

# Course Contents

- ▶ Lexical analysis (Scanning)
- ▶ Syntax Analysis (Parsing)
- ▶ Syntax Directed Translation
- ▶ Intermediate Code Generation
- ▶ Run-time environments
- ▶ Code Generation
- ▶ Machine Independent Optimization

# Compiler learning

- ▶ Isn't it an old discipline?
  - ▶ Yes, it is a well-established discipline
  - ▶ Algorithms, methods and techniques are researched and developed in early stages of computer science growth
  - ▶ There are many compilers around and many tools to generate them automatically
- ▶ So, why we need to learn it?
  - ▶ Although you may never write a full compiler
  - ▶ But the techniques we learn is useful in many tasks like writing an interpreter for a scripting language, validation checking for forms and so on

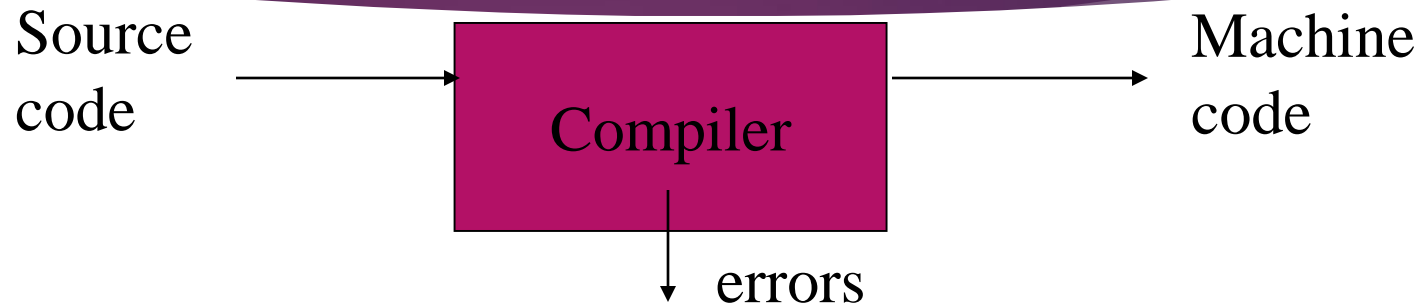
# Terminology

- ▶ **Compiler:**
  - ▶ a program that translates an *executable* program in one language into an *executable* program in another language
  - ▶ we expect the program produced by the compiler to be better, in some way, than the original
- ▶ **Interpreter:**
  - ▶ a program that reads an *executable* program and produces the results of running that program
  - ▶ usually, this involves executing the source program in some fashion
- ▶ Our course is mainly about compilers but many of the same issues arise in interpreters

# Disciplines involved

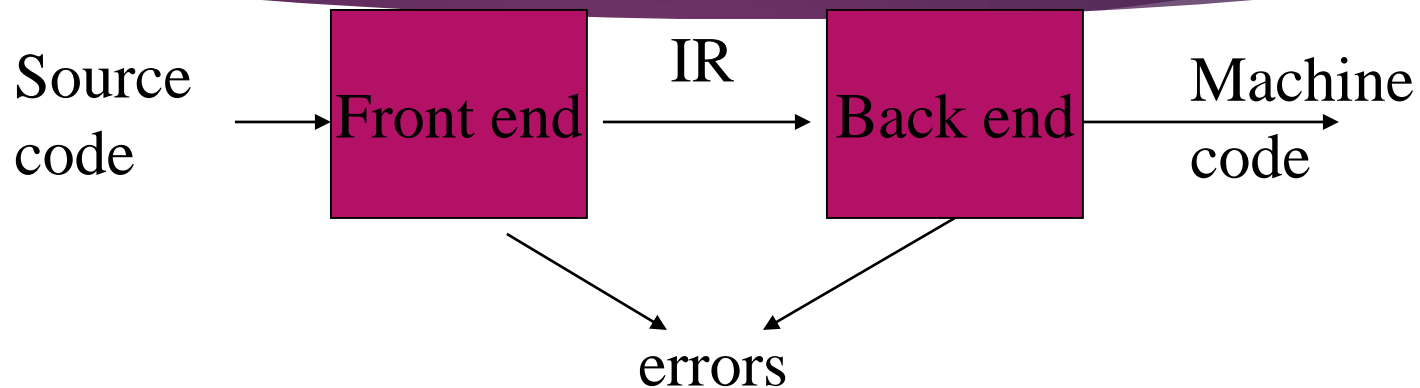
- ▶ Algorithms
- ▶ Languages and machines
- ▶ Operating systems
- ▶ Computer architectures

# Abstract view



- ▶ Recognizes legal (and illegal) programs
- ▶ Generate correct code
- ▶ Manage storage of all variables and code
- ▶ Agreement on format for object (or assembly) code

# Front-end, Back-end division



- ▶ Front end maps legal code into IR
- ▶ Back end maps IR onto target machine
- ▶ Simplify retargeting
- ▶ Allows multiple front ends
- ▶ Multiple passes -> better code

# Front end

Source  
code



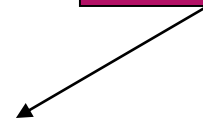
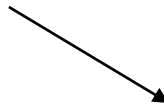
Scanner

tokens



Parser

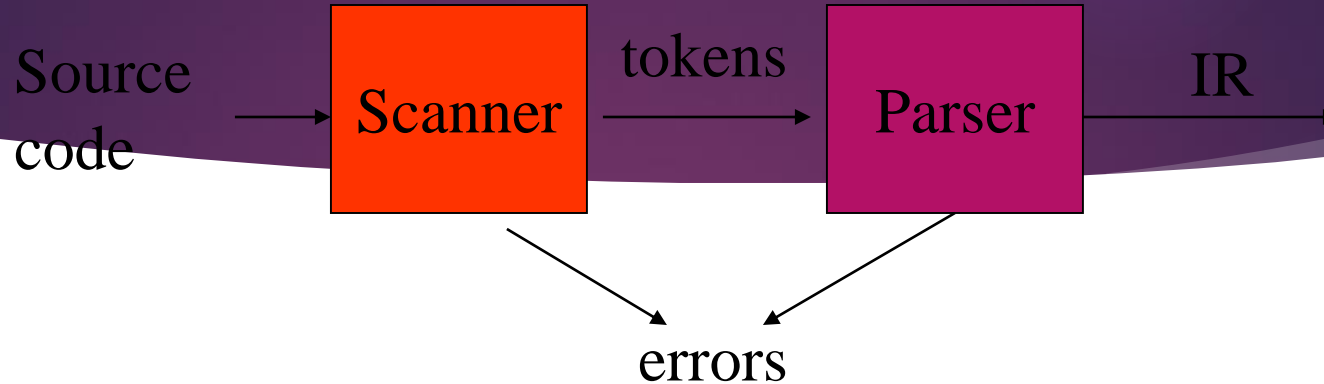
IR



errors

- ▶ Recognize legal code
- ▶ Report errors
- ▶ Produce IR
- ▶ Preliminary storage maps

# Front end



## ► Scanner:

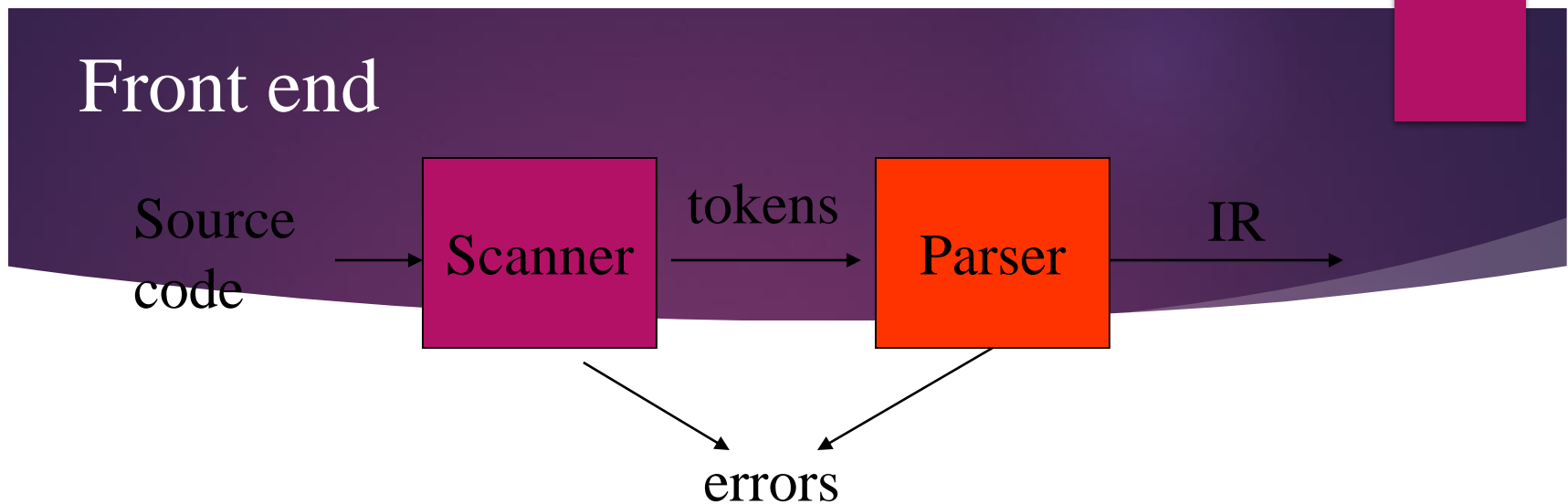
- Maps characters into tokens – the basic unit of syntax

  - $x = x + y$  becomes  $\langle \text{id}, x \rangle = \langle \text{id}, x \rangle + \langle \text{id}, y \rangle$

- Typical tokens: number, id, +, -, \*, /, do, end

- Eliminate white space (tabs, blanks, comments)

- A key issue is speed so instead of using a tool like LEX it sometimes needed to write your own scanner



- ▶ **Parser:**
  - ▶ Recognize context-free syntax
  - ▶ Guide context-sensitive analysis
  - ▶ Construct IR
  - ▶ Produce meaningful error messages
  - ▶ Attempt error correction
- ▶ There are parser generators like YACC which automates much of the work

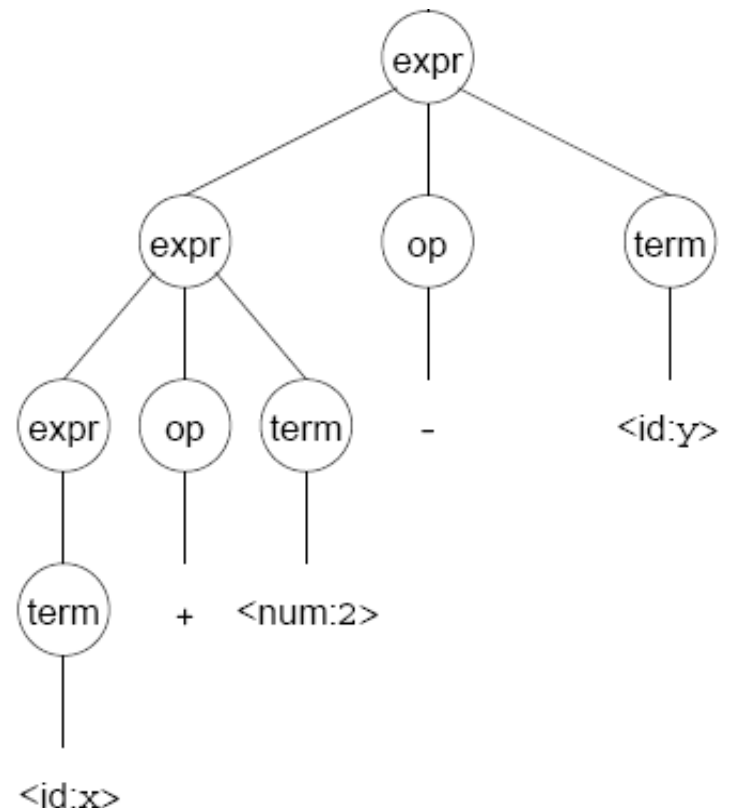
# Front end

- ▶ Context free grammars are used to represent programming language syntaxes:

$$\langle \text{expr} \rangle ::= \langle \text{expr} \rangle \langle \text{op} \rangle \langle \text{term} \rangle \mid \langle \text{term} \rangle$$
$$\langle \text{term} \rangle ::= \langle \text{number} \rangle \mid \langle \text{id} \rangle$$
$$\langle \text{op} \rangle ::= + \mid -$$

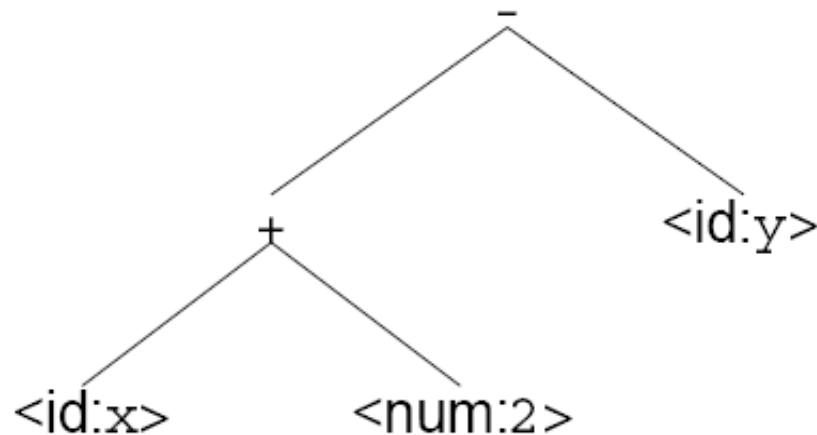
# Front end

- ▶ A parser tries to map a program to the syntactic elements defined in the grammar
- ▶ A parse can be represented by a tree called a parse or syntax tree

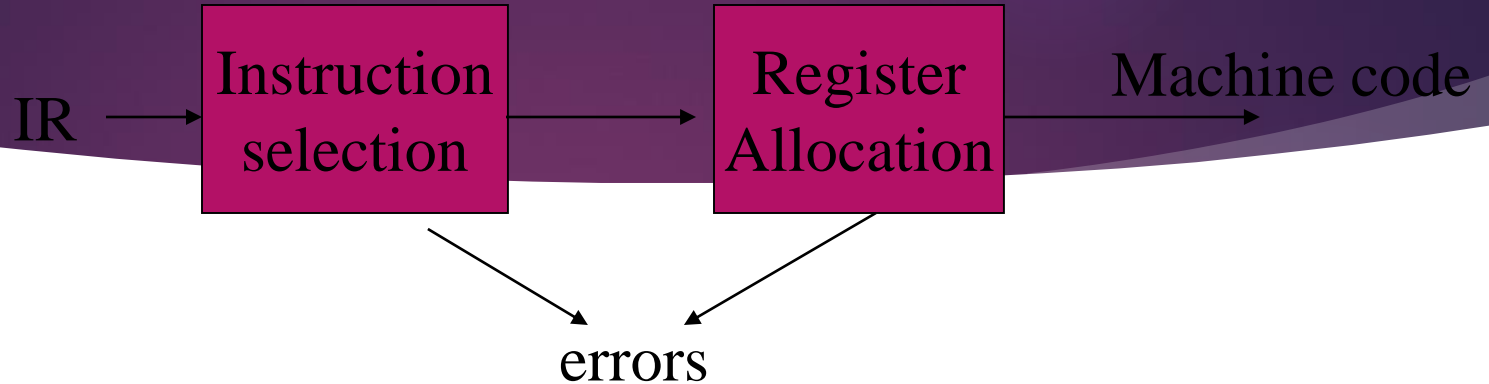


# Front end

- ▶ A parse tree can be represented more compactly referred to as Abstract Syntax Tree (AST)
- ▶ AST is often used as IR between front end and back end

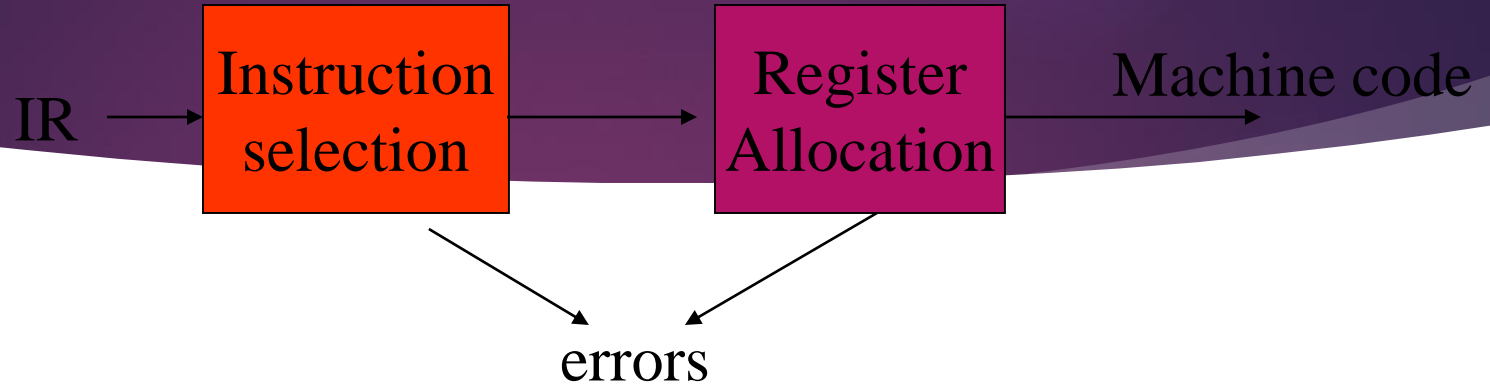


# Back end



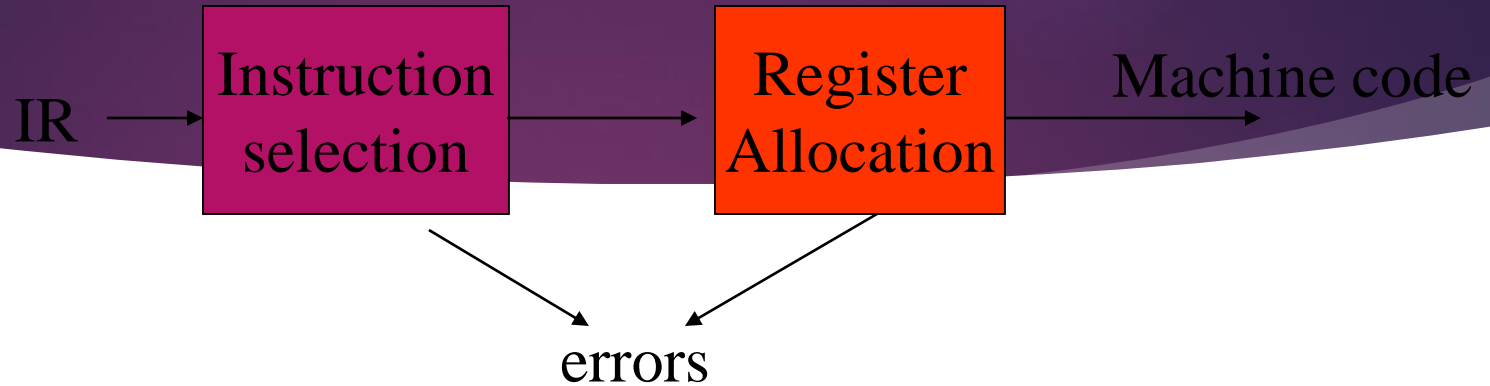
- ▶ Translate IR into target machine code
- ▶ Choose instructions for each IR operation
- ▶ Decide what to keep in registers at each point
- ▶ Ensure conformance with system interfaces

# Back end



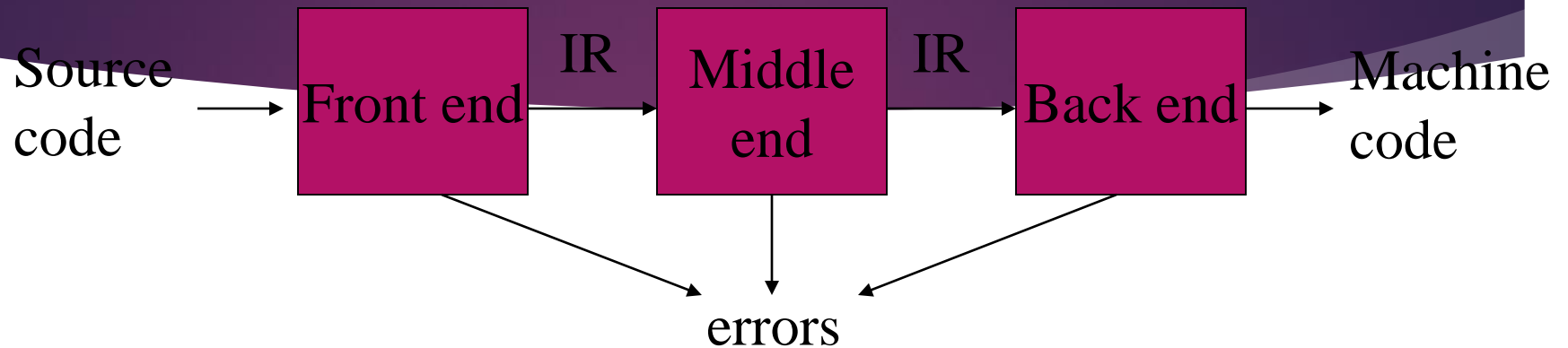
- ▶ Produce compact fast code
- ▶ Use available addressing modes

# Back end



- ▶ Have a value in a register when used
- ▶ Limited resources
- ▶ Optimal allocation is difficult

# Traditional three pass compiler



- ▶ Code improvement analyzes and change IR
- ▶ Goal is to reduce runtime

# Middle end (optimizer)

- ▶ Modern optimizers are usually built as a set of passes
- ▶ Typical passes
  - ▶ Constant propagation
  - ▶ Common sub-expression elimination
  - ▶ Redundant store elimination
  - ▶ Dead code elimination

# What Do Compilers Do

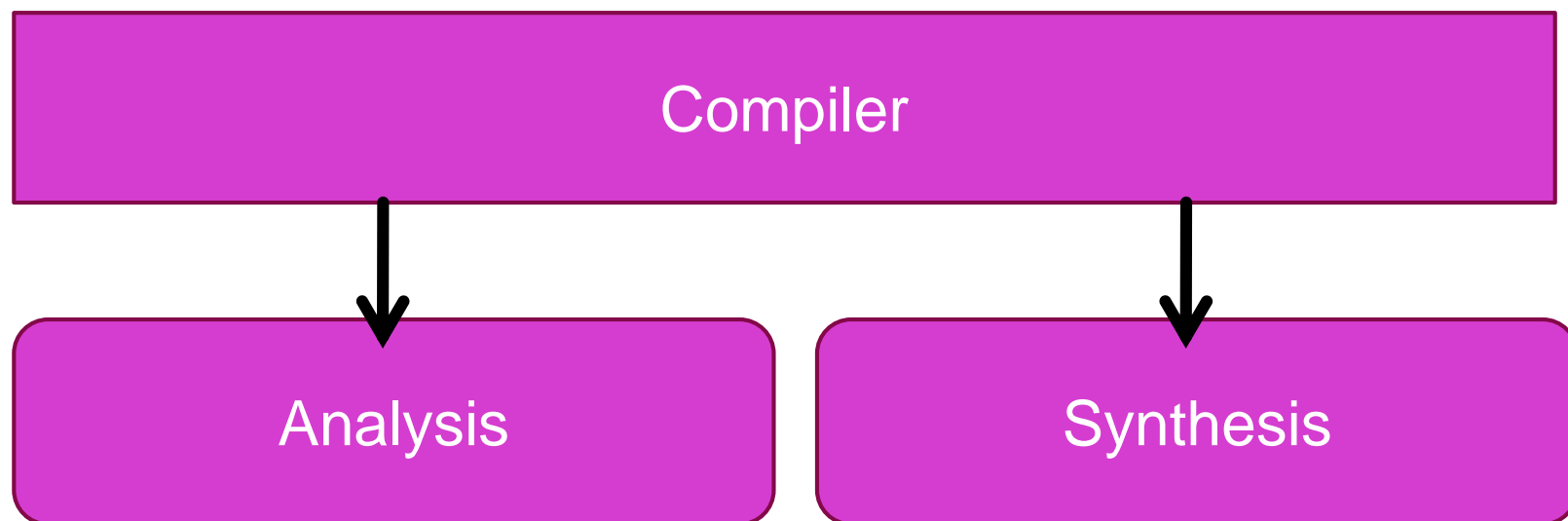
- ▶ A compiler acts as a translator, transforming human-oriented programming languages into computer-oriented machine languages.
- ▶ Ignore machine-dependent details for programmer



# What Do Compilers Do

- ▶ Another way that compilers differ from one another is in the format of the target machine code they generate:
  - ▶ Assembly or other source format
  - ▶ Relocatable binary
    - ▶ Relative address
    - ▶ A linkage step is required
  - ▶ Absolute binary
    - ▶ Absolute address
    - ▶ Can be executed directly

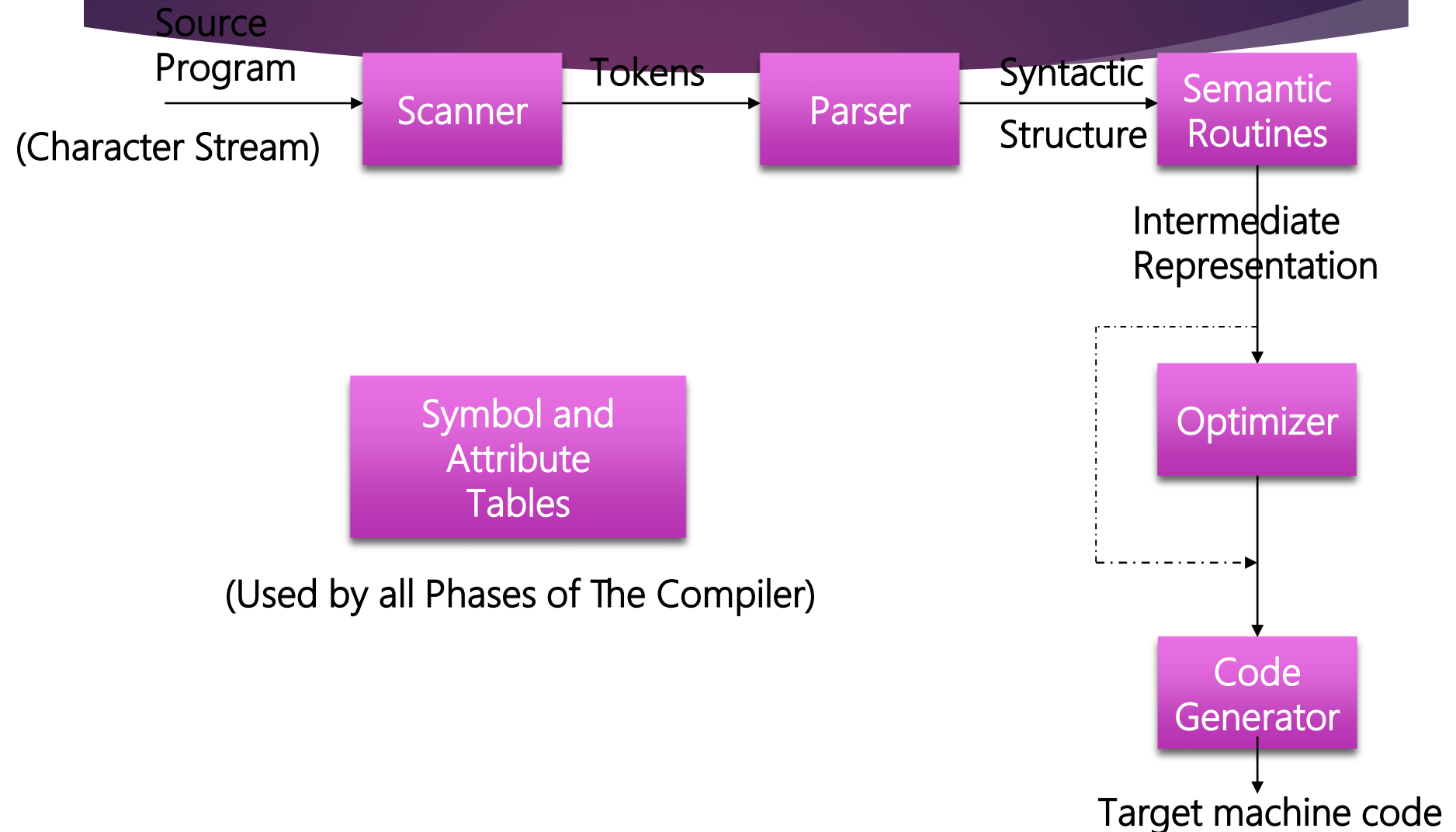
# The Structure of a Compiler



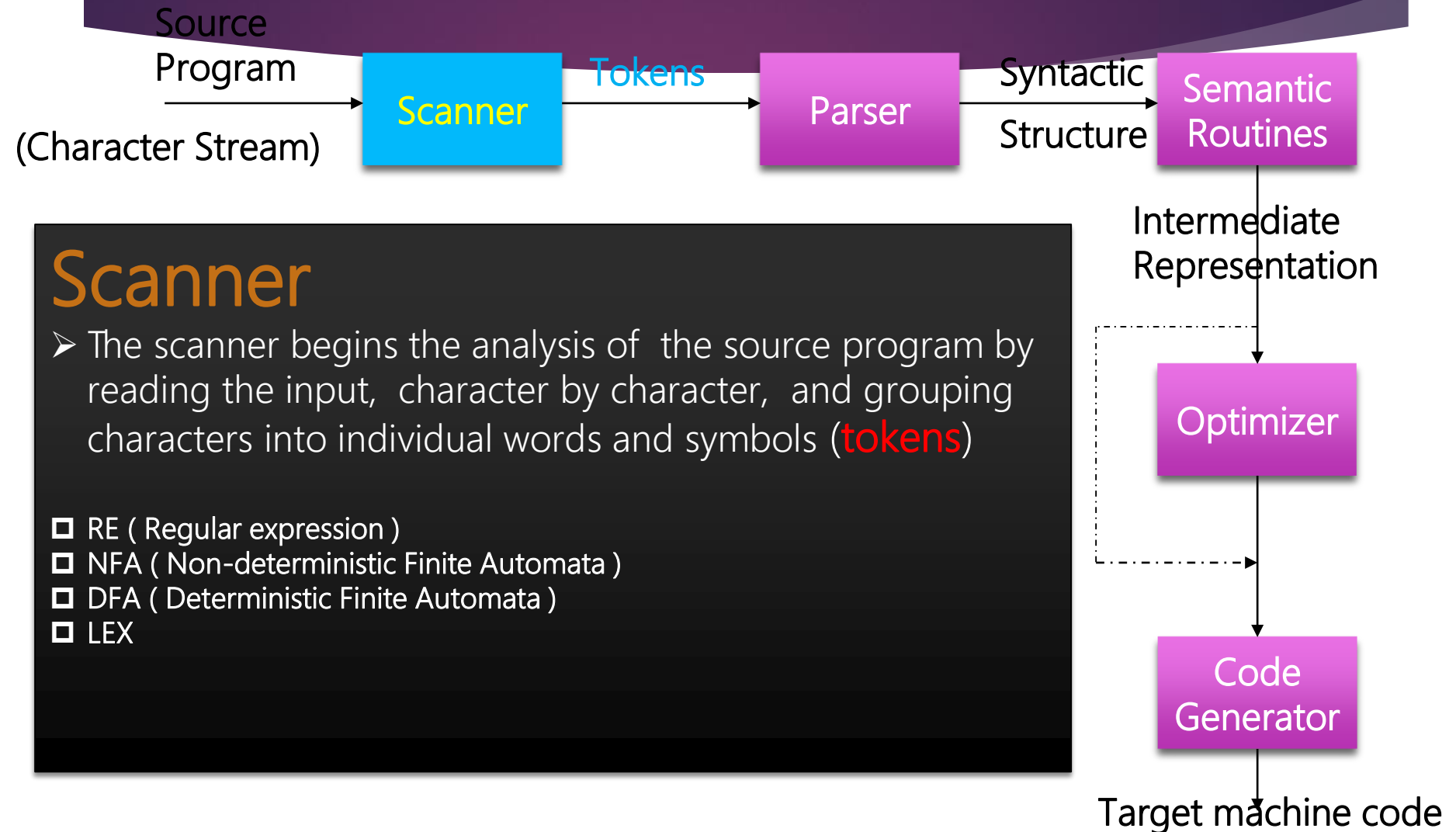
- ▶ Analysis of the source program
- ▶ Synthesis of a machine-language program

# The Structure of a Compiler

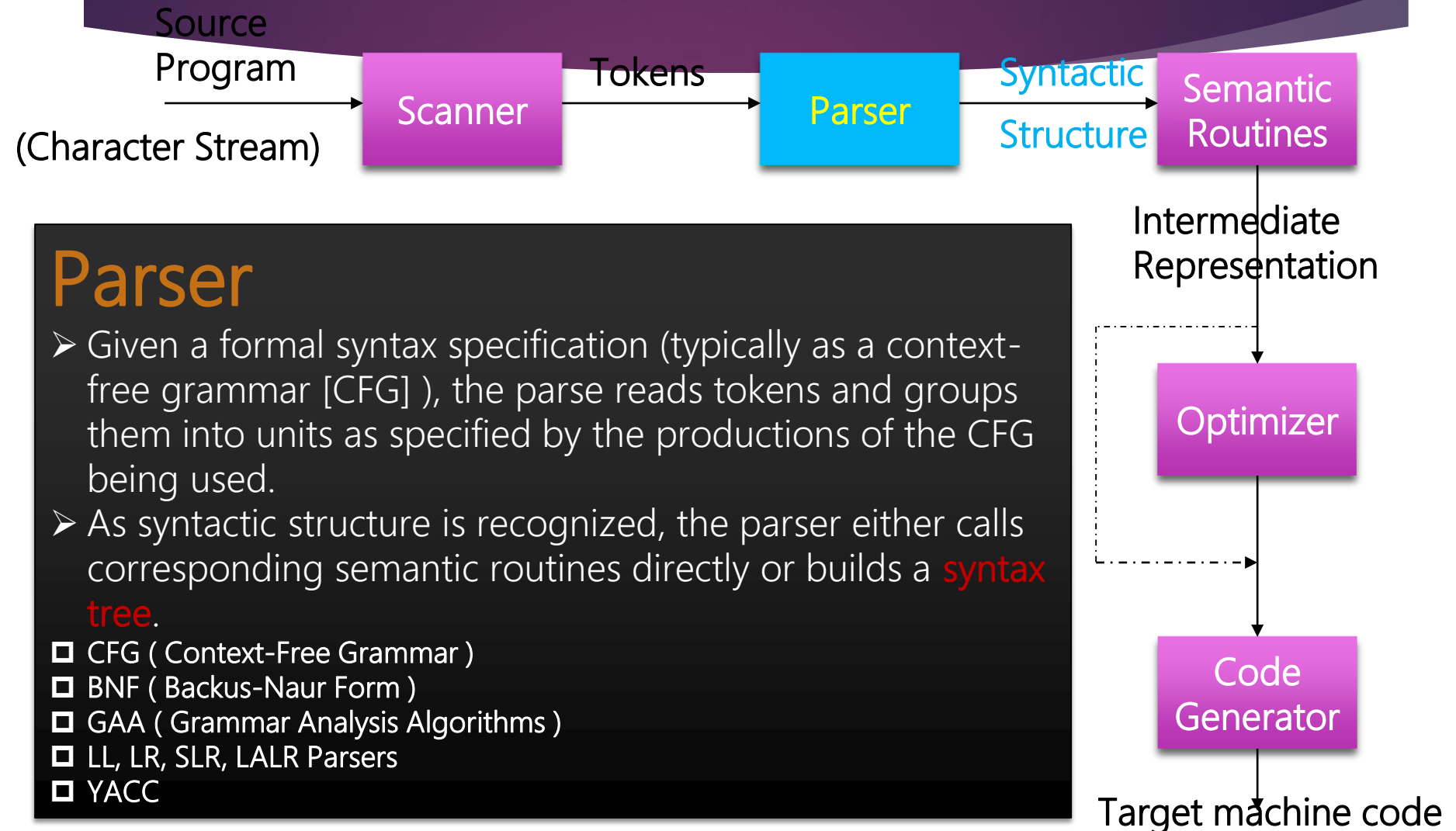
23



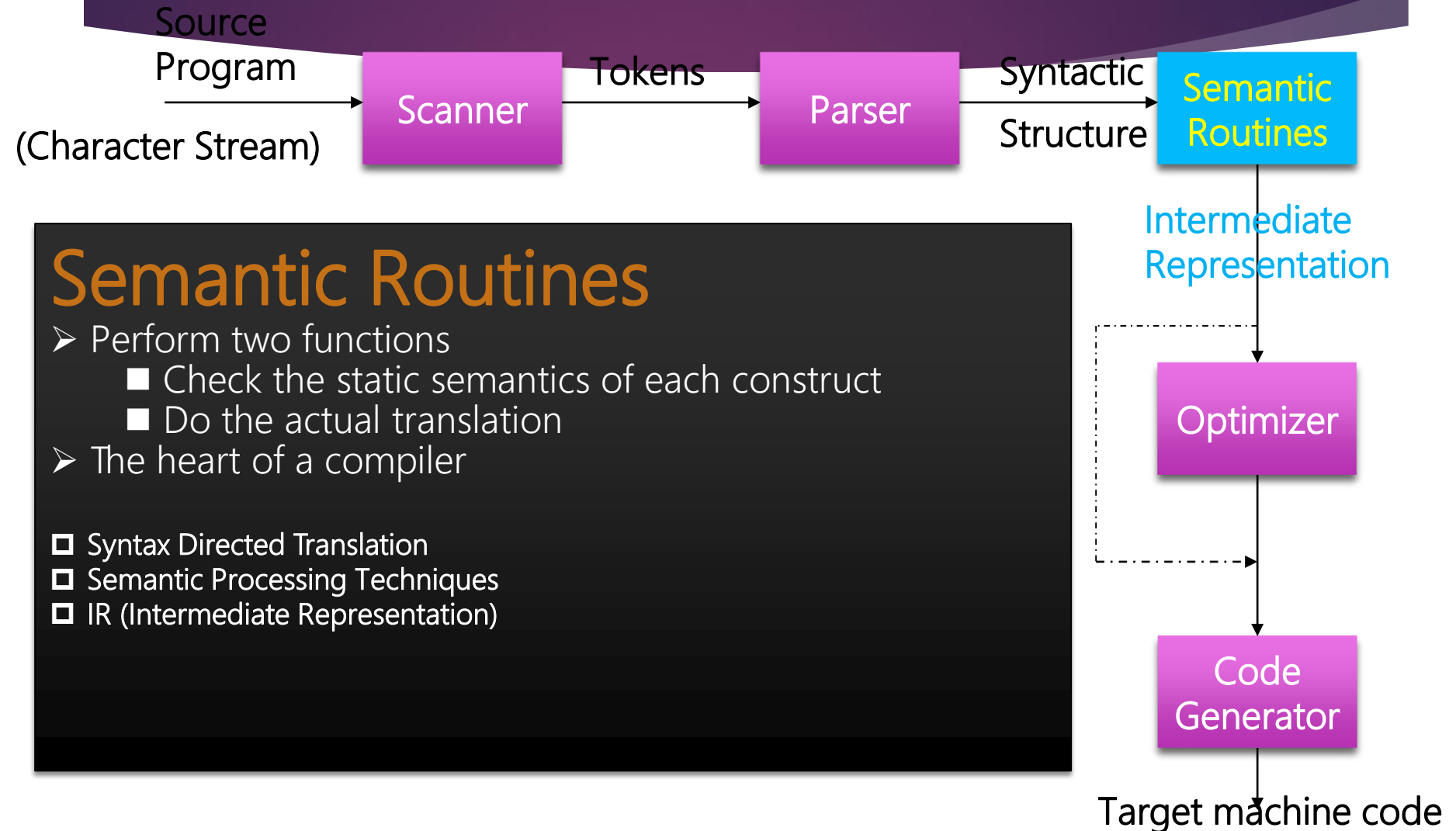
# The Structure of a Compiler



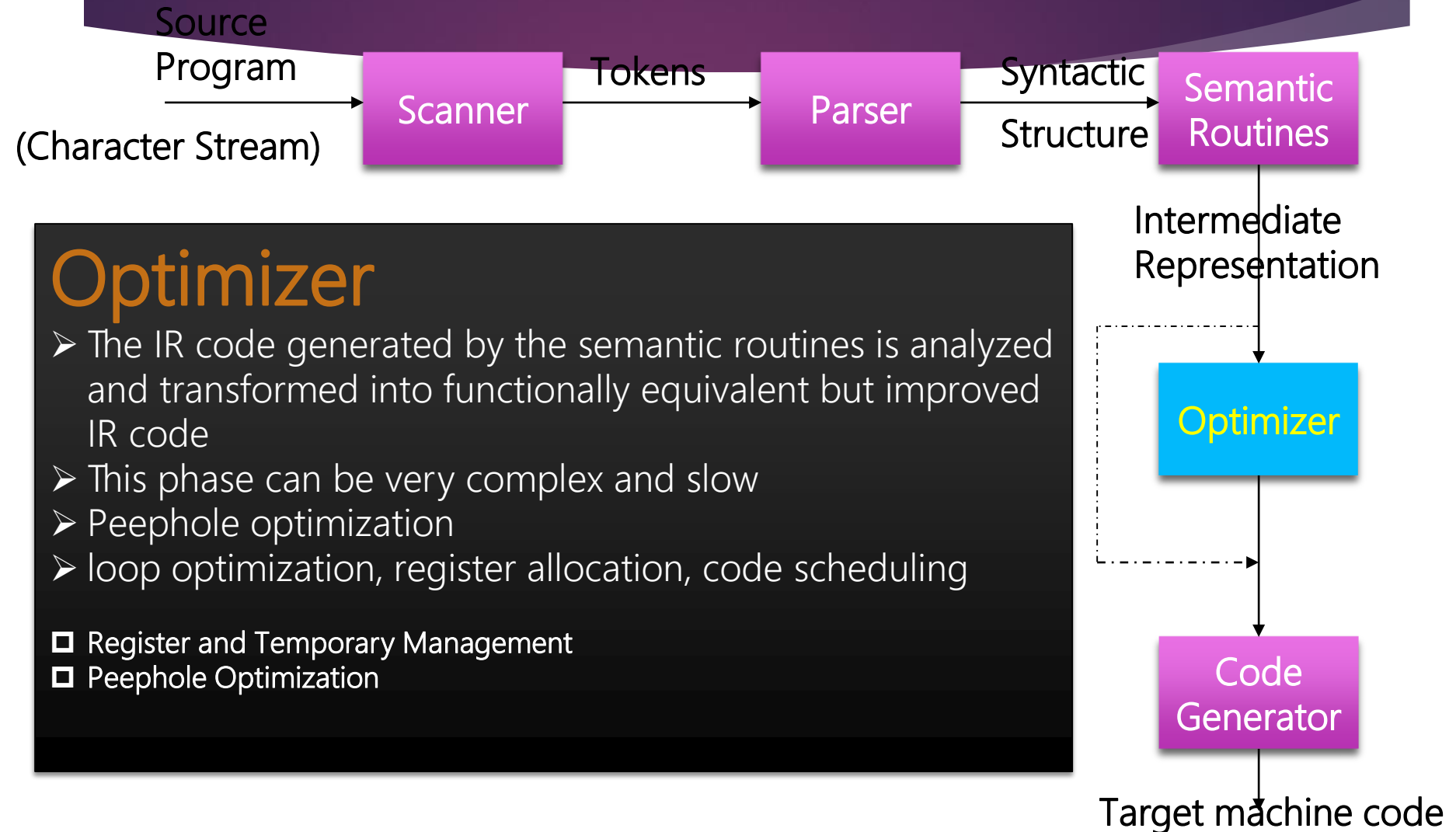
# The Structure of a Compiler



# The Structure of a Compiler

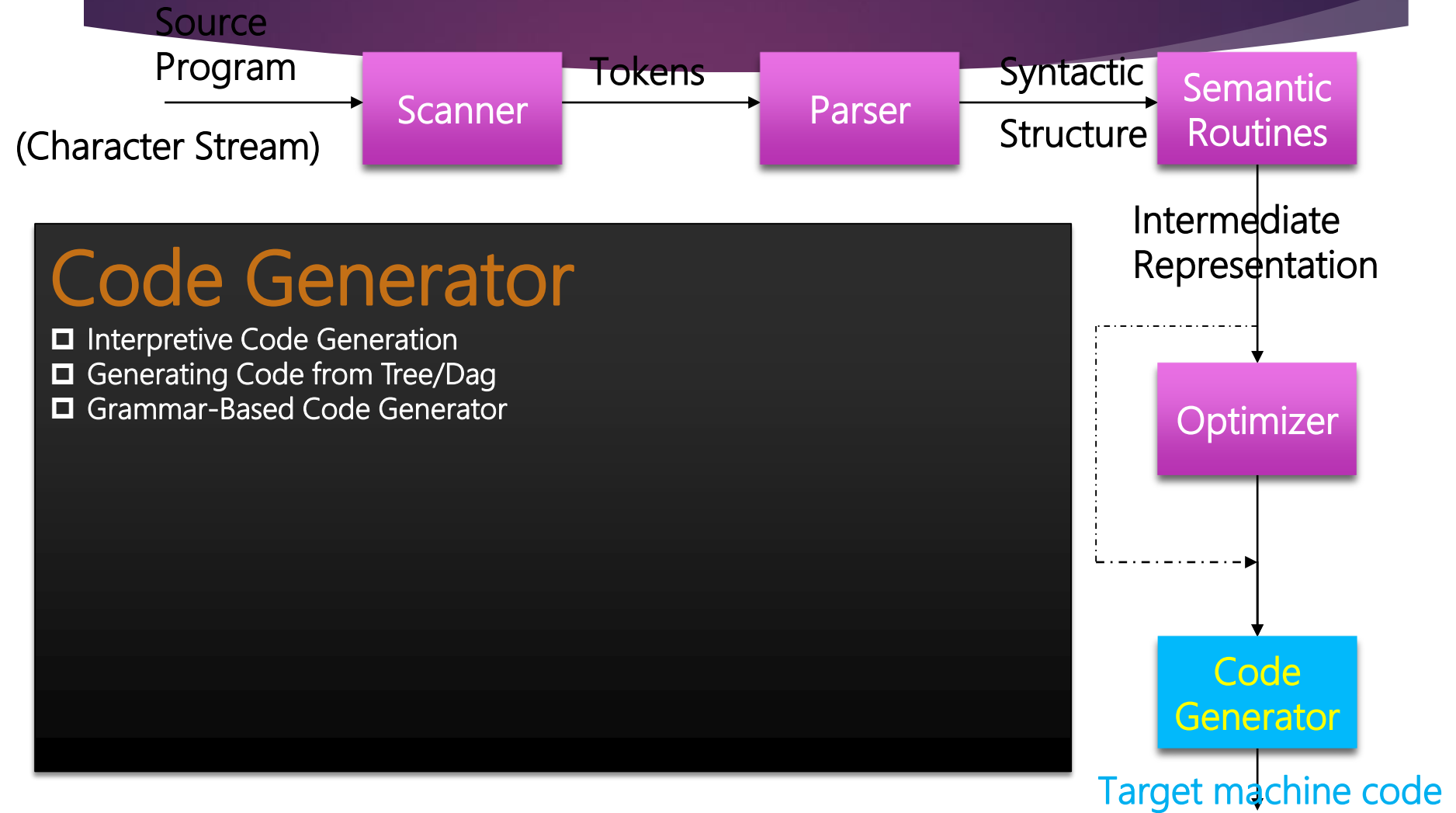


# The Structure of a Compiler



# The Structure of a Compiler

28



# The Structure of a Compiler

SYMBOL TABLE

1	position	...
2	initial	...
3	rate	...
4		

```
position := initial + rate * 60
```

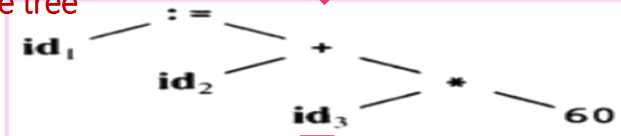
**Scanner**  
[Lexical Analyzer]

Tokens

```
id1 := id2 + id3 * 60
```

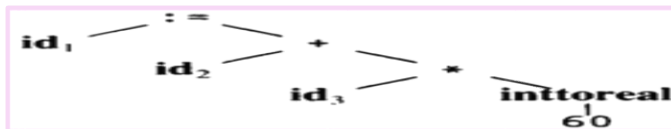
**Parser**  
[Syntax Analyzer]

Parse tree



**Semantic Process**  
[Semantic analyzer]

Abstract Syntax Tree w/ Attributes



**Code Generator**  
[Intermediate Code Generator]

Non-optimized Intermediate Code

```
temp1 := inttoreal(60)
temp2 := id3 * temp1
temp3 := id2 + temp2
id1 := temp3
```

**Code Optimizer**

Optimized Intermediate Code

```
temp1 := id3 * 60.0
id1 := id2 + temp1
```

**Code Optimizer**

Target machine code

```
MOVF id3, R2
MULF #60.0, R2
MOVF id2, R1
ADDF R2, R1
MOVF R1, id1
```

# The Structure of a Compiler

- ✓ Compiler writing tools
  - ▶ Compiler generators or compiler-compilers
    - ▶ E.g. scanner and parser generators
    - ▶ Examples : Yacc, Lex

# The Syntax and Semantics of Programming Language

- ▶ A programming language must include the specification of syntax (structure) and semantics (meaning).
- ▶ Syntax typically means the context-free syntax because of the almost universal use of context-free-grammar (CFGs)
- ▶ Ex.
  - ▶  $a = b + c$  is syntactically legal
  - ▶  $b + c = a$  is illegal

# The Syntax and Semantics of Programming Language

- ▶ The semantics of a programming language are commonly divided into two classes:
  - ▶ Static semantics
    - ▶ Semantics rules that can be checked at compiled time.
    - ▶ Ex. The type and number of a function's arguments
  - ▶ Runtime semantics
    - ▶ Semantics rules that can be checked only at run time

# Compiler Design and Programming Language Design

- ▶ An interesting aspect is how programming language design and compiler design influence one another.
- ▶ Programming languages that are easy to compile have many advantages

# Computer Architecture and Compiler Design

34

- ▶ Compilers should exploit the hardware-specific feature and computing capability to optimize code.
- ▶ The problems encountered in modern computing platforms:
  - ▶ Instruction sets for some popular architectures are highly nonuniform.
  - ▶ High-level programming language operations are not always easy to support.
    - ▶ Ex. exceptions, threads, dynamic heap access ...
  - ▶ Exploiting architectural features such as cache, distributed processors and memory
  - ▶ Effective use of a large number of processors

# Compiler Design Considerations

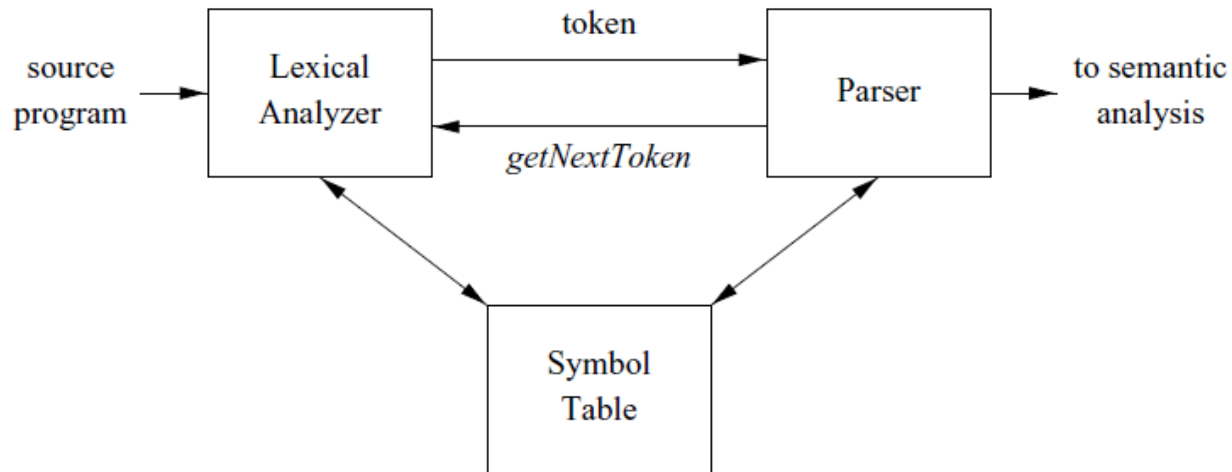
- ▶ Debugging Compilers
  - ▶ Designed to aid in the development and debugging of programs.
- ▶ Optimizing Compilers
  - ▶ Designed to produce efficient target code
- ▶ Retargetable Compilers
  - ▶ A compiler whose target architecture can be changed without its machine-independent components having to be rewritten.

# Tools

- ▶ Parser generator
  - ▶ Yacc
- ▶ Scanner generators
  - ▶ Lex
- ▶ SDT
  - ▶ ICG
- ▶ Code generator generator
- ▶ Data flow analysis engines
  - ▶ Code optimization
- ▶ Compiler construction toolkits

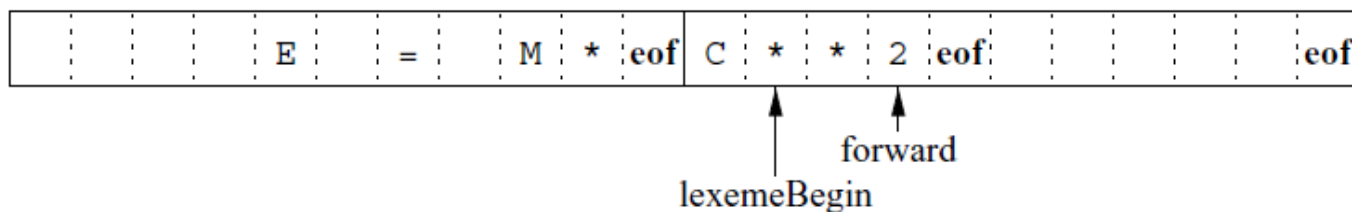
# Role of Lexical analysis

- ▶ Src program
- ▶ Lexical analyzer
  - ▶ Token
- ▶ Parser
  - ▶ Get next token
- ▶ Symbol table
- ▶ To semantic analyzer



# Lexical analysis vs parsing

- ▶ Removal of white spaces
- ▶ Buffering
- ▶ Tokens
- ▶ Patterns
- ▶ Lexemes



# Lookahead buffering

```
switch ( *forward++ ) {  
    case eof:  
        if ( forward is at end of first buffer ) {  
            reload second buffer;  
            forward = beginning of second buffer;  
        }  
        else if ( forward is at end of second buffer ) {  
            reload first buffer;  
            forward = beginning of first buffer;  
        }  
        else /* eof within a buffer marks the end of input */  
            terminate lexical analysis;  
        break;  
    Cases for the other characters  
}
```

# Attributes for tokens

- ▶ Id
- ▶ Refers to symtab

# Lexical errors

- ▶ Panic mode
- ▶ Error recovery actions
  - ▶ Delete
  - ▶ Insert
  - ▶ Replace
  - ▶ Transpose
  - ▶ Examples

# Input buffering

- ▶ Buffer pairs
- ▶ Eof
- ▶ Sentinels

# Specifications of tokens

- ▶ Alphabet
- ▶ Binary
- ▶ Look ahead code with sentinels
- ▶ String
  - ▶ Prefix
  - ▶ Suffix
  - ▶ Substring
  - ▶ Proper (ones of above)
  - ▶ Subsequence
  - ▶ concatenation
- ▶ Empty string
- ▶ Empty set
- ▶ Language

# Operations on languages

- ▶ Kleene closure

- ▶  $L^*$

- ▶  $L^+$

- ▶  $L^0$

- ▶ Union

- ▶ Concatenation

- ▶ Examples

- ▶  $L$

- ▶  $D$

- ▶  $LUD$

- ▶  $L^4$

- ▶  $L.(LUD)^*$

# RE

Example:

- ▶ Letter\_(letter\_|digit)\*
- ▶ 2 basic rules
  - ▶ Empty set
  - ▶  $L(a)$
- ▶ Induction
  - ▶  $L(r), L(s)$
  - ▶  $(r)|(s)$
  - ▶  $r.s$
  - ▶  $(r)^*$
  - ▶  $(r)$

# Precedence

- ▶ Left associative
- ▶ \*
- ▶ .
- ▶ |

# Example

## Example

- ▶ Alphabet =  $\{a,b\}$
- ▶ Write example RE

# Regular set

- ▶ Language defined by RE
- ▶ Algebraic laws
  - ▶ Commutative over  $|$
  - ▶ Associative over  $.$
  - ▶ Concatenation associative
  - ▶ Concatenation distributive
  - ▶ Identity over epsilon and kleene closure

# Extensions of RE

- ▶ One or more instances
- ▶ Zero or more instances
- ▶ Character classes

# Regular definition

- ▶  $D_1 \rightarrow r_1$
- ▶  $D_2 \rightarrow r_2$
- ▶  $D_3 \rightarrow r_3$
- ▶  $D_i$  is symbol
- ▶  $R_i$  is RE
- ▶ Examples
  - ▶ Letter  $\rightarrow a|b|c|\dots|z$
  - ▶ Digit  $\rightarrow 0|1|\dots|9$
  - ▶ Id  $\rightarrow \text{letter\_}(\text{letter\_}|\text{digit})^*$

# Problems

- ▶ Write regular definition for Unsigned numbers
- ▶ Example inputs
  - ▶ 8456
  - ▶ 2.345
  - ▶ 345.56E-56

## Extension of RE

- ▶  $(L(r))^+$
- ▶ Epsilon
- ▶  $L(r)$
- ▶  $[a_1, a_2, \dots, a_n]$
- ▶  $A_1 | a_2 | \dots | a_n$

# Recognition of Tokens

```
stmt  →  if expr then stmt  
        |  if expr then stmt else stmt  
        |   $\epsilon$   
expr  →  term relop term  
        |  term  
term  →  id  
        |  number
```

A grammar for branching statements

# Patterns for recognizing tokens

<i>digit</i>	→	[0-9]
<i>digits</i>	→	<i>digit</i> <sup>+</sup>
<i>number</i>	→	<i>digits</i> ( . <i>digits</i> )? ( E [+-]? <i>digits</i> )?
<i>letter</i>	→	[A-Za-z]
<i>id</i>	→	<i>letter</i> ( <i>letter</i>   <i>digit</i> )*
<i>if</i>	→	if
<i>then</i>	→	then
<i>else</i>	→	else
<i>relop</i>	→	<   >   <=   >=   =   <>

Patterns for tokens

To recognize spaces

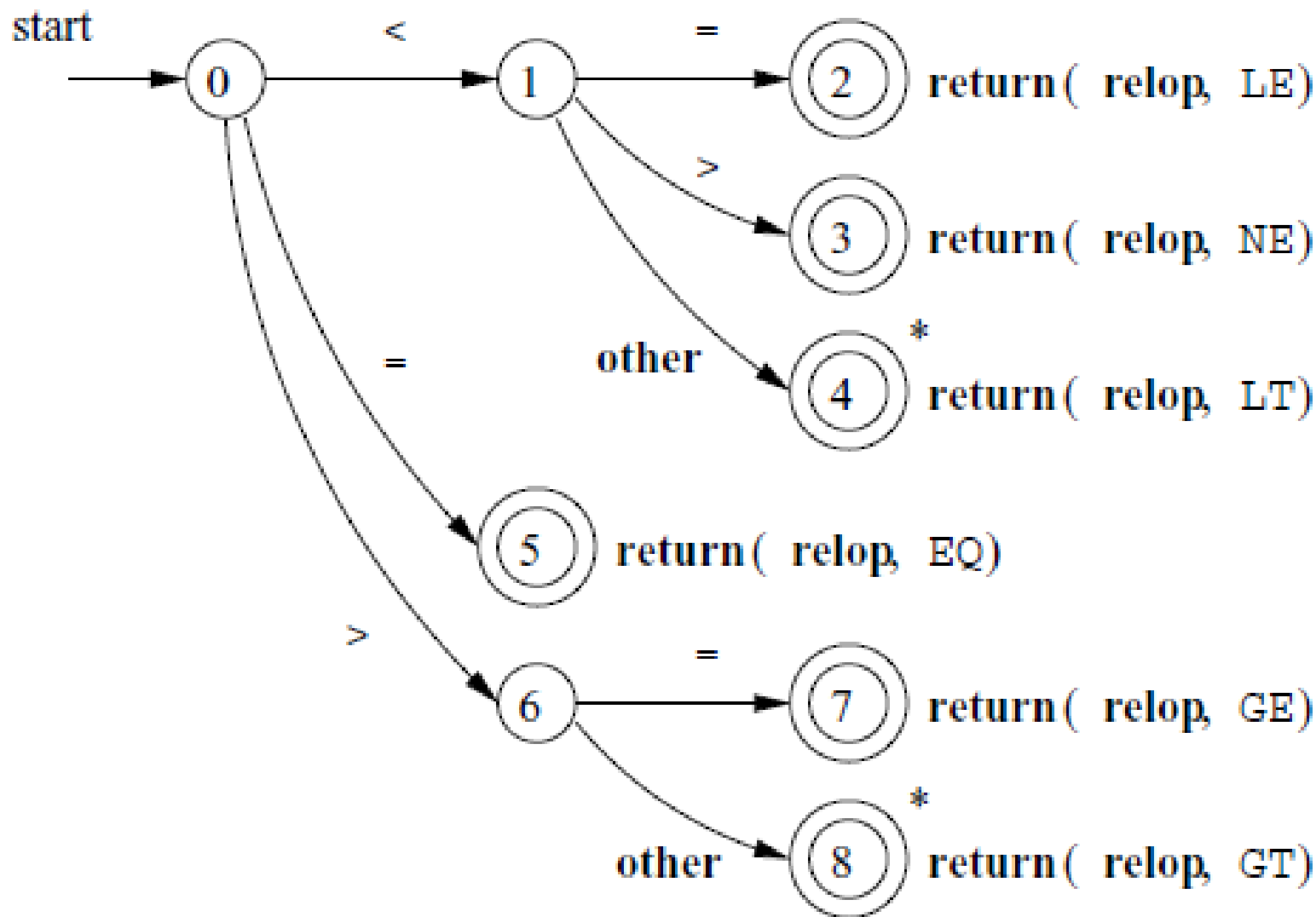
$$ws \rightarrow ( \text{blank} \mid \text{tab} \mid \text{newline} )^+$$

LEXEMES	TOKEN NAME	ATTRIBUTE VALUE
Any <i>ws</i>	—	—
if	if	—
then	then	—
else	else	—
Any <i>id</i>	id	Pointer to table entry
Any <i>number</i>	number	Pointer to table entry
<	relop	LT
<=	relop	LE
=	relop	EQ
<>	relop	NE
>	relop	GT
>=	relop	GE

Tokens, their patterns, and attribute values

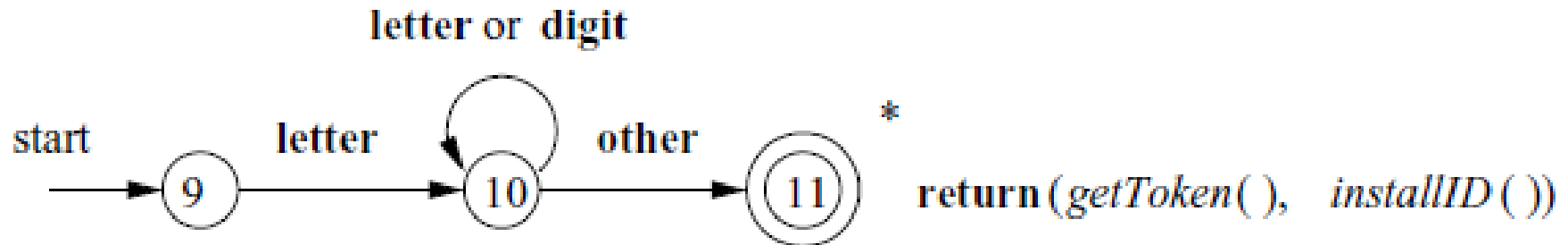
# Transition Diagrams

- ▶ States
- ▶ Lexeme begin
- ▶ Forward
- ▶ Edges
- ▶ Deterministic transition diagrams
- ▶ Accepting / Final state
- ▶ Start state/ Initial state

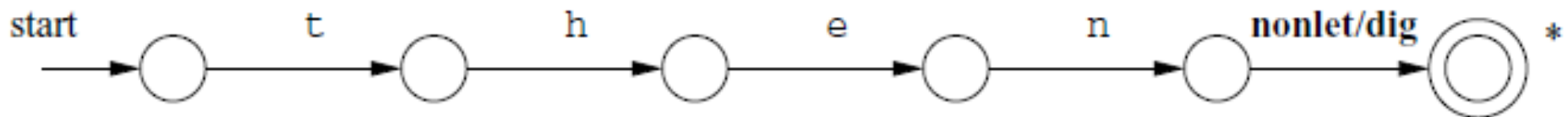


Transition diagram for **relop**

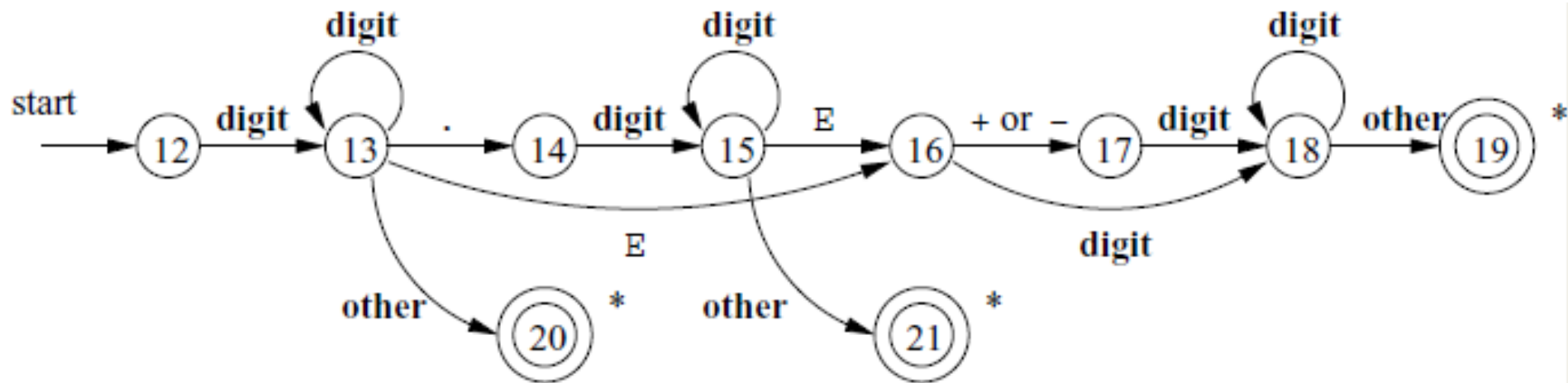
# Recognition of reserved words and identifiers



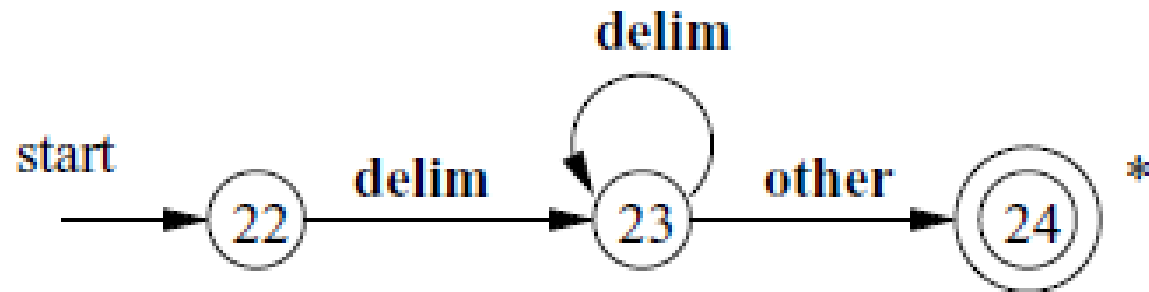
A transition diagram for **id**'s and keywords



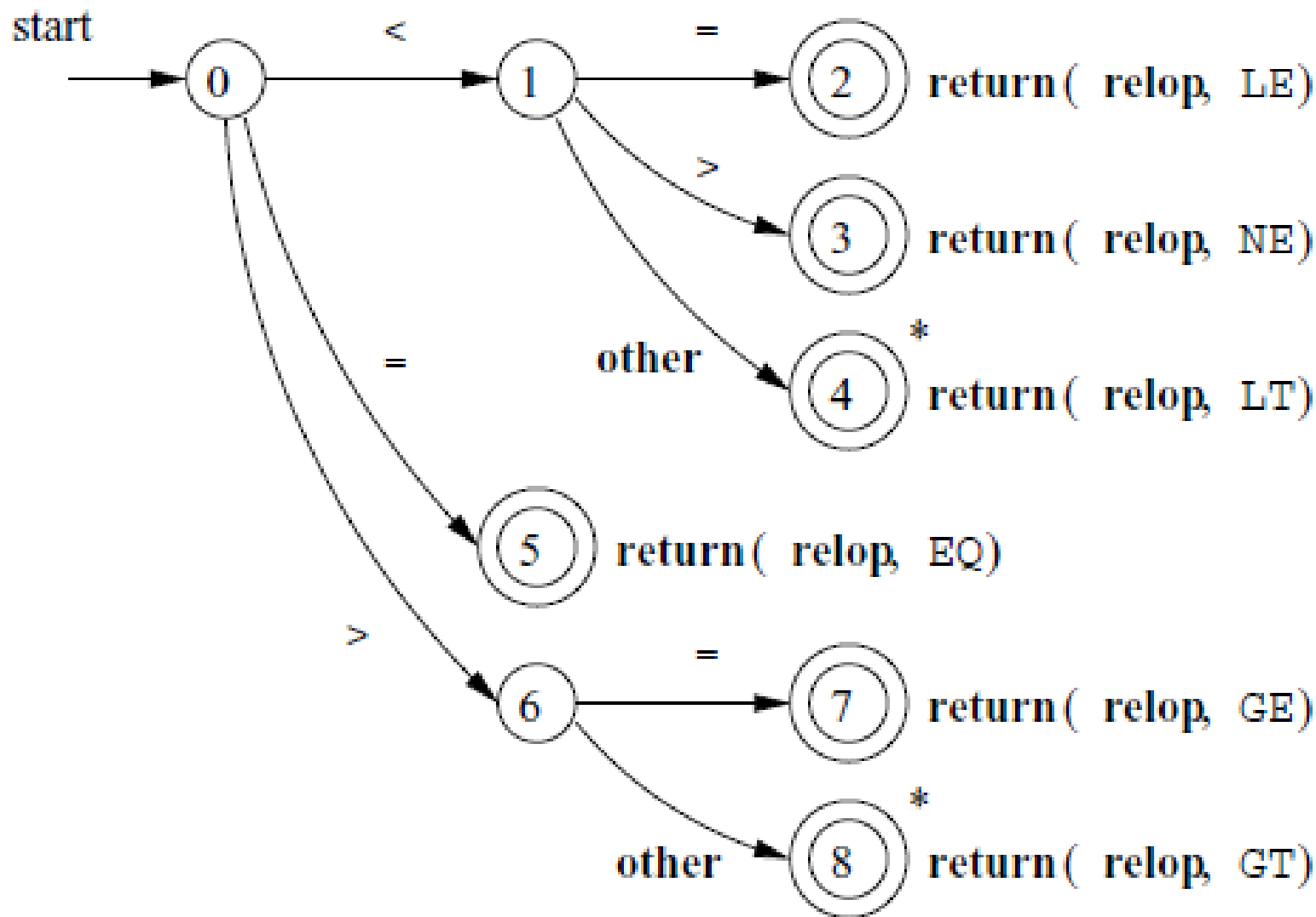
Hypothetical transition diagram for the keyword `then`



A transition diagram for unsigned numbers



A transition diagram for whitespace

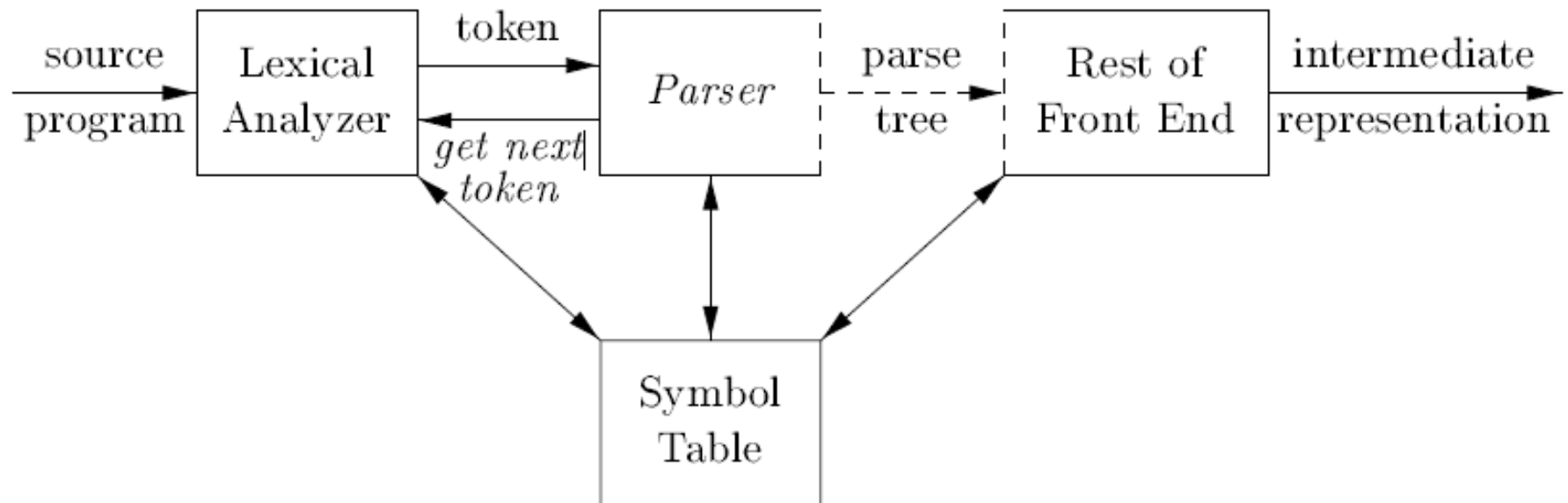


Transition diagram for **relop**

# Sketch of Implementation of relop() Transition diagram

```
TOKEN getRelop()
{
    TOKEN retToken = new(RELOP);
    while(1) { /* repeat character processing until a return
                or failure occurs */
        switch(state) {
            case 0: c = nextChar();
                    if ( c == '<' ) state = 1;
                    else if ( c == '=' ) state = 5;
                    else if ( c == '>' ) state = 6;
                    else fail(); /* lexeme is not a relop */
                    break;
            case 1: ...
                    ...
            case 8: retract();
                    retToken.attribute = GT;
                    return(retToken);
        }
    }
}
```

# Role of Parser



Position of parser in compiler model

# Parsers

- ▶ Type of parsers
  - ▶ Bottom up
  - ▶ Top down
- ▶ Input to parser is scanned from left to right, one symbol at a time

# Errors in different levels

- ▶ Lexical errors
- ▶ Syntactic errors
- ▶ Semantic errors
- ▶ Logical errors
- ▶ Error handler works:
  - ▶ Report error
  - ▶ Correct error

# Error recovery strategies

- ▶ Panic mode
  - ▶ Synchronizing tokens
  - ▶ Eg : { , ;
- ▶ Phrase level
  - ▶ Eg: Insert missing semicolon
- ▶ Error productions
  - ▶ Production rules
- ▶ Global corrections
  - ▶ Replace string x by y

# Grammar

- ▶ Terminal
  - ▶ Token name
- ▶ Non terminal
  - ▶ Set of strings

$expression \rightarrow expression + term$   
 $expression \rightarrow expression - term$   
 $expression \rightarrow term$   
 $term \rightarrow term * factor$   
 $term \rightarrow term / factor$   
 $term \rightarrow factor$   
 $factor \rightarrow ( expression )$   
 $factor \rightarrow \mathbf{id}$

Grammar for simple arithmetic expressions

# Ambiguity

- ▶ Grammar
  - ▶ More than one parse tree for an input string
- ▶ G:
  - $E \rightarrow E + E$
  - $E \rightarrow \text{id} \mid \text{num}$
  - Input :  $a + b + c$

# Lexical vs syntax analysis

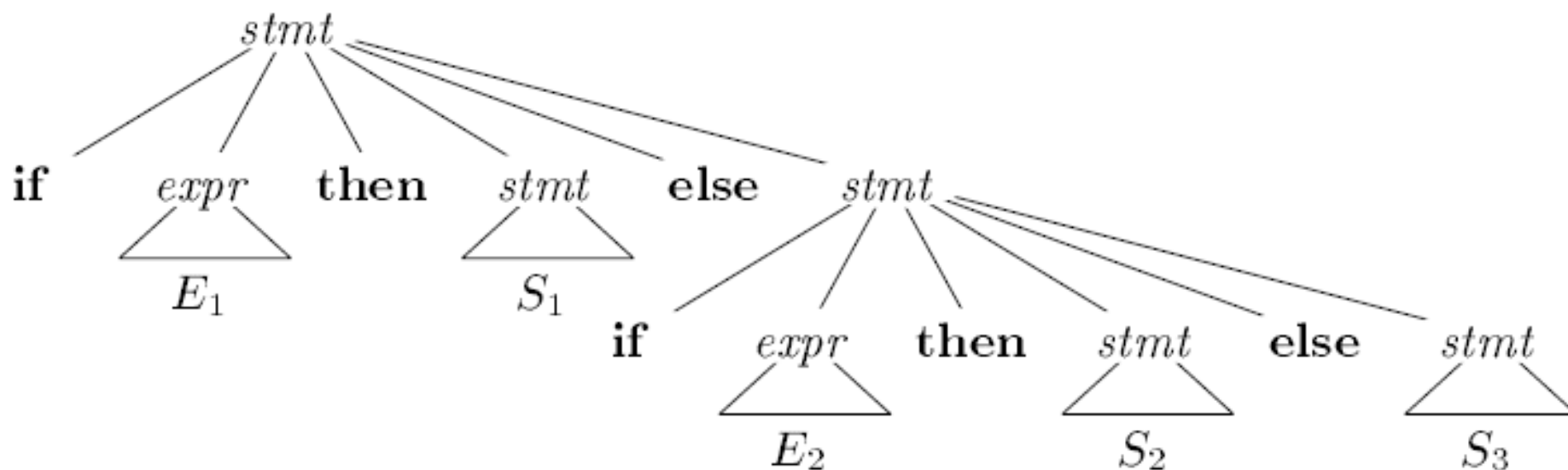
- ▶ Lexer
  - ▶ RE
- ▶ Parser
  - ▶ grammar

## Eliminating ambiguity

*stmt*     $\rightarrow$     **if** *expr* **then** *stmt*  
             |        **if** *expr* **then** *stmt* **else** *stmt*  
             |        **other**

Pbm : Draw parse tree for the given input string

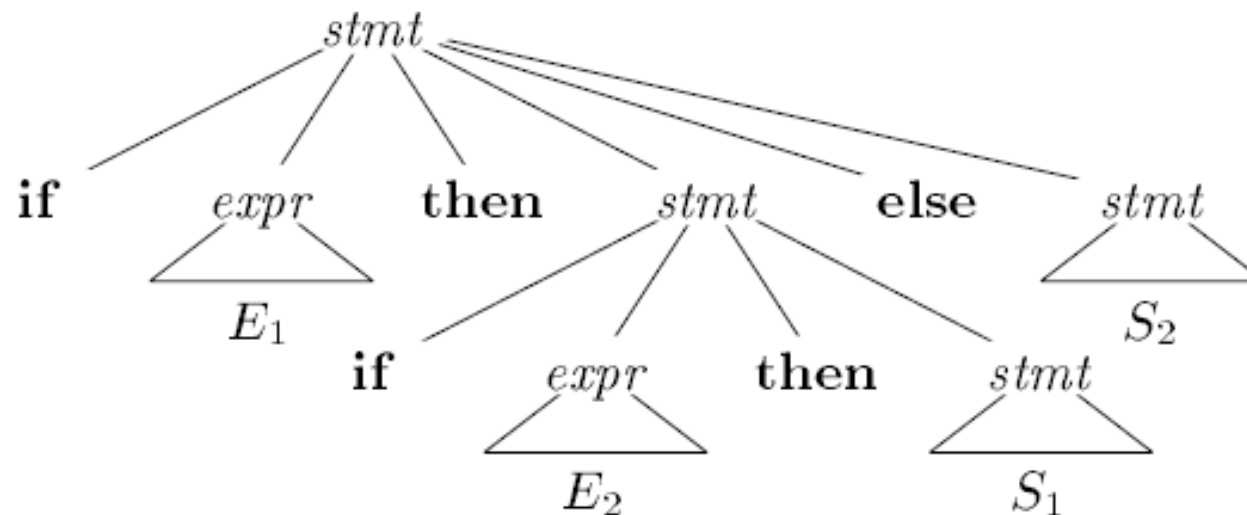
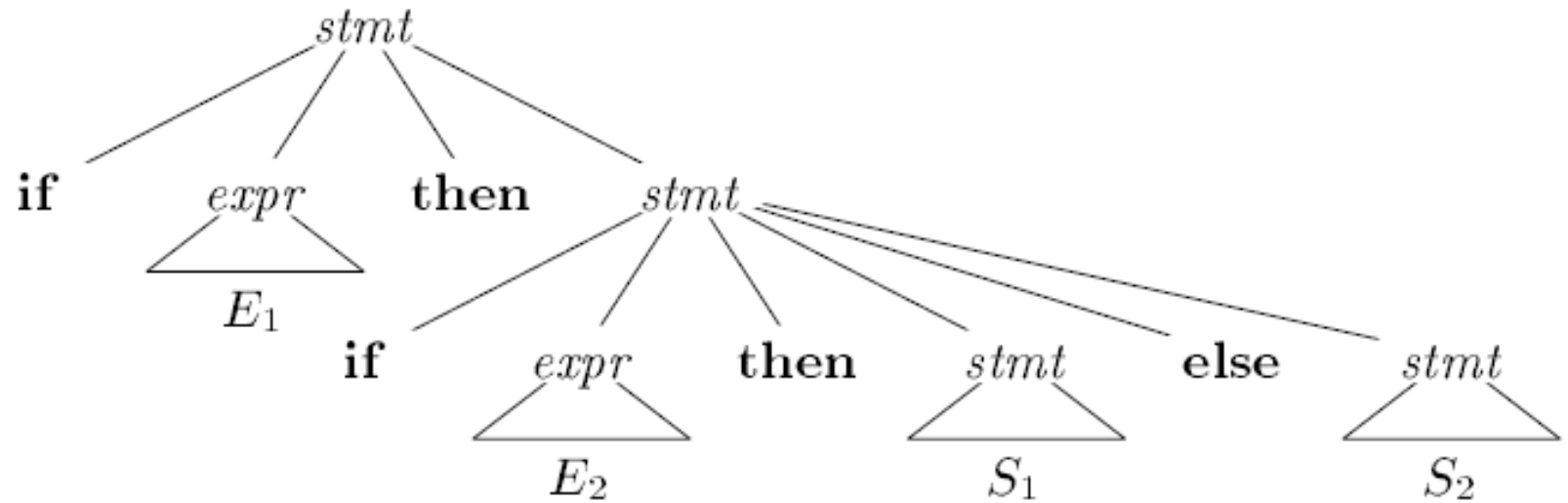
if  $E_1$  then  $S_1$  else if  $E_2$  then  $S_2$  else  $S_3$



Parse tree for a conditional statement

Draw Parse tree for the given input string

if  $E_1$  then if  $E_2$  then  $S_1$  else  $S_2$



$$\begin{array}{ll} stmt & \rightarrow \quad matched\_stmt \\ & \quad | \quad open\_stmt \\ matched\_stmt & \rightarrow \quad \mathbf{if} \ expr \ \mathbf{then} \ matched\_stmt \ \mathbf{else} \ matched\_stmt \\ & \quad | \quad \mathbf{other} \\ open\_stmt & \rightarrow \quad \mathbf{if} \ expr \ \mathbf{then} \ stmt \\ & \quad | \quad \mathbf{if} \ expr \ \mathbf{then} \ matched\_stmt \ \mathbf{else} \ open\_stmt \end{array}$$

Unambiguous grammar for if-then-else statements



**Algorithm**                      Eliminating left recursion.

**INPUT:** Grammar  $G$  with no cycles or  $\epsilon$ -productions.

**OUTPUT:** An equivalent grammar with no left recursion.

**METHOD:** Apply the algorithm                      to  $G$ . Note that the resulting non-left-recursive grammar may have  $\epsilon$ -productions.

- 1)    arrange the nonterminals in some order  $A_1, A_2, \dots, A_n$ .
- 2)    **for** ( each  $i$  from 1 to  $n$  ) {
- 3)         **for** ( each  $j$  from 1 to  $i - 1$  ) {
- 4)                replace each production of the form  $A_i \rightarrow A_j \gamma$  by the  
                         productions  $A_i \rightarrow \delta_1 \gamma \mid \delta_2 \gamma \mid \dots \mid \delta_k \gamma$ , where  
                          $A_j \rightarrow \delta_1 \mid \delta_2 \mid \dots \mid \delta_k$  are all current  $A_j$ -productions
- 5)                }
- 6)        eliminate the immediate left recursion among the  $A_i$ -productions
- 7)    }

# Left recursion

► Cases:  $A \rightarrow A\alpha$

$$A \rightarrow A\alpha \mid \beta$$

# Left recursion elimination

$$\begin{array}{l} A \rightarrow \beta A' \\ A' \rightarrow \alpha A' \quad | \quad \epsilon \end{array}$$

## Example Grammar : Left recursion

$$\begin{aligned} E &\rightarrow E + T \mid E - T \mid T \\ T &\rightarrow T * F \mid T / F \mid F \\ F &\rightarrow ( E ) \mid \mathbf{id} \end{aligned}$$

# Left recursion

G:

$$E \rightarrow E + T \mid T$$

LRE :

$$E \rightarrow T E'$$

$$E' \rightarrow + T E' \mid \epsilon$$

# LRE

$$T \rightarrow T * F \mid F$$

LRE:

$$T \rightarrow F T'$$

$$T' \rightarrow * F T' \mid \epsilon$$

# LRE

$F \rightarrow (E) \mid \text{id}$

No LRE needed

## Left recursion other cases

$$A \rightarrow A\alpha_1 \mid A\alpha_2 \mid \cdots \mid A\alpha_m \mid \beta_1 \mid \beta_2 \mid \cdots \mid \beta_n$$

## LRE

$$\begin{aligned} A &\rightarrow \beta_1 A' \mid \beta_2 A' \mid \cdots \mid \beta_n A' \\ A' &\rightarrow \alpha_1 A' \mid \alpha_2 A' \mid \cdots \mid \alpha_m A' \mid \epsilon \end{aligned}$$

## Left recursion other cases

$$\begin{array}{l} S \rightarrow A a \mid b \\ A \rightarrow A c \mid S d \mid \epsilon \end{array}$$

## Left recursion

$$S \Rightarrow Aa \Rightarrow Sda$$

# Left recursion

$$A \rightarrow A c \mid A a d \mid b d \mid \epsilon$$

## LRE

$$\begin{aligned}
 S &\rightarrow A a \mid b \\
 A &\rightarrow b d A' \mid A' \\
 A' &\rightarrow c A' \mid a d A' \mid \epsilon
 \end{aligned}$$

## Need for Left factoring

$$A \rightarrow \alpha\beta_1 \mid \alpha\beta_2$$

**Algorithm** : Left factoring a grammar.

**INPUT:** Grammar  $G$ .

**OUTPUT:** An equivalent left-factored grammar.

**METHOD:** For each nonterminal  $A$ , find the longest prefix  $\alpha$  common to two or more of its alternatives. If  $\alpha \neq \epsilon$  — i.e., there is a nontrivial common prefix — replace all of the  $A$ -productions  $A \rightarrow \alpha\beta_1 \mid \alpha\beta_2 \mid \cdots \mid \alpha\beta_n \mid \gamma$ , where  $\gamma$  represents all alternatives that do not begin with  $\alpha$ , by

$$\begin{aligned} A &\rightarrow \alpha A' \mid \gamma \\ A' &\rightarrow \beta_1 \mid \beta_2 \mid \cdots \mid \beta_n \end{aligned}$$

Here  $A'$  is a new nonterminal. Repeatedly apply this transformation until no two alternatives for a nonterminal have a common prefix.  $\square$

# Left factoring

$$A \rightarrow \alpha A'$$

$$A' \rightarrow \beta_1 \mid \beta_2$$

## Example

$stmt \rightarrow \begin{array}{l} \text{if } expr \text{ then } stmt \text{ else } stmt \\ \text{if } expr \text{ then } stmt \end{array}$

# Example

$$\begin{aligned} S &\rightarrow i \ E \ t \ S \mid i \ E \ t \ S \ e \ S \mid a \\ E &\rightarrow b \end{aligned}$$

## Left factoring applied

$$\begin{aligned} S &\rightarrow i \ E \ t \ S \ S' \mid a \\ S' &\rightarrow e \ S \mid \epsilon \\ E &\rightarrow b \end{aligned}$$

# Problems to solve

► Pbm 1:

Grammar: Apply LRE

$S \rightarrow Sa \mid Sb \mid c \mid d$

► Pbm 2:

Grammar : Apply LRE

$A \rightarrow Br$

$B \rightarrow Cd$

$C \rightarrow At$

# Problems

► Apply left factoring:

$S \rightarrow 0S1 \mid 01$