COMPILER DESIGN

DR. ABHISHEK RAY

Associate Professor & Dean (T&P) Industry Engagements

Mobile: 78944 27741

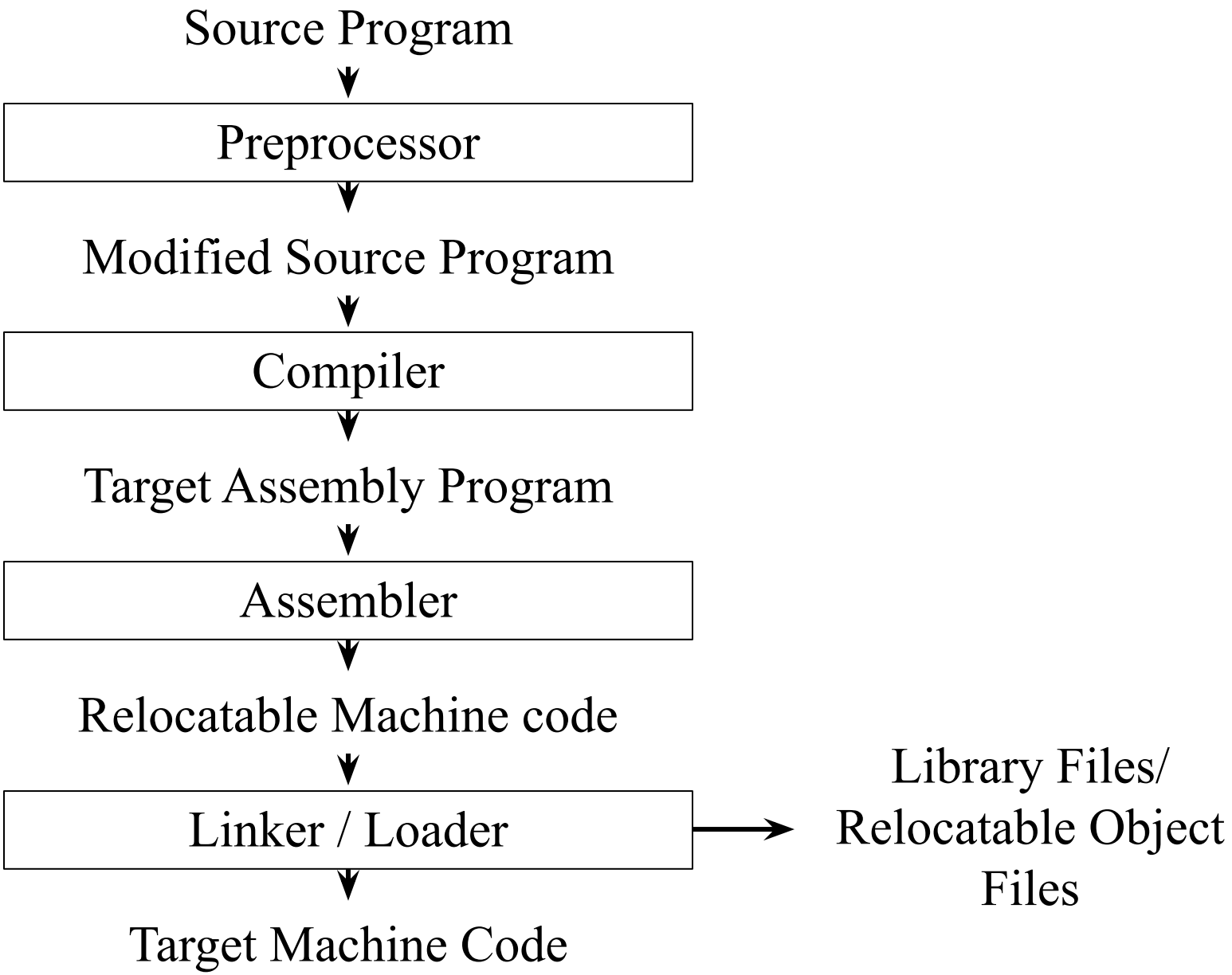
E_mail: arayfcs@kiit.ac.in

Introduction to Compiler Design

Text Book:
Principles of Compiler Design
By

Aho, Ullmann & Sethi

Language Processing System



Overview and History

- Cause
  - Software for early computers was written in assembly language
  - The benefits of reusing software on different CPUs started to become significantly greater than the cost of writing a compiler.

- The first real compiler
  - FORTRAN compilers of the late 1950s
  - 18 person-years to build

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

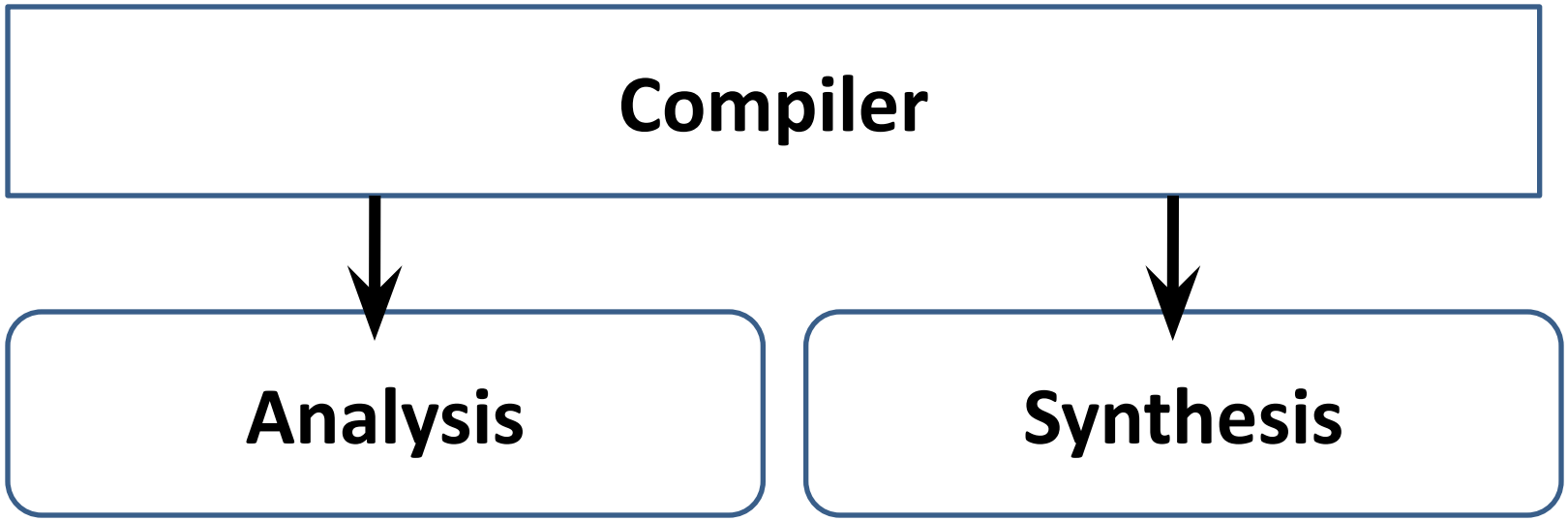
- **Compilers may generate three types of code:**
  - **Pure Machine Code**
    - **Machine instruction set without assuming the existence of any operating system or library.**
    - **Mostly being OS or embedded applications.**
  - **Augmented Machine Code**
    - **Code with OS routines and runtime support routines.**
  - **Virtual Machine Code**
    - **Virtual instructions, can be run on any architecture with a virtual machine interpreter or a just-in-time compiler**
    - **Ex. Java**

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

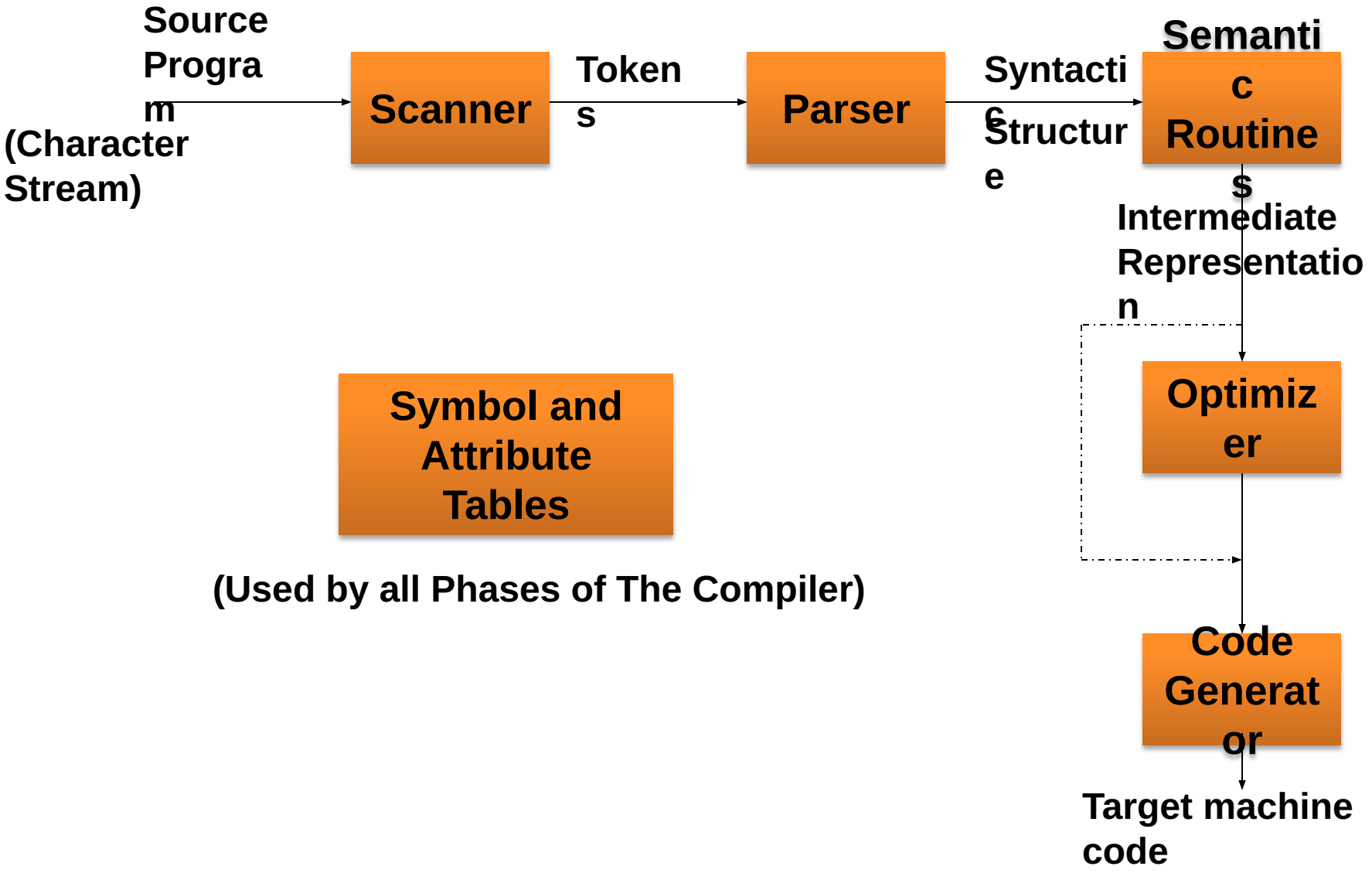
The Structure of a Compiler

- **Any compiler must perform two major tasks**

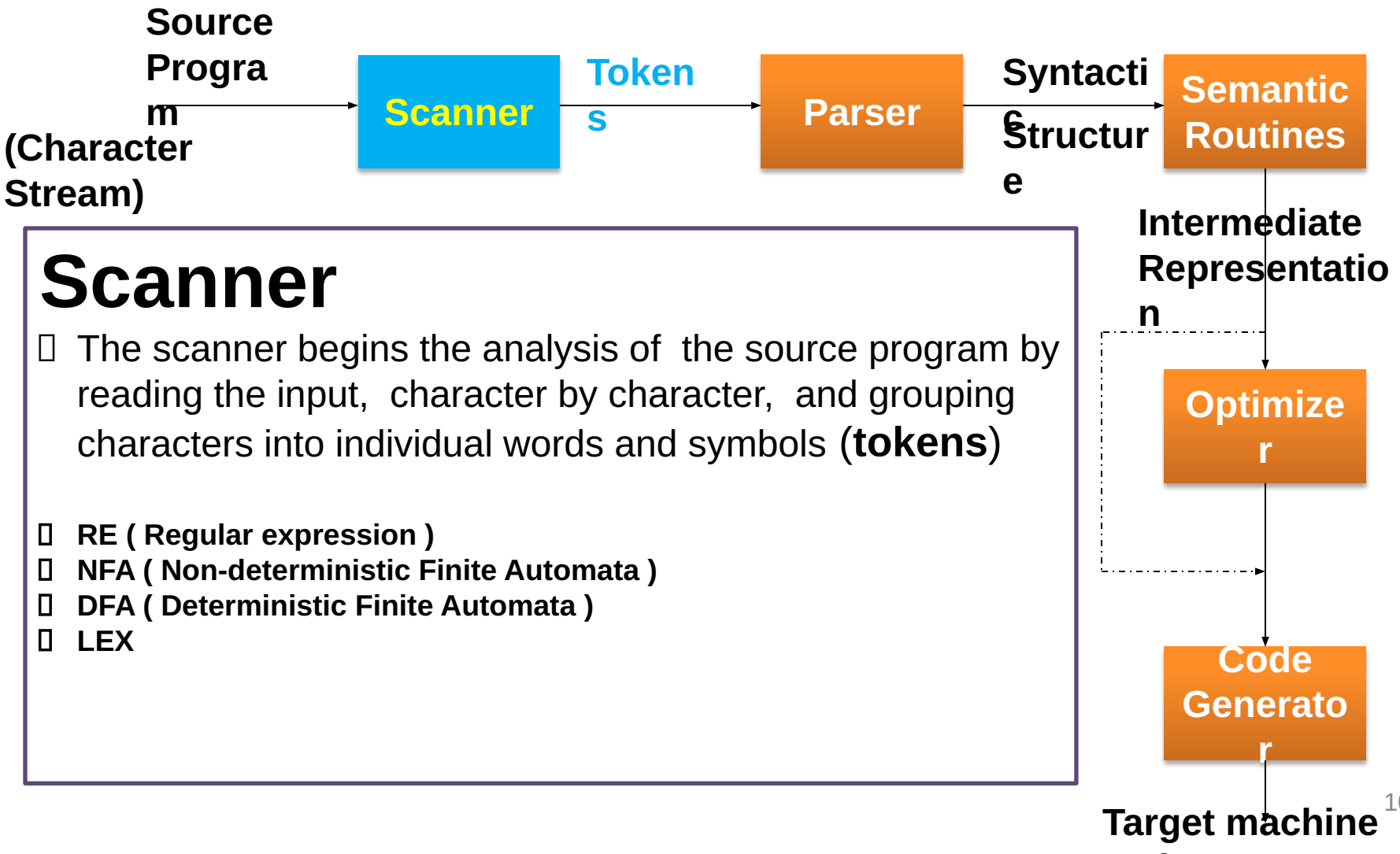


  - ***Analysis*** **of the source program**
  - ***Synthesis*** **to a machine-language program**

The Structure of a Compiler



The Structure of a Compiler

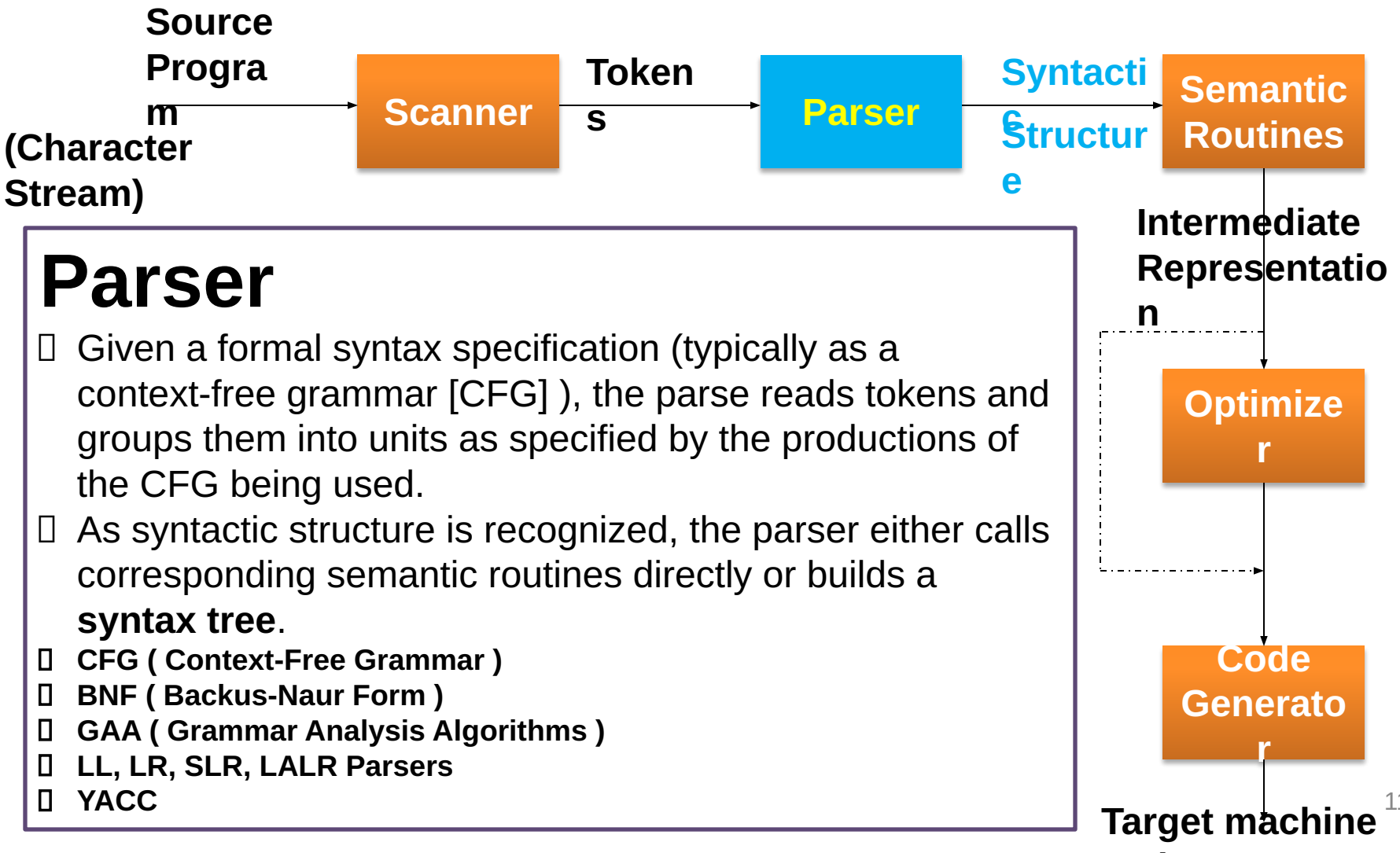


Scanner

- The scanner begins the analysis of the source program by reading the input, character by character, and grouping characters into individual words and symbols (**tokens**)

- **RE (Regular expression)**
- **NFA (Non-deterministic Finite Automata)**
- **DFA (Deterministic Finite Automata)**
- **LEX**

The Structure of a Compiler

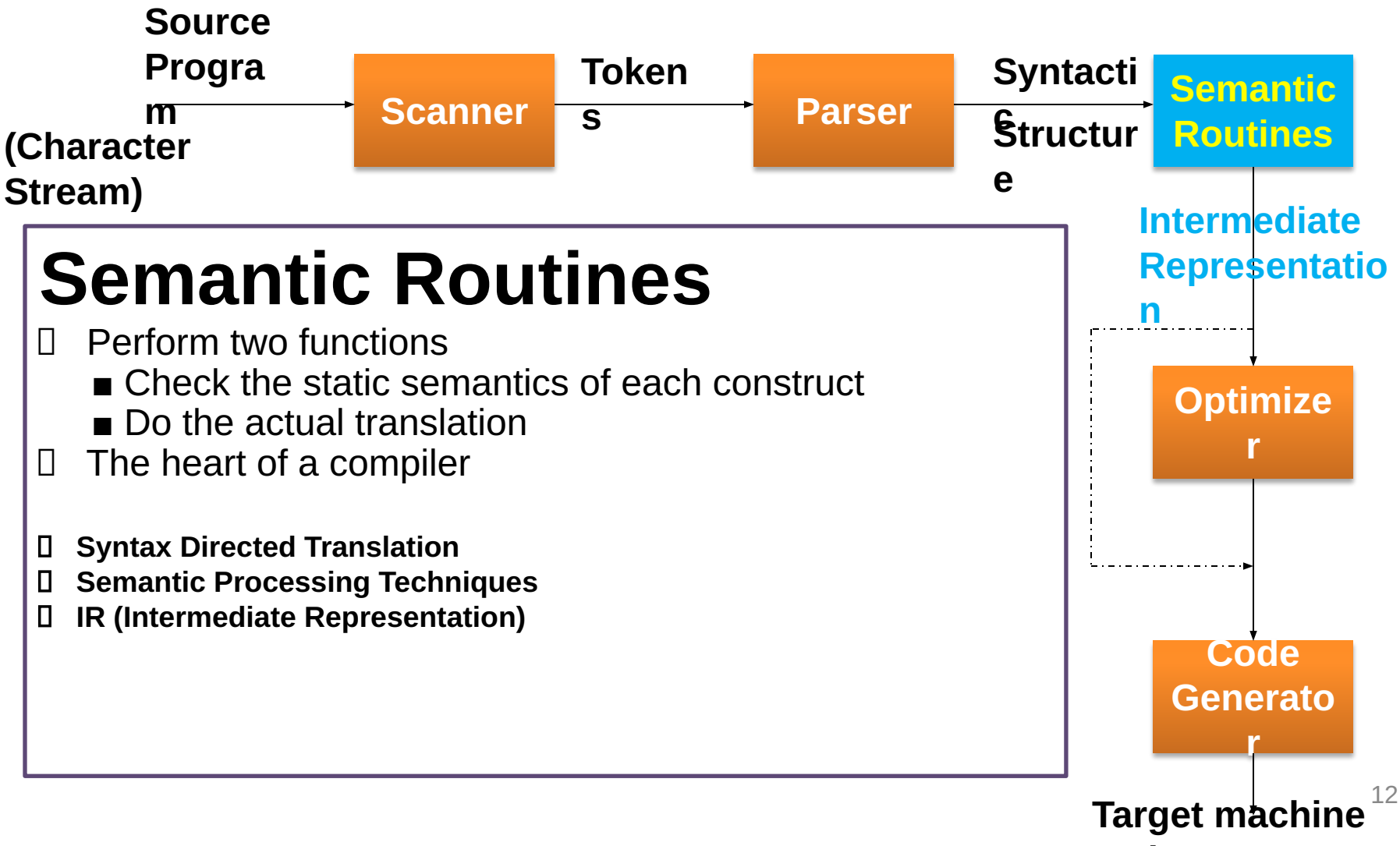


Parser

- Given a formal syntax specification (typically as a context-free grammar [CFG]), the parse reads tokens and groups them into units as specified by the productions of the CFG being used.
- As syntactic structure is recognized, the parser either calls corresponding semantic routines directly or builds a **syntax tree**.

- **CFG (Context-Free Grammar)**
- **BNF (Backus-Naur Form)**
- **GAA (Grammar Analysis Algorithms)**
- **LL, LR, SLR, LALR Parsers**
- **YACC**

The Structure of a Compiler

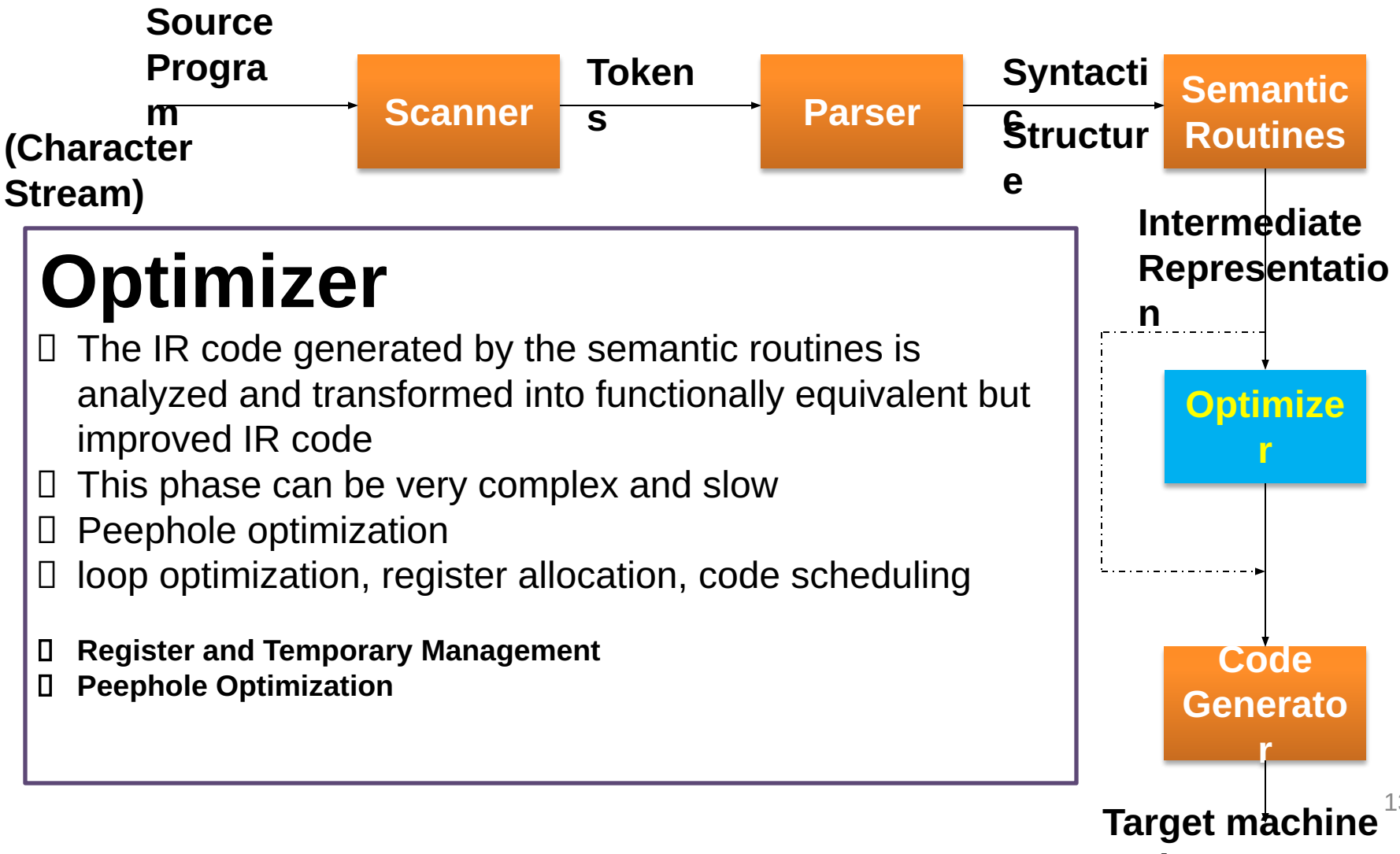


Semantic Routines

- Perform two functions
  - Check the static semantics of each construct
  - Do the actual translation
- The heart of a compiler

- **Syntax Directed Translation**
- **Semantic Processing Techniques**
- **IR (Intermediate Representation)**

The Structure of a Compiler

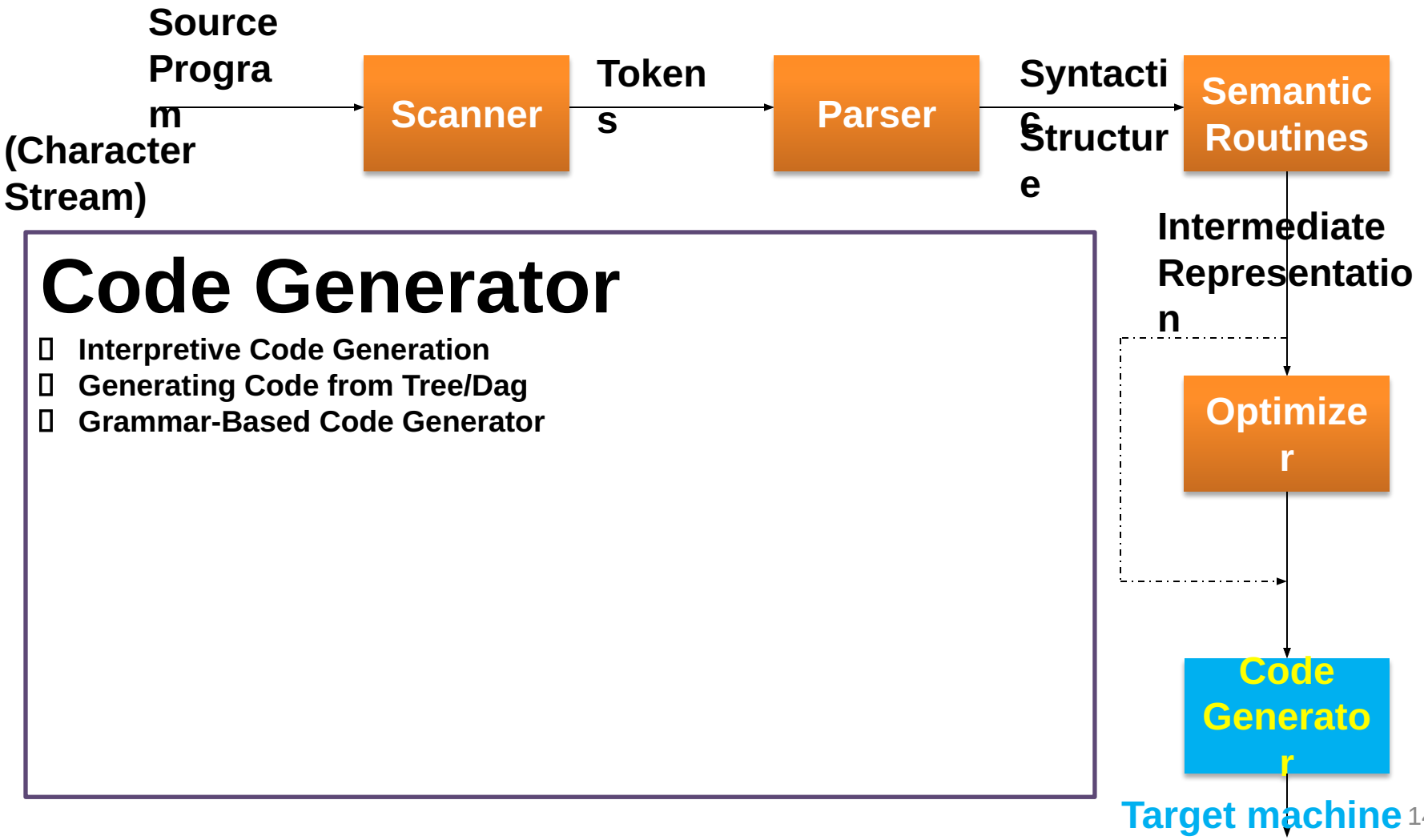


Optimizer

- The IR code generated by the semantic routines is analyzed and transformed into functionally equivalent but improved IR code
- This phase can be very complex and slow
- Peephole optimization
- loop optimization, register allocation, code scheduling

- **Register and Temporary Management**
- **Peephole Optimization**

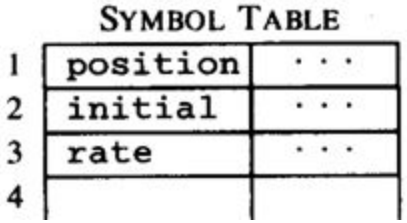
The Structure of a Compiler



Code Generator

- Interpretive Code Generation
- Generating Code from Tree/Dag
- Grammar-Based Code Generator

The Structure of a Compiler




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

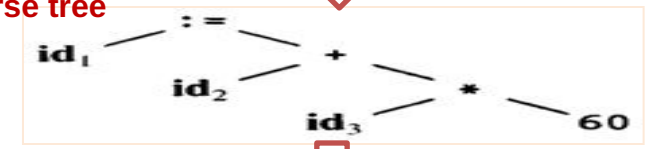
Scanner
[Lexical Analyzer]

Tokens

$id_1 := id_2 + id_3 * 60$

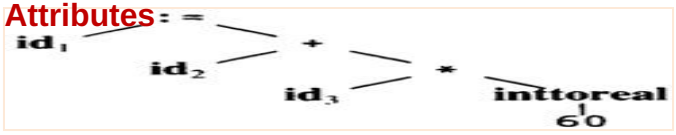
Parser
[Syntax Analyzer]

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 Generator

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

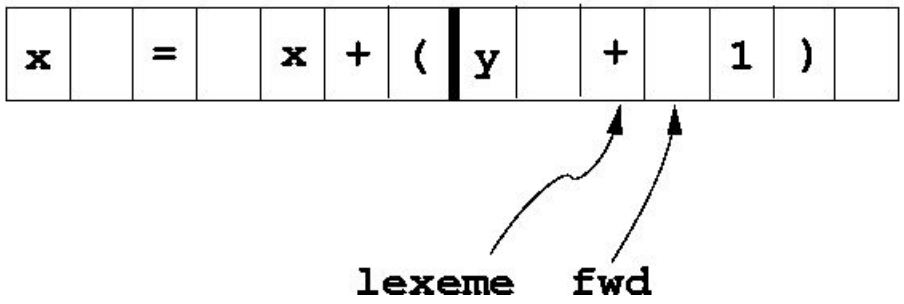
Lexical Analysis in Fortran

- Fortran rule: white space is insignificant
  - Example: "VAR1" is the same as "VA R1"
  - Left-to-right reading is not enough
    - DO 5 I = 1,25 ==> DO 5 I = 1 , 25
    - DO 5 I = 1.25 ==> DO5I = 1.25
  - Reading left-to-right cannot tell whether DO5I is a variable or a DO statement until "." or "," is reached
  - "Lookahead" may be needed to decide where a token ends and the next token begins
  - Even our simple example has lookahead issues
    - e.g, "=" and "=="

Input Buffering

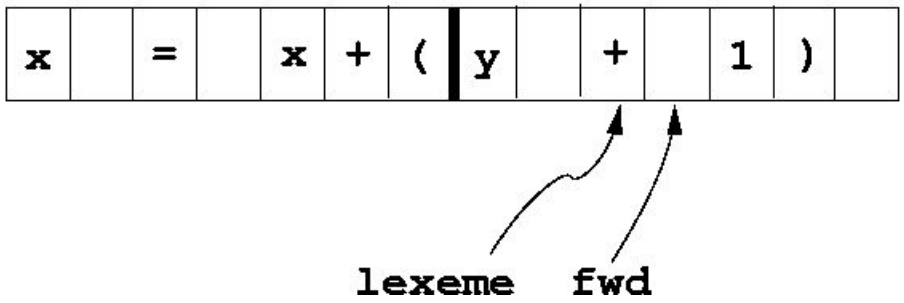
- Scanner performance is crucial:
  - This is the only part of the compiler that examines the entire input program one character at a time.
  - Disk input can be slow.
  - The scanner accounts for ~25-30% of total compile time.
- We need look ahead to determine when a match has been found.
- Scanners use *double-buffering* to minimize the overheads associated with this.

Buffer Pairs



- Use two *N*-byte buffers (*N* = size of a disk block; typically, *N* = 1024 or 4096).
- Read *N* bytes into one half of the buffer each time. If input has less than *N* bytes, put a special EOF marker in the buffer.
- When one buffer has been processed, read *N* bytes into the other buffer ("*circular buffers*").

Buffer pairs (cont'd)

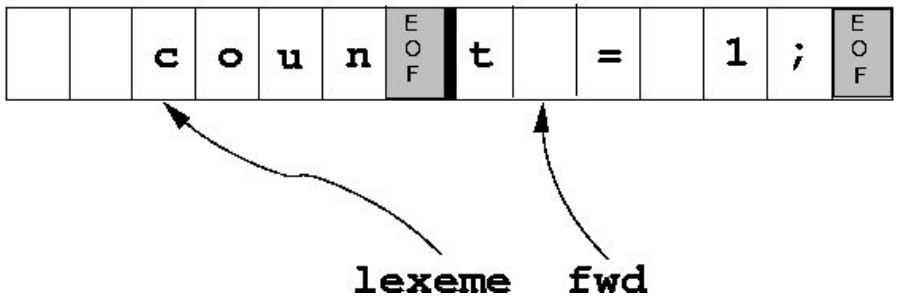


Code:

```
if (fwd at end of first half)
        reload second half;
        set fwd to point to beginning of second half;
else if (fwd at end of second half)
        reload first half;
        set fwd to point to beginning of first half;
else
        fwd++;
```

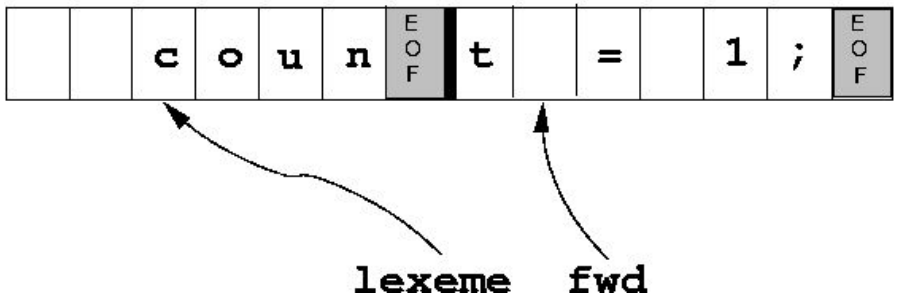
it takes two tests for each advance of the fwd pointer.

Buffer pairs: Sentinels



- *Objective*: Optimize the common case by reducing the number of tests to one per advance of fwd.
- *Idea*: Extend each buffer half to hold a *sentinel* at the end.
  - This is a special character that cannot occur in a program (e.g., EOF).
  - It signals the need for some special action (fill other buffer-half, or terminate processing).

Buffer pairs with sentinels (cont'd)



Code:

```
fwd++;
if ( *fwd == EOF ) {          /* special processing needed */
     if (fwd at end of first half)
          . . .
     else if (fwd at end of second half)
          . . .
     else      /* end of input */
          terminate processing.
}
```

common case now needs just a single test per character.

The Role of Lexical Analyzer

- Lexical analyzer is the first phase of a compiler.
- Its main task is to read input characters and produce a sequence of tokens as output that parser uses for syntax analysis.

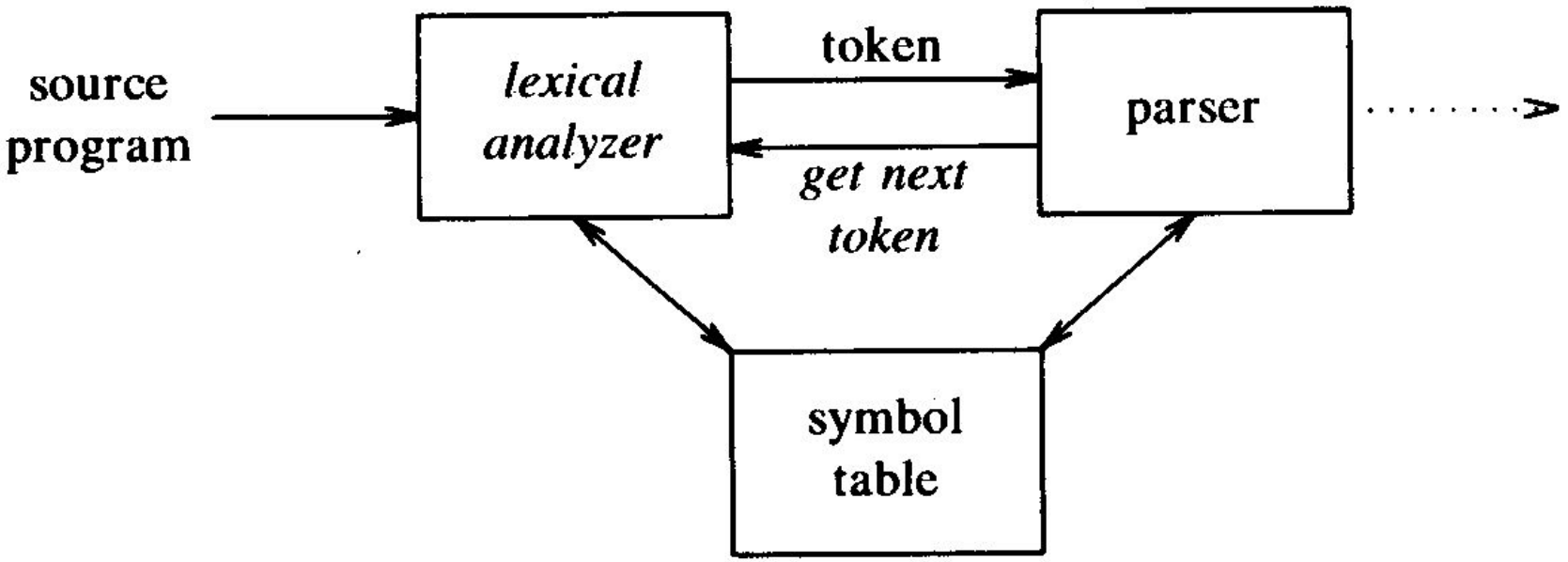


Fig. 3.1. Interaction of lexical analyzer with parser.

Issues in Lexical Analysis

- There are several reasons for separating the analysis phase of compiling into lexical analysis and parsing:
  - Simpler design
  - Compiler efficiency
  - Compiler portability
- Specialized tools have been designed to help automate the construction of lexical analyzer and parser when they are separated.

Tokens, Patterns, Lexemes

- A lexeme is a sequence of characters in the source program that is matched by the pattern for a token.
- A lexeme is a basic lexical unit of a language comprising one or several words, the elements of which do not separately convey the meaning of the whole.
- The lexemes of a programming language include its identifier, literals, operators, and special words.
- A token of a language is a category of its lexemes.
- A pattern is a rule describing the set of lexemes that can represent as particular token in source program.

Examples of Tokens

```
const pi = 3.1416;
```

The substring *"pi"* is a lexeme for the token "identifier".

TOKEN	SAMPLE LEXEMES	INFORMAL DESCRIPTION OF PATTERN
const	`const`	`const`
if	`if`	`if`
relation	`<, <=, =, <>, >, >=`	< or <= or = or <> or >= or >
id	`pi, count, D2`	letter followed by letters and digits
num	`3.1416, 0, 6.02E23`	any numeric constant
literal	`"core dumped"`	any characters between " and " except "

Lexeme and Token

```
Index = 2 * count +17;
```

Lexemes	Tokens
Index	Identifier
=	equal_sign
2	int_literal
*	multi_op
Count	identifier
+	plus_op
17	int_literal
;	semicolon

Lexical Errors

- Few errors are discernible at the lexical level alone, because a lexical analyzer has a very localized view of a source program.
- Let some other phase of compiler handle any error.
- Panic mode
- Error recovery

Specification of Tokens

- Regular expressions are an important notation for specifying patterns.
- Operation on languages
- Regular expressions
- Regular definitions
- Notational shorthands

Operations on Languages

OPERATION	DEFINITION
union of L and M written $L \cup M$	$L \cup M = \{ s \mid s \text{ is in } L \text{ or } s \text{ is in } M \}$
concatenation of L and M written LM	$LM = \{ st \mid s \text{ is in } L \text{ and } t \text{ is in } M \}$
Kleene closure of L written L^*	$L^* = \bigcup_{i=0}^{\infty} L^i$ L^* denotes "zero or more concatenations of" L.
positive closure of L written L^+	$L^+ = \bigcup_{i=1}^{\infty} L^i$ L^+ denotes "one or more concatenations of" L.

Regular Expressions

- Regular expression is a compact notation for describing string.
- In Pascal, an identifier is a letter followed by zero or more letter or digits
  $\rightarrow$letter(letter|digit)*
- | : or
- *: zero or more instance of
- $a(a|d)^*$

Rules

- ε is a regular expression that denotes {ε}, the set containing empty string.
- If *a* is a symbol in Σ, then *a* is a regular expression that denotes {*a*}, the set containing the string *a*.
- Suppose *r* and *s* are regular expressions denoting the language L(*r*) and L(*s*), then
  - (*r*) |(*s*) is a regular expression denoting L(*r*) ∪ L(*s*).
  - (*r*)(*s*) is regular expression denoting L (*r*) L(*s*).
  - (*r*) * is a regular expression denoting (L (*r*))*.
  - (*r*) is a regular expression denoting L (*r*).

Precedence Conventions

- The unary operator * has the highest precedence and is left associative.
- Concatenation has the second highest precedence and is left associative.
- | has the lowest precedence and is left associative.
- $(a)|(b)^*(c) \rightarrow a|b^*c$

Example of Regular Expressions

Example 3.3. Let $\Sigma = \{a, b\}$.

1. The regular expression $a|b$ denotes the set $\{a, b\}$.
2. The regular expression $(a|b)(a|b)$ denotes $\{aa, ab, ba, bb\}$, the set of all strings of a's and b's of length two. Another regular expression for this same set is $aa \mid ab \mid ba \mid bb$.
3. The regular expression a^* denotes the set of all strings of zero or more a's, i.e., $\{\epsilon, a, aa, aaa, \cdots\}$.
4. The regular expression $(a|b)^*$ denotes the set of all strings containing zero or more instances of an a or b, that is, the set of all strings of a's and b's. Another regular expression for this set is $(a^*b^*)^*$.
5. The regular expression $a \mid a^*b$ denotes the set containing the string a and all strings consisting of zero or more a's followed by a b. □

Properties of Regular Expression

AXIOM	DESCRIPTION
$r\|s = s\|r$	$\|$ is commutative
$r\|(s\|t) = (r\|s)\|t$	$\|$ is associative
$(rs)t = r(st)$	concatenation is associative
$r(s\|t) = rs\|rt$ $(s\|t)r = sr\|tr$	concatenation distributes over $\|$
$\epsilon r = r$ $r\epsilon = r$	ϵ is the identity element for concatenation
$r^* = (r\|\epsilon)^*$	relation between * and ϵ
$r^{**} = r^*$	* is idempotent

Regular Definitions

- If Σ is an alphabet of basic symbols, then a *regular definition* is a sequence of definitions of the form:

  $d_1 \rightarrow r_1$
  $d_2 \rightarrow r_2$

  …
  $d_n \rightarrow r_n$

- where each d_i is a distinct name, and each r_i is a regular expression over the symbols in $\Sigma \cup \{d_1, d_2, \ldots, d_{i-1}\}$, i.e., the basic symbols and the previously defined names.

Examples of Regular Definitions

Example 3.4. As we have stated, the set of Pascal identifiers is the set of strings of letters and digits beginning with a letter. Here is a regular definition for this set.

$$\begin{aligned} \textbf{letter} &\rightarrow \texttt{A} \mid \texttt{B} \mid \cdots \mid \texttt{Z} \mid \texttt{a} \mid \texttt{b} \mid \cdots \mid \texttt{z} \\ \textbf{digit} &\rightarrow \texttt{0} \mid \texttt{1} \mid \cdots \mid \texttt{9} \\ \textbf{id} &\rightarrow \textbf{letter} \; (\; \textbf{letter} \mid \textbf{digit} \;)^* \end{aligned}$$

□

Example 3.5. Unsigned numbers

$$\begin{aligned} \textbf{digit} &\rightarrow \texttt{0} \mid \texttt{1} \mid \cdots \mid \texttt{9} \\ \textbf{digits} &\rightarrow \textbf{digit} \; \textbf{digit}^* \\ \textbf{optional\_fraction} &\rightarrow \; . \; \textbf{digits} \mid \epsilon \\ \textbf{optional\_exponent} &\rightarrow (\; \texttt{E} \; (\; + \mid - \mid \epsilon \;) \; \textbf{digits} \;) \mid \epsilon \\ \textbf{num} &\rightarrow \textbf{digits} \;\; \textbf{optional\_fraction} \;\; \textbf{optional\_exponent} \end{aligned}$$

Notational Shorthands

Certain constructs occur so frequently in regular expressions that it is convenient to introduce notational shorthands for them.

1. *One or more instances.* The unary postfix operator $^+$ means "one or more instances of." If r is a regular expression that denotes the language $L(r)$, then $(r)^+$ is a regular expression that denotes the language $(L(r))^+$. Thus, the regular expression a^+ denotes the set of all strings of one or more a's. The operator $^+$ has the same precedence and associativity as the operator *. The two algebraic identities $r^* = r^+ | \epsilon$ and $r^+ = rr^*$ relate the Kleene and positive closure operators.

2. *Zero or one instance.* The unary postfix operator ? means "zero or one instance of." The notation $r?$ is a shorthand for $r|\epsilon$. If r is a regular expression, then $(r)?$ is a regular expression that denotes the language $L(r) \cup \{\epsilon\}$. For example, using the $^+$ and ? operators, we can rewrite the regular definition for **num** in Example 3.5 as

$$
\begin{aligned}
\textbf{digit} &\rightarrow \texttt{0} \mid \texttt{1} \mid \cdots \mid \texttt{9} \\
\textbf{digits} &\rightarrow \textbf{digit}^+ \\
\textbf{optional\_fraction} &\rightarrow (\ \texttt{.}\ \textbf{digits}\ )? \\
\textbf{optional\_exponent} &\rightarrow (\ \texttt{E}\ (\ \texttt{+} \mid \texttt{-}\ )?\ \textbf{digits}\ )? \\
\textbf{num} &\rightarrow \textbf{digits}\ \ \textbf{optional\_fraction}\ \ \textbf{optional\_exponent}
\end{aligned}
$$

3. *Character classes.* The notation [abc] where a, b, and c are alphabet symbols denotes the regular expression a | b | c. An abbreviated character class such as [a−z] denotes the regular expression a | b | · · · | z. Using character classes, we can describe identifiers as being strings generated by the regular expression

```
[A−Za−z][A−Za−z0−9]*
```

Finite Automata

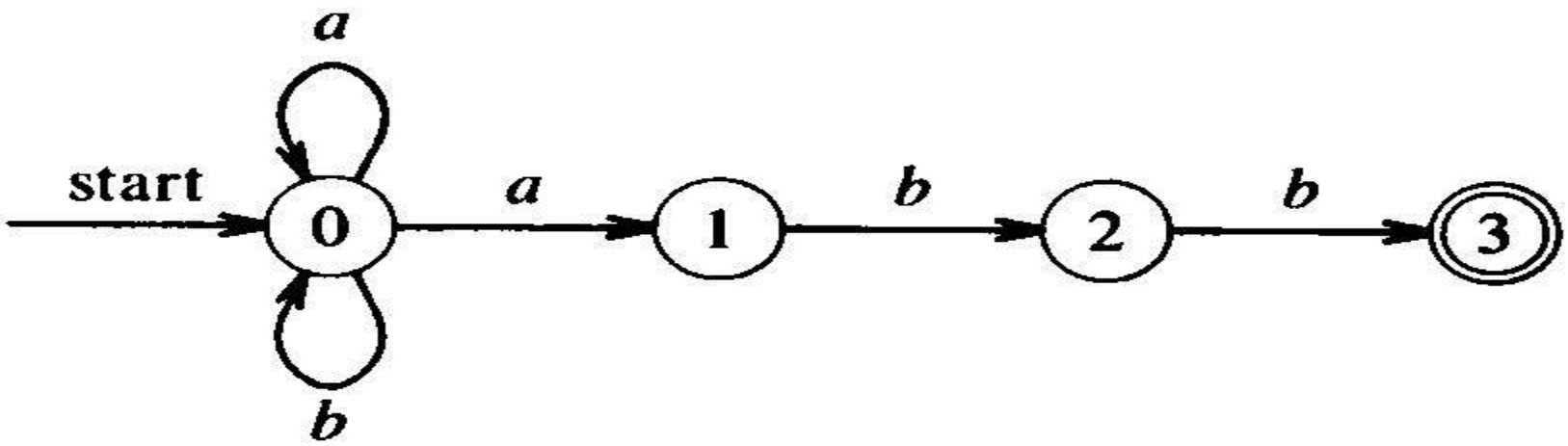
- A *recognizer* for a language is a program that takes as input a string *x* and answer “yes” if *x* is a sentence of the language and “no” otherwise.
- We compile a regular expression into a recognizer by constructing a generalized transition diagram called a *finite automaton*.
- A finite automaton can be *deterministic* or *nondeterministic*, where nondeterministic means that more than one transition out of a state may be possible on the same input symbol.

Nondeterministic Finite Automata (NFA)

- A set of states S
- A set of input symbols Σ
- A transition function *move* that maps state-symbol pairs to sets of states
- A state s_0 that is distinguished as the start *(initial) state*
- A set of states F distinguished as *accepting (final) states*.

NFA

- An NFA can be represented diagrammatically by a labeled directed graph, called a *transition graph*, in which the nodes are the states and the labeled edges represent the transition function.
- *(a|b)*abb*



A nondeterministic finite automaton.

NFA Transition Table

- The easiest implementation is a transition table in which there is a row for each state and a column for each input symbol and ε, if necessary.

STATE	INPUT SYMBOL	
	a	*b*
0	{0, 1}	{0}
1	–	{2}
2	–	{3}

Fig. 3.20. Transition table for the finite automaton of Fig. 3.19.

Example of NFA

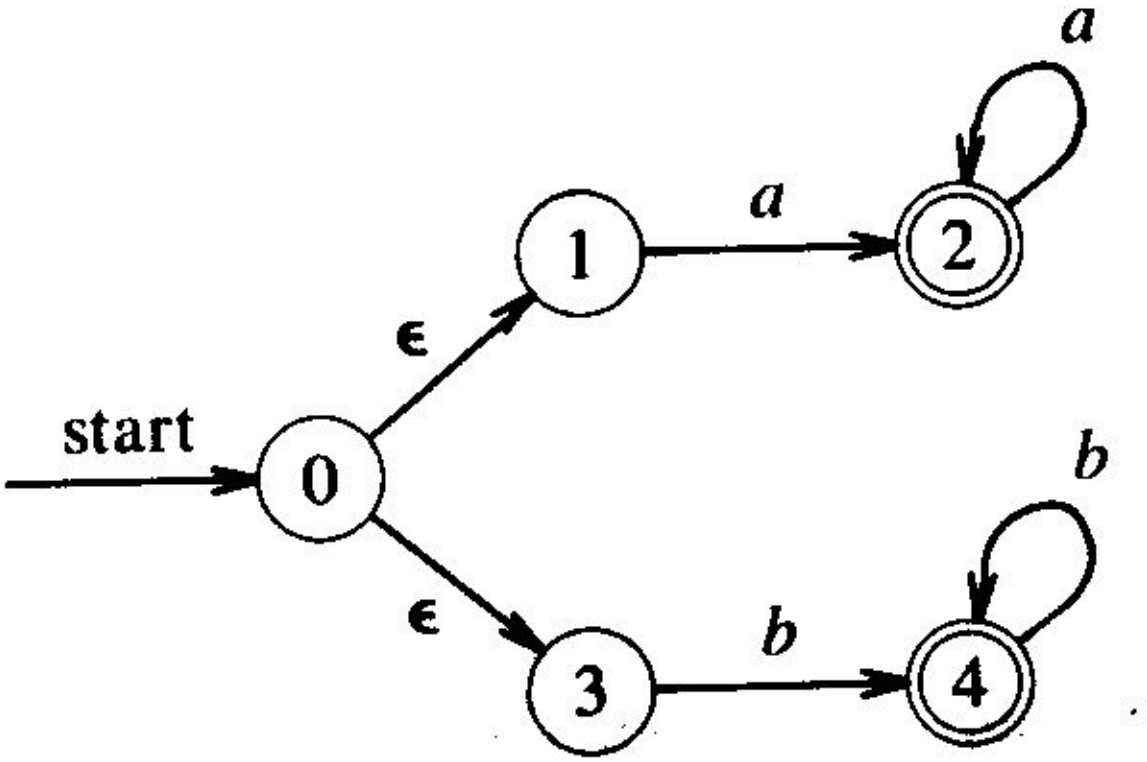


Fig. 3.21. NFA accepting $aa^* \mid bb^*$.

Deterministic Finite Automata (DFA)

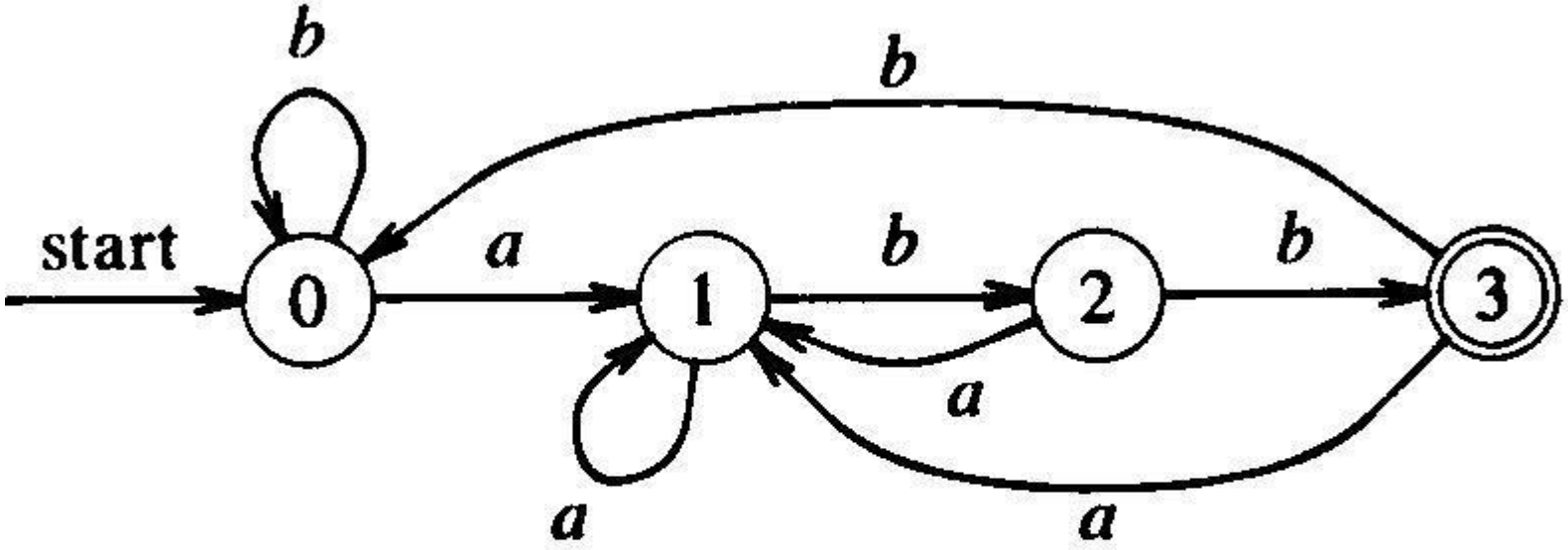
- A DFA is a special case of a NFA in which
  - no state has an ε-transition
  - for each state *s* and input symbol *a*, there is at most one edge labeled *a* leaving *s*.

Simulating a DFA

```
s := s0;
c := nextchar;
while c ≠ eof do
        s := move(s, c);
        c := nextchar
end;
if s is in F then
        return "yes"
else return "no";
```

Simulating a DFA.

Example of DFA



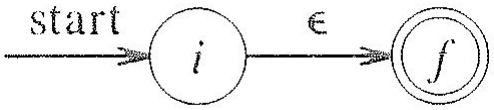
DFA accepting $(a|b)^*abb$

From Regular Expression to NFA

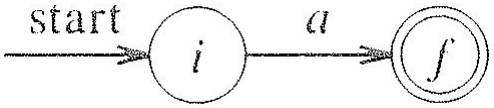
- Thompson's construction - an NFA from a regular expression
- Input: a regular expression *r* over an alphabet Σ.
- Output: an NFA *N* accepting *L(r)*

Method

- First parse *r* into its constituent subexpressions.
- Construct NFA's for each of the basic symbols in *r*.
  - for ε

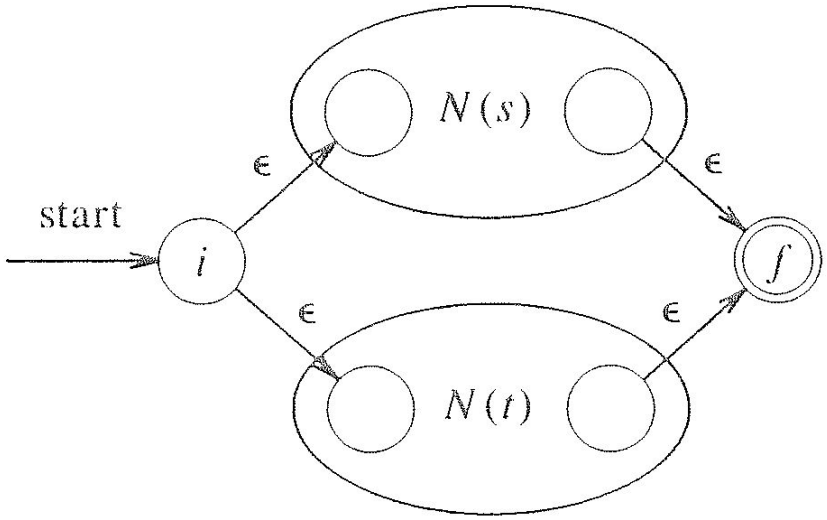


  - for *a* in Σ

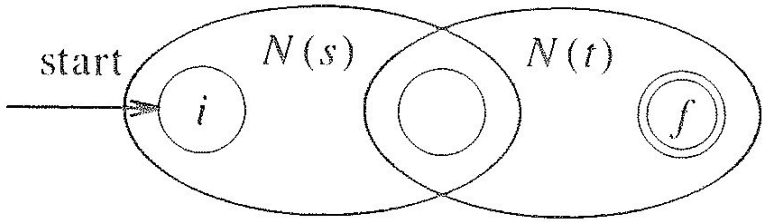


Method (II)

- For the regular expression $s|t$,

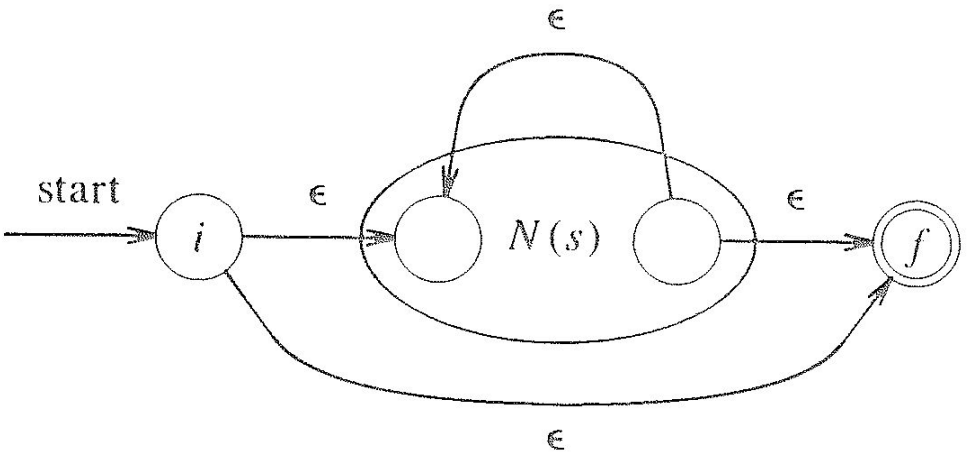


- For the regular expression st,



Method (III)

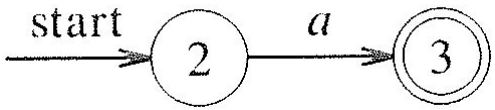
- For the regular expression *s**,

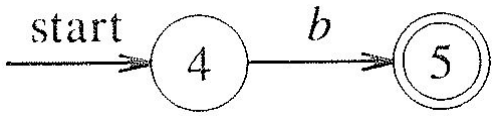


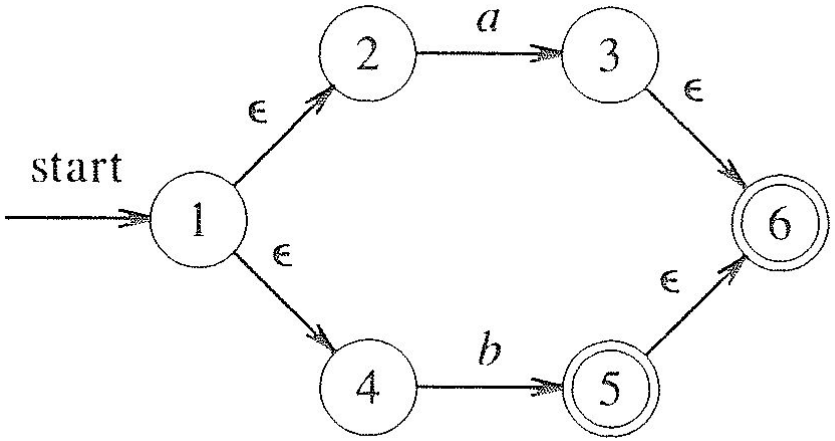
- For the parenthesized regular expression (*s*), use *N(s)* itself as the NFA.

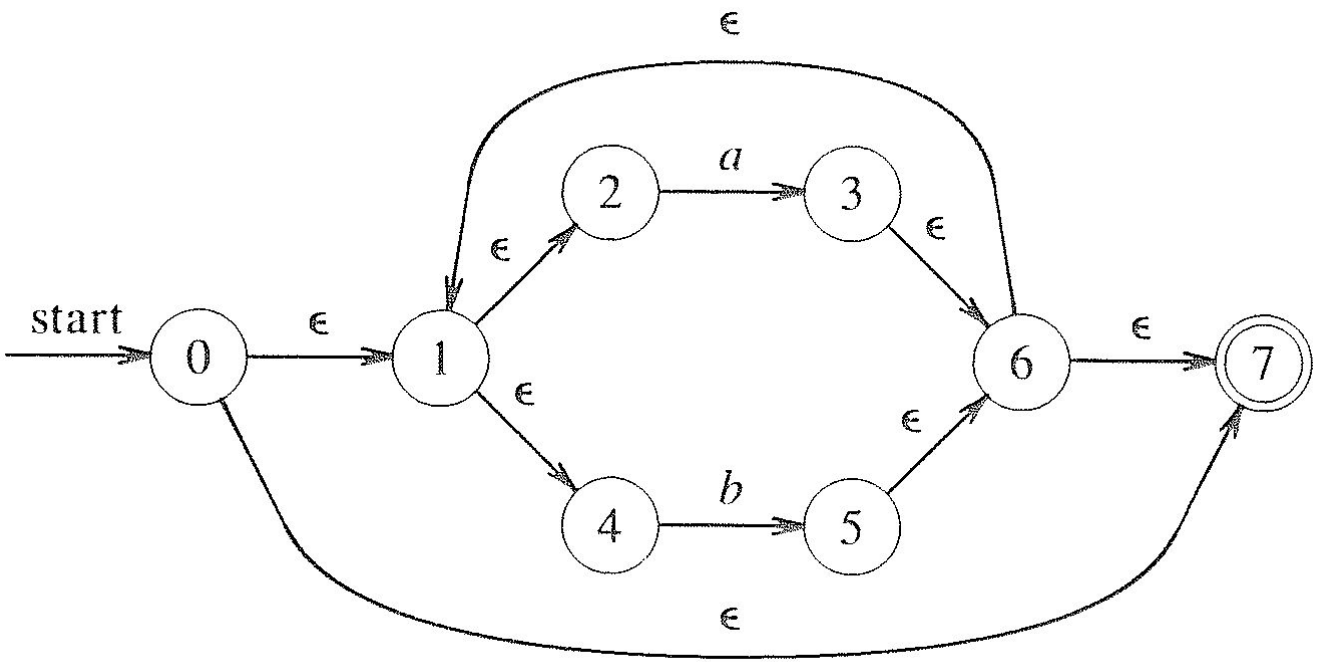
Every time we construct a new state, we give it a distinct name.

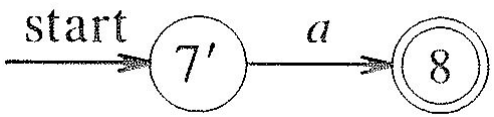
Example - construct *N(r)* for *r=(a|b)*abb*

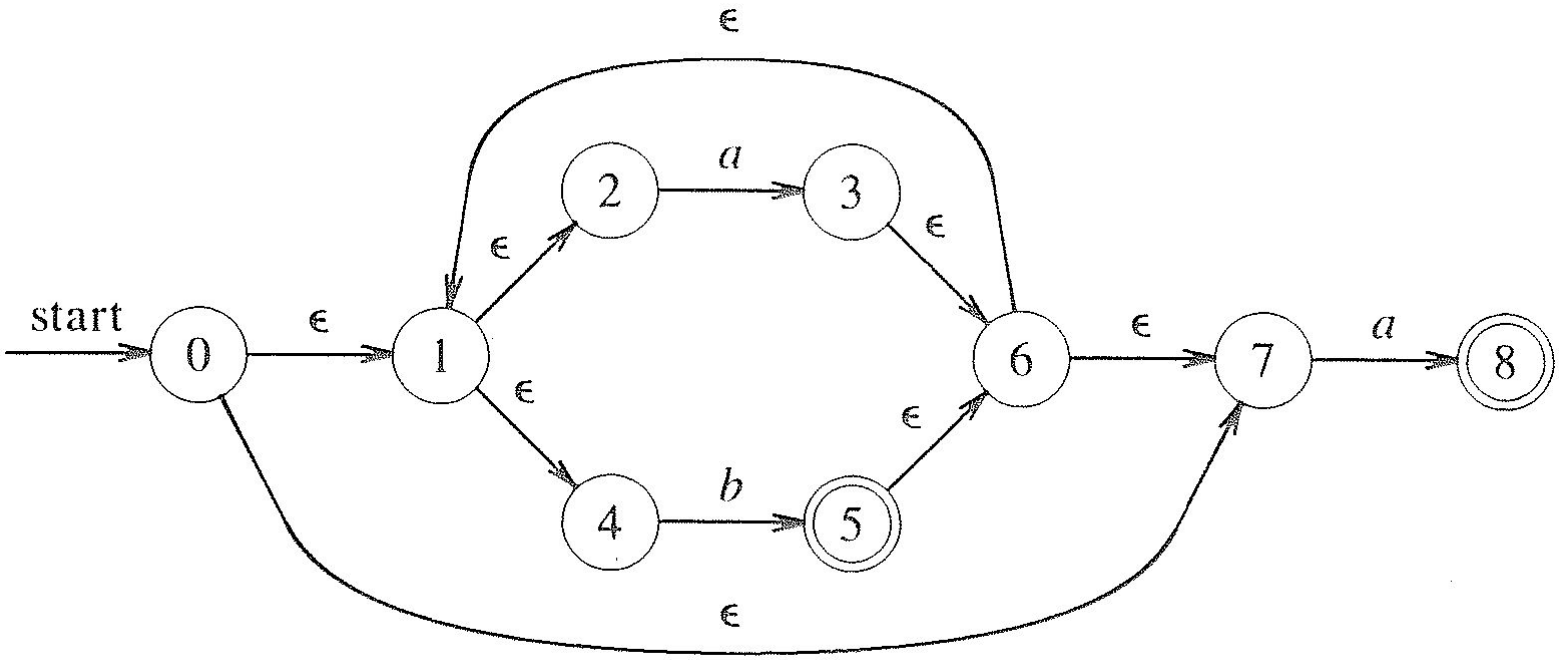




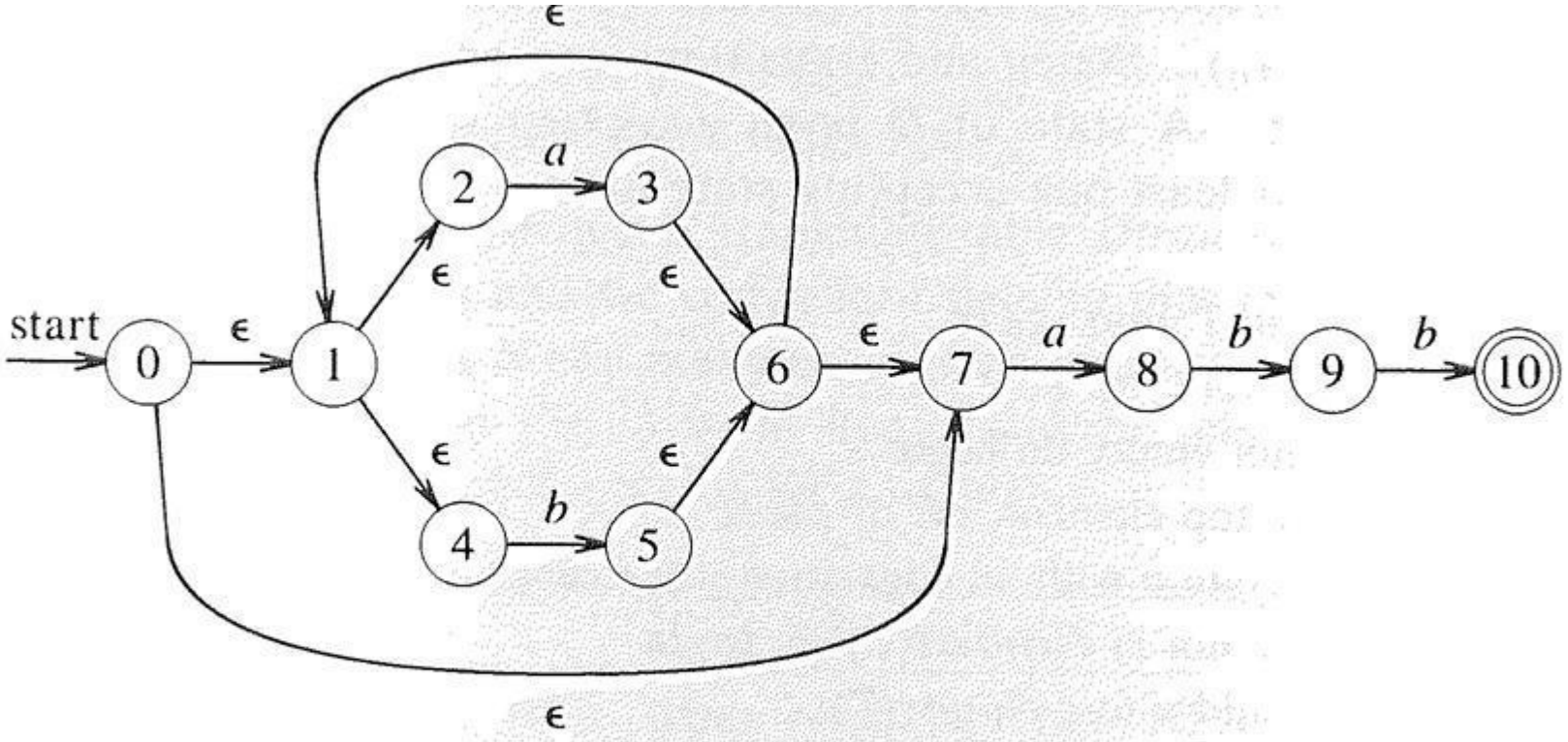



start
0
1
2
3
4
5
6
7
a
b
ϵ


start
7′
a
8


start
0
1
2
3
4
5
6
7
8
a
b
ϵ

Example (III)



NFA N for $(a|b)^*abb$.

Conversion of an NFA into DFA

- *Subset construction* algorithm is useful for simulating an NFA by a computer program.
- In the transition table of an NFA, each entry is a set of states; in the transition table of a DFA, each entry is just a single state.
- The general idea behind the NFA-to-DFA construction is that each DFA state corresponds to a set of NFA states.
- The DFA uses its state to keep track of all possible states the NFA can be in after reading each input symbol.

Subset Construction

- constructing a DFA from an NFA

- Input: An NFA N.
- Output: A DFA D accepting the same language.
- Method: We construct a transition table Dtran for D. Each DFA state is a set of NFA states and we construct Dtran so that D will simulate "in parallel" all possible moves N can make on a given input string.

Subset Construction (II)

s : represents an NFA state

T : represents a set of NFA states.

OPERATION	DESCRIPTION
ϵ-*closure*(s)	Set of NFA states reachable from NFA state s on ϵ-transitions alone.
ϵ-*closure*(T)	Set of NFA states reachable from some NFA state s in T on ϵ-transitions alone.
move(T, a)	Set of NFA states to which there is a transition on input symbol a from some NFA state s in T.

Operations on NFA states.

Subset Construction (III)

initially, ϵ-*closure*(s_0) is the only state in *Dstates* and it is unmarked;
while there is an unmarked state T in *Dstates* **do begin**
    mark T;
    for each input symbol a **do begin**
        $U := \epsilon$-*closure*(*move*(T, a));
        if U is not in *Dstates* **then**
            add U as an unmarked state to *Dstates*;
        Dtran [T, a] := U
    end
end

The subset construction

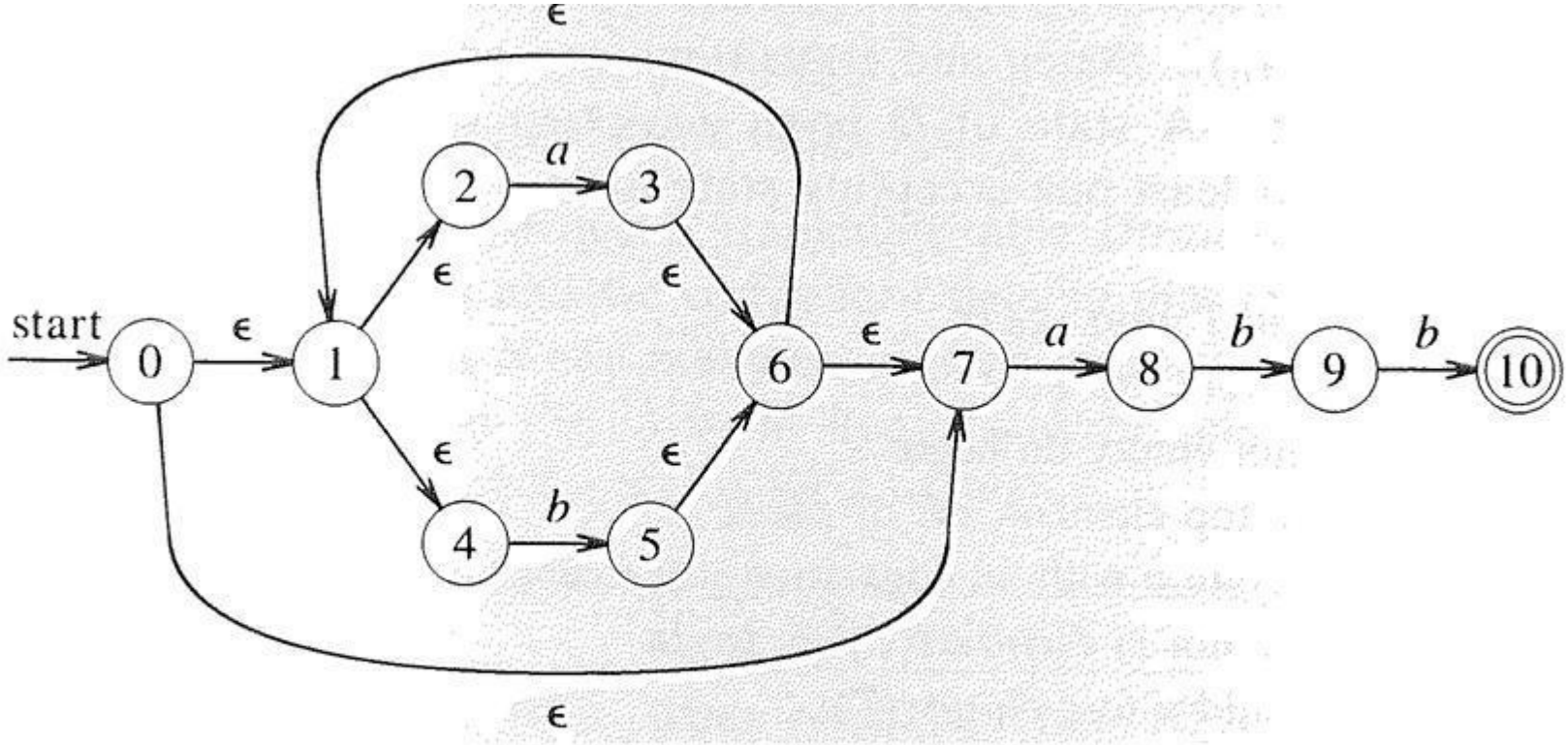
Subset Construction (IV)

(ε-closure computation)

```
push all states in T onto stack;
initialize ϵ-closure(T) to T;
while stack is not empty do begin
        pop t, the top element, off of stack;
        for each state u with an edge from t to u labeled ϵ do
            if u is not in ϵ-closure(T) do begin
                add u to ϵ-closure(T);
                push u onto stack
            end
end
```

Computation of ϵ-*closure*

Example



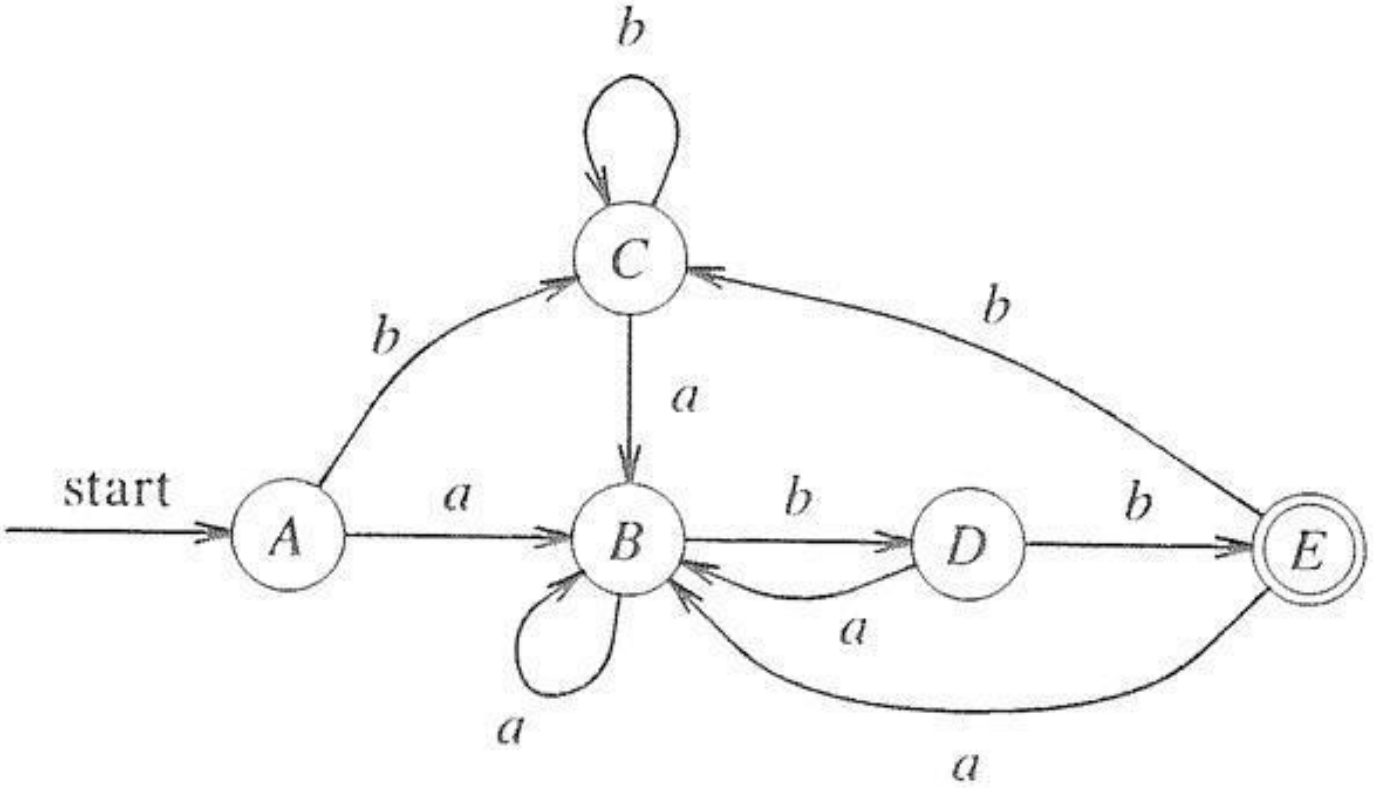
NFA N for $(a|b)^*abb$.

Example (II)

STATE	INPUT SYMBOL	
	a	*b*
A	*B*	*C*
B	*B*	*D*
C	*B*	*C*
D	*B*	*E*
E	*B*	*C*

Transition table *Dtran* for DFA

Example (III)



Minimizing the number of states in DFA

- Minimize the number of states of a DFA by finding all groups of states that can be distinguished by some input string.

- Each group of states that cannot be distinguished is then merged into a single state.

Minimizing the number of states in DFA (II)

Algorithm: Minimizing the number of states of a DFA

- Input. A DFA M with set of states S, set of inputs $\sum$, transitions defined for all states and inputs, start state s_0, and a set of accepting states F.
- Output. A DFA M' accepting the same language as M and having as few states as possible.
- Method.

  1. Construct an initial partition $\prod$ of the set of states with two groups: the accepting states F and non-accepting states $S - F$.
  2. Partition $\prod$ to $\prod_{new}$.
  3. If $\prod_{new} = \prod$, let $\prod_{final} = \prod$ and go to step (4). Otherwise, repeat step (2) with $\prod := \prod_{new}$.
  4. Choose one state in each group of the partition $\prod_{final}$ as the *representative* for that group.
  5. Remove dead states.

Construct New Partition

An example in class

for each group G of Π **do begin**
    partition G into subgroups such that two states s and t
        of G are in the same subgroup if and only if for all
        input symbols a, states s and t have transitions on a
        to states in the same group of Π;
    /* at worst, a state will be in a subgroup by itself */
    replace G in Π_{new} by the set of all subgroups formed
end

Fig. 3.45. Construction of Π_{new}.

Regular Expressions ⇒ Grammars

Rule #	Regular Expression Productions:	Grammar Productions:
R.1	$A \rightarrow xy$	$A \rightarrow xB$ $B \rightarrow y$
R.2	$A \rightarrow x^*y$	$A \rightarrow xB\|y$ $B \rightarrow xB\|y$
R.3	$A \rightarrow x\|y$	$A \rightarrow x$ $A \rightarrow y$
R.4	$A \rightarrow B$ $B \rightarrow x$	$A \rightarrow x$ $B \rightarrow x$
R.5	$A \rightarrow \in$ $B \rightarrow xA$	$B \rightarrow xA$ $B \rightarrow x$
R.6	$S \rightarrow \in$ {S *is goal symbol*}	$G \rightarrow S$ $G \rightarrow \in$

Table 3.3. Transforming regular expressions to regular grammars.

More in class …

Grammars ⇒ Regular Expressions

Rule #	Grammar Productions:		Regular Expression Productions:
R.1	$A \rightarrow xB$	$B \rightarrow y$	$A \rightarrow xy$
R.2	$A \rightarrow xA \mid y$		$A \rightarrow x^*y$
R.3	$A \rightarrow x$	$A \rightarrow y$	$A \rightarrow x \mid y$

Table 3.4. Transforming regular grammars to regular expressions

Automata ⇒ Grammars

A Language for Specifying Lexical Analyzer

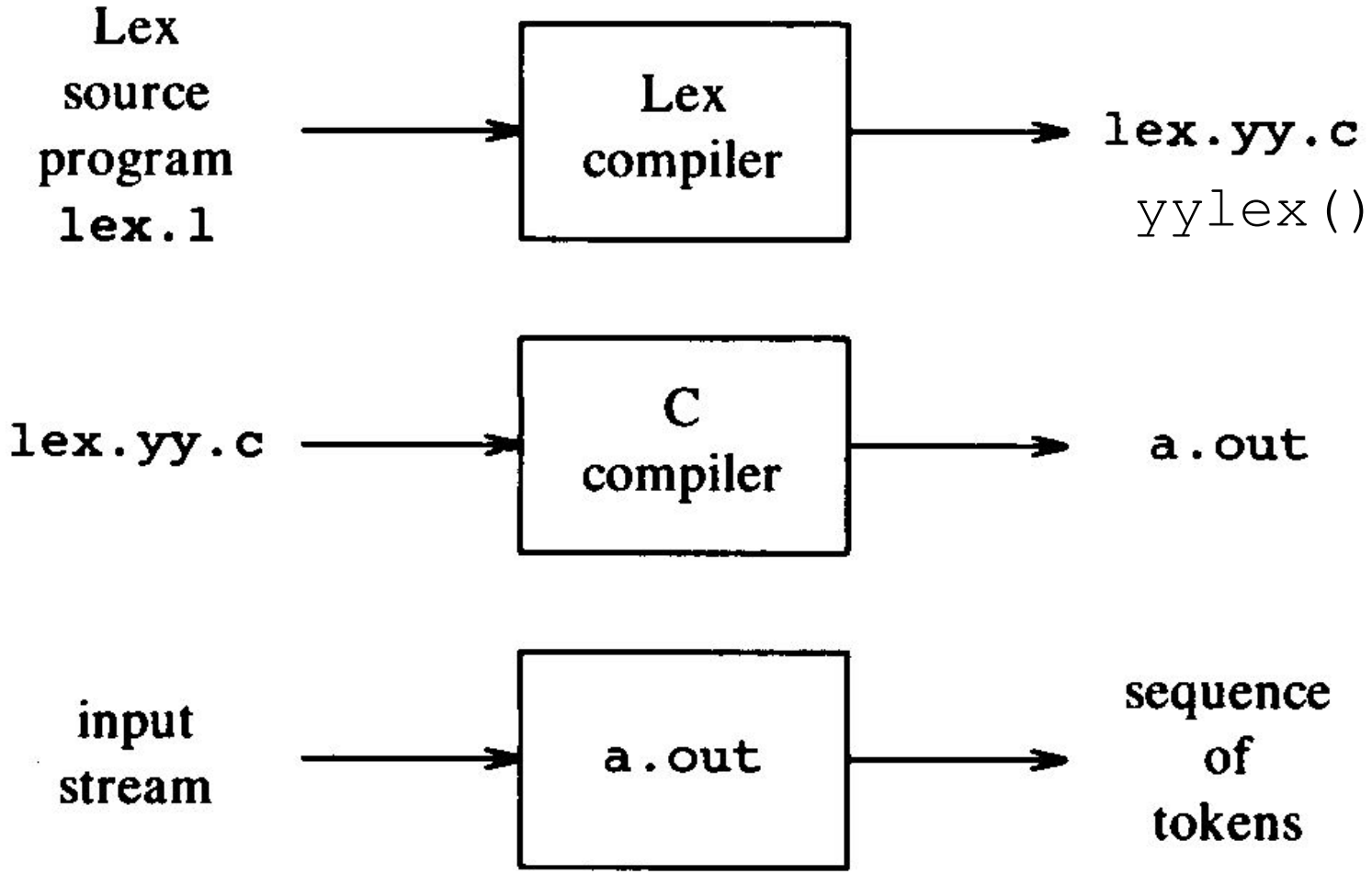


Fig. 3.17. Creating a lexical analyzer with Lex.

Simple Lex Example

```
        int num_lines = 0, num_chars = 0;
%%
\n      ++num_lines; ++num_chars;
.       ++num_chars;
%%
main()
{
        yylex();
        printf( "# of lines = %d,
                # of chars = %d\n",
                num_lines, num_chars );
}
```

```
%{
#include <math.h>  /* need this for the call to atof() below */
#include <stdio.h> /* need this for printf(), fopen() and stdin below */
%}
DIGIT    [0-9]
ID       [a-z][a-z0-9]*
%%
{DIGIT}+                    {
                            printf("An integer: %s (%d)\n", yytext,
                            atoi(yytext));
                            }
{DIGIT}+"."{DIGIT}*         {
                            printf("A float: %s (%g)\n", yytext,
                            atof(yytext));
                            }
if|then|begin|end|procedure|function        {
                            printf("A keyword: %s\n", yytext);
                            }
{ID}                        printf("An identifier: %s\n", yytext);
"+"|"-"|"*"|"/"             printf("An operator: %s\n", yytext);
"{"[^}\n]*"}"               /* eat up one-line comments */
[ \t\n]+                    /* eat up white space */
.                           printf("Unrecognized character: %s\n", yytext);
%%
int main(int argc, char *argv[]){
                            ++argv, --argc;  /* skip over program name */
                            if (argc > 0)
                            yyin = fopen(argv[0], "r");
                            else
                            yyin = stdin;
                            yylex();
}
```

AUTOMATIC SCANNER CONSTRUCTION

To convert a specification into code:

1. **Write down the RE for the input language**
2. **Build a big NFA**
3. **Build the DFA that simulates the NFA**
4. **Systematically shrink the DFA**
5. **Turn it into code**

Automatic Scanner Generators

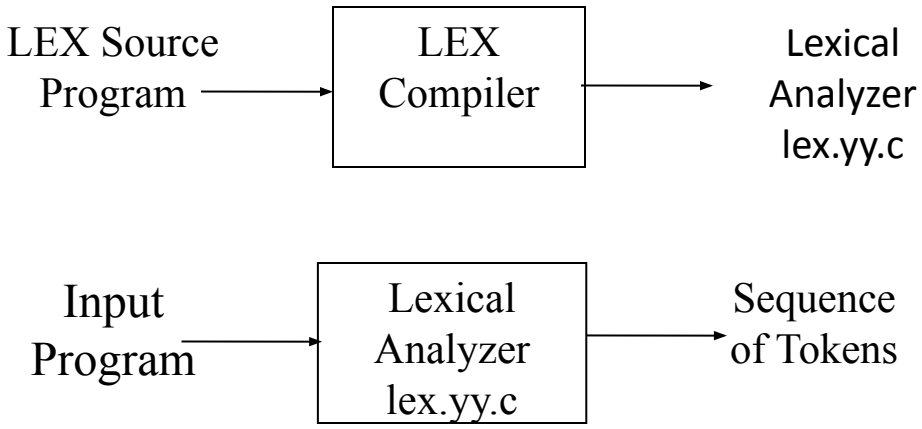
- **Which should work along these lines**
- **Algorithm are well known and well understood**
- **Key issue is interface to parser**
- **Easier to build.**

? How one can build a lexical analyzer from the specification of tokens present in the form of Regular Expression.

LEX a common tool to specify Lexical Analyzer

i.e. LEX is a Compiler which takes an input as a LEX Program and gives the output as Lexical Analyzer.

Pictorially represented as follows:



The format of the LEX input File

The format consists of Three Sections:

- **A collection of definitions**
- **A collection of Rules**
- **A collection of user subroutines**

These three sections are separated by **%%** (double percent sign) as shown below:

{definition}
%%

{rules}
%%

{user subroutines}

- **The first %% is compulsory as it represents the beginning of rules.**
- **The definitions and user subroutines are optional.**
- **The second %% is optional.**

The definition section

It may contain two things:

- **Any C code** that we want to be inserted external to any function (i.e. not as a part of any function) is required to be put in this section between delimiters %{ and %}.

- If we want to use **names for regular expressions**, then these names are also required to be defined in this section.

  - ✔ The definition of name should start on a separate line and in the first column.
  - ✔ The definition has format:

    name regular-expression

For Example: LETTER [a-z]

The rule section

This section contain the rules having the following format:

```
regular-expression    {
        Action in form of C code which will be
executed when regular expression is
matched
        }
```

The user subroutine section

- **The last section contains C code for any user defined subroutines that are called in the second section, and not defined elsewhere**
- **We can also put main function in this section, which will call yylex().**

If want to compile lex.yy.c as a standalone program.

Internal names used by the Lex:

Lex Internal Name	Use
lex.yy.c or lexyy.c	Name of the output file containing C source code generated by Lex.
yylex	Name of the Scanner routine generated by the Lex.
yytext	Name of the string used for holding the string matched by regular expression.

Example-1(Test1.l)

```
%%
```

This is an absolute minimum Lex Program

This generates a scanner that copies the input to the output unchanged.

Example-2(Test2.l)

- **This specifies a scanner that recognizes the keywords if and begin and an identifier, which is defined as any string that starts with a letter and followed by letters or digits.**

```
LETTER  [a-zA-Z]
DIGIT     [0-9]
%%
begin   {printf("Recognized KeyWORD: %s\n",yytext);}
if   {printf("Recognized KeyWORD: %s\n",yytext);}
{LETTER}({LETTER}|{DIGIT})*  {
          printf("Recognized ID: %s\n",yytext);}
%%
main()
{
  yylex();
}
```

To Execute the lex program we need to give the following sequence of commands

```
$lex Text2.l
$cc lex.yy.c –ll
$./a.out < data
```

Where, data is a file containing the input to be given to the scanner generated by Lex.

The content of data file is as follows:

```
if
begin
xyz
ABC
```

The output produced for the above input is:

Recognized KeyWORD: if

Recognized KeyWORD: begin

Recognized ID : xyz

Recognized ID : ABC

Example-3(Test3.l)

- **This specifies a scanner that recognizes the keywords if and begin and an identifier, which is defined as any string that starts with a letter and followed by letters or digits.**

```
LETTER    [a-zA-Z]
DIGIT     [0-9]
%%
{LETTER}({LETTER}|{DIGIT})*     {
      printf("Recognized ID: %s\n",yytext);}
begin   {printf("Recognized KeyWORD: %s\n",yytext);}
if   {printf("Recognized KeyWORD: %s\n",yytext);}
%%
main()
{
  yylex();
}
```

By giving the following sequence of commands:

```
$lex Text3.l

$cc lex.yy.c –ll

$./a.out < data
```

Where, data is a file containing the input to be given to the scanner generated by Lex.

The content of data file is as follows:

```
if  begin xyz ABC
```

The output produced for the above input is:

Recognized ID : if

Recognized ID : begin

Recognized ID : xyz

Recognized ID : ABC

Ambiguity Resolution

- **The scanner or lexical analyzer generated by Lex always first match the longest possible substring of the input to a rule.**
- **If the longest substring still matches two or more rules, then the scanner picks up the rule listed first in the rules section.**
- **If no rule matches any nonempty substring of input, then the default action is carried out which outputs the character or string not matching to the output.**

Example-4(Test4.l)

- This specifies a scanner that recognizes some of the keywords like begin, if, some of the operators, and identifiers, which is defined as any string that starts with a letter and followed by letters or digits and counts the number of identifiers, keywords and operators encountered in the input given to the scanner. The action taken by the scanner is to increment a counter named key when it recognizes a keyword, increment a counter op when it finds an operator, and increment a counter id, when it encounters an identifier.

The content of data1 is as follows:

```
if a + b then x + y;
else p / q;
while a <= b do
x = y + z;
```

```
%{
int key=0,op=0,id=0;
%}
LETTER    [a-zA-Z]
DIGIT       [0-9]
%%
(begin | if | while | do | then | else)   {key++;}
[-+*/<>=]    {op++;}
(<= | >= | != | ==)    {op++;}
[,;\.]      ;
{LETTER}({LETTER}|{DIGIT})*      { id++;}
%%
main()
{
yylex();
printf("Number of ID's = %d\n Number of KEYWORDS = %d\n
Number of Operator = %d\n", id,key,op);
}
```

The output:

Number of ID's = 11

Number of KEYWORDS = 5

Number of Operator = 6

Write a LEX program to generate a scanner which will take a decimal number between 1 to 999 in words and prints its numeric value as output.

Input:

one hundred ten

Output:

110

```
%{
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <ctype.h>
#define TRUE 1
#define FALSE 0
char *unitList[] =
{"ZERO","ONE","TWO","THREE","FOUR","FIVE","SIX","SEVEN","EIGHT","NINE",NULL};
char *tenList[]  =
{"TEN","TWENTY","THIRTY","FORTY","FIFTY","SIXTY","SEVENTY","EIGHTY",
"NINETY",NULL};
char *restList[] =     {"ELEVEN","TWELEVE","THIRTEEN","FOURTEEN",
"FIFTEEN","SIXTEEN","SEVENTEEN","EIGHTEEN","NINETEEN",NULL};
int unitNum = 0;
int isNonZeroHundred = FALSE;
int isHundred = FALSE;
int number = 0;
char reference[6];
```

```
int lookupList(char *data,char *list[])
{
        int i = 0;
        char *temp;
        while((temp = *(list + i++)) != NULL)
                if(!strcmp(data,temp))
                        return (i-1);
        return -1;
}
char *toUpperCase(char *string)
{
        int i = 0;
        char *temp = (char *)malloc(strlen(string)+1);
                while(string[i] != '\0')
                temp[i] = toupper(string[i++]);
        temp[i] = '\0';
        return temp;
}
%}
%%
```

```
(one|two|three|four|five|six|seven|eight|nine)
	{
		if( isNonZeroHundred == FALSE) isNonZeroHundred = TRUE;		unitNum
= lookupList(toUpperCase(yytext),unitList);		sprintf(reference,"%s",yytext);

		number+= unitNum;
	}
(hundred|Hundred)
	{
		if( isHundred == FALSE && isNonZeroHundred == TRUE)
		{
			 number*= 100;
			isHundred = TRUE;
		}
		else
		{
			if(isNonZeroHundred == FALSE)				printf("Error:No
number specified before hundred!\n");		if(isHundred == TRUE)
					printf("Error:\"hundred\" is used multiple
times!\n");
			exit(1);
		}
	}
```

```
(ten|twenty|thirty|forty|fifty|sixty|seventy|eighty|ninety)
        {
                if( isNonZeroHundred == TRUE && isHundred == FALSE)
                {
                        printf("Error:\"%s\" cannot precede \"%s\" without
\"hundred\"!\n",reference,yytext);                              exit(2);
                }
                number+= 10 * (lookupList(toUpperCase(yytext),tenList) + 1);   }

(eleven|twelve|thirteen|fourteen|fifteen|sixteen|seventeen|eighteen|nineteen)
        {
                if( isNonZeroHundred == TRUE && isHundred == FALSE)
                {
                        printf("Error:\"%s\" cannot precede \"%s\" without
        \"hundred\"!\n",reference,yytext);
                        exit(3);
                }
                number+= 10+(lookupList(toUpperCase(yytext),restList) + 1);
        }
```

```
%%
main()
{
        yylex();
        printf("Required Number:%d\n",number);
}
```

Write a LEX Program to Recognize a valid ‘C’ Program.

Write a LEX Program to count all comment statements (if any) from a given ‘C’ Program. Remove all identified comment statements and print the program without comment statements.

```
/* program name is lexp.l */
%{
/* program to recognize a c program */
int COMMENT=0;
%}
identifier [a-zA-Z][a-zA-Z0-9]*
%%
#.* { printf("\n%s is a PREPROCESSOR DIRECTIVE",yytext);}
int | float |char |double |while |for |do |if |break |continue |void |
switch |case |long |struct |const |typedef |return |else |goto
      {printf("\n\t%s is a KEYWORD",yytext);}
"/*" {COMMENT = 1;}
/* {printf("\n\n\t%s is a COMMENT\n",yytext);}*/
"*/" {COMMENT = 0;}
/* printf("\n\n\t%s is a COMMENT\n",yytext);}*/
{identifier}\(
```

```
{if(!COMMENT)printf("\n\nFUNCTION\n\t%s",yytext);}
\{ {if(!COMMENT) printf("\n BLOCK BEGINS");}
\} {if(!COMMENT) printf("\n BLOCK ENDS");}
{identifier}(\[[0-9]*\])? {if(!COMMENT) printf("\n %s IDENTIFIER",yytext);}
\".*\" {if(!COMMENT) printf("\n\t%s is a STRING",yytext);}
[0-9]+ {if(!COMMENT) printf("\n\t%s is a NUMBER",yytext);}
\)(\;)? {if(!COMMENT) printf("\n\t");ECHO;printf("\n");}
\( ECHO;
= {if(!COMMENT)printf("\n\t%s is an ASSIGNMENT OPERATOR",yytext);}
\<= |
\>= |
\< |
== |
\> {if(!COMMENT) printf("\n\t%s is a RELATIONAL OPERATOR",yytext);}
```

```
%%
int main(int argc, char **argv)
{
if (argc > 1)
{
FILE *file; file = fopen(argv[1],"r"); if(!file)
{
printf("could not open %s \n",argv[1]); exit(0);
}
yyin = file;
}
yylex();
printf("\n\n");
return 0;
} int yywrap()
{
return 0;
}
```

Context-Free Grammar

Definition

- Informally,
  - ✔ A CFG is a set of rules for deriving (or generating) strings (or sentences) in a language.

Example: CFG

⟨sentence⟩ -> ⟨noun-phrase⟩ ⟨verb-phrase⟩ (1)
⟨noun-phrase⟩ -> ⟨proper-noun⟩ (2)
⟨noun-phrase⟩ -> ⟨determiner⟩ ⟨common-noun⟩ (3)
⟨proper-noun⟩ -> John (4)
⟨proper-noun⟩ -> Jill(5)
⟨common-noun⟩ -> car (6)
⟨common-noun⟩ -> hamburger (7)
⟨determiner⟩ -> a (8)
⟨determiner⟩ -> the (9)
⟨verb-phrase⟩ -> ⟨verb⟩ ⟨adverb⟩ (10)
⟨verb-phrase⟩ -> <verb> (11)
⟨verb⟩ -> drives(12)
⟨verb⟩ -> eats (13)
⟨adverb⟩ -> slowly(14)
⟨adverb⟩ -> frequently (15)

Example: CFG

Find the sentence: "Jill drives frequently" by using the above grammar?

⟨sentence⟩ → ⟨noun-phrase⟩ ⟨verb-phrase⟩ by (1)

→ ⟨proper-noun⟩ ⟨verb-phrase⟩ by (2)

→ Jill ⟨verb-phrase⟩ by (5)

→ Jill ⟨verb⟩ ⟨adverb⟩ by (10)

→ Jill drives ⟨adverb⟩ by (12)

→ Jill drives frequently by (15)

Elements of CFG

- Informally a CFG consists of:
  - ✔ A set of replacement rules, each having a Left-Hand Side (LHS) and a Right-Hand Side (RHS).
  - ✔ Two types of symbols; variables and terminals.
  - ✔ LHS of each rule is a single variable (no terminals).
  - ✔ RHS of each rule is a string of zero or more variables and terminals.
  - ✔ A string consists of only terminals.

Formal Definition of CFG

A Context-Free Grammar (CFG) is a 4-tuple:

G = (V, T, P, S)

V — A finite set of variables or *non-terminals*

T — A finite set of *terminals* (V and T do not intersect)

P — A finite set of *productions*, each of the form A -> α, where A is in V and α is in (V U T)*

Note that α may be ε

S — A starting non-terminal (S is in V)

Example: CFG

G = ({S}, {0, 1}, P, S)

P:

1. S -> 0S1
2. S -> ε

Example Derivations:

S => 0S1
  => 01

S => 0S1
  => 00S11
  => 000S111
  => 000111

Note that G "generates" the language $\{0^k1^k \mid k>=0\}$

Example: CFG

$G = (\{A, B, C, S\}, \{a, b, c\}, P, S)$

P:

(1) S -> ABC
(2) A -> aA
(3) A -> ε
(4) B -> bB
(5) B -> ε
(6) C -> cC
(7) C -> ε

Example Derivations:

S => ABC
=> BC
=>C
=> ε

Example: CFG

Example Derivations:

S => ABC (1)
=> aABC (2)
=> aaABC (2)
=> aaBC (3)
=> aabBC (4)
=> aabC (5)
=> aabcC (6)
=> aabc (7)

G generates the language a*b*c*

Write a CFG for the following languages:

L = {w | w ∈ 0(10)*}

S -> 0A, A→10A | ∈ S -> S10 | 0

L = {w | w ∈ (a + b)*}

S -> aS | bS | ∈

L = {w | w is a string of alternative sequence of a & b}

S → aV | bU
V → bU | b
U → aV | a

Write a CFG for the following languages:

L = {w | w is a even palindrome over the alphabet (0, 1)}

S -> 0S0 | 1S1 | ∈

L = {w | w is a odd palindrome over the alphabet (0, 1)}

S -> 0S0 | 1S1 | 0 | 1

L = {w | w is a palindrome over the alphabet (0, 1)}

S -> 0S0 | 1S1 | 0 | 1 |∈

Write a CFG for the following languages:

$L = \{a^n b^n c^m d^m \mid n, m \geq 0\}$

$S \rightarrow AB,\ A \rightarrow aAb \mid \in,\ B \rightarrow cBd \mid \in$

$L = \{a^n b^m c^m d^n \mid n, m \geq 0\}$

$S \rightarrow aSd \mid A,\ A \rightarrow bAc \mid \in$

$L = \{a^i b^j c^k \mid i \neq j \text{ or } j \neq k\}$

$X \rightarrow AB$
$A \rightarrow aAb \mid C$
$C \rightarrow aD \mid bE$
$D \rightarrow aD \mid \in$
$E \rightarrow bE \mid \in$
$B \rightarrow cB \mid \in$

$Y \rightarrow PQ$
$P \rightarrow aP \mid \in$
$Q \rightarrow bQc \mid M$
$M \rightarrow bN \mid cO$
$N \rightarrow bN \mid \in$
$O \rightarrow cO \mid \in$

$S \rightarrow X \mid Y$

Write a CFG for the following languages:

$L = \{a^n b^m \mid n > m\}$

$S \rightarrow AB,\ A \rightarrow aA \mid a,\ B \rightarrow aBb \mid \in$

$L = \{a^n b^m \mid n \neq m\}$

$S \rightarrow X \mid Y$

$X \rightarrow AB$

$A \rightarrow aA \mid a$

$B \rightarrow aBb \mid \in$

$Y \rightarrow CD$

$C \rightarrow aCb \mid \in$

$D \rightarrow bC \mid b$

Write a CFG for the following languages:

L = {w | w is a string of {a, b} where w do not contain 3 consecutive b's}

$$S \rightarrow aS \mid a \mid b \mid bT$$
$$T \rightarrow bU \mid a \mid b \mid aS$$
$$U \rightarrow aS \mid a$$

L = {equal number of a's & b's}

$$S \rightarrow aSbS \mid bSaS \mid \in$$

Derivation

v is **one-step derivable** from u, written u ⇒ v, if:

- u = xαz
- v = xβz
- α → β in P

v is **derivable** from u, written u ⇒* v, if:

There is a chain of one-derivations of the form:

$$u \Rightarrow u_1 \Rightarrow u_2 \Rightarrow \ldots \Rightarrow v$$

Context-Free Languages

Definition. Given a context-free grammar $G = (V, T, P, S)$, the **language generated** or derived from G is the set:

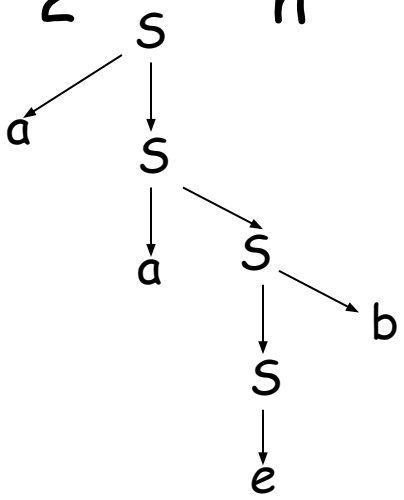
$$L(G) = \{w : \ S \Rightarrow^* w \}$$

Definition. A language L is context-free if there is a context-free grammar $G = (V, T, P, S)$, such that L is generated from G

Parse Tree

A parse tree of a derivation is a tree in which:

- Each internal node is labeled with a non-terminal or variable.
- If a rule $A \rightarrow A_1A_2 \ldots A_n$ occurs in the derivation then A is a parent node of nodes labeled $A_1, A_2, \ldots, A_n$



Parse Tree

$S \rightarrow A \mid A\ B$
$A \rightarrow \varepsilon \mid \mathbf{a} \mid A\ \mathbf{b} \mid A\ A$
$B \rightarrow \mathbf{b} \mid \mathbf{b}\ \mathbf{c} \mid B\ \mathbf{c} \mid \mathbf{b}\ B$

Sample derivations:

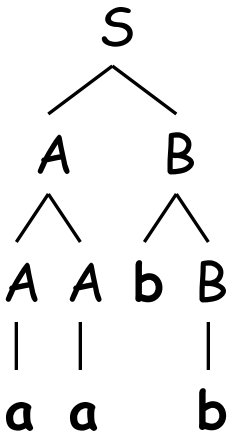
$S \Rightarrow AB \Rightarrow AAB \Rightarrow \mathbf{a}AB \Rightarrow \mathbf{aa}B \Rightarrow \mathbf{aab}B \Rightarrow \mathbf{aabb}$

$S \Rightarrow AB \Rightarrow A\mathbf{b}B \Rightarrow A\mathbf{bb} \Rightarrow AA\mathbf{bb} \Rightarrow A\mathbf{abb} \Rightarrow \mathbf{aabb}$

These two derivations use same productions, but in different orders.

This ordering difference is often uninteresting.

Derivation trees give way to abstract away ordering differences.



Root label = start node.

Each interior label = variable.

Each parent/child relation = derivation step.

Each leaf label = terminal or ε.

All leaf labels together = derived string = *yield*.

Leftmost, Rightmost Derivations

A **left-most derivation** of a sentential form is one in which rules transforming the left-most non-terminal are always applied

A **right-most derivation** of a sentential form is one in which rules transforming the right-most non-terminal are always applied

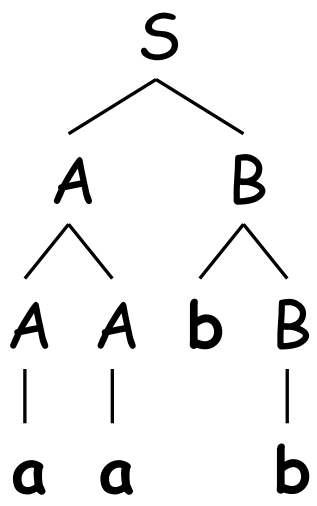
Leftmost, Rightmost Derivations

$S \rightarrow A \mid A\ B$
$A \rightarrow \varepsilon \mid \mathbf{a} \mid A\ \mathbf{b} \mid A\ A$
$B \rightarrow \mathbf{b} \mid \mathbf{b}\ \mathbf{c} \mid B\ \mathbf{c} \mid \mathbf{b}\ B$

Sample derivations:

$S \Rightarrow AB \Rightarrow AAB \Rightarrow \mathbf{a}AB \Rightarrow \mathbf{aa}B \Rightarrow \mathbf{aab}B \Rightarrow \mathbf{aabb}$

$S \Rightarrow AB \Rightarrow A\mathbf{b}B \Rightarrow A\mathbf{bb} \Rightarrow AA\mathbf{bb} \Rightarrow A\mathbf{abb} \Rightarrow \mathbf{aabb}$



These two derivations are special.

1st derivation is *leftmost.* Always picks leftmost variable.

2nd derivation is *rightmost.* Always picks rightmost variable.

Derivation Trees

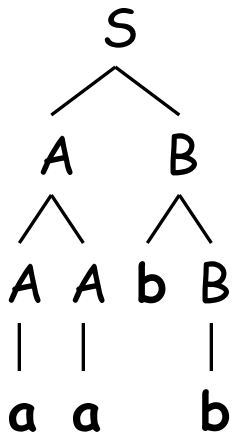
$S \rightarrow A \mid A\ B$
$A \rightarrow \varepsilon \mid \mathbf{a} \mid A\ \mathbf{b} \mid A\ A$
$B \rightarrow \mathbf{b} \mid \mathbf{b}\ \mathbf{c} \mid B\ \mathbf{c} \mid \mathbf{b}\ B$

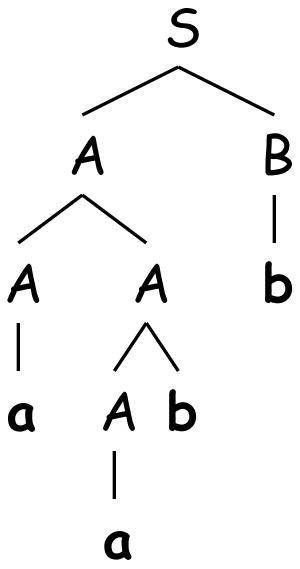
w = **aabb**

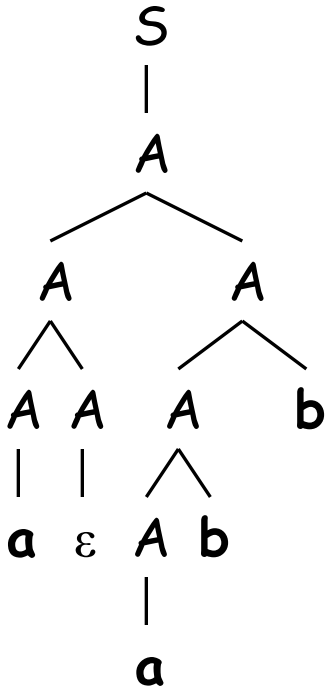
Other derivation trees for this string?

?

?







Infinitely many others possible.

Ambiguous Grammar

A grammar G is ambiguous if there is a word w ∈ L(G) having least two different parse trees

Consider a grammar G is for representing a basic arithmetic expression.

1. E→E + E
2. E→E - E
3. E→E * E
4. E→E / E
5. E→E ↑ E
6. E→ -E
7. E→(E)
8. E→ id

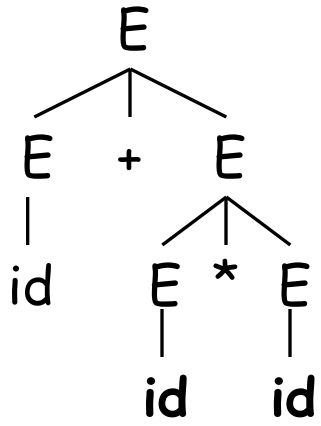
E →E + E	E →E * E
→id + E	→E + E * E
→id + E * E	→id + E * E
→id + id * E	→id + id * E
→id + id * id	→id + id * id

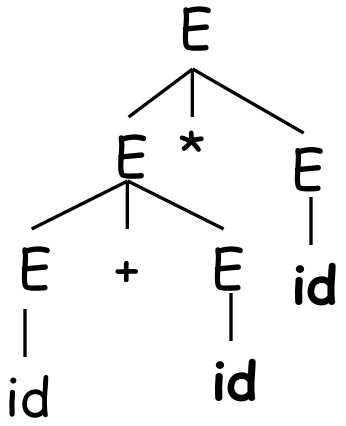
Notice that id + id * id has at least two left-most derivations

Ambiguous Grammar

E → E + E
→ id + E
→ id + E * E
→ id + id * E
→ id + id * id

E → E * E
→ E + E * E
→ id + E * E
→ id + id * E
→ id + id * id





Notice that id + id * id has two parse trees.

Ambiguity Conditions

CFG *ambiguous* ⇔ any of following equivalent statements:

- ∃ string w with multiple derivation trees.
- ∃ string w with multiple leftmost derivations.
- ∃ string w with multiple rightmost derivations.

Defining ambiguity of grammar, not language.

Ambiguity & Disambiguation

Given an ambiguous grammar, would like an equivalent unambiguous grammar.

- Allows you to know more about structure of a given derivation.
- Simplifies inductive proofs on derivations.
- Can lead to more efficient parsing algorithms.
- In programming languages, want to impose a canonical structure on derivations. E.g., for **1+2×3**.

Strategy: Force an ordering on all derivations.

Disambiguation: Example 1

1. E → E + E
2. E → E - E
3. E → E * E
4. E → E / E
5. E → E ↑ E
6. E → -E
7. E → (E)
8. E → id

Uses:

operator precedence
left- associativity or right associativity

Disambiguation: Example 1

operator precedence:

+, -

*, /

↑

-(Unary)

left- associative:

+, -, *, /

right associativity

↑

-(Unary)

Disambiguation: Example 1

P → (E) | id

O → -O | P

N → O ↑ N | O

M → M * N | M / N | N

E → E + M | E - M | M

1. E → E + E
2. E → E - E
3. E → E * E
4. E → E / E
5. E → E ↑ E
6. E → -E
7. E → (E)
8. E → id

Disambiguation: Example 2

⟨if_statement⟩ → if ⟨cond^n⟩ then ⟨if_statement⟩
| if ⟨cond^n⟩ then ⟨if_statement⟩
else ⟨if_statement⟩
| S_1 | S_2 | ... | S_n

⟨cond^n⟩ → C_1 | C_2 | ... | C_n

Disambiguation: Example 2

Solution:

⟨if_statement⟩ → ⟨MI⟩ | ⟨UMI⟩

⟨MI⟩ → if ⟨condⁿ⟩ then ⟨MI⟩ else ⟨MI⟩
| S_1|S_2| … |S_n

⟨UMI⟩ → if ⟨condⁿ⟩ then ⟨if_statement⟩
| if ⟨condⁿ⟩ then ⟨MI⟩ else ⟨UMI⟩

⟨condⁿ⟩ → C_1|C_2| … |C_n

Inherent Ambiguity

A Context-free Language L is said to be inherently ambiguous if all its grammars are ambiguous.

If even one of the grammar for L is unambiguous, then L is an unambiguous language.

Consider the following example:

$L = \{a^n b^n c^m d^m \mid n \geq 1, m \geq 1\} \cup \{a^n b^m c^m d^n \mid n \geq 1, m \geq 1\}$

The possible CFG:

S → AB | C
A → aAb | ab
B → cBd | cd
C → aCd | aDd
D → bDc | bc

Properties of Context-Free Languages

- Normal forms for Context-Free Grammar
- Closure Properties of Context-Free Language
- The pumping lemma for CFL
- Decision properties of CFL's

Normal forms for Context-Free Grammar

- Chomsky Normal Form.

  - Eliminating Useless symbols
  - Eliminating ∈-Productions
  - Eliminate Unit Productions

Eliminating useless symbols

✔ To Find the useless symbols in the grammar we need to perform two tests as stated below **"to find all the useful symbols of the grammar which can generate the resultant string"**.

- Reachability
- Generating

Eliminating useless symbols

Reachability:

A symbol X is *reachable* if there exists:

$$S \Rightarrow^* \alpha X \beta$$

Generating:

A symbol X is *generating* if there exists:

$X \Rightarrow^* w$, for some $w \in T^*$

Eliminating useless symbols

- For a symbol X to be "useful", it has to be both reachable *and* generating
  - ✔ $S \Rightarrow^* \alpha X \beta \Rightarrow^* w'$, for some $w' \in T^*$

Algorithm to detect useless symbols

1. First, eliminate all symbols that are *not* generating.

2. Next, eliminate all symbols that are *not* reachable.

Is the order of these steps important, or can we switch?

Example: Useless symbols

- S□AB | a
- A□ b

1. *A, S* are generating
2. *B* is *not generating* (and therefore B is useless)
3. ==> Eliminating B… (i.e., remove all productions that involve B)
   1. S□ a
   2. A □ b
4. Now, *A* is *not reachable* and therefore is useless

5. Simplified *G*:
   1. S □ a

What would happen if you reverse the order:
i.e., test reachability before generating?

Will fail to remove:
A □ b

Algorithm to find all generating symbols

$X \Rightarrow^* w$

- Given: G=(V,T,P,S)
- Basis:
  - Every symbol in T is obviously generating.
- Induction:
  - Suppose for a production $A \rightarrow \alpha$, where α is generating
  - Then, A is also generating

Algorithm to find all reachable symbols

$S \Rightarrow^* \alpha X \beta$

- Given: G=(V,T,P,S)
- Basis:
  - S is obviously reachable (from itself)
- Induction:
  - Suppose for a production $A \Rightarrow a_1 a_2 \ldots a_k$, where A is reachable
  - Then, all symbols on the right hand side, $\{a_1, a_2, \ldots a_k\}$ are also reachable.

Eliminating ε-productions

Caveat: It is *not* possible to eliminate ε-productions for languages which include ε in their word set.

So we will target the grammar for the *rest* of the language.

Theorem: If G=(V,T,P,S) is a CFG for a language L, then L-{ε} has a CFG without ε-productions.

Definition: *A is "nullable" if* $A \Rightarrow^* \varepsilon$

- If A is nullable, then any production of the form "B CAD" can be simulated by:
  - B CD | CAD
    - This can allow us to remove ε transitions for A.

Algorithm to detect all nullable variables

- Basis:
  - If $A \rightarrow \varepsilon$ is a production in G, then A is nullable. (note: A can still have other productions)

- Induction:
  - If there is a production $B \rightarrow C_1C_2\ldots C_k$, where *every* C_i is nullable, then B is also nullable.

Eliminating ε-productions

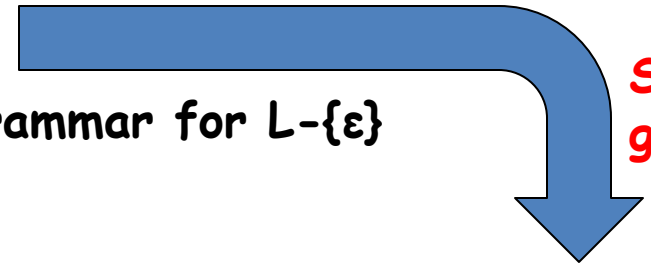
Given: G=(V,T,P,S)

Algorithm:

1. Detect all nullable variables in G
2. Then construct $G_1=(V,T,P_1,S)$ as follows:
   i. For each production of the form: $A \rightarrow X_1X_2\ldots X_k$, where k≥1, suppose ***m*** out of the ***k*** X_i's are nullable symbols
   ii. Then G_1 will have 2^m versions for this production
      i. i.e, all combinations where each X_i is either present or absent
   iii. Alternatively, if a production is of the form: $A \rightarrow \varepsilon$, then remove it

Example: Eliminating ε-productions

- Let L be the language represented by the following CFG G:
  i. S□AB
  ii. A□aAA | ε
  iii. B□bBB | ε



Goal: To construct G1, which is the grammar for L-{ε}

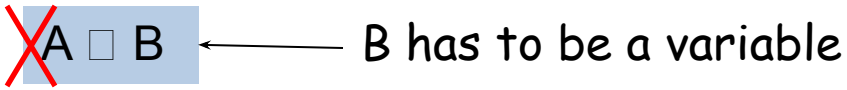
- Nullable symbols: {A, B}
- G_1 can be constructed from G as follows:
  - B □ b | bB | bB | bBB
    - ==>B □ b | bB | bBB
  - Similarly, A □ a | aA | aAA
  - Similarly, S □ A | B | AB
- Note: L(G) = L(G_1) U {ε}

G_1:
- S □ A | B | AB
- A □ a | aA | aAA
- B □ b | bB | bBB

+

- S □ ε

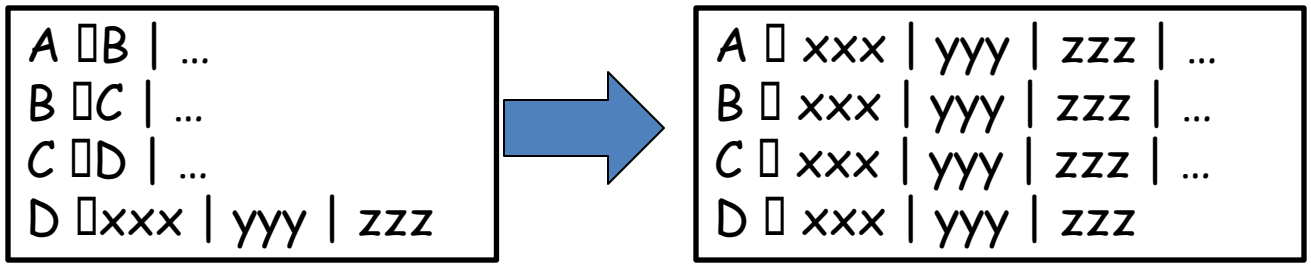
Eliminating unit productions



What's the point of removing unit transitions ?

Will save #substitutions

E.g.,



before *after*

Eliminating unit productions

- Unit production is one which is of the form A → B, where both A & B are variables
- E.g.,
  1. E → T | E+T
  2. T → F | T*F
  3. F → I | (E)
  4. I → a | b | Ia | Ib | I0 | I1
  - How to eliminate unit productions?
    - Replace E→ T with E → F | T*F
    - Then, upon recursive application wherever there is a unit production:
      - E→ F | T*F | E+T (substituting for T)
      - E→ I | (E) | T*F| E+T (substituting for F)
      - E→ a | b | Ia | Ib | I0 | I1 | (E) | T*F | E+T (substituting for I)
      - Now, E has no unit productions
    - Similarly, eliminate for the remainder of the unit productions

The Unit Pair Algorithm: to remove unit productions

- Suppose $A \Rightarrow B_1 \Rightarrow B_2 \Rightarrow \ldots \Rightarrow B_n \Rightarrow \alpha$
- Action: Replace all intermediate productions to produce α directly
  - i.e., $A \Rightarrow \alpha$; $B_1 \Rightarrow \alpha$; ... $B_n \Rightarrow \alpha$;

Definition: (A,B) to be a **"unit pair"** if $A \Rightarrow^* B$

- We can find all unit pairs inductively:
  - Basis: Every pair (A,A) is a unit pair (by definition). Similarly, if $A \Rightarrow B$ is a production, then (A,B) is a unit pair.
  - Induction: If (A,B) and (B,C) are unit pairs, and $A \Rightarrow C$ is also a unit pair.

The Unit Pair Algorithm: to remove unit productions

Input: $G=(V,T,P,S)$

Goal: To build $G_1=(V,T,P_1,S)$ devoid of unit productions

Algorithm:

1. Find all unit pairs in G
2. For each unit pair (A,B) in G:
    1. Add to P_1 a new production $A \rightarrow a$, for every $B \rightarrow a$ which is a *non-unit* production
    2. If a resulting production is already there in P, then there is no need to add it.

Example: eliminating unit productions

G:

1. E → T | E+T
2. T → F | T*F
3. F → I | (E)
4. I → a | b | Ia | Ib | I0 | I1

Unit pairs	*Only non-unit productions to be added to P_1*
(E,E)	E → E+T
(E,T)	E → T*F
(E,F)	E → (E)
(E,I)	E → a|b|Ia | Ib | I0 | I1
(T,T)	T → T*F
(T,F)	T → (E)
(T,I)	T → a|b| Ia | Ib | I0 | I1
(F,F)	F → (E)
(F,I)	F → a| b| Ia | Ib | I0 | I1
(I,I)	I → a| b | Ia | Ib | I0 | I1

G_1:

1. E → E+T | T*F | (E) | a| b | Ia | Ib | I0 | I1
2. T → T*F | (E) | a| b | Ia | Ib | I0 | I1
3. F → (E) | a| b | Ia | Ib | I0 | I1
4. I → a | b | Ia | Ib | I0 | I1

Putting all this together…

- Theorem: If G is a CFG for a language that contains at least one string other than ε, then there is another CFG G_1, such that $L(G_1) = L(G) - \varepsilon$, *and* G_1 has:
  - no ε -productions
  - no unit productions
  - no useless symbols

- Algorithm:

  Step 1) eliminate ε -productions
  Step 2) eliminate unit productions
  Step 3) eliminate useless symbols

Normal Forms

Why normal forms?

- If all productions of the grammar could be expressed in the same form(s), then:

  a. It becomes easy to design algorithms that use the grammar.

  b. It becomes easy to show proofs and properties.

Chomsky Normal Form (CNF)

Let G be a CFG for some L-{ε}

Definition:

G is said to be in ***Chomsky Normal Form*** *if all its productions are in one of the following two forms:*

i. $A \rightarrow BC$ *where A,B,C are variables, or*

ii. $A \rightarrow a$ *where a is a terminal*

- *G has no useless symbols*
- *G has no unit productions*
- *G has no ε-productions*

CNF checklist

Is this grammar in CNF?

G_1:

1. E → E+T | T*F | (E) | Ia | Ib | IO | I1
2. T → T*F | (E) | Ia | Ib | IO | I1
3. F → (E) | Ia | Ib | IO | I1
4. I → a | b | Ia | Ib | IO | I1

Checklist:

- G has no ε-productions
- G has no unit productions
- G has no useless symbols
- But…
  - the normal form for productions is violated

⟶ So, the grammar is not in CNF

How to convert a G into CNF?

- <u>Assumption:</u> G has no ε-productions, unit productions or useless symbols

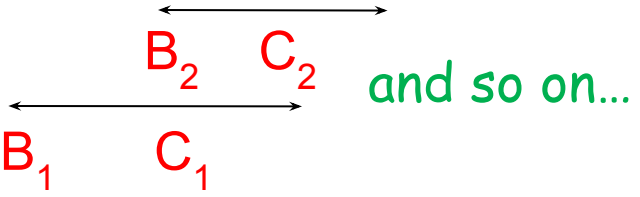
- For every terminal **a** that appears in the body of a production:

  i. create a unique variable, say X_a, with a production $X_a \rightarrow a$, and

  ii. replace all other instances of a in G by X_a

How to convert a G into CNF?

- Now, all productions will be in one of the following two forms:
  - $A \rightarrow B_1B_2 \ldots B_k$ $(k \geq 3)$ or $A \rightarrow a$
- Replace each production of the form $A \rightarrow B_1B_2B_3 \ldots B_k$ by:



  - $A \rightarrow B_1C_1$ $C_1 \rightarrow B_2C_2$ ... $C_{k-3} \rightarrow B_{k-2}C_{k-2}$ $C_{k-2} \rightarrow B_{k-1}B_k$

Example #1

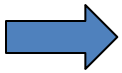
G:

$S \rightarrow AS \mid BABC$

$A \rightarrow A1 \mid 0A1 \mid 01$

$B \rightarrow 0B \mid 0$

$C \rightarrow 1C \mid 1$



G in CNF:

$X_0 \rightarrow 0$

$X_1 \rightarrow 1$

$S \rightarrow AS \mid BY_1$

$Y_1 \rightarrow AY_2$

$Y_2 \rightarrow BC$

$A \rightarrow AX_1 \mid X_0Y_3 \mid X_0X_1$

$Y_3 \rightarrow AX_1$

$B \rightarrow X_0B \mid 0$

$C \rightarrow X_1C \mid 1$

All productions are of the form: $A \rightarrow BC$ or $A \rightarrow a$

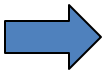
Languages with ε

- For languages that include ε,
  - Write down the rest of grammar in CNF
  - Then add production "S → ε" at the end

E.g., consider:

G:

S → AS | BABC
A → A1 | 0A1 | 01 | ε
B → 0B | 0 | ε
C → 1C | 1 | ε



G in CNF:

$X_0 \rightarrow 0$
$X_1 \rightarrow 1$

$S \rightarrow AS \mid BY_1 \mid \varepsilon$
$Y_1 \rightarrow AY_2$
$Y_2 \rightarrow BC$

$A \rightarrow AX_1 \mid X_0Y_3 \mid X_0X_1$
$Y_3 \rightarrow AX_1$
$B \rightarrow X_0B \mid 0$
$C \rightarrow X_1C \mid 1$

Other Normal Forms

- Griebach Normal Form (GNF)
  - All productions of the form:

$$A \rightarrow a\alpha$$

Where $A \in V$, $a \in T$ and $\alpha \in V^*$

Griebach Normal Form (GNF) - Example

S → AB

A → aA | bB | b

B → b

Grammar in GNF:

S → aAB | bBB | bB

A → aA | bB | b

B → b

Griebach Normal Form (GNF) - Example

S → abSb | aa

Grammar in GNF:

S → aBSB | aA

A → a

B → b

CFL Closure Properties

Closure Property:

- CFLs are closed under:
  - Union
  - Concatenation
  - Kleene closure operator
  - Substitution
  - Homomorphism, inverse homomorphism
  - reversal

- CFLs are *not* closed under:
  - Intersection
  - Difference
  - Complementation

Note: Reg languages are closed under these operators

Strategy for Closure Property Proofs

- First prove "closure under **substitution**"
- Using the above result, prove other closure properties
- CFLs are closed under:
  - Union
  - Concatenation
  - Kleene closure operator
  - Homomorphism, inverse homomorphism
  - Reversal

The Substitution operation

Note: s(L) can use a different alphabet

For each $a \in \Sigma$, then let $s(a)$ be a language

If $w=a_1a_2\ldots a_n \in L$, then:

- $s(w) = \{ x_1x_2 \ldots \} \in s(L)$, s.t., $x_i \in s(a_i)$

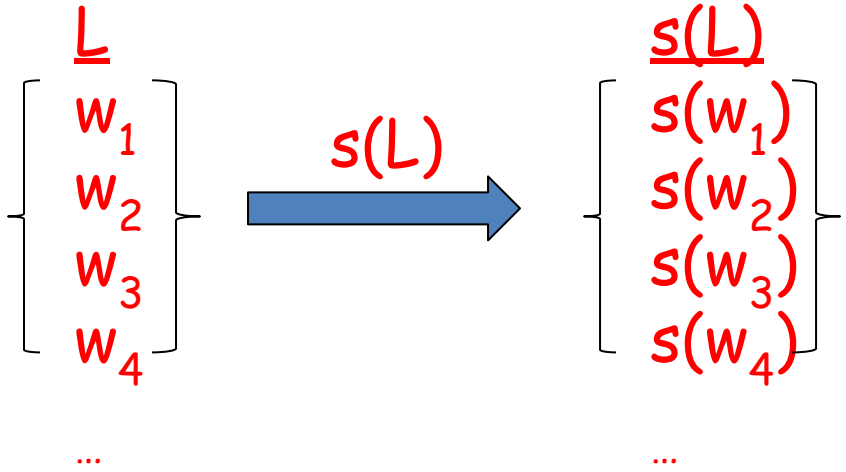
Example:

- Let $\Sigma=\{0,1\}$
- Let: $s(0) = \{a^nb^n \mid n \geq 1\}$, $s(1) = \{aa,bb\}$
- If $w=01$, $s(w)=s(0).s(1)$
  - E.g., s(w) contains $a^1 b^1 aa$, $a^1 b^1 bb$, $a^2 b^2 aa$, $a^2 b^2 bb$, ... and so on.

CFLs are closed under Substitution

- IF L is a CFL over Σ and s substitution defined on Σ, s.t., s(a) is a CFL for every symbol a $\in \Sigma$, THEN:
  - s(L) is a CFL.

What is s(L)?



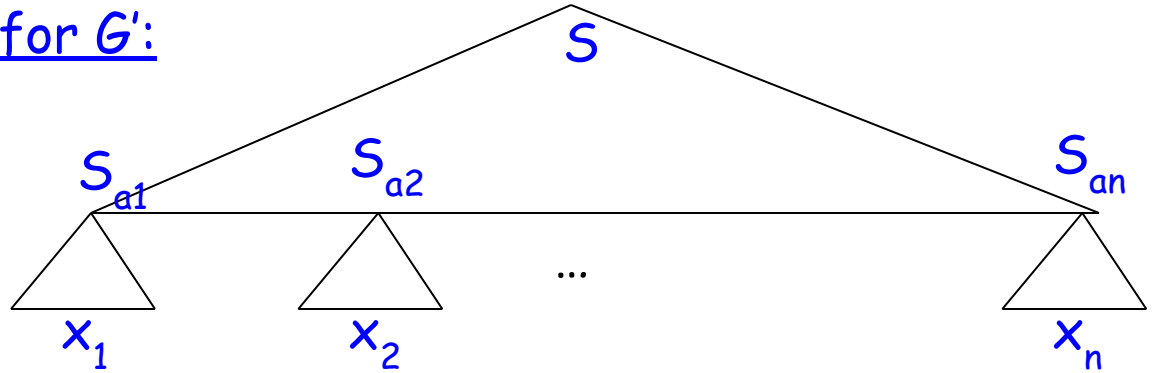
Note: each s(w) is itself a set of strings

CFLs are closed under Substitution

- $G=(V,T,P,S)$: CFG for L
- Because <u>every s(a) is a CFL</u>, there is a CFG for each s(a)
  - Let $G_a = (V_a, T_a, P_a, S_a)$
- Construct $G'=(V',T',P',S)$ for s(L)

- P′ consists of:
  - The productions of P, but with every occurrence of terminal "a" in their bodies replaced by S_a.
  - All productions in any P_a, for any $a \in \Sigma$

Parse tree for G′:



Substitution of a CFL: example

- Let L = language of binary palindromes s.t., substitutions for 0 and 1 are defined as follows:
  - $s(0) = \{a^n b^n \mid n \geq 1\}$, $s(1) = \{xx, yy\}$
- Prove that s(L) is also a CFL.

CFG for L:

$S \Rightarrow 0S0 \mid 1S1 \mid \varepsilon$

CFG for s(0):

$S_0 \Rightarrow aS_0b \mid ab$

CFG for s(1):

$S_1 \Rightarrow xx \mid yy$

Therefore, CFG for s(L):

$S \Rightarrow S_0SS_0 \mid S_1 S S_1 \mid \varepsilon$

$S_0 \Rightarrow aS_0b \mid ab$

$S_1 \Rightarrow xx \mid yy$

CFLs are closed under union

Let L_1 and L_2 be CFLs

<u>To show:</u> $L_1 \cup L_2$ is also a CFL

<u>Let us show by using the result of *Substitution*</u>

- Make a new language:
  - L_{new} = {a,b} s.t., s(a) = L_1 and s(b) = L_2

  ==> s(L_{new}) == same as == $L_1 \cup L_2$

- A more direct, alternative proof
  - Let S_1 and S_2 be the starting variables of the grammars for L_1 and L_2
    - Then, $S_{new} \Rightarrow S_1 \mid S_2$

CFLs are closed under concatenation

- Let L_1 and L_2 be CFLs

Let us show by using the result of *Substitution*

- Make L_{new} = {ab} s.t.,
  $s(a) = L_1$ and $s(b) = L_2$
  ==> $L_1 L_2 = s(L_{new})$

- A proof without using substitution?

CFLs are closed under Kleene Closure

- Let L be a CFL

– Let $L_{new} = \{a\}^*$ and $s(a) = L_1$

  – Then, $L^* = s(L_{new})$

CFLs are closed under Reversal

- Let L be a CFL, with grammar $G=(V,T,P,S)$
- For L^R, construct $G^R=(V,T,P^R,S)$ s.t.,
  - If $A \Rightarrow \alpha$ is in P, then:
    - $A \Rightarrow \alpha^R$ is in P^R
    - (that is, reverse every production)

CFLs are not closed under Intersection

- Existential proof:
  - $L_1 = \{0^n1^n2^i \mid n \geq 1, i \geq 1\}$
  - $L_2 = \{0^i1^n2^n \mid n \geq 1, i \geq 1\}$
- Both L_1 and L_1 are CFLs
  - Grammars?
- But $L_1 \cap L_2$ *cannot* be a CFL
  - Why?
- We have an example, where intersection is not closed.
- Therefore, CFLs are not closed under intersection

CFLs are not closed under complementation

- Follows from the fact that CFLs are not closed under intersection

- $L_1 \cap L_2 = \overline{\overline{L_1} \cup \overline{L_2}}$

Logic: if CFLs were to be closed under complementation

- **the whole right hand side becomes a CFL (because CFL is closed for union)**
- **the left hand side (intersection) is also a CFL**
- **but we just showed CFLs are NOT closed under intersection!**
- **CFLs *cannot* be closed under complementation.**

CFLs are not closed under difference

- Follows from the fact that CFLs are not closed under complementation

- Because, if CFLs are closed under difference, then:
  - $L = \Sigma^* - L$
  - So L has to be a CFL too
  - Contradiction

Decision Properties

- Emptiness test
  - Generating test
  - Rechability test
- Membership test
  - PDA acceptance

"Undecidable" problems for CFL

- Is a given CFG G ambiguous?
- Is a given CFL inherently ambiguous?
- Is the intersection of two CFLs empty?
- Are two CFLs the same?
- Is a given L(G) equal to Σ^*?

The Pumping Lemma !!

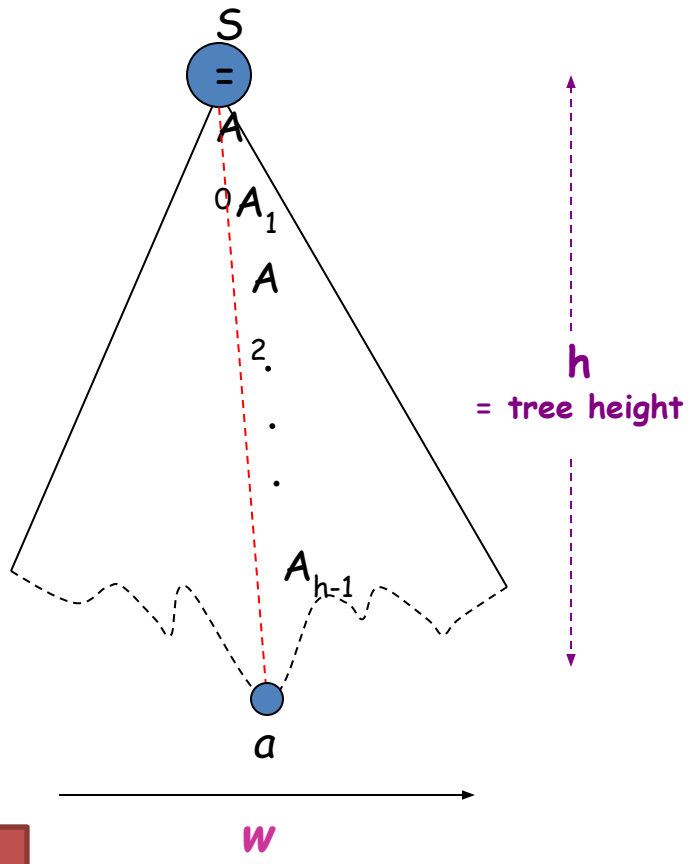
Why pumping lemma?

- A result that will be useful in proving languages that *are not* CFLs
  - (just like we did for regular languages)

The "parse tree theorem"

Observe that any parse tree generated by a CNF will be a binary tree, where all internal nodes have exactly two children (except those nodes connected to the leaves).

Parse tree for w



Given:

- Suppose we have a parse tree for a string **w**, according to a CNF grammar, G=(V,T,P,S)
- Let *h* be the height of the parse tree

Implies:

- $|w| \leq 2^{h-1}$

In other words, a CNF parse tree's string yield (w) can no longer be 2^{h-1}

Proof…The size of parse trees

To show: $|w| \leq 2^{h-1}$

Proof: (using induction on h)

Basis: h = 1

- Derivation will have to be "S⇒a"
- $|w| = 1 = 2^{1-1}$.

Ind. Hyp: h = k-1

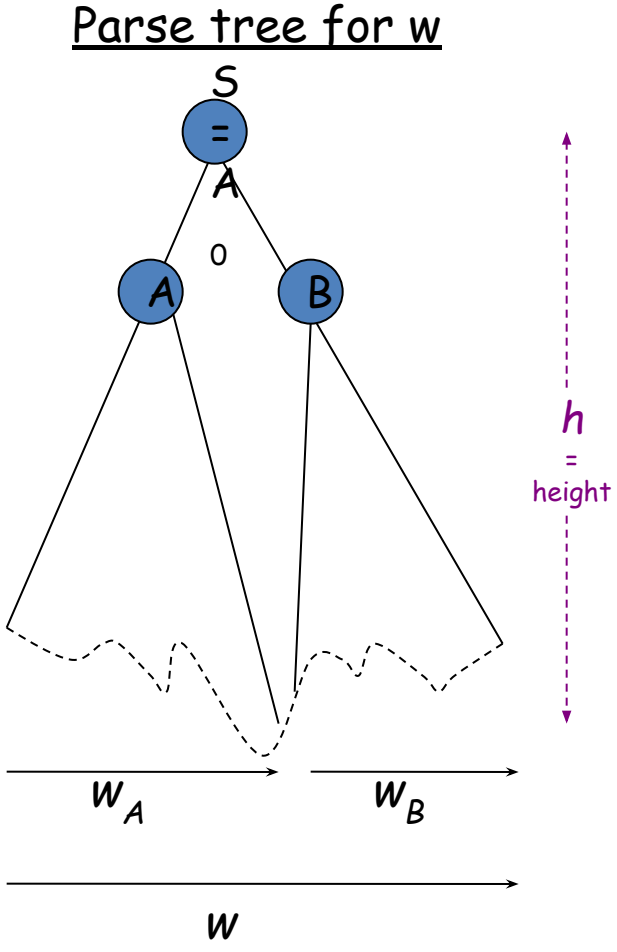
- $|w| \leq 2^{k-2}$

Ind. Step: h = k

S will have exactly two children:

S⇒AB

- Heights of A & B subtrees are at most h-1
- $w = w_A w_B$, where $|w_A| \leq 2^{k-2}$ and $|w_B| \leq 2^{k-2}$
- $|w| \leq 2^{k-1}$



Implication of the Parse Tree Theorem (assuming CNF)

Fact:

- If the height of a parse tree is h, then
  - $|w| \leq 2^{h-1}$

Implication:

- If $|w| \geq 2^{h}$, then
  - Its parse tree's height is *at least* h+1

The Pumping Lemma for CFLs

Let L be a CFL.

Then there exists a constant N, s.t.,

- if $z \in L$ s.t. $|z| \geq N$, then we can write $z = uvwxy$, such that:
  1. $|vwx| \leq N$
  2. $vx \neq \varepsilon$
  3. For all $k \geq 0$: $uv^k wx^k y \in L$

Note: we are pumping in two places (v & x)

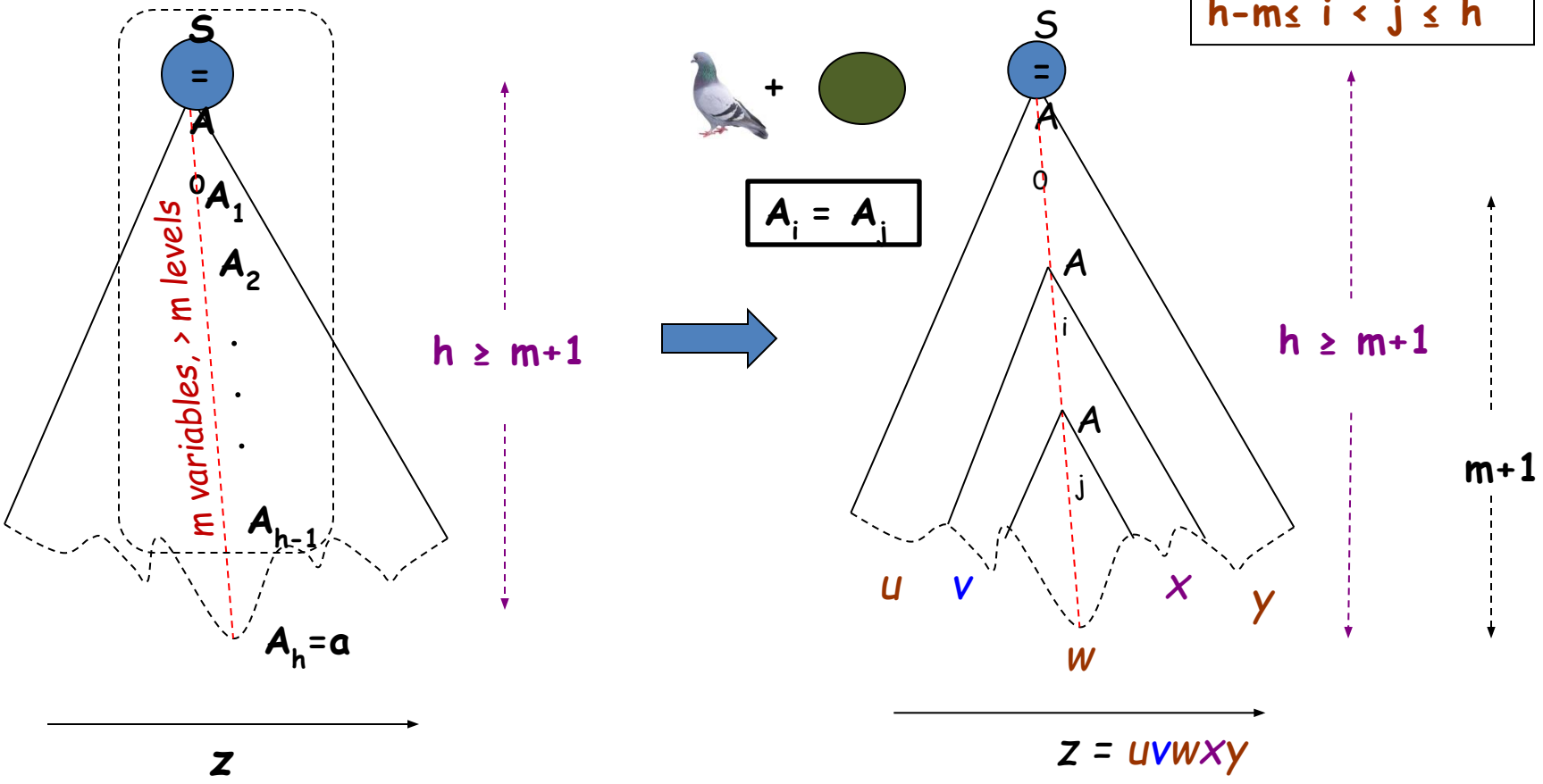
Proof: Pumping Lemma for CFL

- If L = ∅ or contains only ε, then the lemma is trivially satisfied (as it cannot be violated)

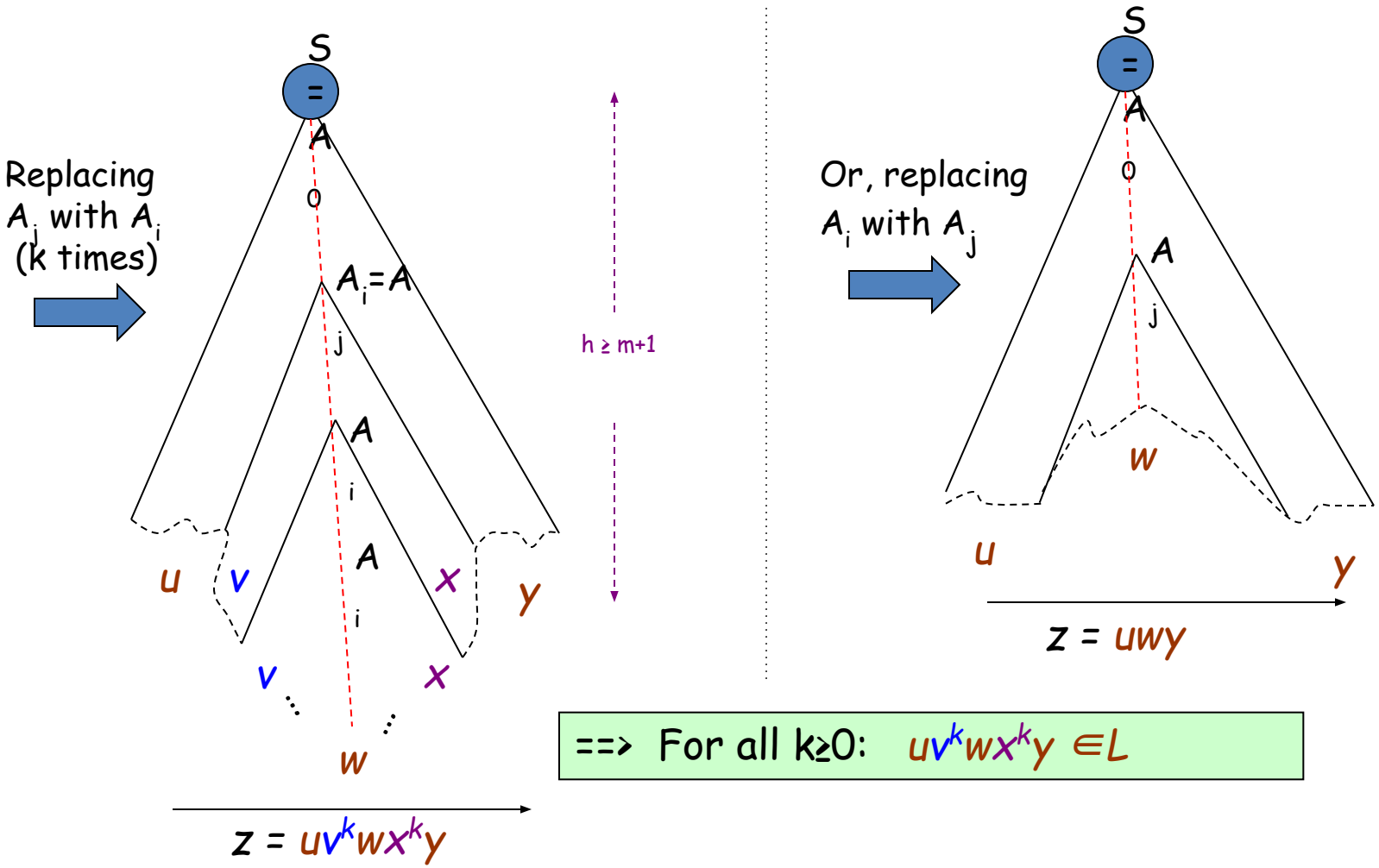
- For any other L which is a CFL:
  - Let G be a CNF grammar for L
  - Let m = number of variables in G
  - Choose N = 2^m.
  - Pick any z ∈ L s.t. |z| ≥ N

∴ the parse tree for z should have a height ≥ m+1
(by the parse tree theorem)

Parse tree for z



Extending the parse tree…



==> For all k≥0: $uv^kwx^ky \in L$

Proof contd..

- Also, since A_i's subtree no taller than m+1

  ==> the string generated under A_i's subtree, which is vwx, cannot be longer than 2^m (=N)

  But, 2^m =N

  ==> |vwx| ≤ N

  This completes the proof for the pumping lemma.

Application of Pumping Lemma for CFLs

Example 1: $L = \{a^m b^m c^m \mid m>0 \}$

Claim: L is not a CFL

Proof:

- Let N <== P/L constant
- Pick $z = a^N b^N c^N$
- Apply pumping lemma to z and show that there exists at least one other string constructed from z (obtained by pumping up or down) that is $\notin$ L

Proof contd…

- $z = uvwxy$
- *As* $z = a^N b^N c^N$ *and* $|vwx| \leq N$ *and* $vx \neq \varepsilon$
  - ==> v, x cannot contain all three symbols (a,b,c)
  - ==> we can pump up or pump down to build another string which is $\notin L$

Example #2 for P/L application

- L = { ww | w is in {0,1}*}
- Show that L is not a CFL
  - Try string z = 0^N0^N
    - what happens?
  - Try string z = $0^N1^N0^N1^N$
    - what happens?

Example 3

- L = { 0^{k^2} | k is any integer)

- Prove L is not a CFL using Pumping Lemma

Example 4

- $L = \{a^i b^j c^k \mid i<j<k \}$

- Prove that L is not a CFL