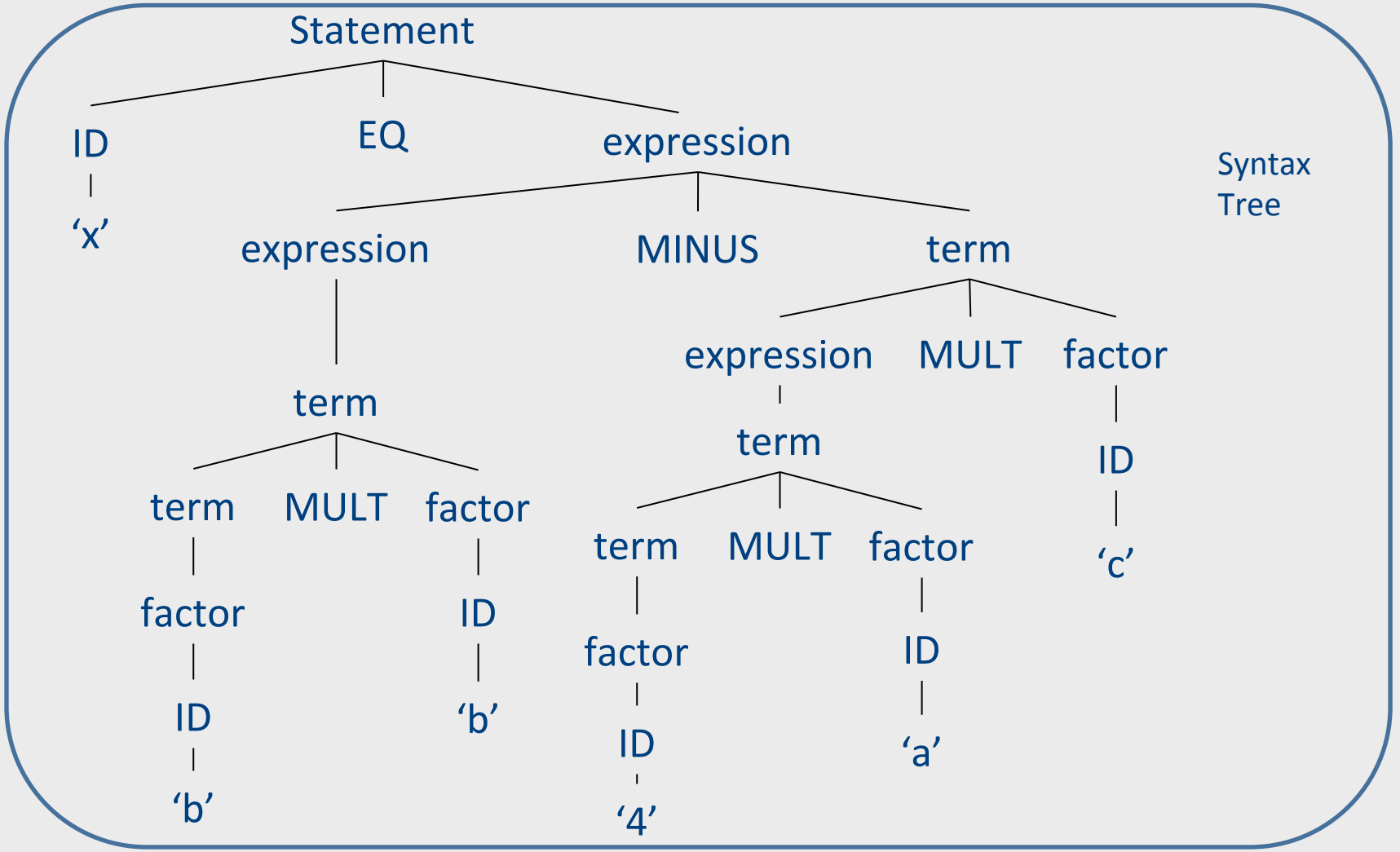


Token Stream

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

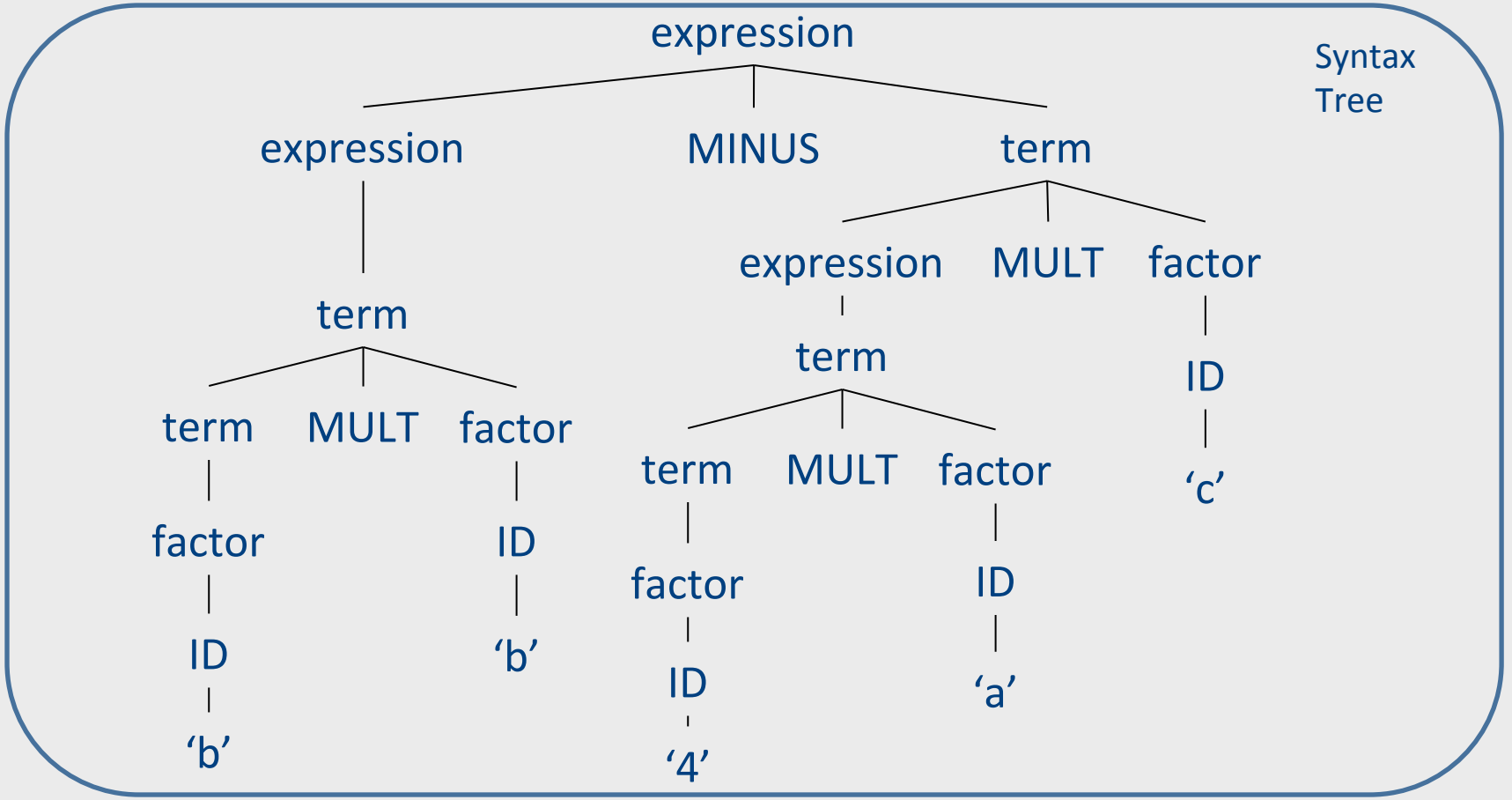
Summary: Journey inside a compiler

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



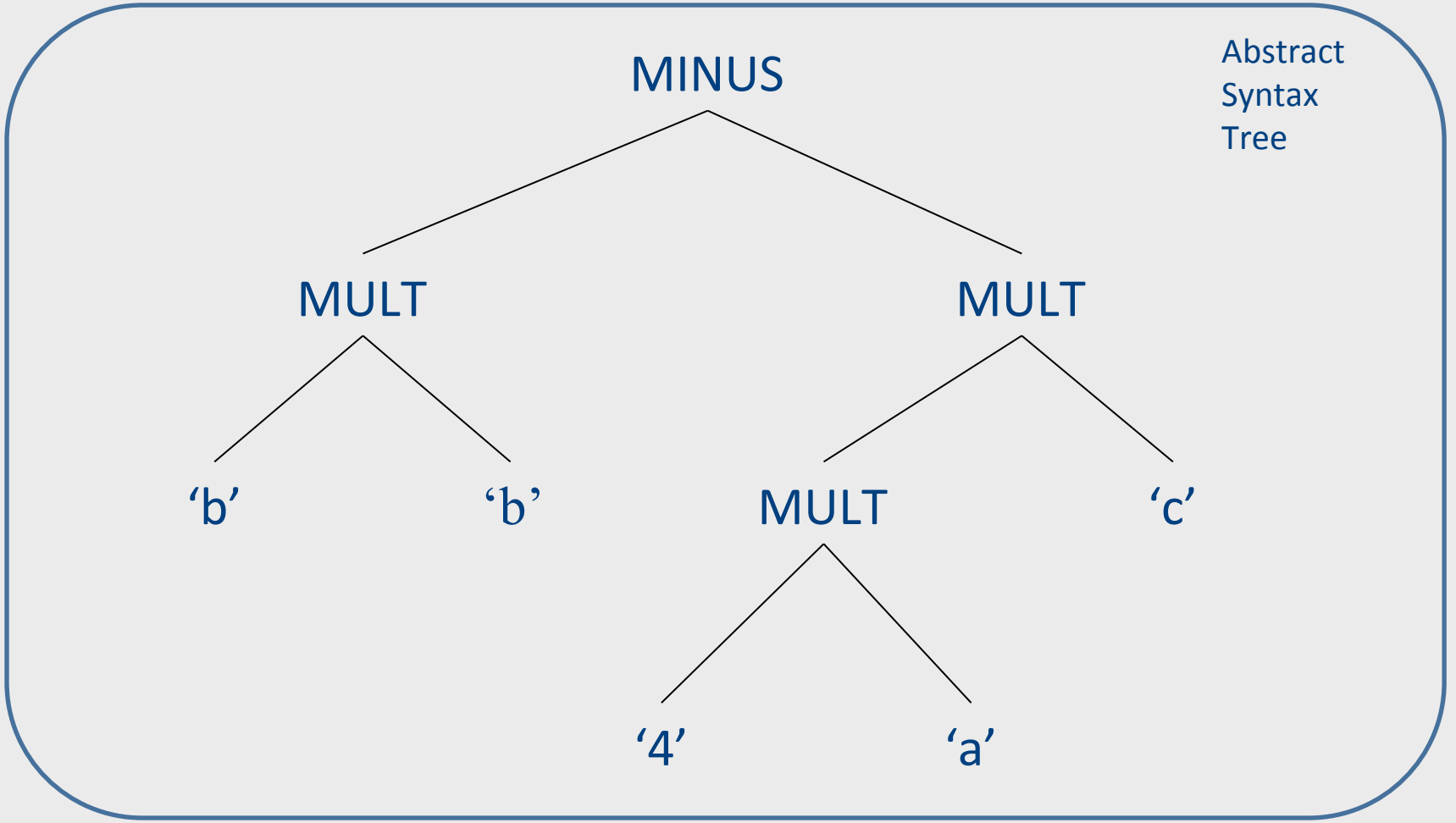
Summary: Journey inside a compiler

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

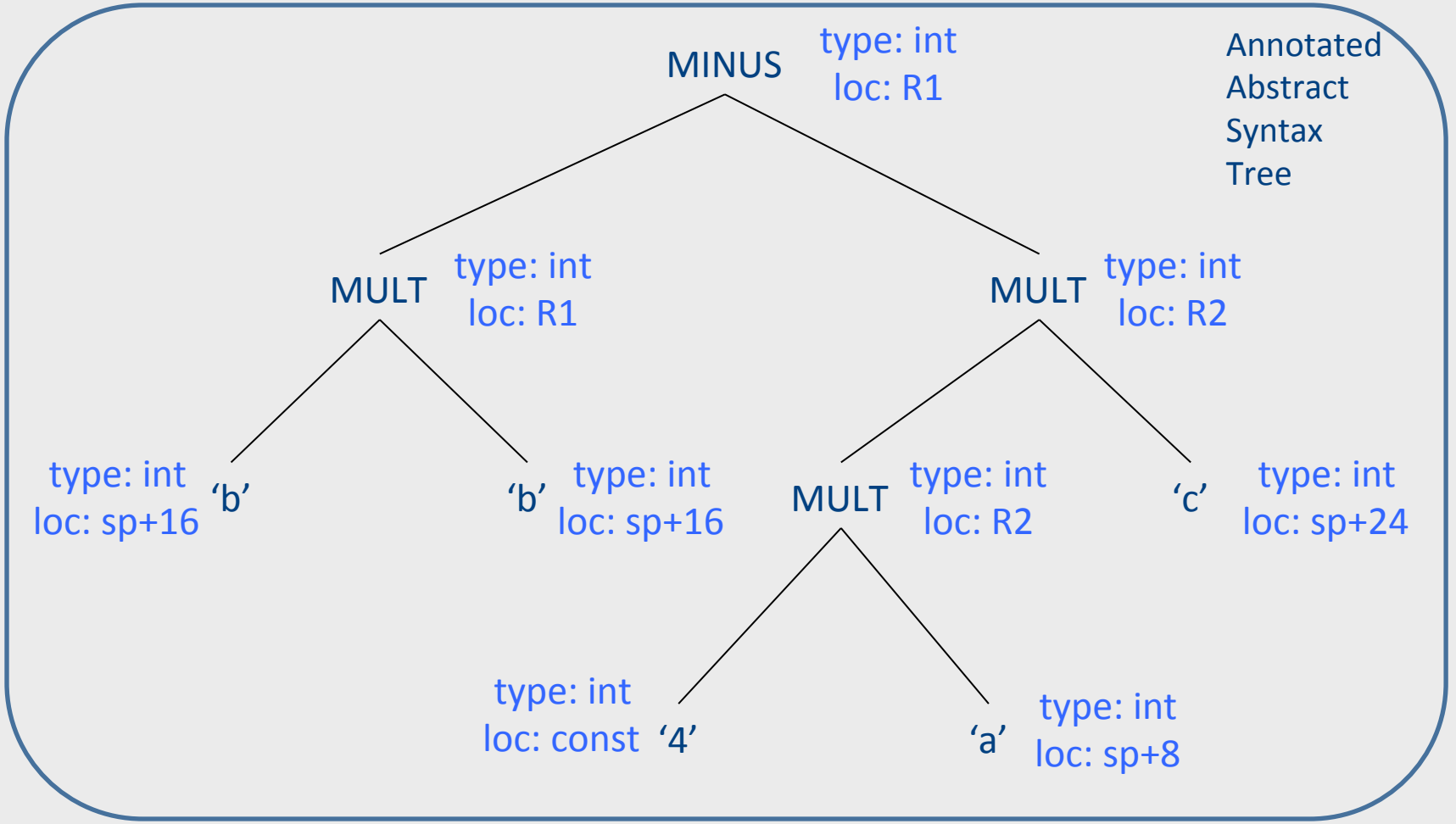


Lexical Analysis | Syntax Analysis | Sem. Analysis | Inter. Rep. | Code Gen.

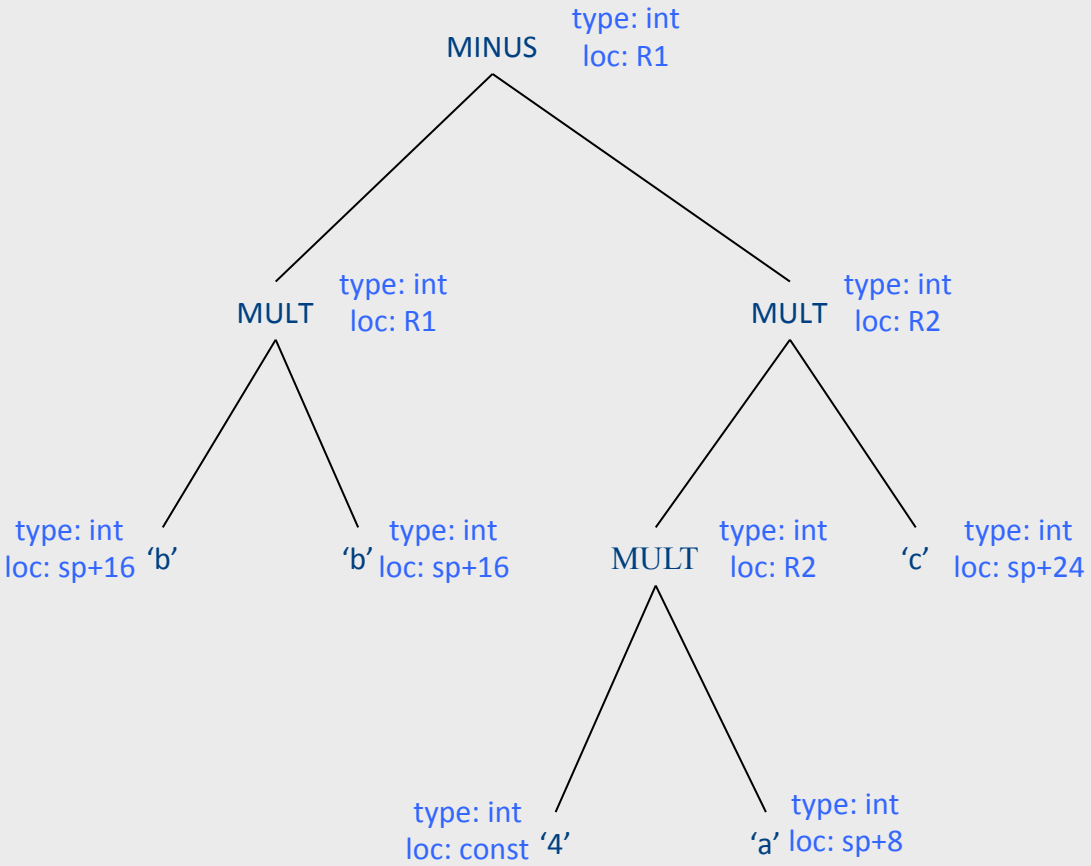
Summary: Journey inside a compiler



Summary: Journey inside a compiler



Journey inside a compiler



Intermediate Representation

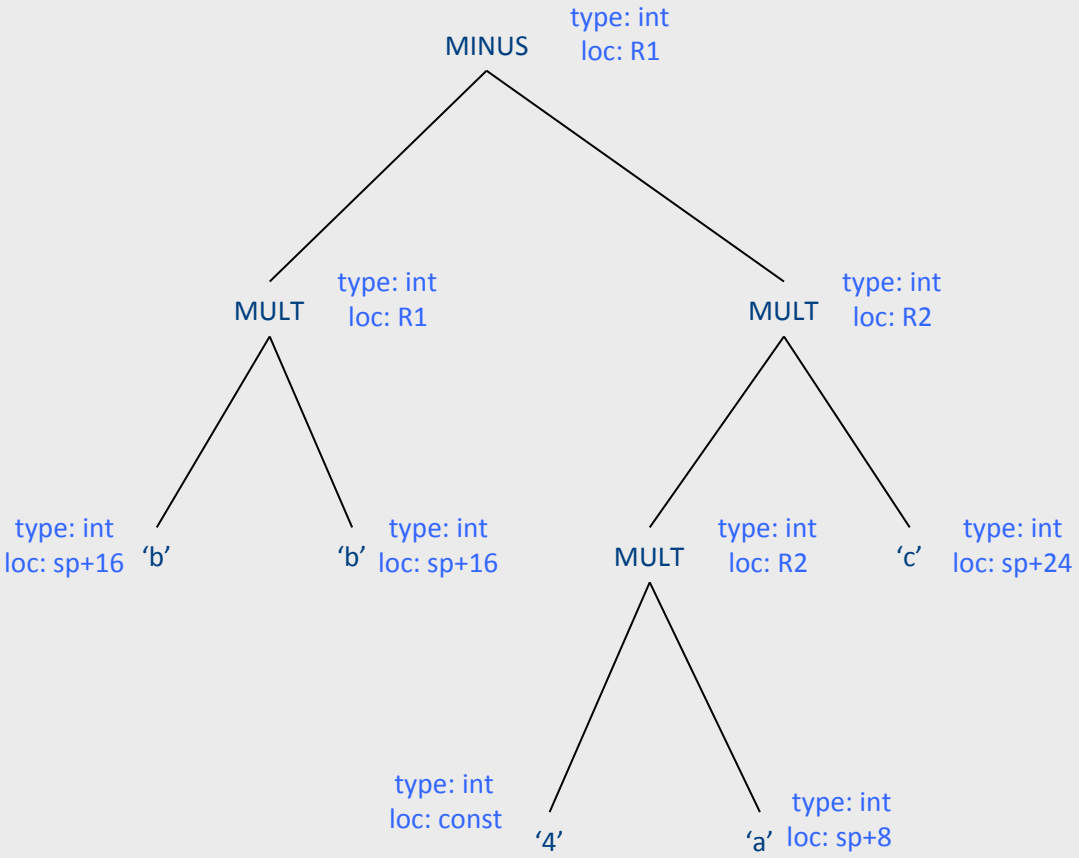
R2 = 4*a

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

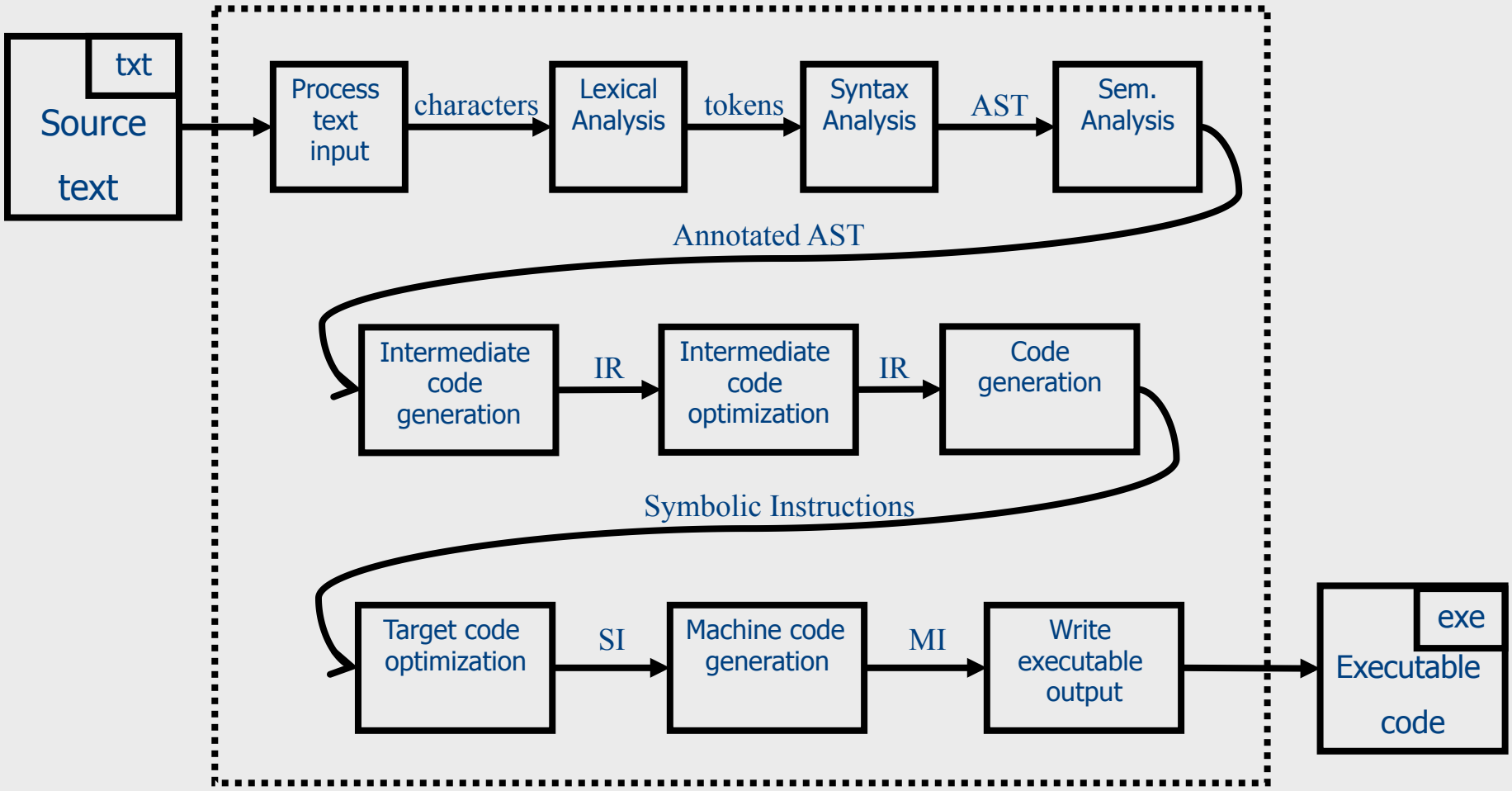
Error Checking

- 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

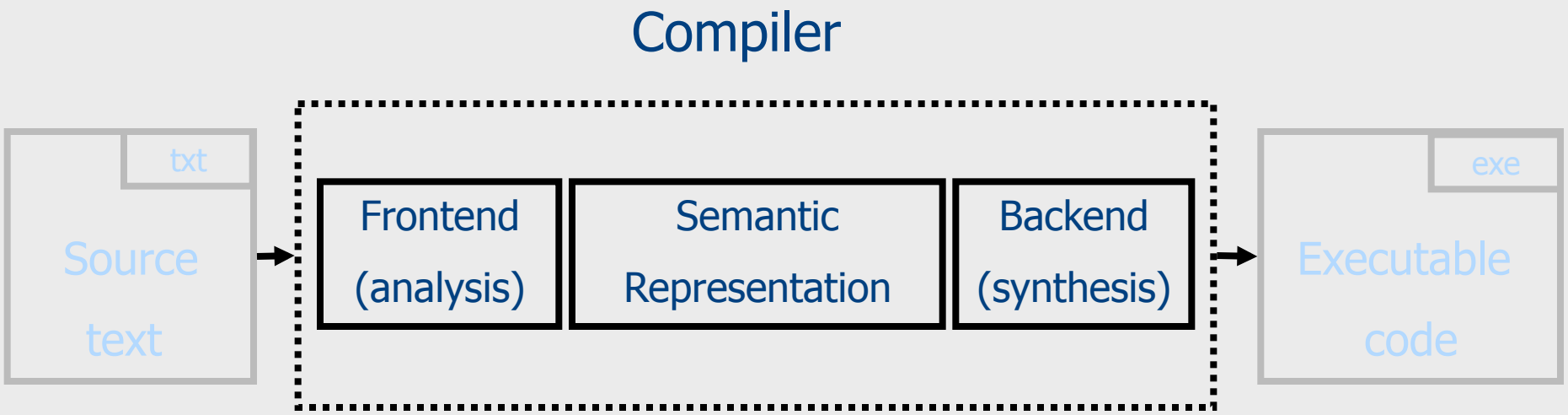
- What is a compiler
- How does it work
- (Reusable) techniques & tools

- Programming language implementation
  - runtime systems
- Execution environments
  - Assembly, linkers, loaders, OS

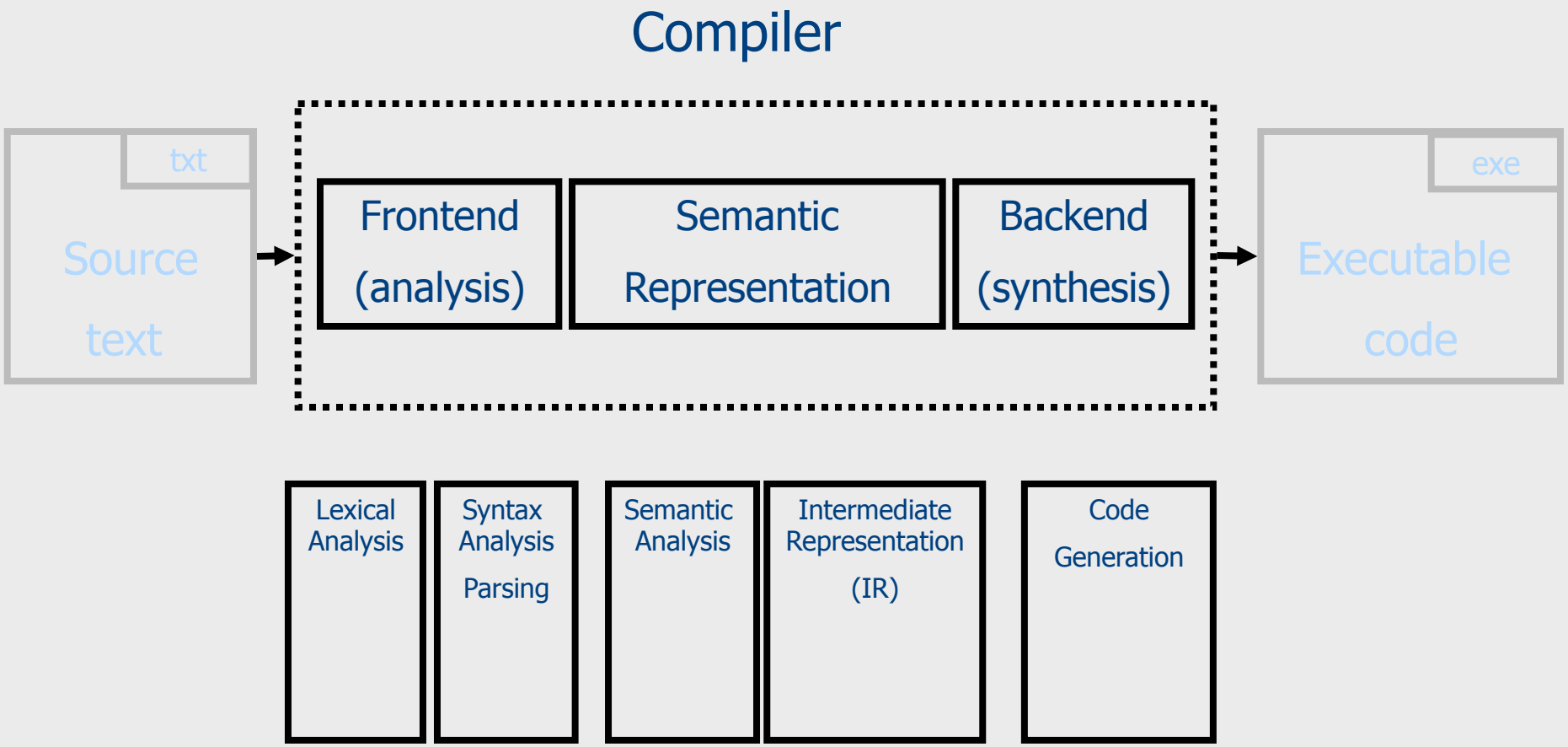
To Compilers, and Beyond …

- 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

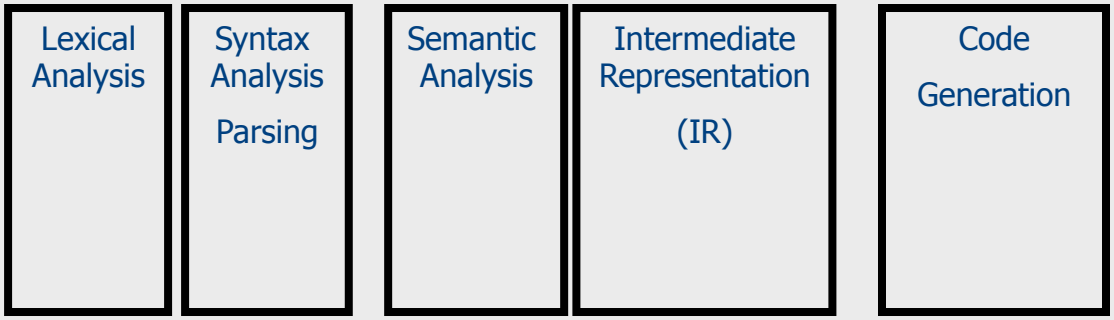


Conceptual Structure of a Compiler



Judicious use of formalisms

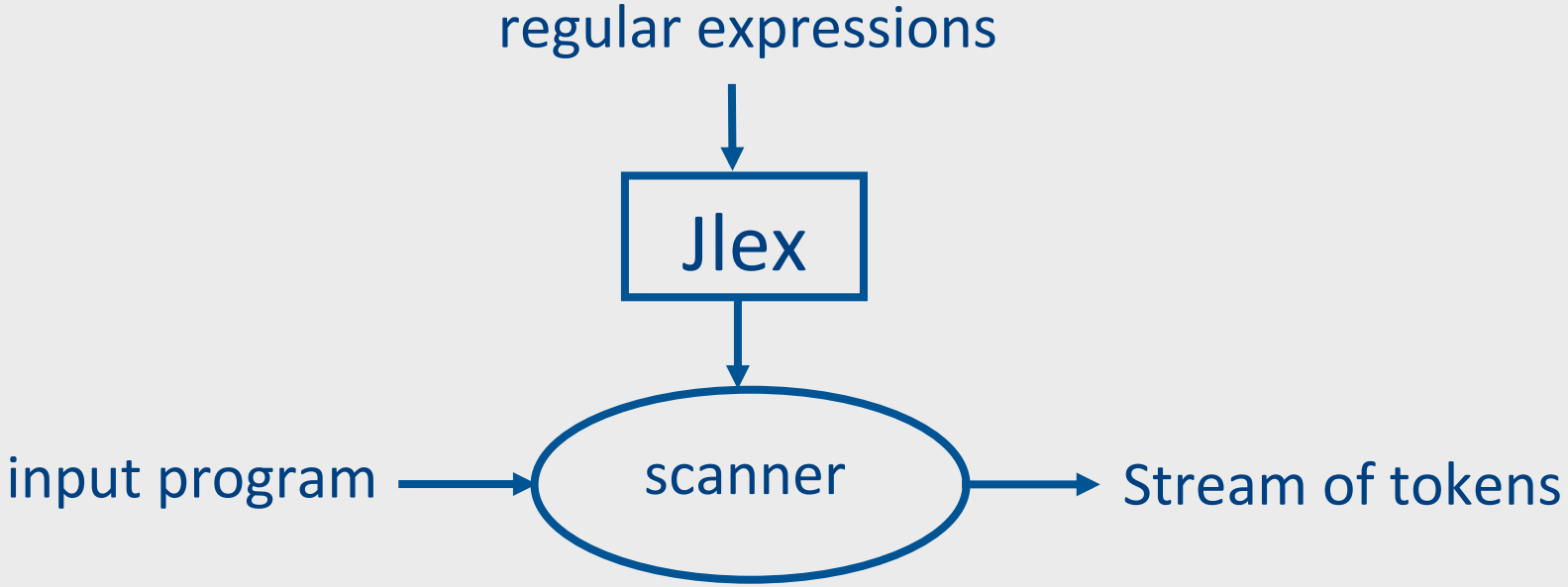
- Regular expressions (lexical analysis)
- Context-free grammars (syntactic analysis)
- Attribute grammars (context analysis)
- Code generator generators (dynamic programming)



- But also some nitty-gritty programming

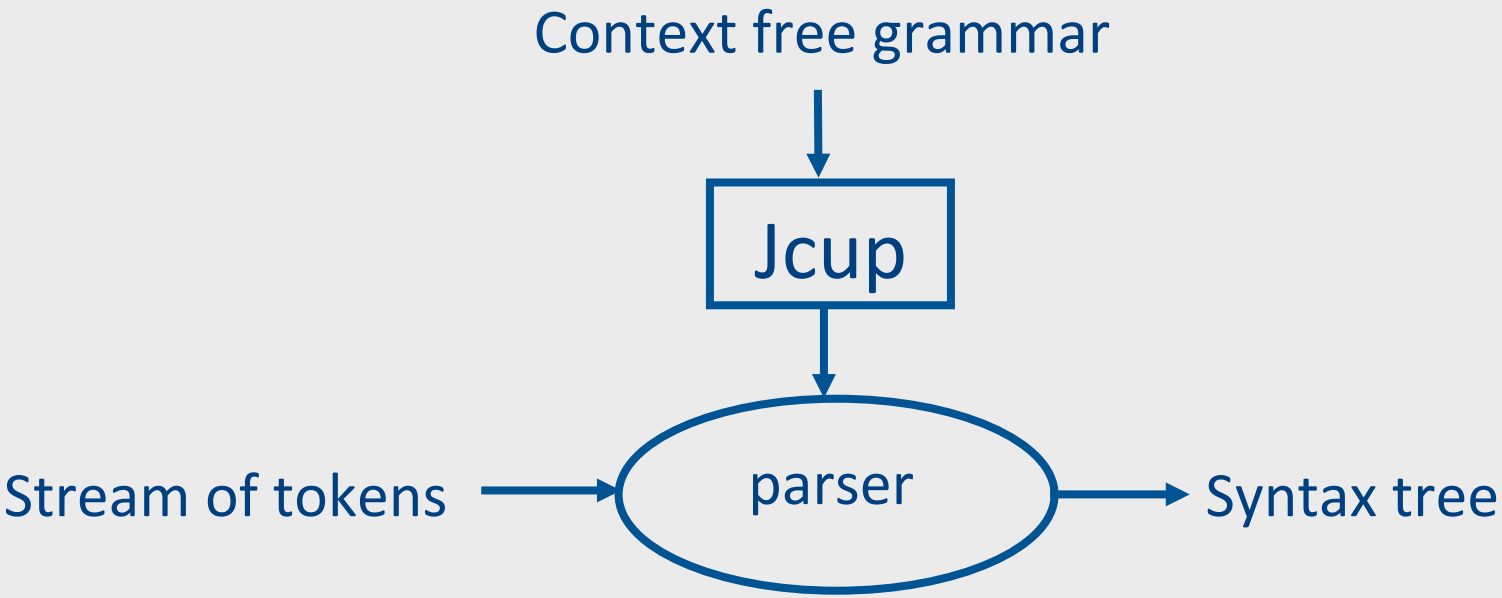
Use of program-generating tools

- Parts of the compiler are automatically generated from specification



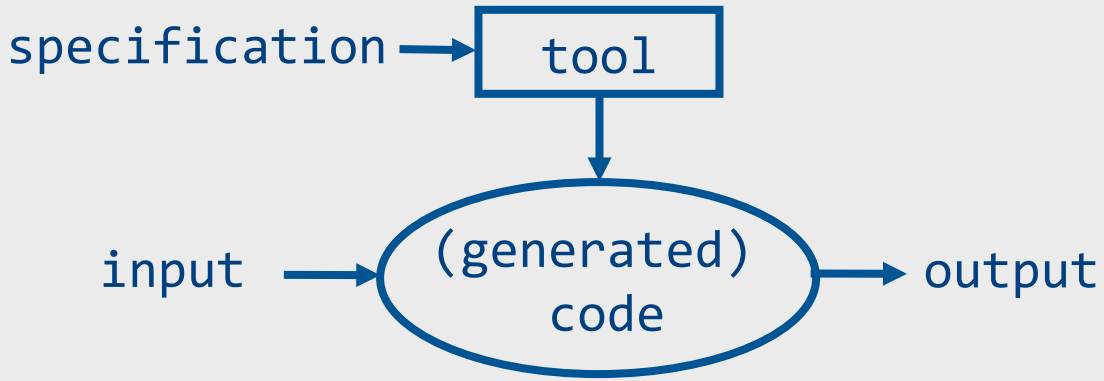
Use of program-generating tools

- 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

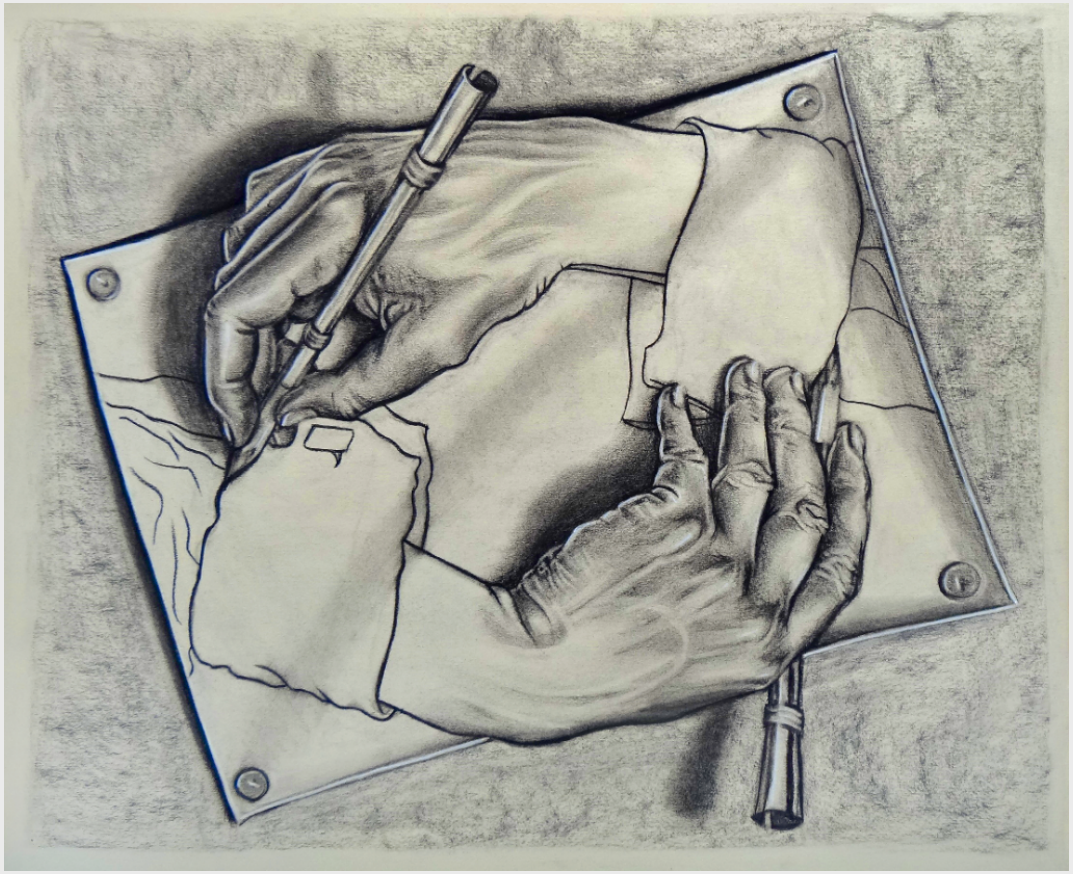
Wide applicability

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

Generally useful algorithms

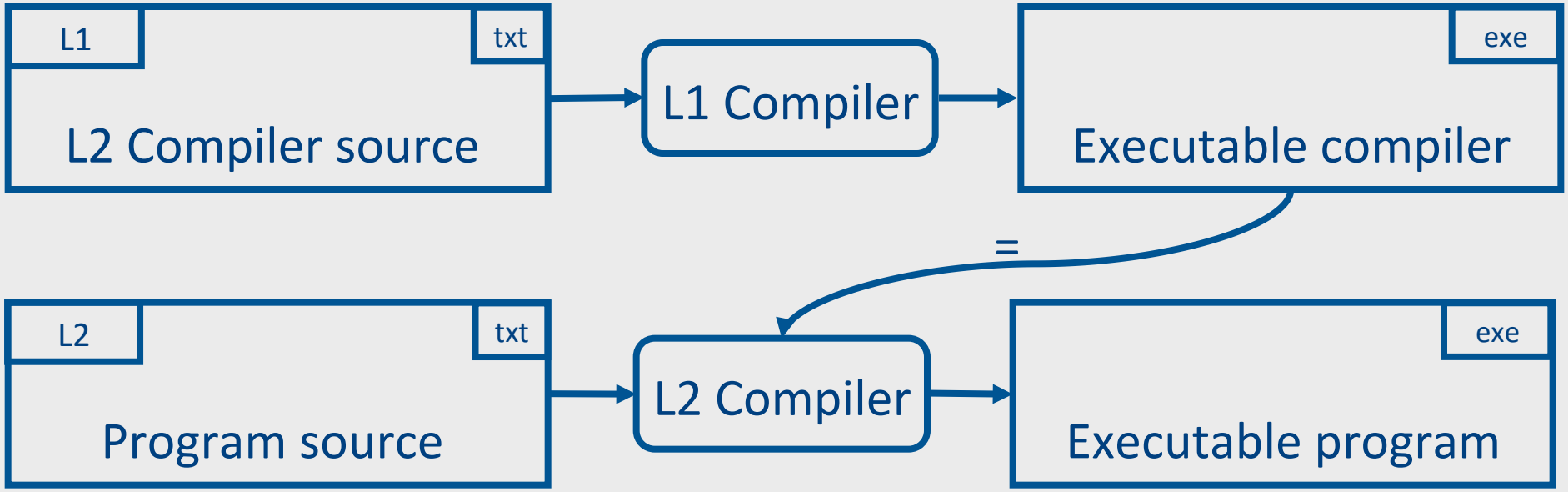
- Parser generators
- Garbage collection
- Dynamic programming
- Graph coloring

How to write a compiler?

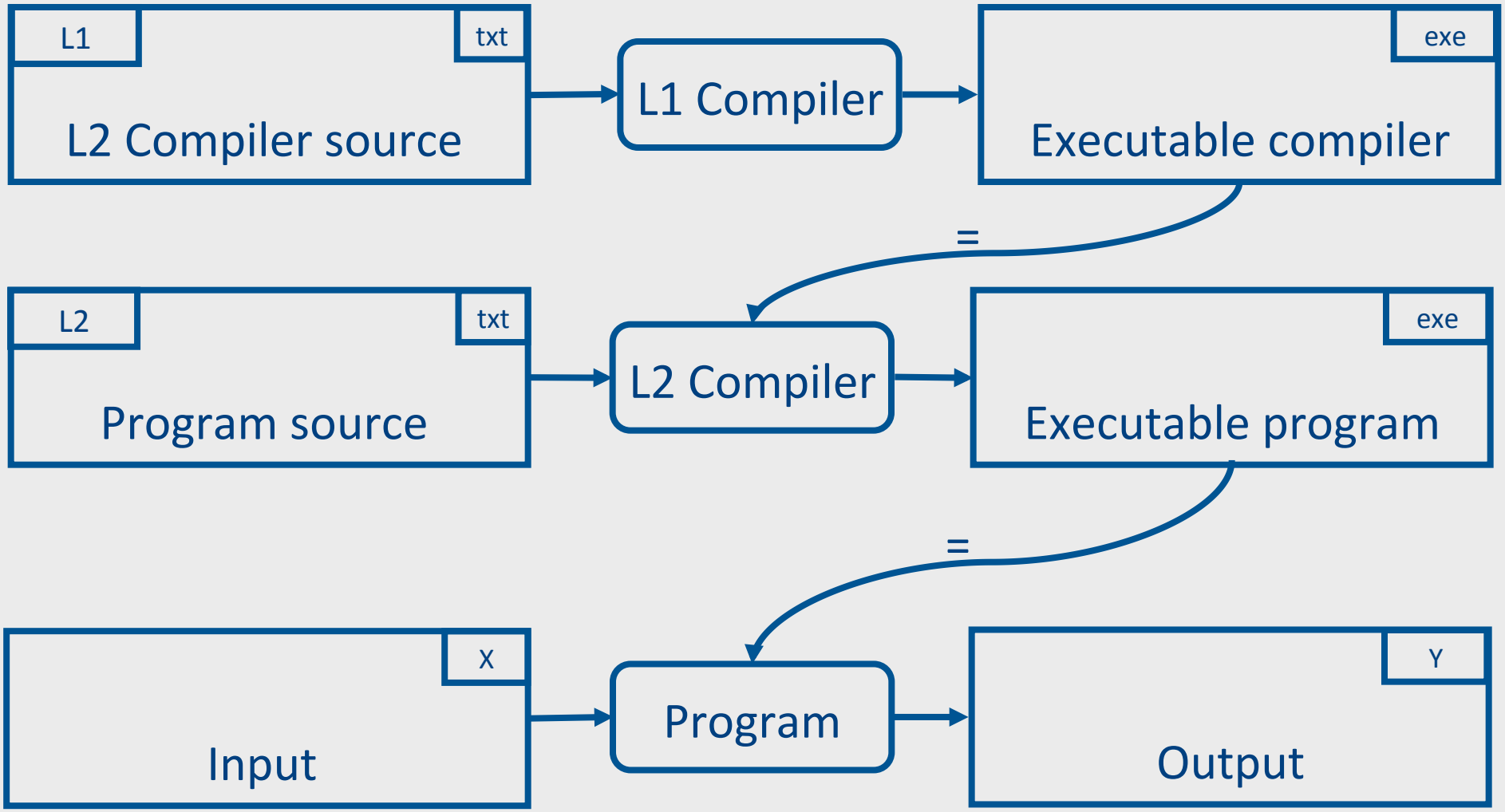


How to write a compiler?

How to write a compiler?

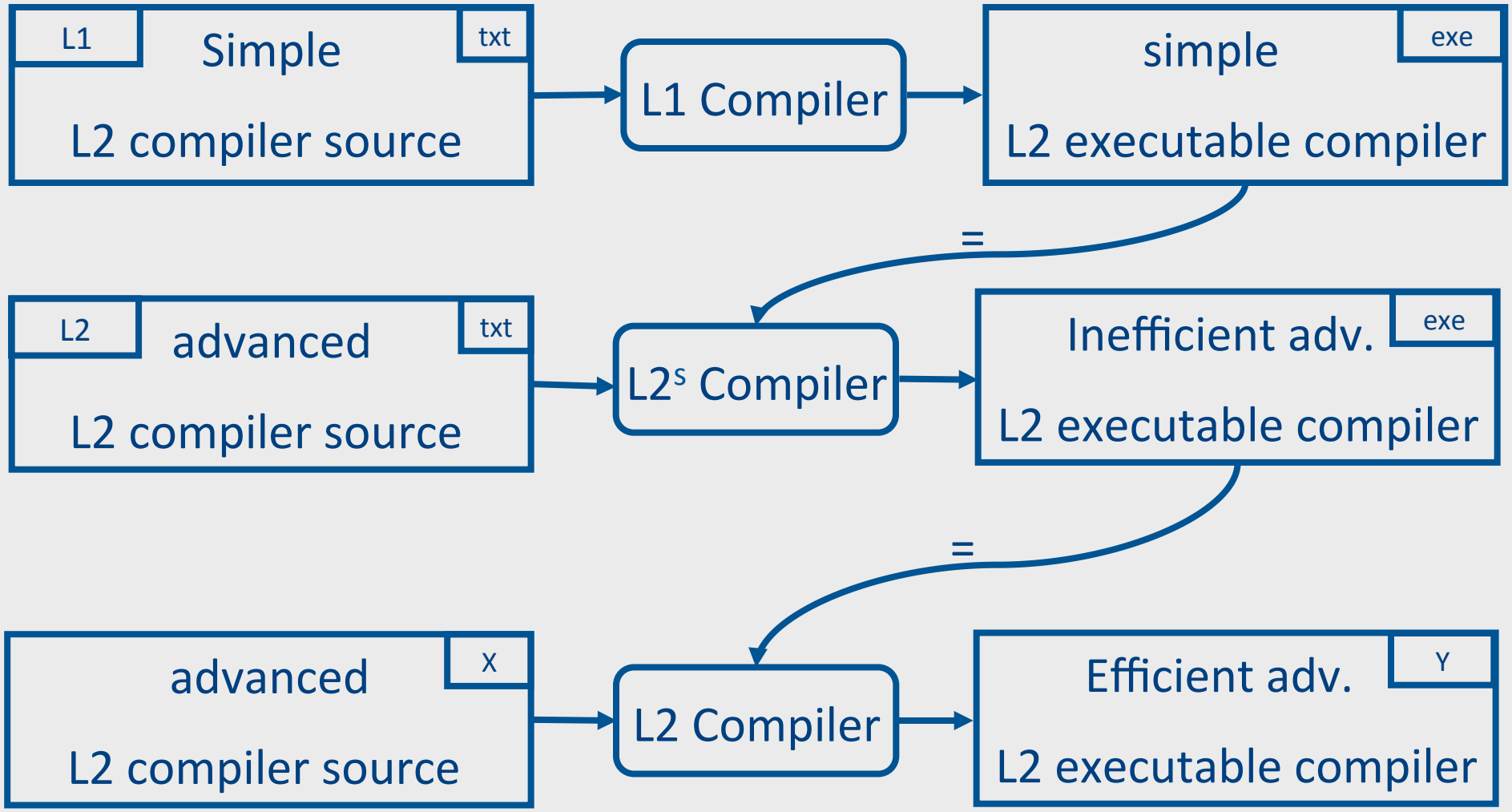


How to write a compiler?



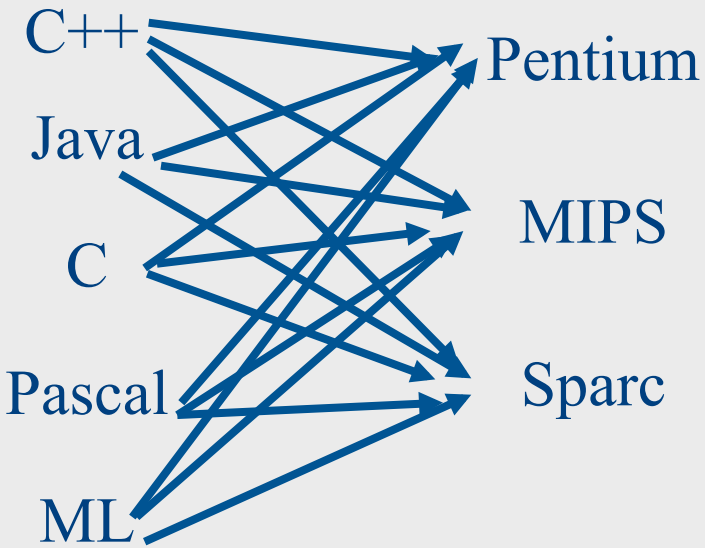
Bootstrapping a compiler

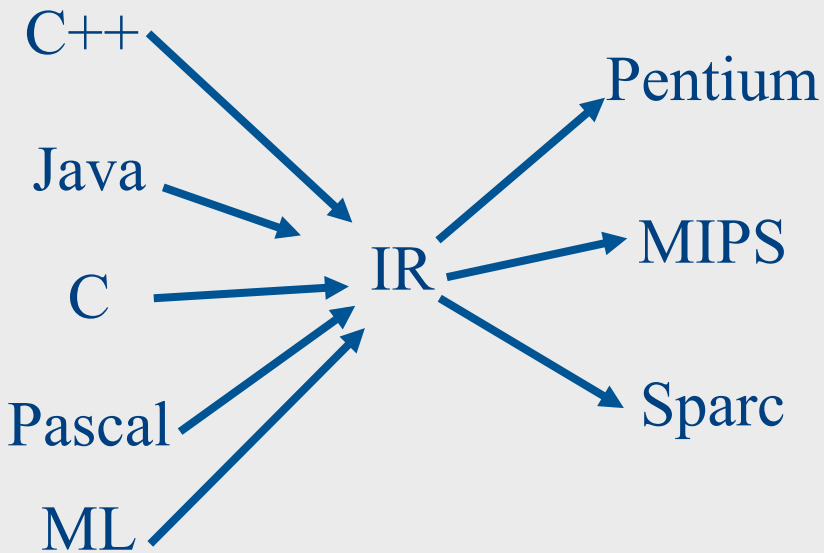
Bootstrapping a compiler



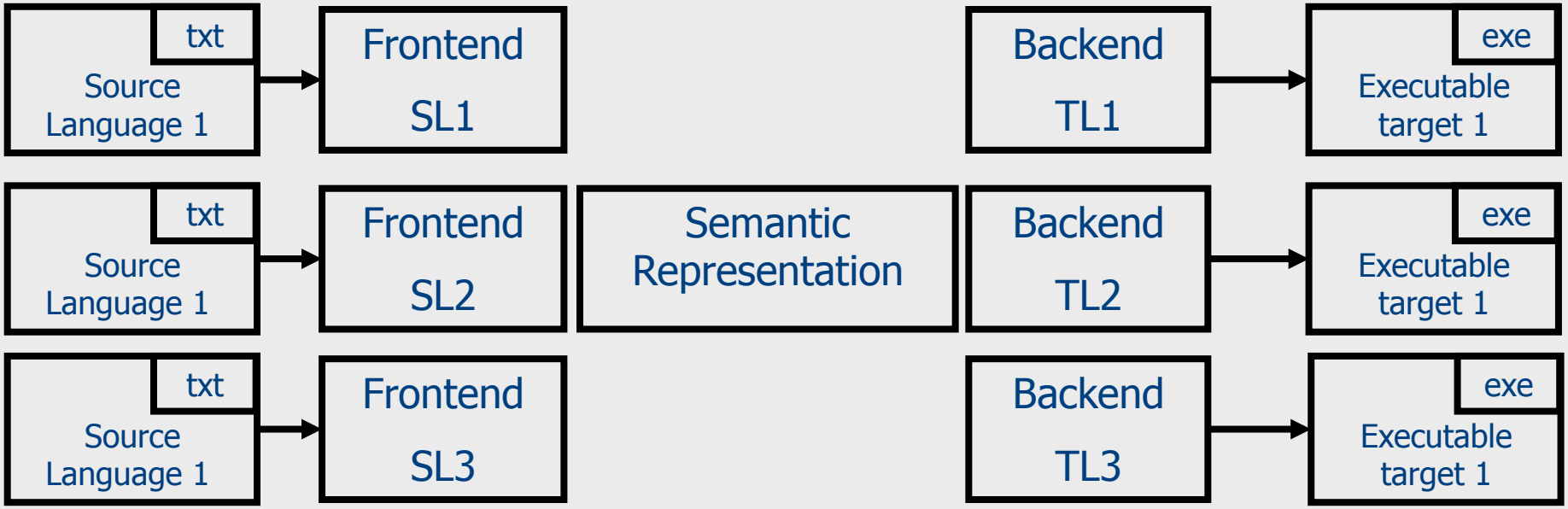
Proper Design

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





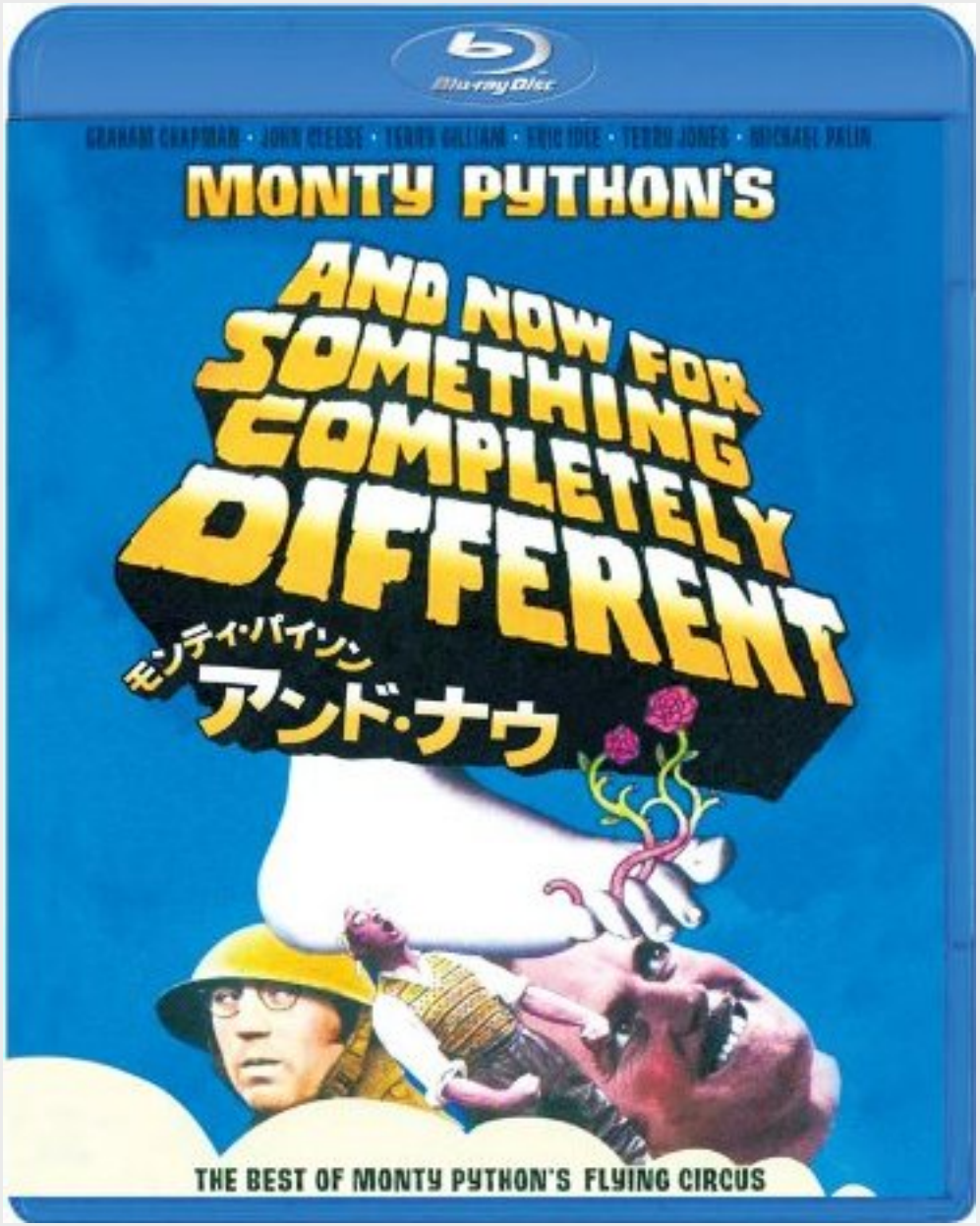
Modularity




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

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

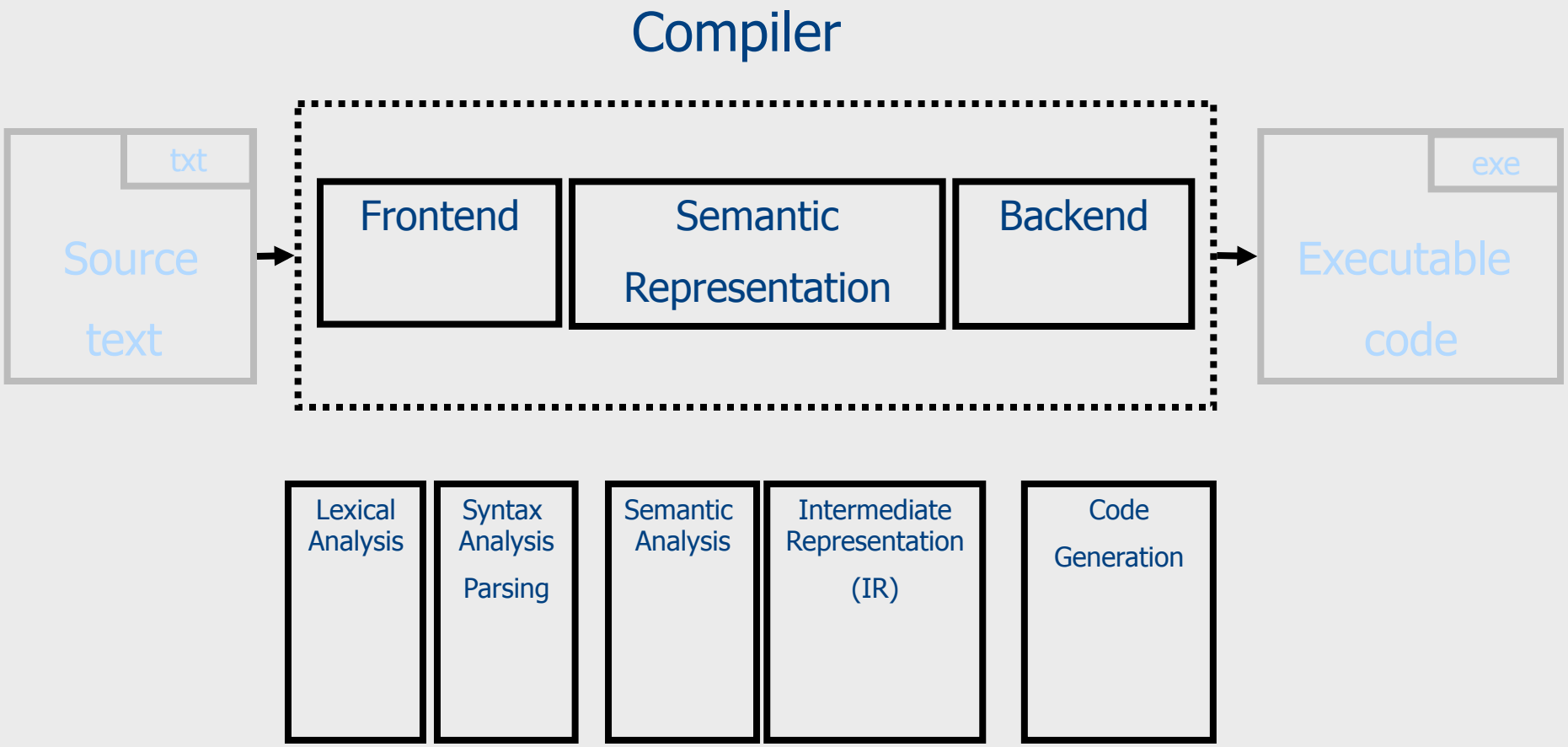
```
MOV R1,2
SAL R1
INC R1
MOV R2,R1
```


Blu-ray Disc
GRAHAM CHAPMAN · JOHN CLEESE · TERRY GILLIAM · ERIC IDLE · TERRY JONES · MICHAEL PALIN
MONTY PYTHON'S
AND NOW FOR SOMETHING COMPLETELY DIFFERENT
モンティ・パイソン
アンド・ナウ
THE BEST OF MONTY PYTHON'S FLYING CIRCUS

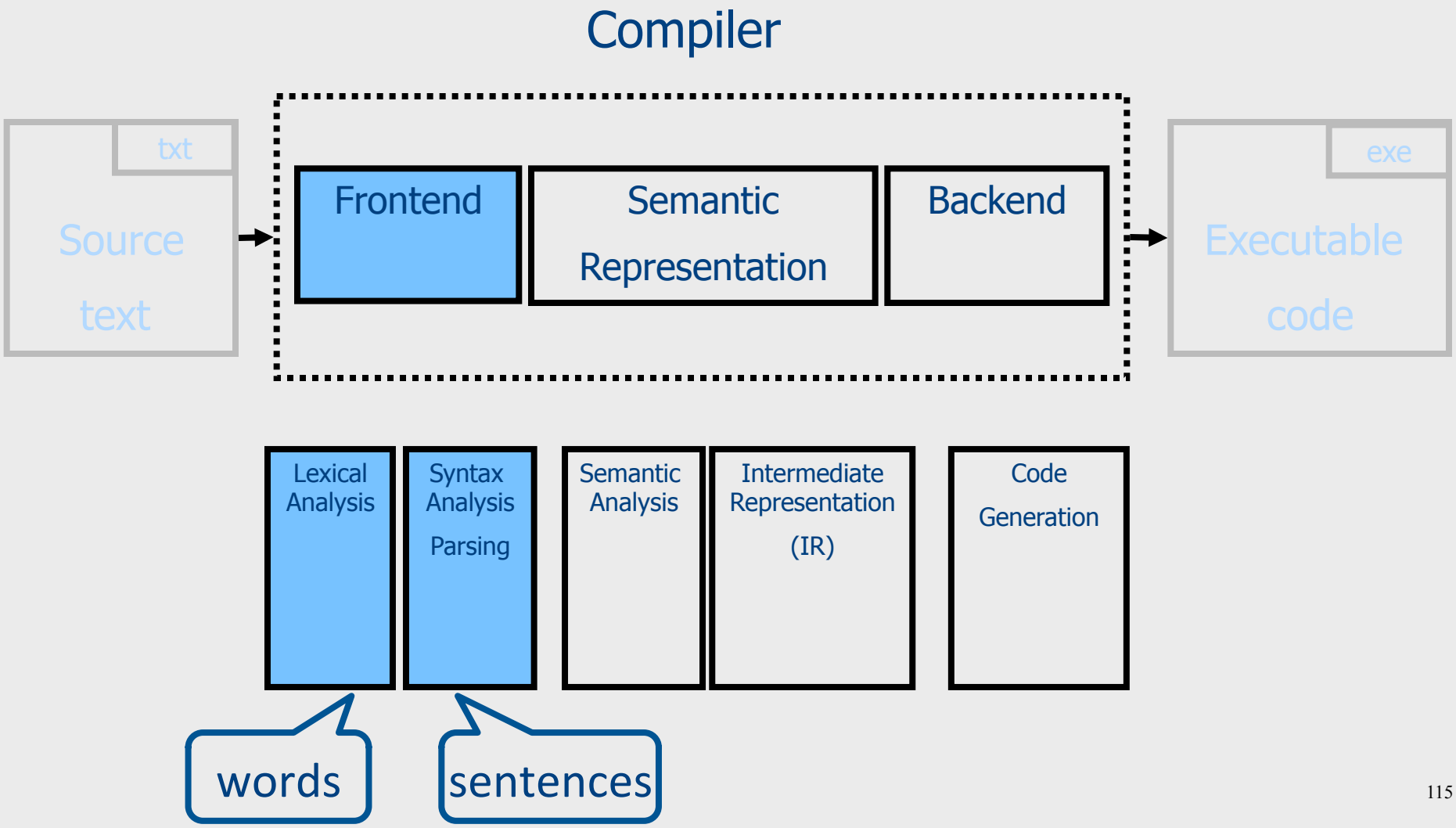
Lexical Analysis

Modern Compiler Design: Chapter 2.1

Conceptual Structure of a Compiler



Conceptual Structure of a Compiler



What does Lexical Analysis do?

- Language: fully parenthesized expressions

Expr → Num | LP Expr Op Expr RP

Num → Dig | Dig Num

Dig → ‘0’ | ‘1’ | ‘2’ | ‘3’ | ‘4’ | ‘5’ | ‘6’ | ‘7’ | ‘8’ | ‘9’

LP → ‘(’

RP → ‘)’

Op → ‘+’ | ‘*’

(	(	23	+	7	)	*	19	)

What does Lexical Analysis do?

- Language: fully parenthesized expressions

Context free language

Expr → Num | LP Expr Op Expr RP

Regular languages

Num → Dig | Dig Num

Dig → ‘0’ | ‘1’ | ‘2’ | ‘3’ | ‘4’ | ‘5’ | ‘6’ | ‘7’ | ‘8’ | ‘9’

LP → ‘(’

RP → ‘)’

Op → ‘+’ | ‘*’

(	(	23	+	7	)	*	19	)

What does Lexical Analysis do?

- Language: fully parenthesized expressions

Context free language

Expr → Num | LP Expr Op Expr RP

Regular languages

Num → Dig | Dig Num

Dig → ‘0’ | ‘1’ | ‘2’ | ‘3’ | ‘4’ | ‘5’ | ‘6’ | ‘7’ | ‘8’ | ‘9’

LP → ‘(’

RP → ‘)’

Op → ‘+’ | ‘*’

(	(	23	+	7	)	*	19	)

What does Lexical Analysis do?

- Language: fully parenthesized expressions

Context free language

Expr → Num | LP Expr Op Expr RP

Regular languages

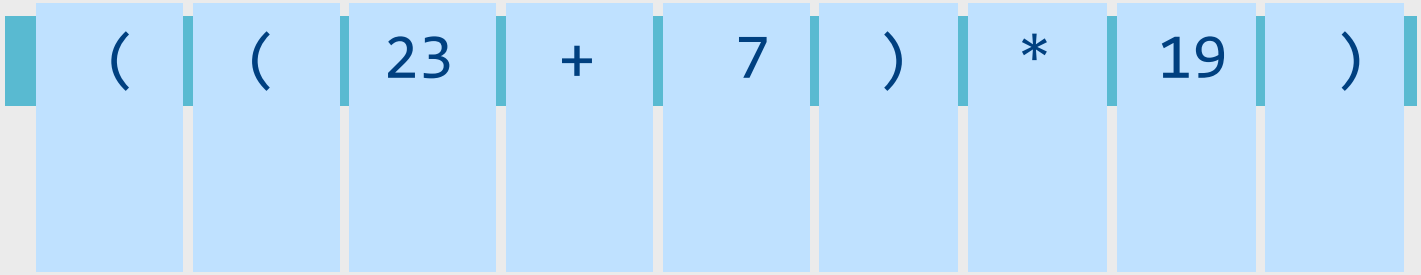
Num → Dig | Dig Num

Dig → ‘0’ | ‘1’ | ‘2’ | ‘3’ | ‘4’ | ‘5’ | ‘6’ | ‘7’ | ‘8’ | ‘9’

LP → ‘(’

RP → ‘)’

Op → ‘+’ | ‘*’



What does Lexical Analysis do?

- Language: fully parenthesized expressions

Context free language

Expr → Num | LP Expr Op Expr RP

Regular languages

Num → Dig | Dig Num

Dig → ‘0’ | ‘1’ | ‘2’ | ‘3’ | ‘4’ | ‘5’ | ‘6’ | ‘7’ | ‘8’ | ‘9’

LP → ‘(’

RP → ‘)’

Op → ‘+’ | ‘*’

(	(	23	+	7	)	*	19	)
LP	LP	Num	Op	Num	RP	Op	Num	RP

What does Lexical Analysis do?

- Language: fully parenthesized expressions

Context free language

Expr → Num | LP Expr Op Expr RP

Regular languages

Num → Dig | Dig Num

Dig → ‘0’ | ‘1’ | ‘2’ | ‘3’ | ‘4’ | ‘5’ | ‘6’ | ‘7’ | ‘8’ | ‘9’

LP → ‘(’

RP → ‘)’

Op → ‘+’ | ‘*’

Value	(	(	23	+	7	)	*	19	)
Kind	LP	LP	Num	Op	Num	RP	Op	Num	RP

What does Lexical Analysis do?

- Language: fully parenthesized expressions

Context free language

Expr → Num | LP Expr Op Expr RP

Regular languages

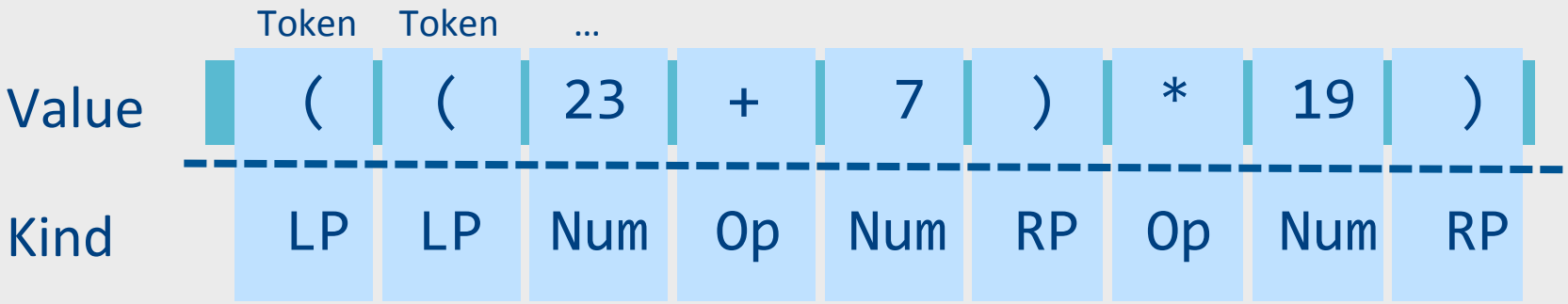
Num → Dig | Dig Num

Dig → ‘0’ | ‘1’ | ‘2’ | ‘3’ | ‘4’ | ‘5’ | ‘6’ | ‘7’ | ‘8’ | ‘9’

LP → ‘(’

RP → ‘)’

Op → ‘+’ | ‘*’



What does Lexical Analysis do?

- Partitions the input into stream of tokens
  - Numbers
  - Identifiers
  - Keywords
  - Punctuation

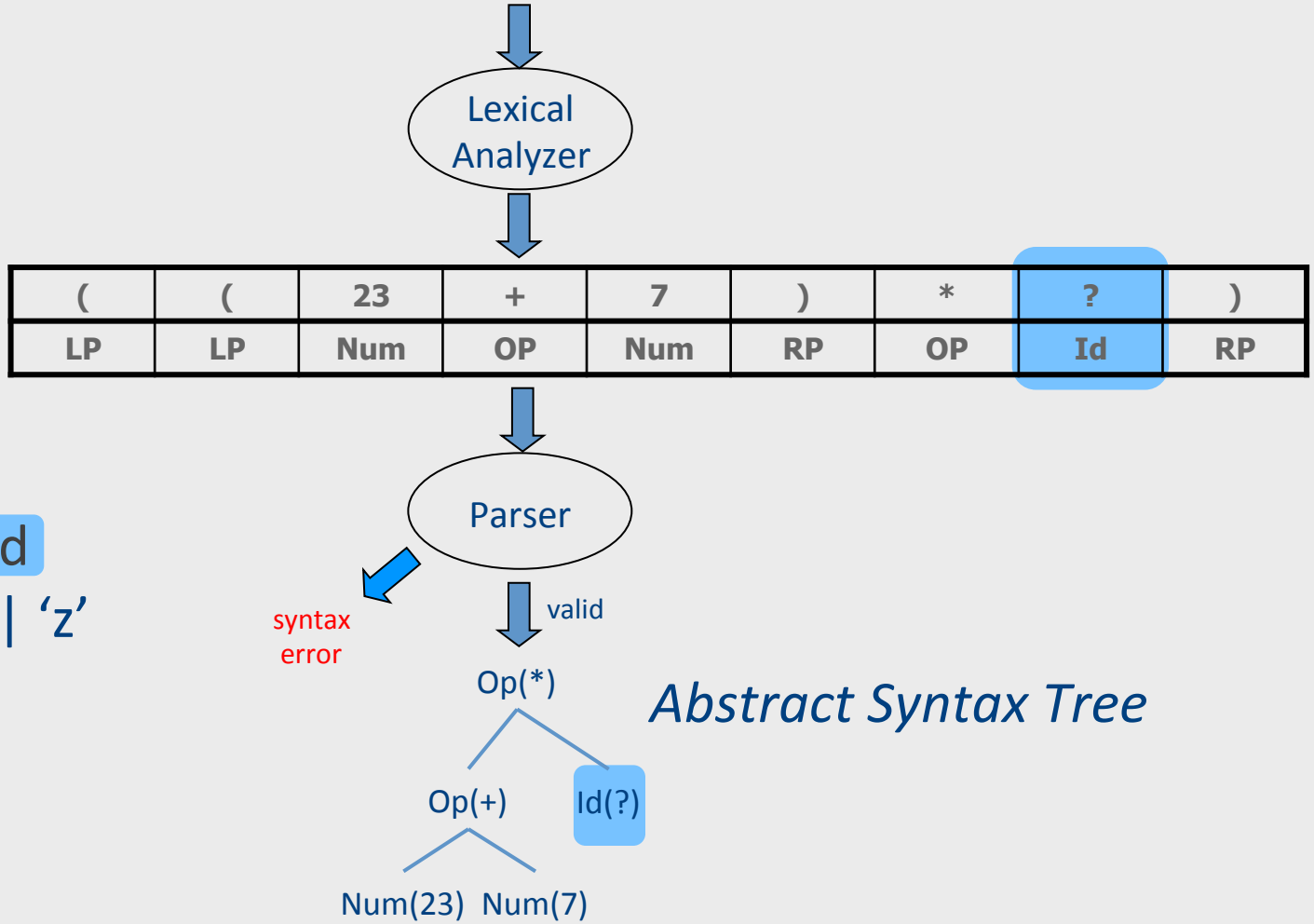
- "word" in the source language
- "meaningful" to the syntactical analysis

- Usually represented as (kind, value) pairs
  - (Num, 23)
  - (Op, '*')

From scanning to parsing

program text

((23 + 7) * x)



token stream

Grammar:

Expr → ... | Id

Id → ‘a’ | ... | ‘z’

Abstract Syntax Tree

Why Lexical Analysis?

- Well, not strictly necessary, but …
  - Regular languages $\subseteq$ Context-Free languages

- Simplifies the syntax analysis (parsing)
  - And language definition
- Modularity
- Reusability
- Efficiency

Lecture goals

- Understand role & place of lexical analysis
- Lexical analysis theory
- Using program generating tools

Lecture Outline

- ✓ Role & place of lexical analysis
- What is a token?
- Regular languages
- Lexical analysis
- Error handling
- Automatic creation of lexical analyzers

What is a token? (Intuitively)

- A “word” in the source language
  - Anything that should appear in the input to syntax analysis
    - Identifiers
    - Values
    - Language keywords
- Usually, represented as a pair of (kind, value)

Example Tokens

Type	**Examples**
ID	`foo, n_14, last`
NUM	`73, 00, 517, 082`
REAL	`66.1, .5, 5.5e-10`
IF	`if`
COMMA	`,`
NOTEQ	`!=`
LPAREN	`(`
RPAREN	`)`

Example Non Tokens

Type	**Examples**
comment	`/* ignored */`
preprocessor directive	`#include <foo.h>`
	`#define NUMS 5.6`
macro	`NUMS`
whitespace	`\t, \n, \b, ‘ ‘`

Some basic terminology

- Lexeme (aka symbol) - a series of letters separated from the rest of the program according to a convention (space, semi-column, comma, etc.)

- Pattern - a rule specifying a set of strings.
  Example: “an identifier is a string that starts with a letter and continues with letters and digits”
  - (Usually) a regular expression

- Token - a pair of (pattern, attributes)

Example

```
void match0(char *s) /* find a zero */
{
  if (!strncmp(s, “0.0”, 3))
    return 0. ;
}
```

```
VOID ID(match0) LPAREN CHAR DEREF ID(s) RPAREN
LBRACE
IF LPAREN NOT ID(strncmp) LPAREN ID(s) COMMA STRING(0.0)
COMMA NUM(3) RPAREN RPAREN
RETURN REAL(0.0) SEMI
RBRACE
EOF
```

Example Non Tokens

Type	**Examples**
comment	`/* ignored */`
preprocessor directive	`#include <foo.h>`
	`#define NUMS 5.6`
macro	`NUMS`
whitespace	`\t, \n, \b, ‘ ‘`

- Lexemes that are recognized but get consumed rather than transmitted to parser
  - If
  - i/*comment*/f

How can we define tokens?

- Keywords – easy!
  - if, then, else, for, while, …

- Identifiers?
- Numerical Values?
- Strings?

- Characterize unbounded sets of values using a bounded description?

Lecture Outline

- ✓ Role & place of lexical analysis
- ✓ What is a token?
- Regular languages
- Lexical analysis
- Error handling
- Automatic creation of lexical analyzers

Regular languages

- Formal languages
  - Σ = finite set of letters
  - Word = sequence of letter
  - Language = set of words

- Regular languages defined equivalently by
  - Regular expressions
  - Finite-state automata

Common format for reg-exps

Basic Patterns	Matching
x	The character x
.	Any character, usually except a new line
[xyz]	Any of the characters x,y,z
^x	Any character except x
Repetition Operators	
R?	An R or nothing (=optionally an R)
R*	Zero or more occurrences of R
R+	One or more occurrences of R
Composition Operators	
R1R2	An R1 followed by R2
R1|R2	Either an R1 or R2
Grouping	
(R)	R itself

Examples

- ab*|cd? =
- (a|b)* =
- (0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9)* =

Escape characters

- What is the expression for one or more + symbols?
  - `(+)+` won't work
  - `(\+)+` will
- backslash \ before an operator turns it to `standard character`
  - `\*, \?, \+, a\(b\+\*, (a\(b\+\*)+, …`
- backslash double quotes surrounds text
  - `“a(b+*”, “a(b+*”+`

Shorthands

- Use names for expressions
  - letter = a | b | … | z | A | B | … | Z
  - letter_ = letter | _
  - digit = 0 | 1 | 2 | … | 9
  - id = letter_ (letter_ | digit)*
- Use hyphen to denote a range
  - letter = a-z | A-Z
  - digit = 0-9

Examples

- if = if
- then = then
- relop = < | > | <= | >= | = | <>

- digit = 0-9
- digits = digit+

Example

- A number is

```
number = ( 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 )+
     ( ε | . ( 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 )+
           ( ε | E ( 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 )+
           )
)
```

- Using shorthands it can be written as

```
number = digits (ε | .digits (ε | E (ε|+|-) digits ) )
```

Exercise 1 - Question

- Language of rational numbers in decimal representation (no leading, ending zeros)
  - 0
  - 123.757
  - .933333
  - Not 007
  - Not 0.30

Exercise 1 - Answer

- Language of rational numbers in decimal representation (no leading, ending zeros)

Digit = 1|2|…|9
Digit0 = 0|Digit
Num = Digit Digit0*
Frac = Digit0* Digit
Pos = Num | .Frac | 0.Frac| Num.Frac
PosOrNeg = (Є|-)Pos
R = 0 | PosOrNeg

Exercise 2 - Question

- Equal number of opening and closing parenthesis: $[^n]^n$ = [], [[]], [[[]]], …

Exercise 2 - Answer

- Equal number of opening and closing parenthesis: $[^n]^n$ = [], [[]], [[[]]], …
- Not regular
- Context-free
- Grammar: S ::= [] | [S]

Challenge: Ambiguity

- if = if
- id = letter_ (letter_ | digit)*

- "if" is a valid word in the language of identifiers… so what should it be?
- How about the identifier "iffy"?

- Solution
  - Always find longest matching token
  - Break ties using order of definitions… first definition wins (=> list rules for keywords before identifiers)

Creating a lexical analyzer

- Given a list of token definitions (pattern name, regex), write a program such that
  - Input: String to be analyzed
  - Output: List of tokens

- How do we build an analyzer?


That's all Folks