

Course Overview

Mooly Sagiv

msagiv@tau.ac.il

TA: Guy Golan-Gueta

guyg.guyg@gmail.com

<http://www.cs.tau.ac.il/~msagiv/courses/wcc10.html>

Textbook: Modern Compiler Design

Grune, Bal, Jacobs, Langendoen

CS0368-3133-01@listserv.tau.ac.il

Outline

- Course Requirements
- High Level Programming Languages
- Interpreters vs. Compilers
- Why study compilers (1.1)
- A simple traditional modern compiler/interpreter (1.2)
- Subjects Covered
- Summary

Course Requirements

- Compiler Project 50%
 - Translate Java Subset into X86
- Final exam 50% (must pass)

Lecture Goals

- Understand the basic structure of a compiler
- Compiler vs. Interpreter
- Techniques used in compilers

High Level Programming Languages

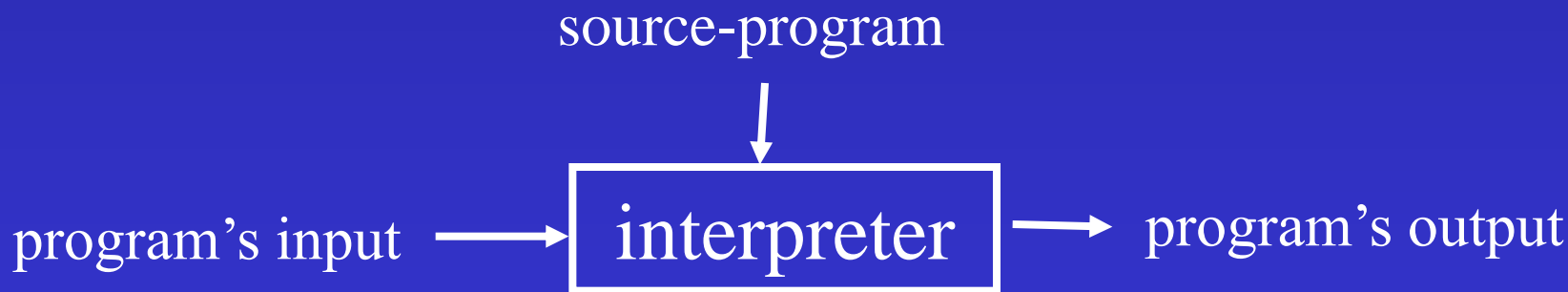
- Imperative
 - Algol, PL1, Fortran, Pascal, Ada, Modula, and C
 - Closely related to “von Neumann” Computers
- Object-oriented
 - Simula, Smalltalk, Modula3, C++, Java, C#, Python
 - Data abstraction and ‘evolutionary’ form of program development
 - **Class** An implementation of an abstract data type (data+code)
 - **Objects** Instances of a class
 - **Fields** Data (structure fields)
 - **Methods** Code (procedures/functions with overloading)
 - **Inheritance** Refining the functionality of a class with different fields and methods
- Functional
 - Lisp, Scheme, ML, Miranda, Hope, Haskell, OCaml, F#
- Functional/Imperative
 - Ruby
- Logic Programming
 - Prolog

Other Languages

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

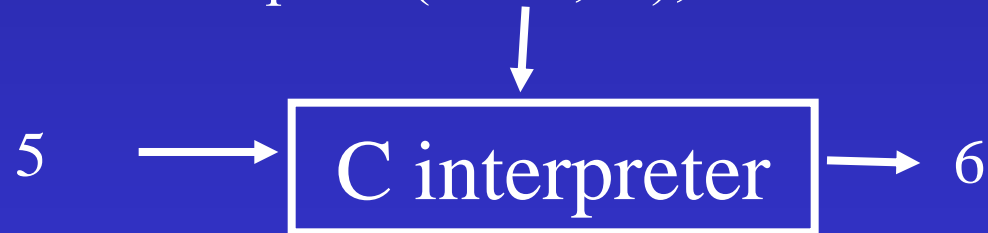
Interpreter

- A program which interprets instructions
- **Input**
 - A program
 - An input for the program
- **Output**
 - The required output



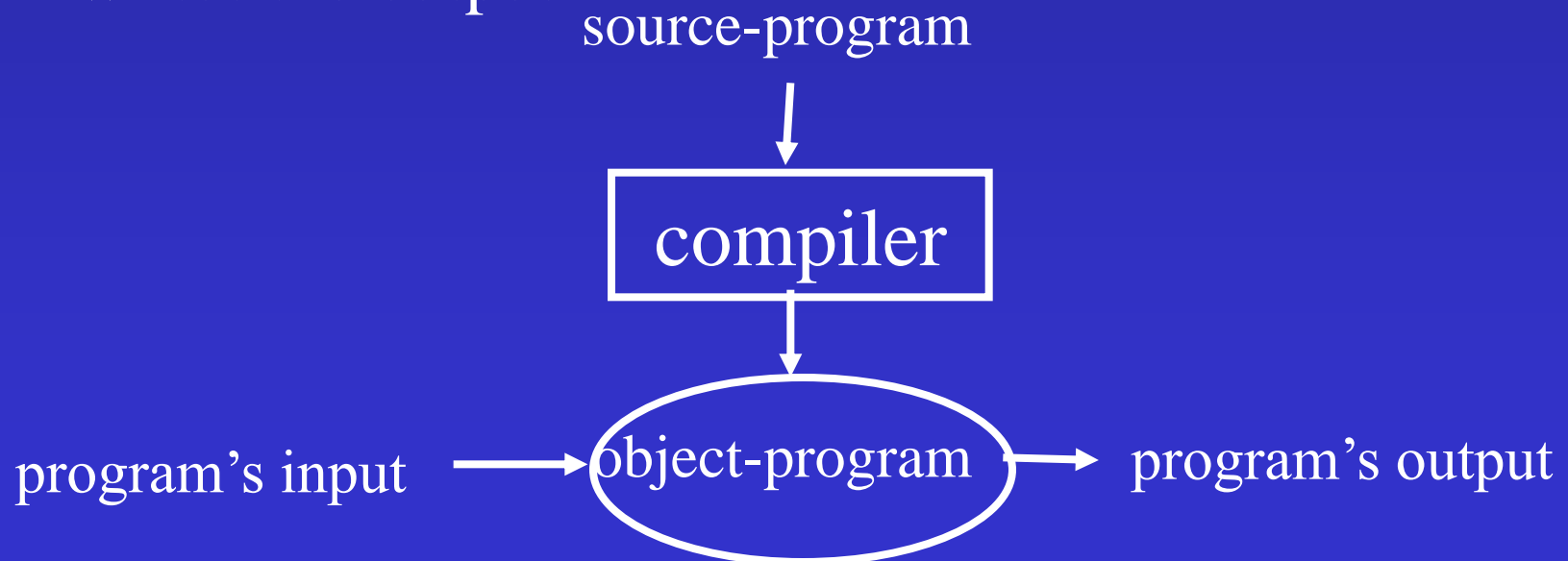
Example

```
int x;  
scanf("%d", &x);  
x = x + 1 ;  
printf("%d", x);
```



Compiler

- A program which compiles instructions
- **Input**
 - A program
- **Output**
 - An object program that reads the input and writes the output



Example

```
int x;  
scanf("%d", &x);  
x = x + 1 ;  
printf("%d", x);
```

Sparc-cc-compiler

```
add  %fp,-8, %11  
mov  %11, %o1  
call scanf  
ld   [%fp-8],%10  
add  %10,1,%10  
st   %10,[%fp-8]  
ld   [%fp-8], %11  
mov  %11, %o1  
call printf
```

assembler/linker

object-program

5

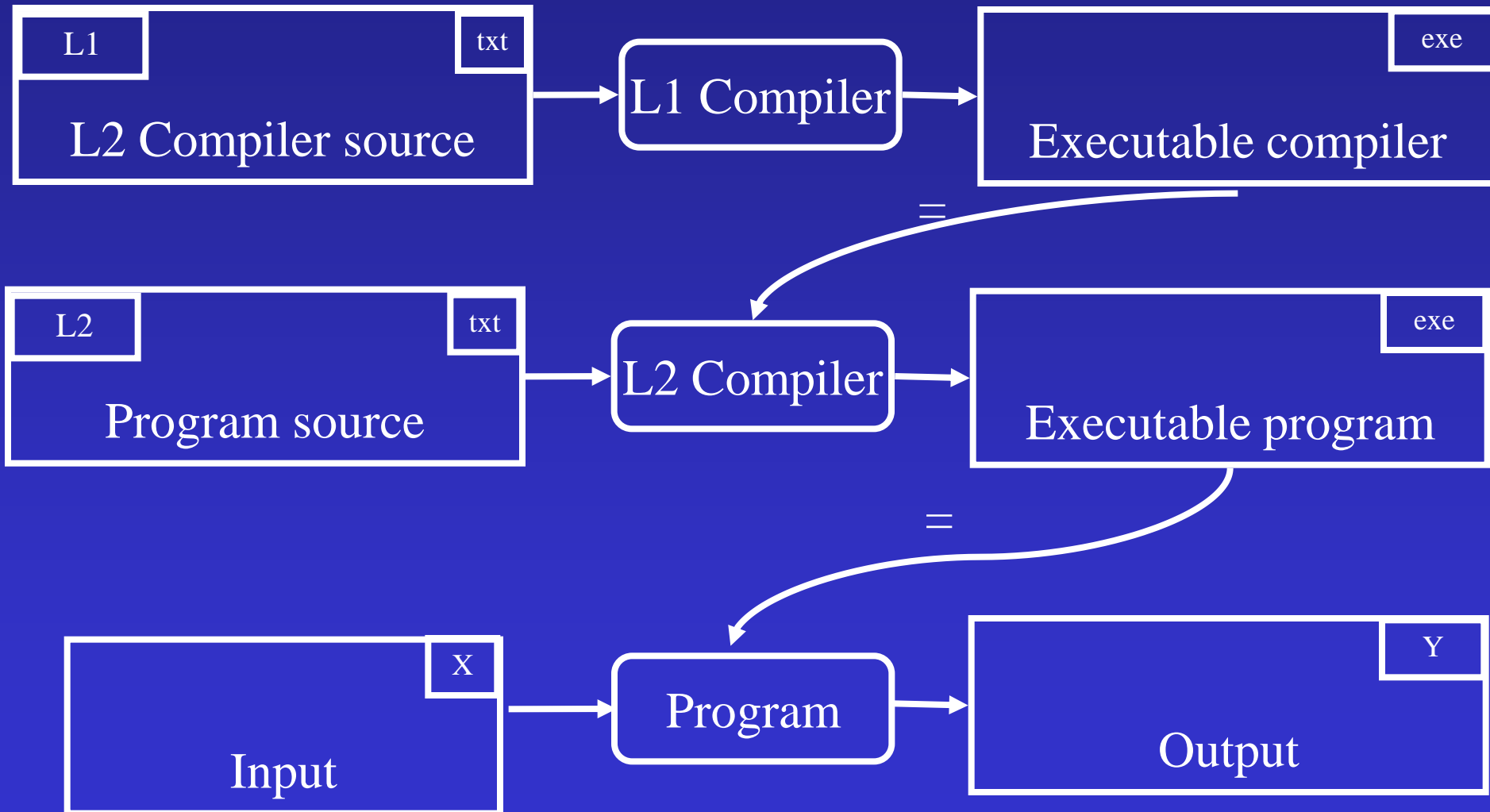


6

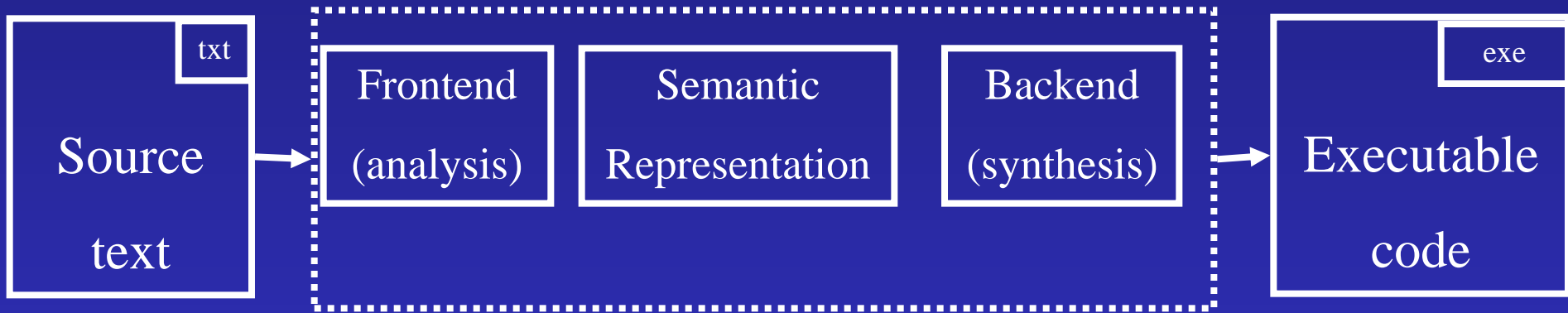
Remarks

- Both compilers and interpreters are programs written in high level languages
- Requires additional step to compile the compiler/interpreter
- Compilers and interpreters share functionality

Bootstrapping a compiler

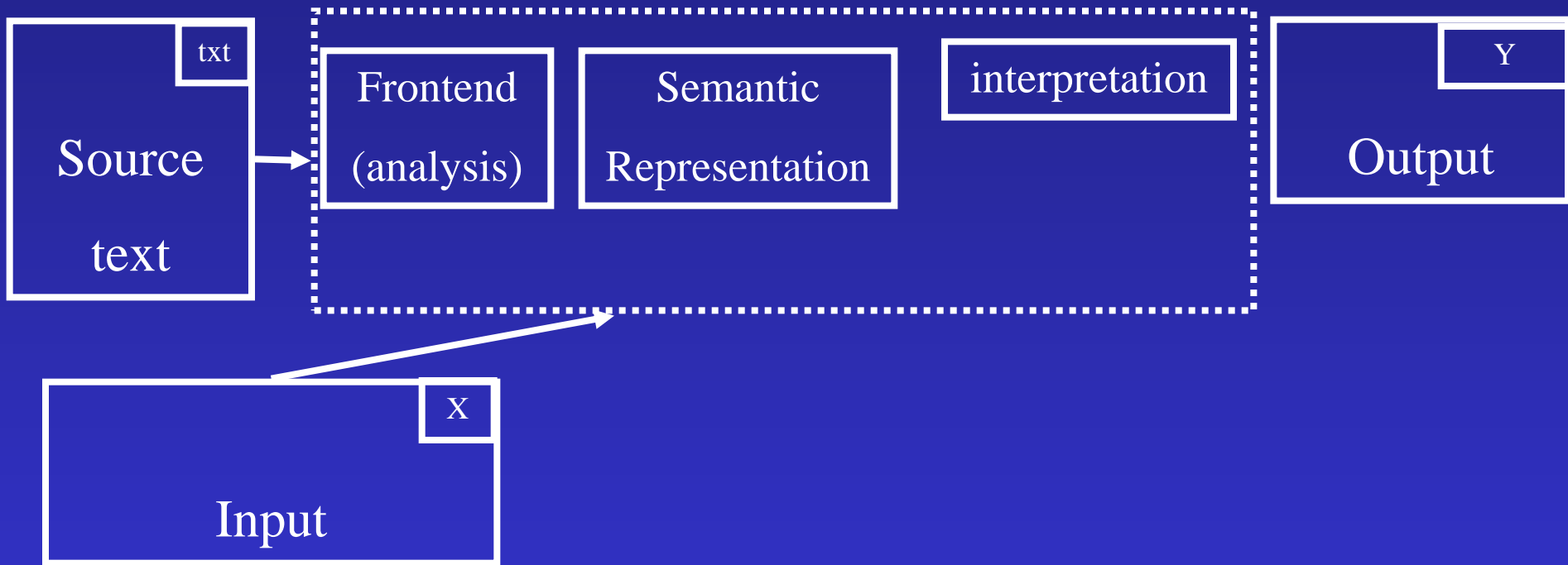


Conceptual structure of a compiler



Compiler

Conceptual structure of an interpreter



Interpreter vs. Compiler

- Conceptually simpler (the definition of the programming language)
- Easier to port
- Can provide more specific error report
- Normally faster
- [More secure]
- Can report errors before input is given
- More efficient
 - Compilation is done once for all the inputs --- many computations can be performed at compile-time
 - Sometimes even $compile-time + execution-time < interpretation-time$

Interpreters provide specific error report

- **Input-program**

```
scanf("%d", &y);
```

```
if (y < 0)
```

```
    x = 5;
```

```
...
```

```
if (y <= 0)
```

```
    z = x + 1;
```

- **Input data** $y=0$

Compilers can provide errors before actual input is given

- **Input-program**

```
scanf("%", &y);
```

```
if (y < 0)
```

```
    x = 5;
```

```
...
```

```
if (y <= 0)
```

```
/* line 88 */    z = x + 1;
```

- **Compiler-Output**

“line 88: x may be used before set”

Compilers can provide errors before actual input is given

- **Input-program**

```
int a[100], x, y ;  
scanf("%d", &y) ;  
if (y < 0)  
/* line 4*/      y = a ;
```

- **Compiler-Output**

“line 4: improper pointer/integer combination: op =”

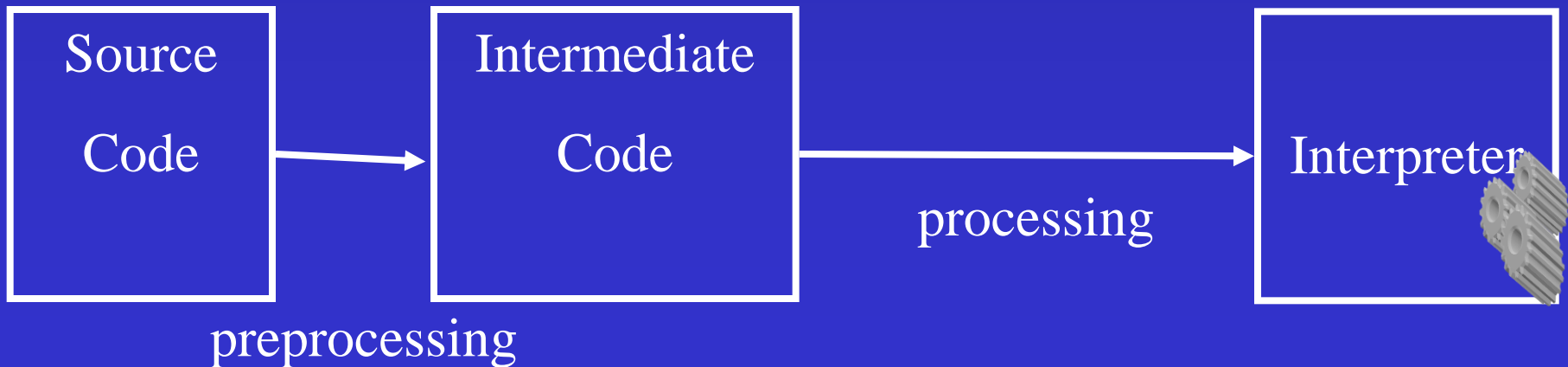
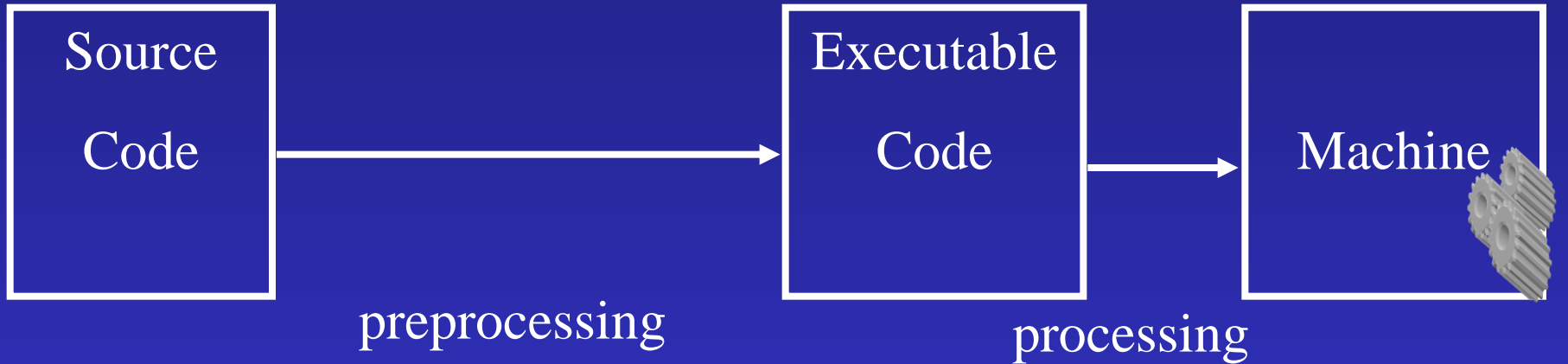
Compilers are usually more efficient

```
scanf("%d", &x);  
y = 5 ;  
z = 7 ;  
x = x +y*z;  
printf("%d", x);
```

Sparc-cc-compiler

```
add  %fp,-8, %11  
mov  %11, %o1  
call scanf  
mov  5, %10  
st  %10,[%fp-12]  
mov  7,%10  
st  %10,[%fp-16]  
ld  [%fp-8], %10  
ld  [%fp-8],%10  
add  %10, 35, %10  
st  %10,[%fp-8]  
ld  [%fp-8], %11  
mov  %11, %o1  
call printf
```

Compiler vs. Interpreter



Why Study Compilers?

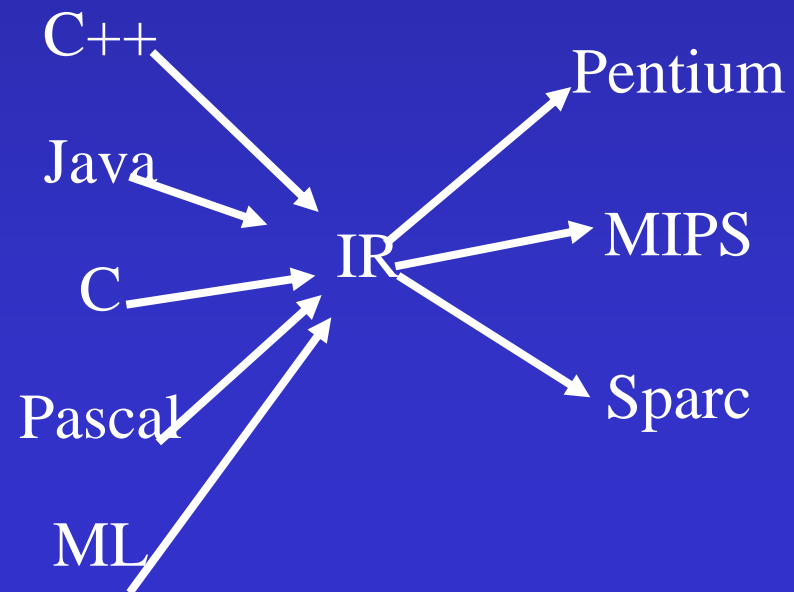
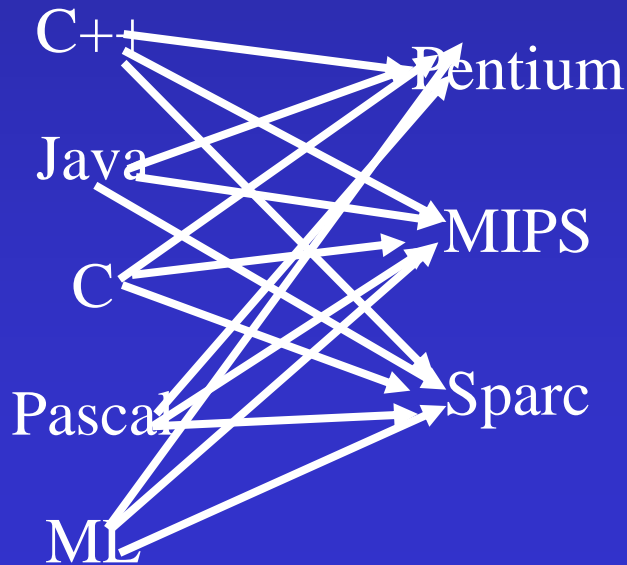
- Become a compiler writer
 - New programming languages
 - New machines
 - New compilation modes: “just-in-time”
- Using some of the techniques in other contexts
- Design a very big software program using a reasonable effort
- Learn applications of many CS results (formal languages, decidability, graph algorithms, dynamic programming, ...)
- Better understating of programming languages and machine architectures
- Become a better programmer

Why study compilers?

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

Proper Problem Structure

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

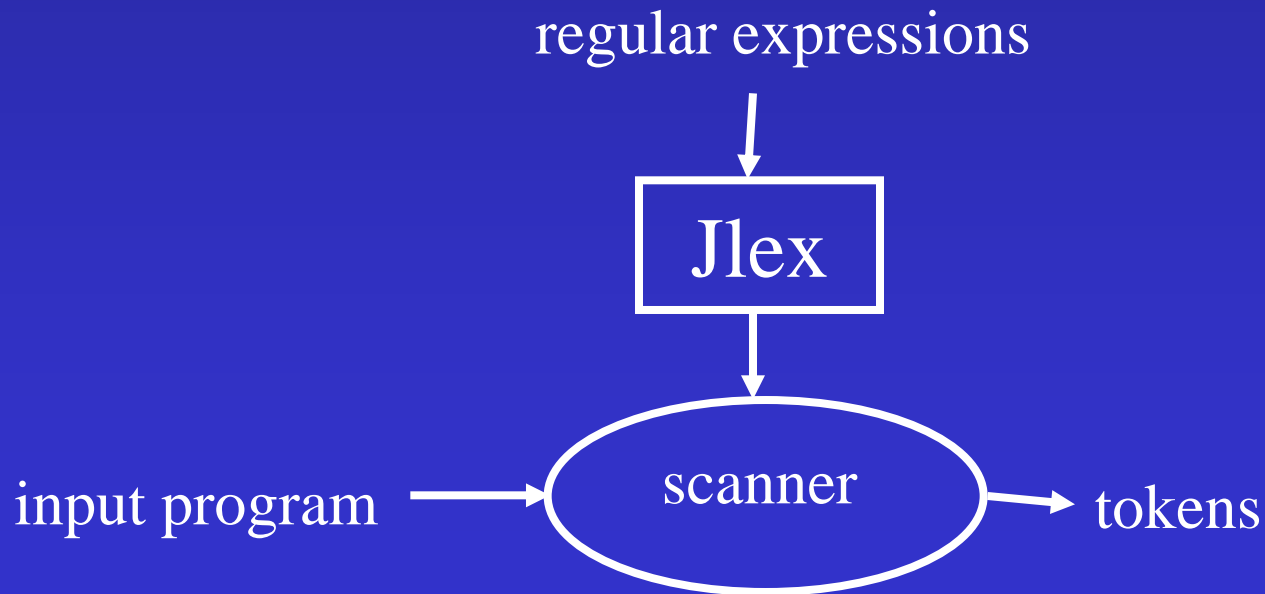


Judicious use of formalisms

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

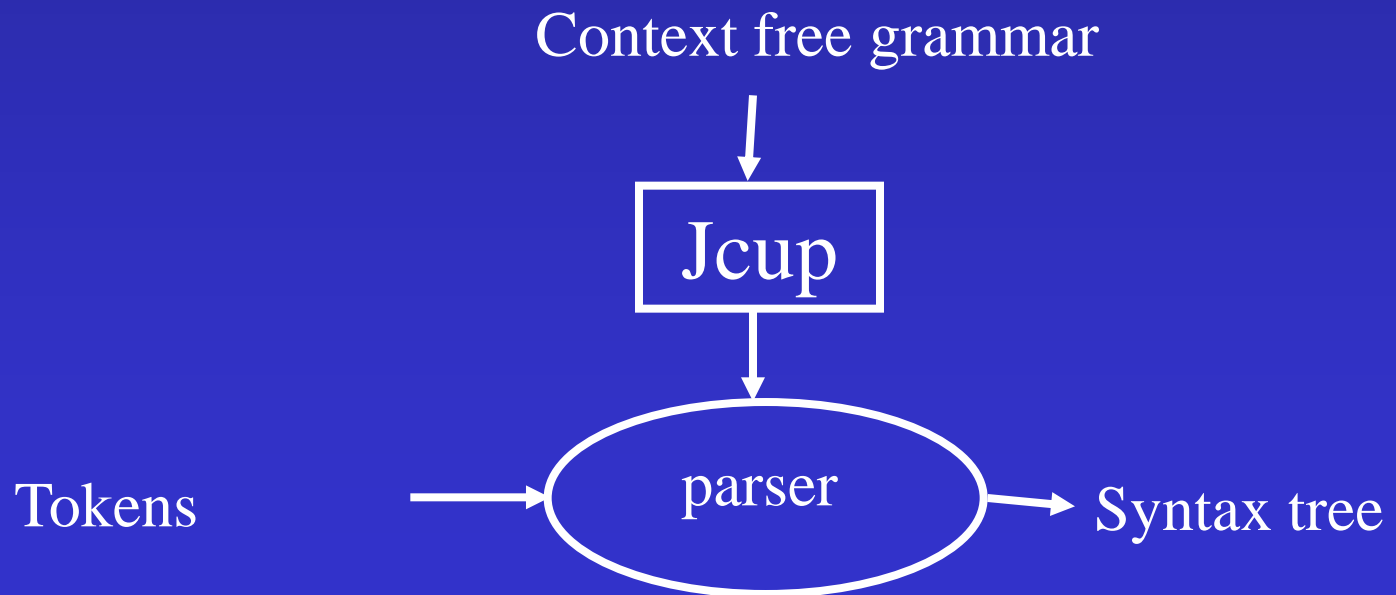
Use of program-generating tools

- Parts of the compiler are automatically generated from specification

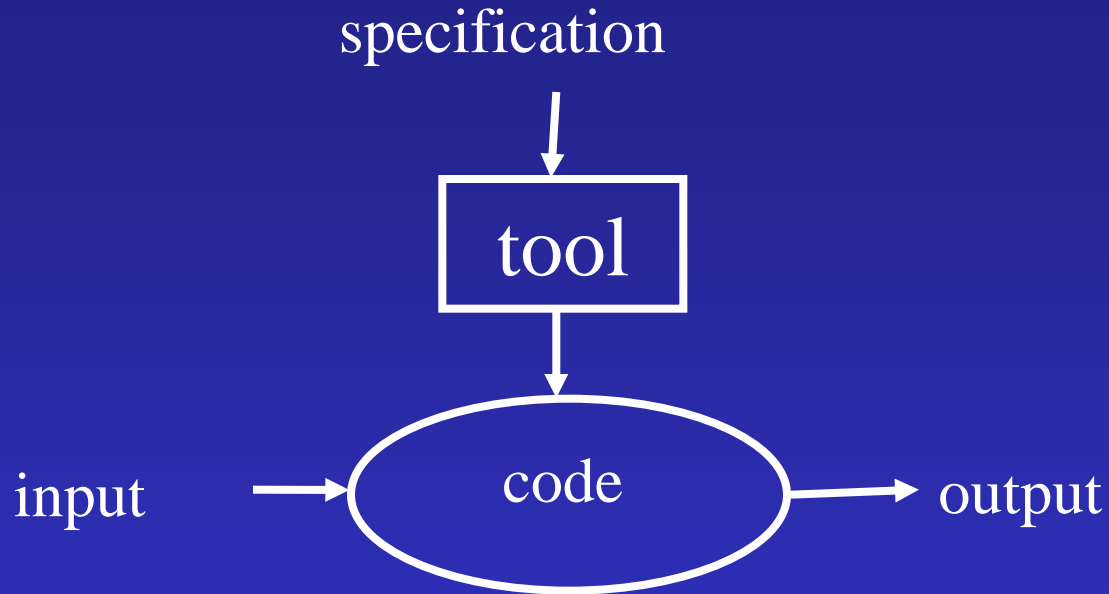


Use of program-generating tools

- 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

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

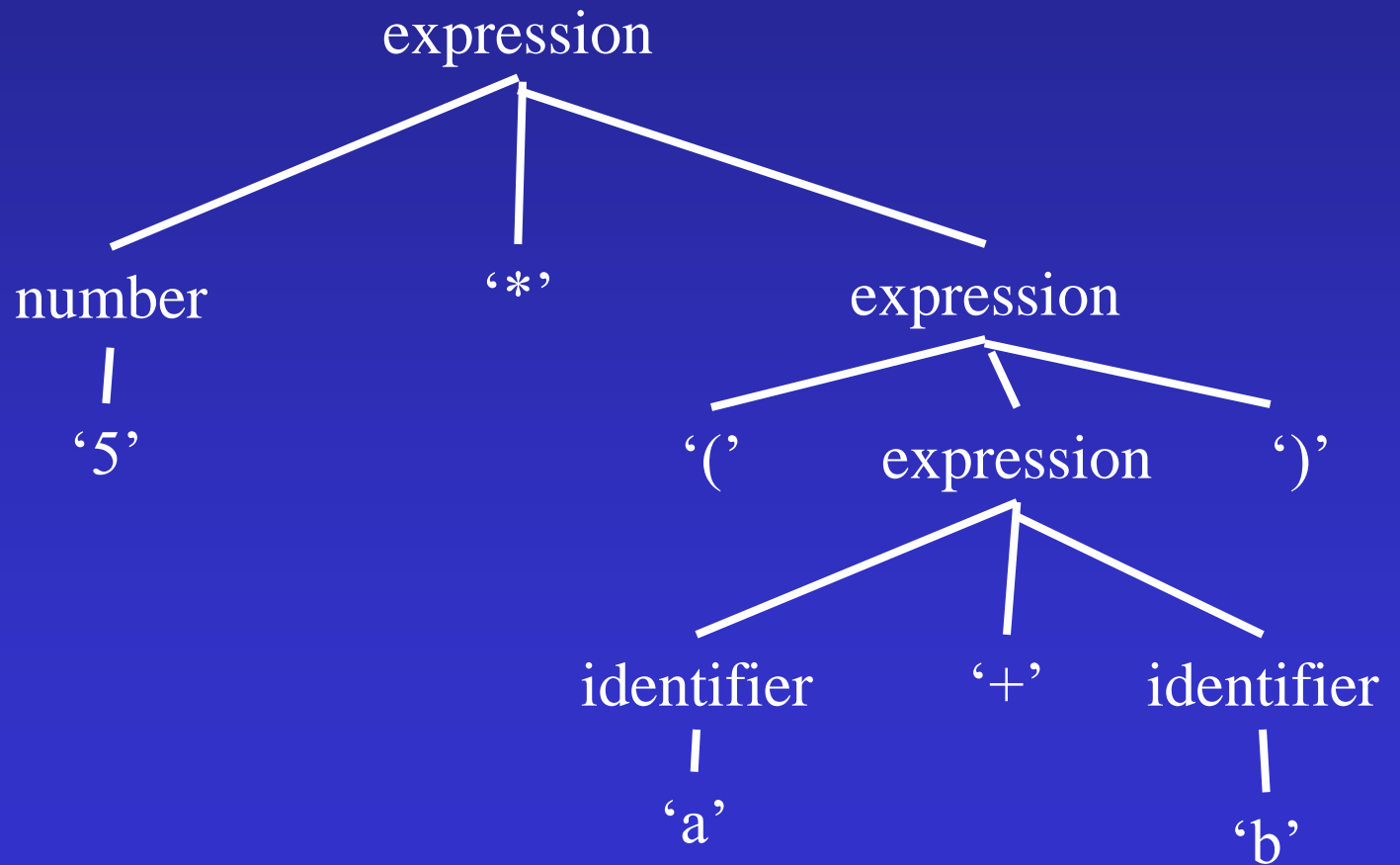
A simple traditional modular compiler/interpreter (1.2)

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

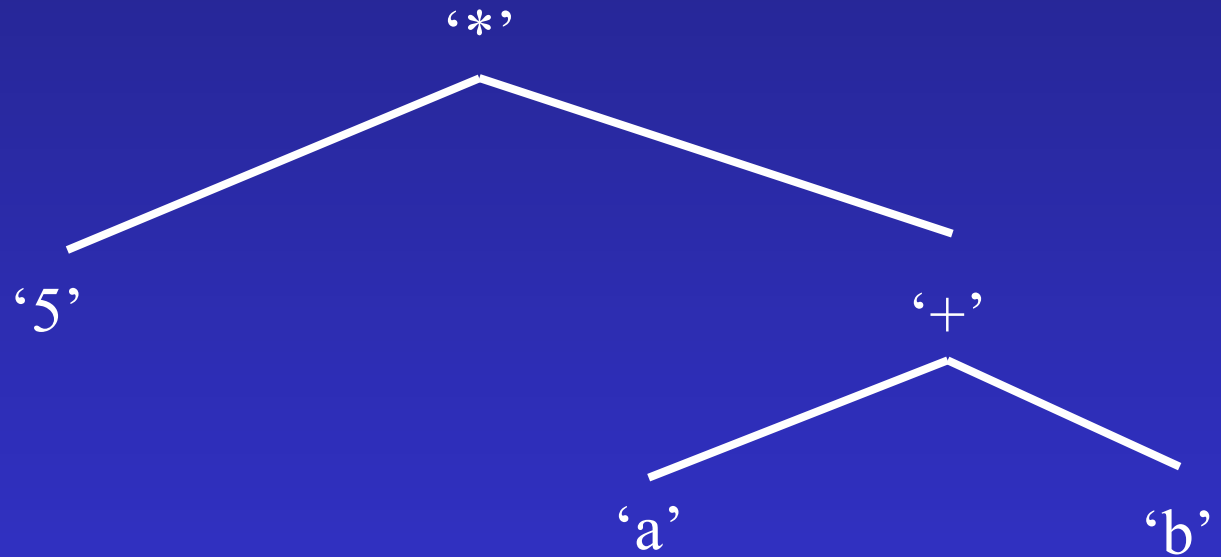
The abstract syntax tree (AST)

- 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)

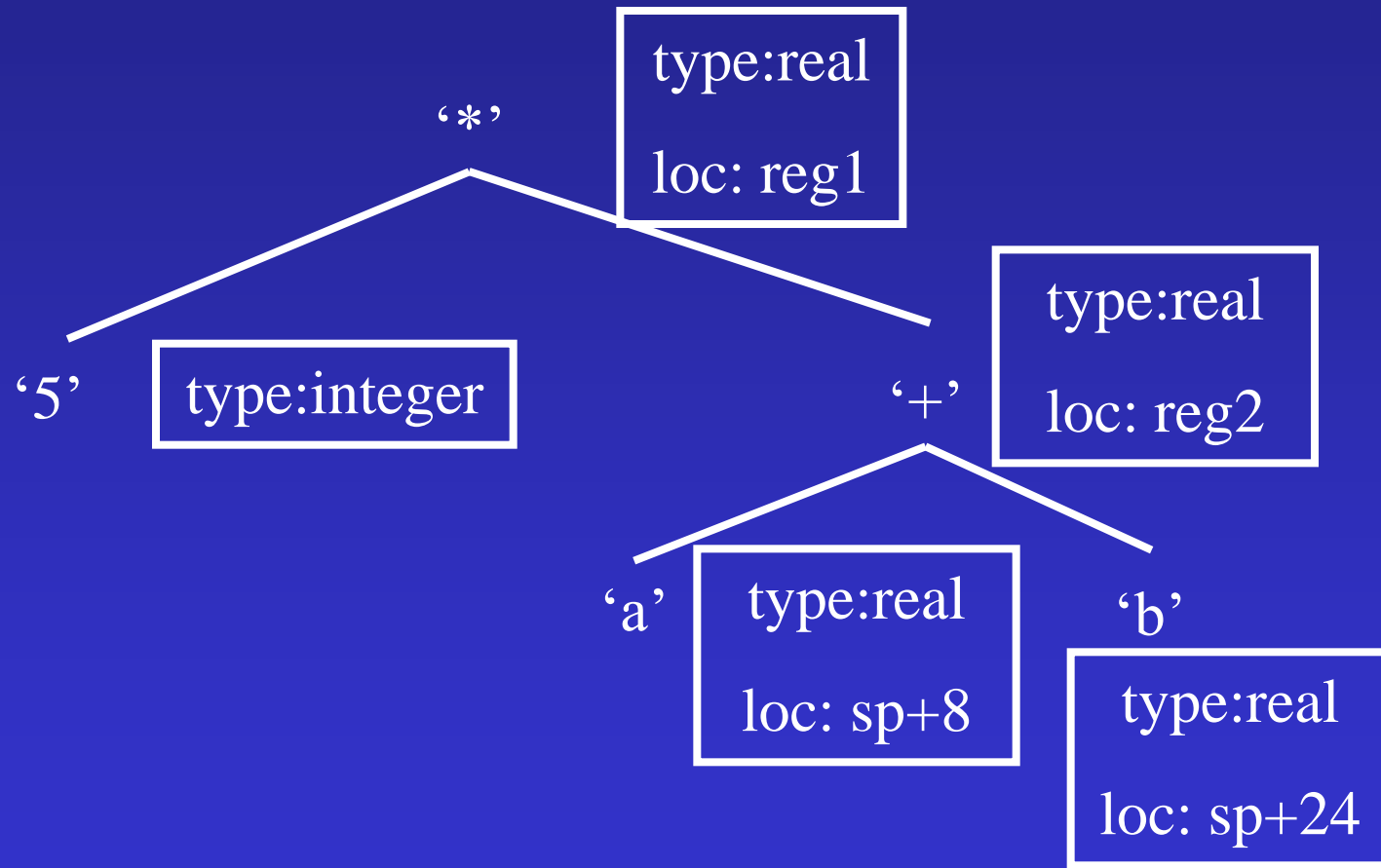
Syntax tree



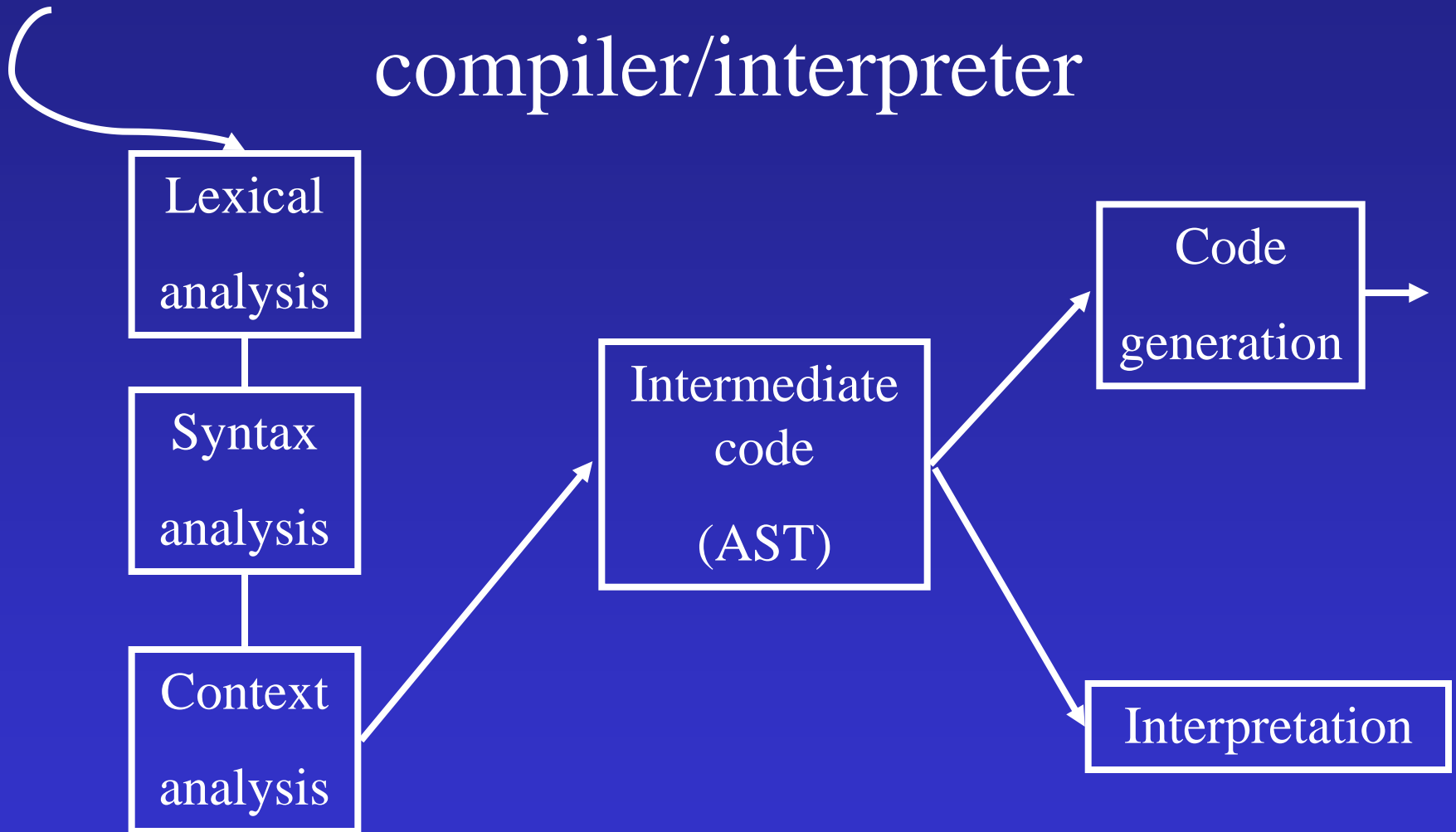
Abstract Syntax tree



Annotated Abstract Syntax tree



Structure of a demo compiler/interpreter



Input language

- Fully parameterized expressions
- Arguments can be a single digit

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

operator \rightarrow ‘+’ | ‘*’

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

Driver for the demo 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;
}
```

Lexical Analysis

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

Header file lex.h 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);
```

```
#include "lex.h"
static int Layout_char(int ch) {
    switch (ch) {
        case ' ': case '\t': case '\n': return 1;
        default: return 0;
    }
}
token_type Token;
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;
}
```


Parser

- Invokes lexical analyzer
- Reports syntax errors
- Constructs AST

Parser Environment

```
#include "lex.h"
#include "error.h"
#include "parser.h"
static Expression *new_expression(void) {
    return (Expression *)malloc(sizeof (Expression));
}
static void free_expression(Expression *expr) {free((void *)expr);}
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;
}
```

Parser Header File

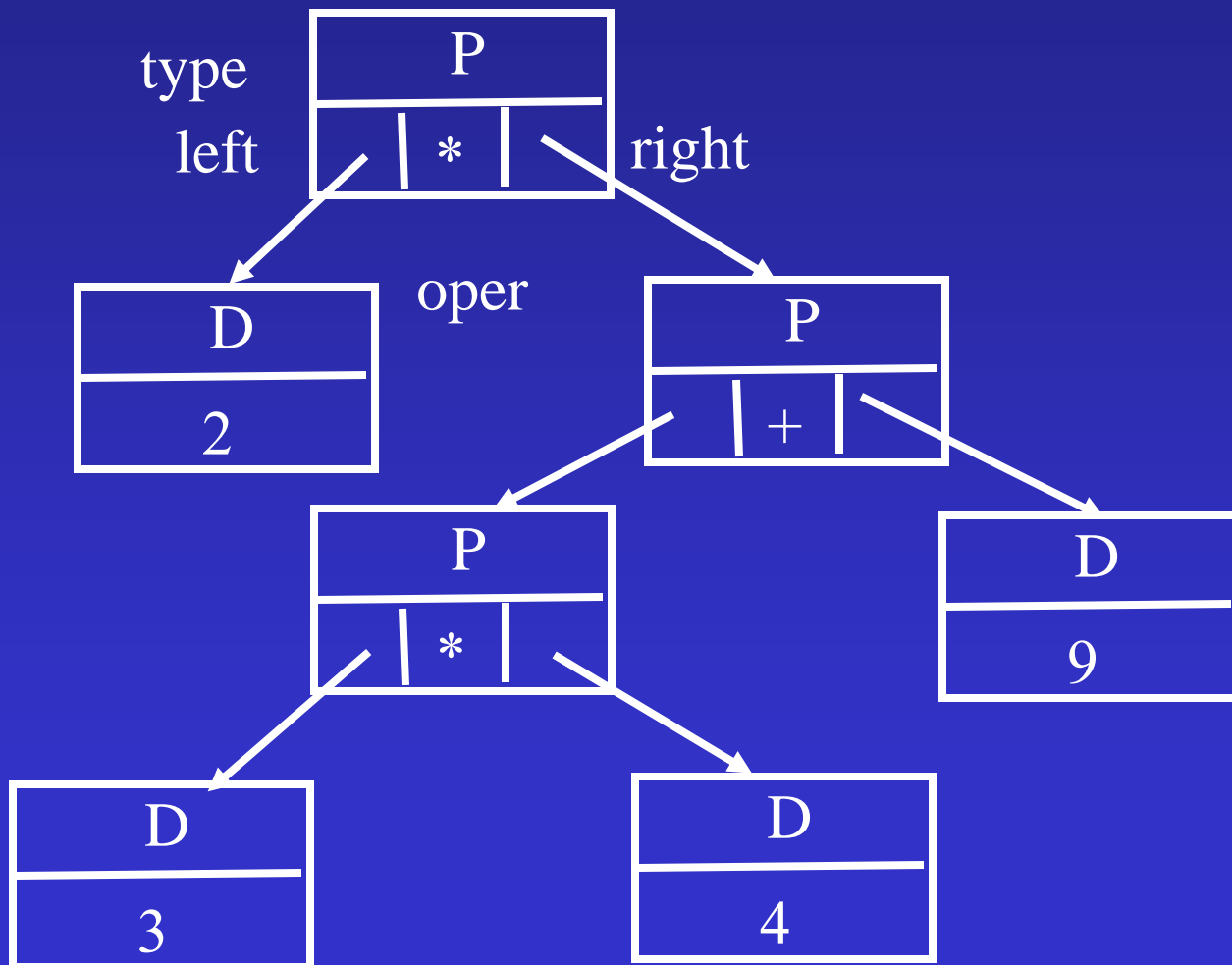
```
typedef int Operator;

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

typedef Expression AST_node; /* the top node is an Expression */

extern int Parse_program(AST_node **);
```

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



Top-Down Parsing

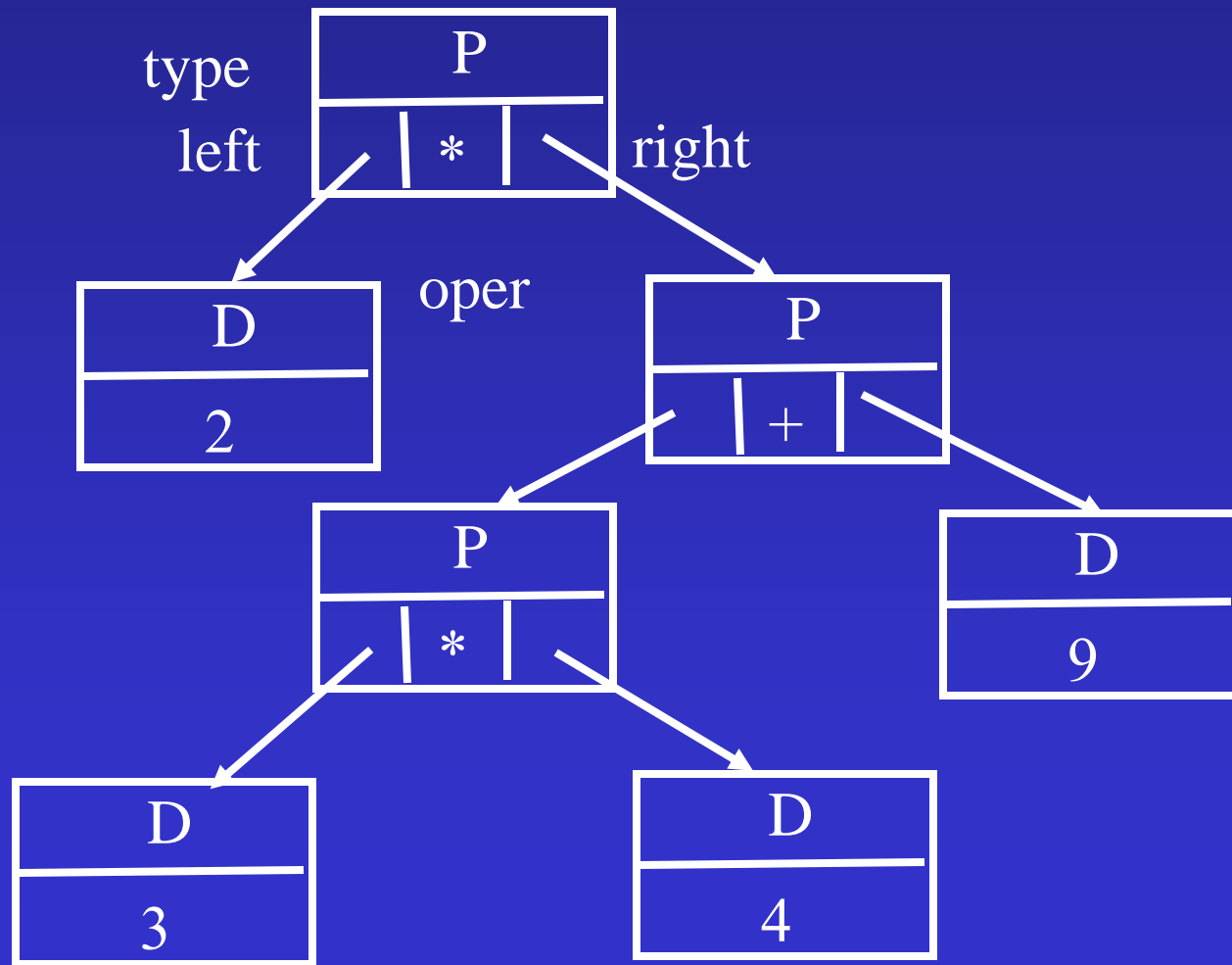
- Optimistically build the tree from the root to leaves
- Try every alternative production
 - For $P \rightarrow A_1 A_2 \dots A_n \mid B_1 B_2 \dots B_m$
 - If A_1 succeeds
 - If A_2 succeeds
 - if A_3 succeeds
 - » ...
 - **Otherwise fail**
 - **Otherwise fail**
 - If B_1 succeeds
 - If B_2 succeeds
 - ...
 - **No backtracking**
- Recursive descent parsing
- Can be applied for certain grammars

Parse_Operator

```
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;  
}
```

```
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;
}
```

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



Context handling

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

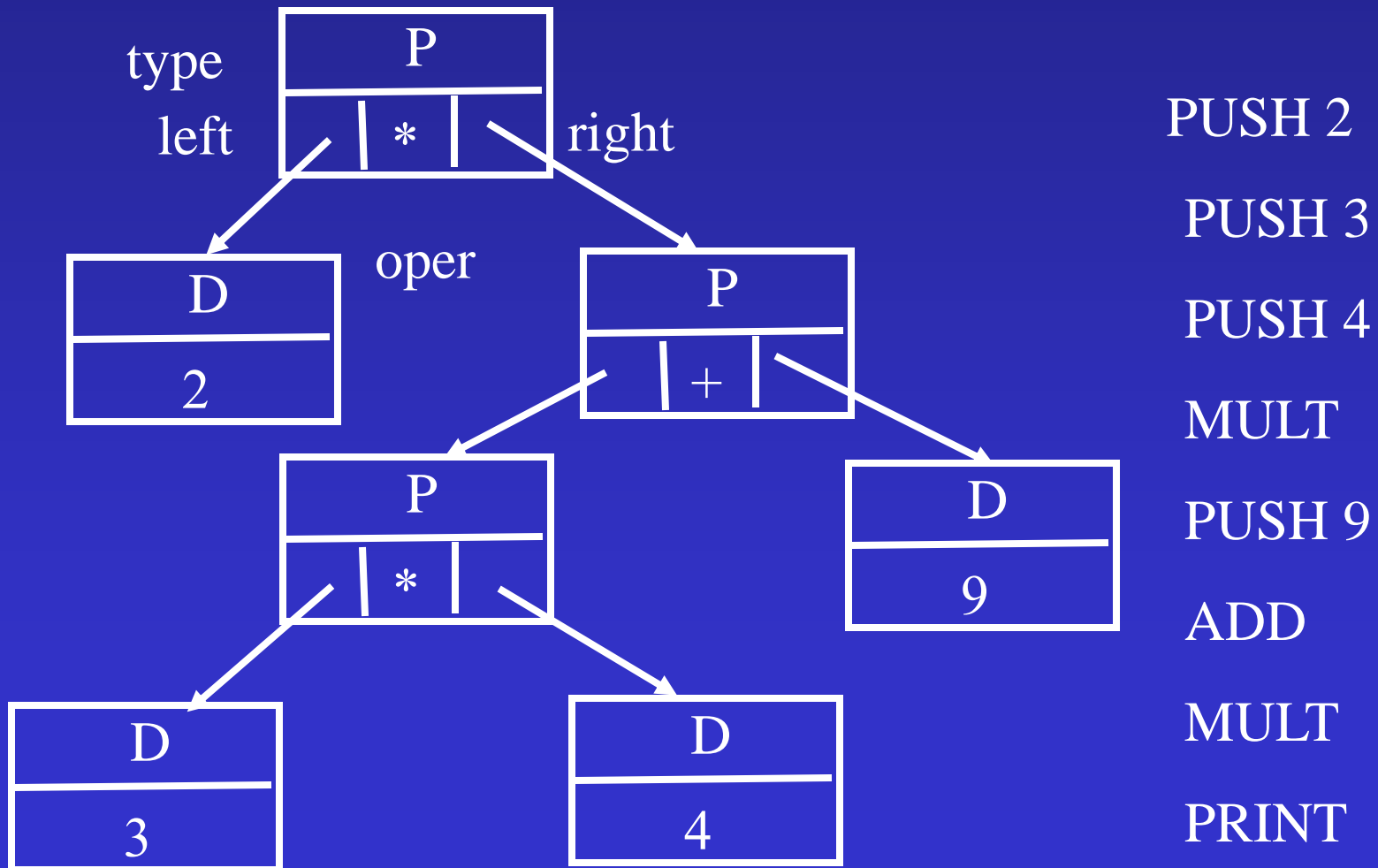
Code generation

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

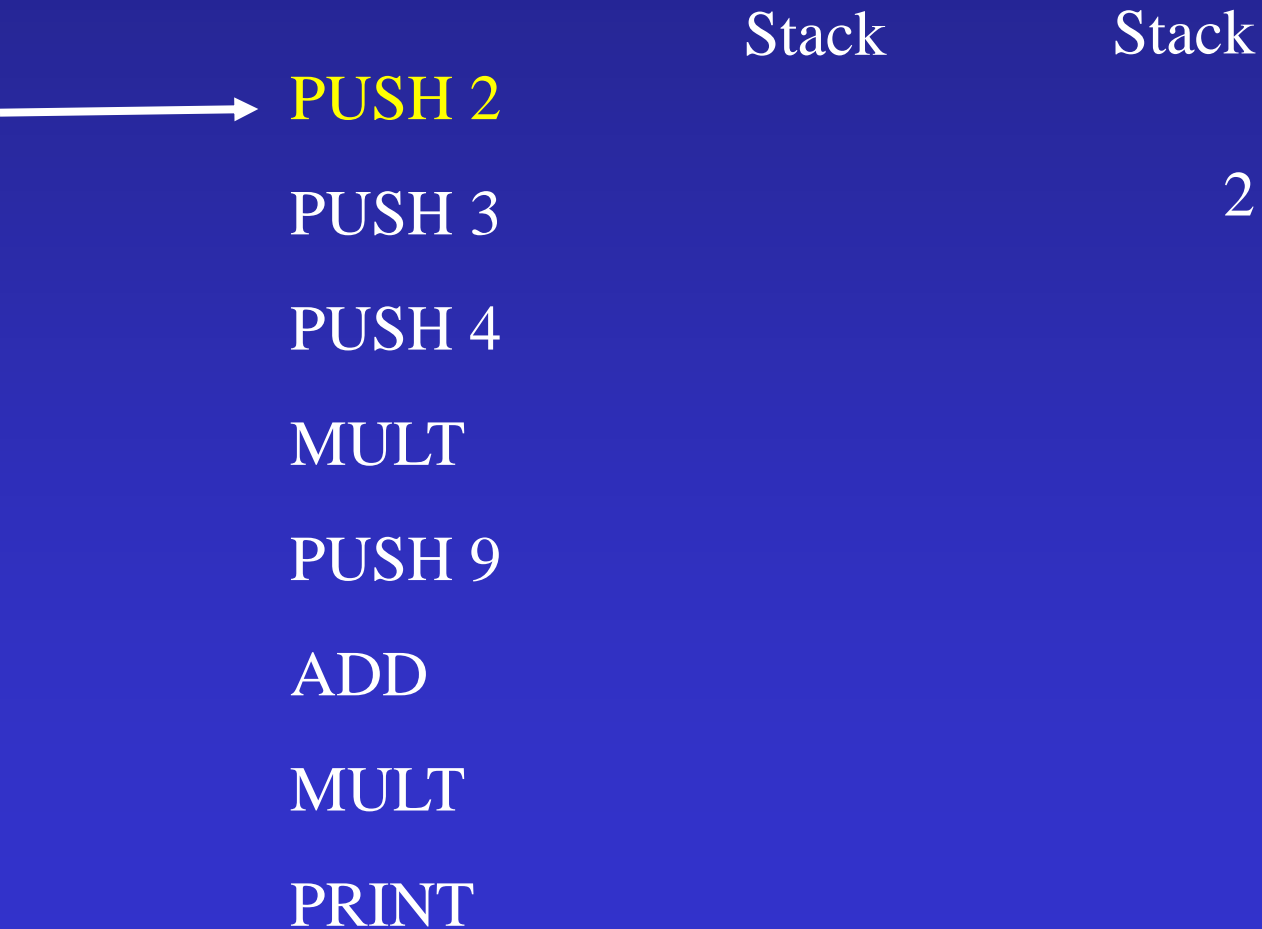
Code generation

```
#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 $(2 * ((3 * 4) + 9))$




Generated Code Execution




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

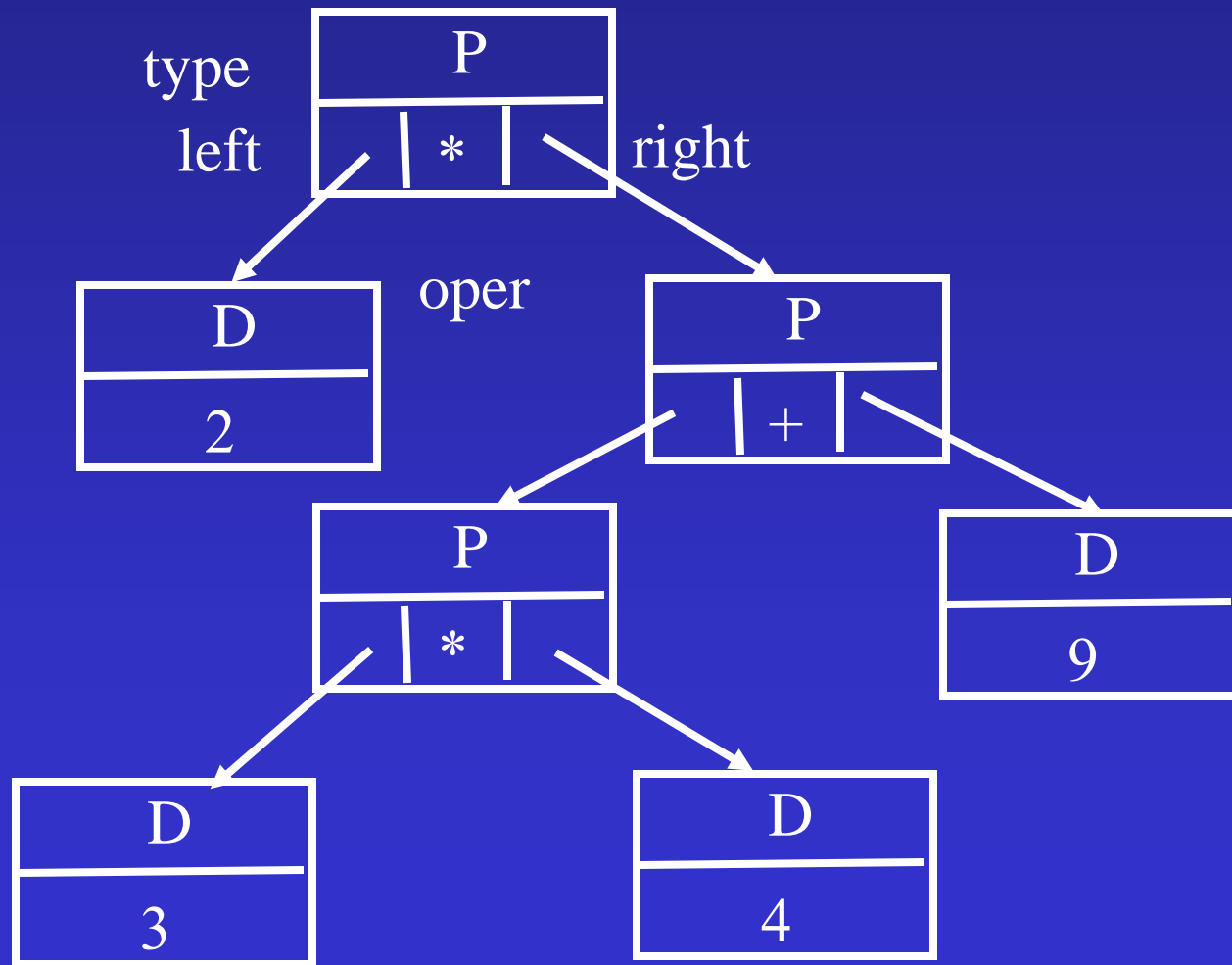
	Stack	Stack
PUSH 2		
PUSH 3	42	
PUSH 4		
MULT		
PUSH 9		
ADD		
MULT		
 PRINT		

Interpretation

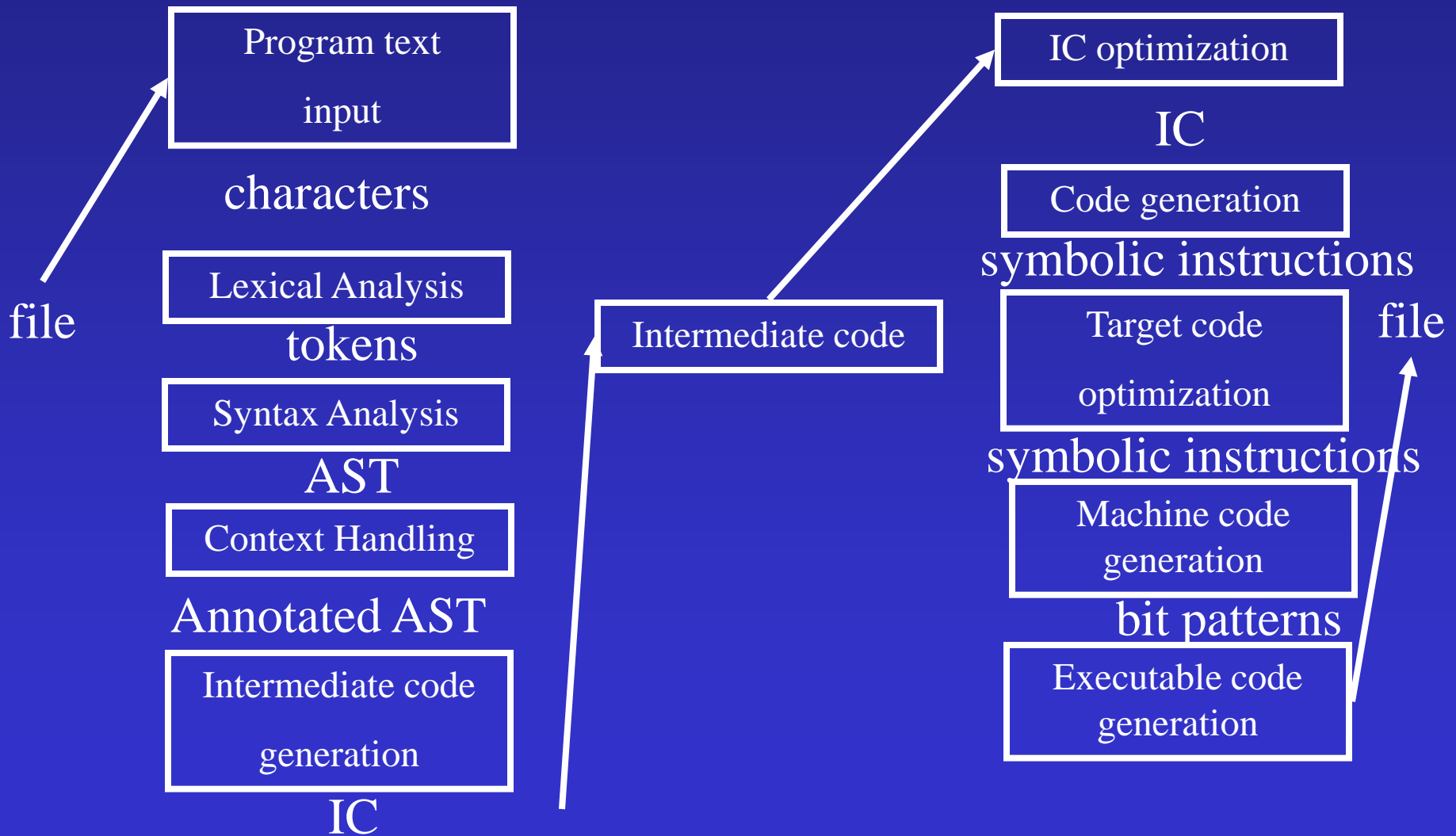
- Bottom-up evaluation of expressions
- The same interface of the compiler

```
#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 $(2 * ((3 * 4) + 9))$



A More Realistic Compiler



Runtime systems

- Responsible for language dependent dynamic resource allocation
- Memory allocation
 - Stack frames
 - Heap
- Garbage collection
- I/O
- Interacts with operating system/architecture
- Important part of the compiler

Shortcuts

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

Tentative Syllabus

- Overview (1)
- Lexical Analysis (1)
 - Regular expressions to Finite State Automaton
- Parsing (2 lectures)
 - Grammars, Ambiguity, Early Parsers, Efficient Parsers
- Semantic analysis (1)
 - Type checking
- Code generation (4)
- Assembler/Linker Loader (1)
- Object Oriented (1)
- Garbage Collection (1)

Summary

- Phases drastically simplifies the problem of writing a good compiler
- The frontend is shared between compiler/interpreter