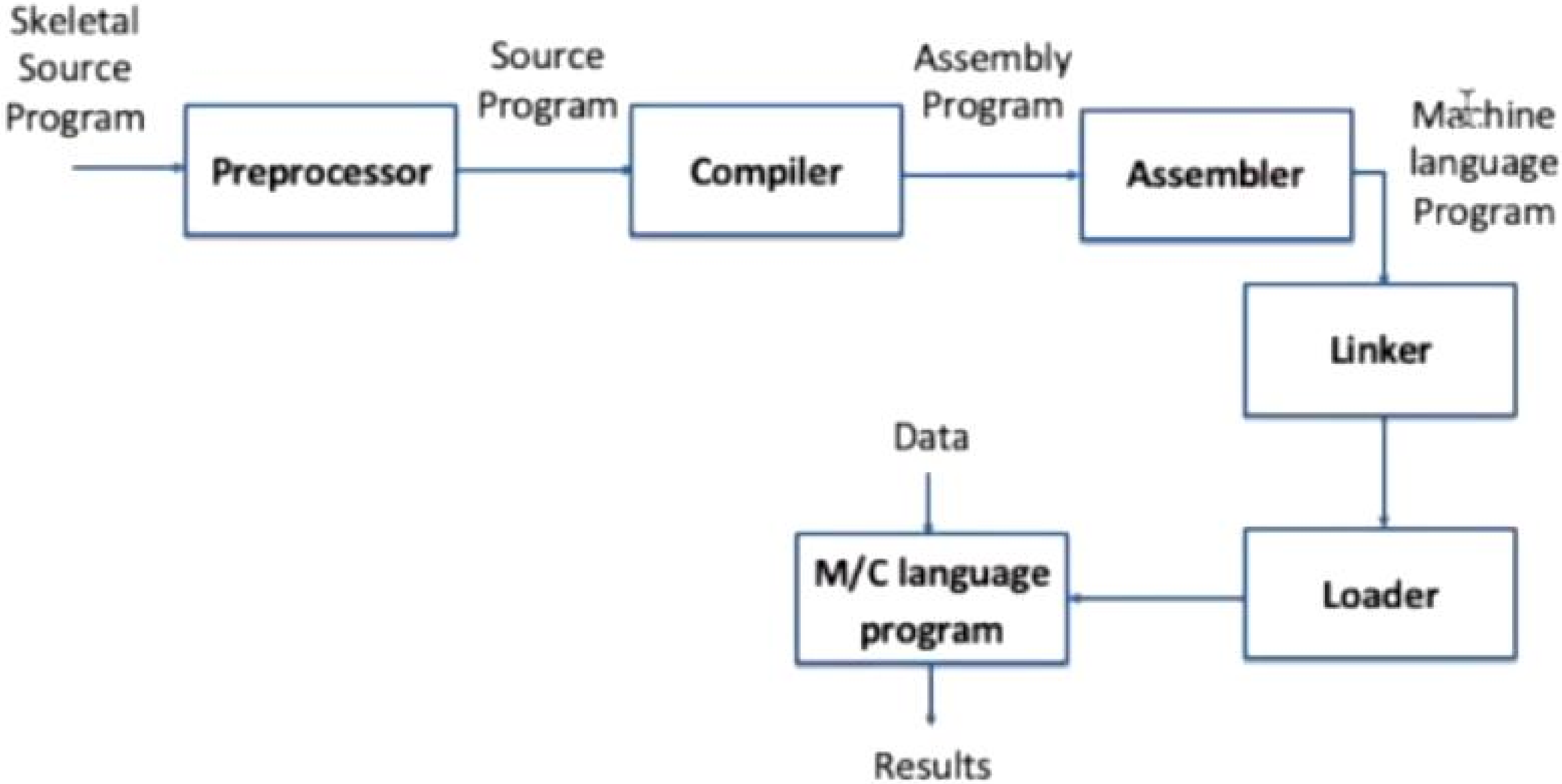
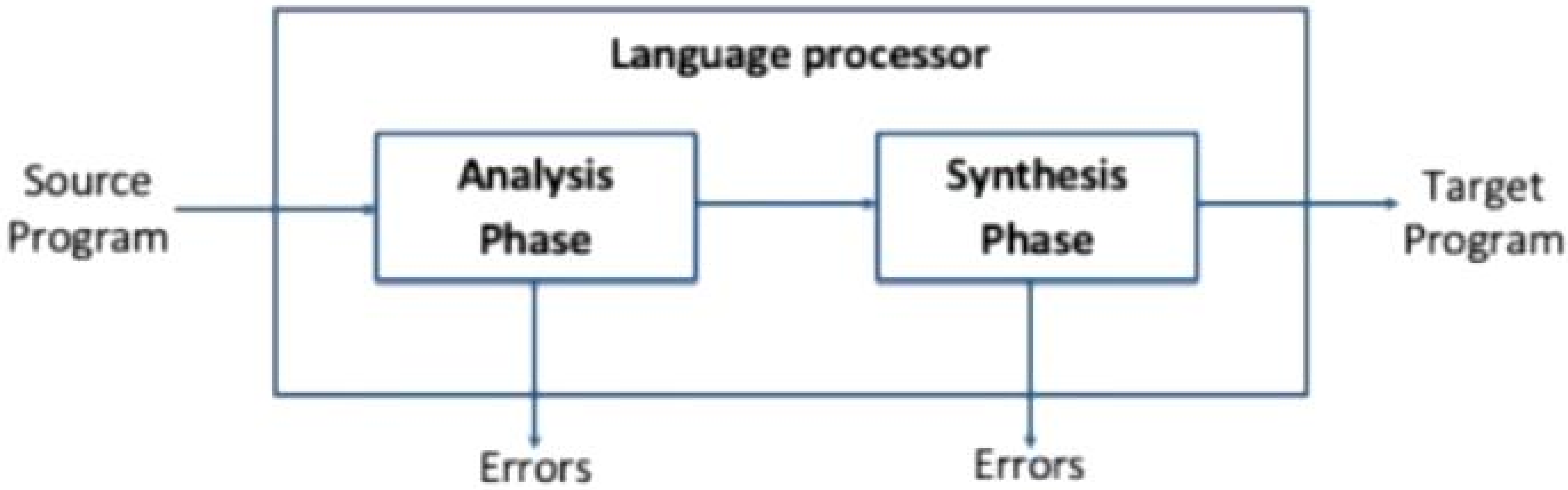
Practical arrangement of language processors

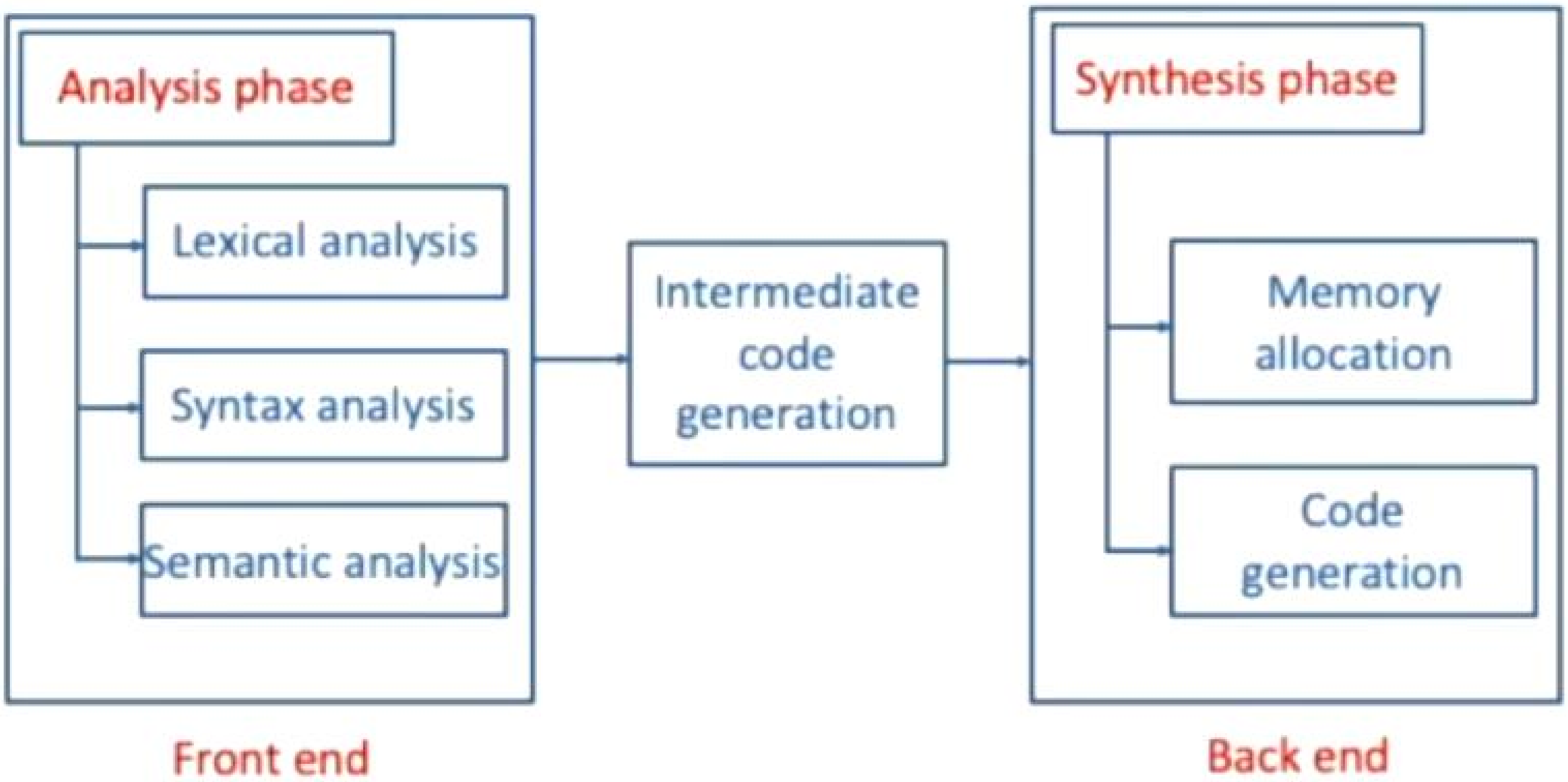


Overview of Compiler

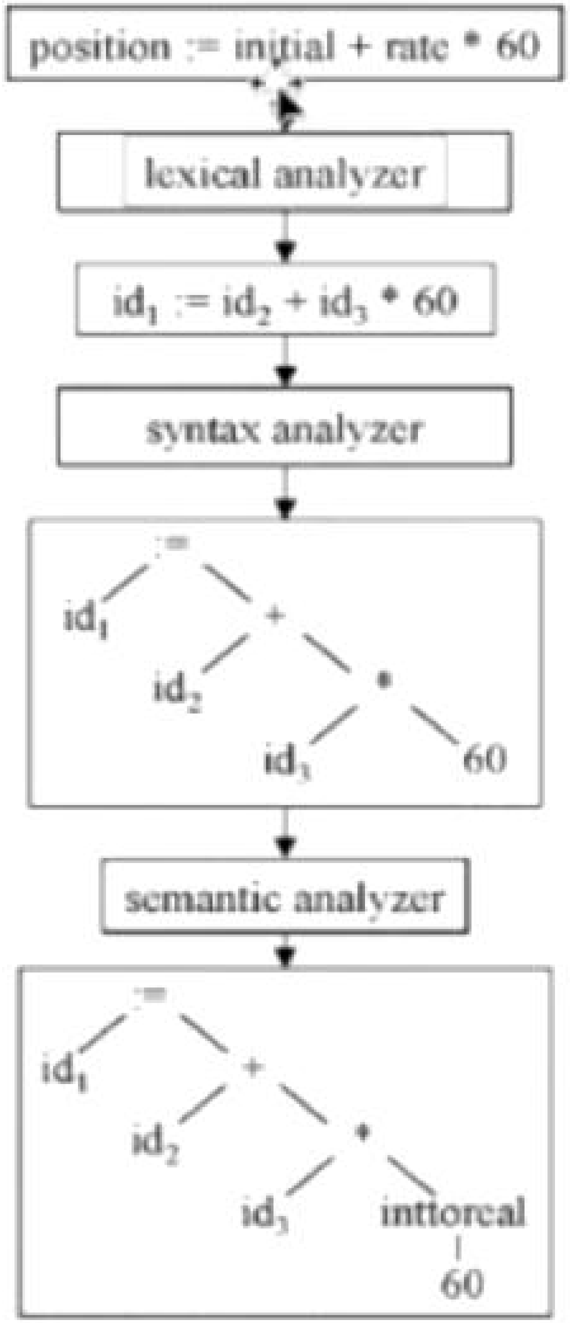
Language processing = Analysis of Source Program + Synthesis of Target Program

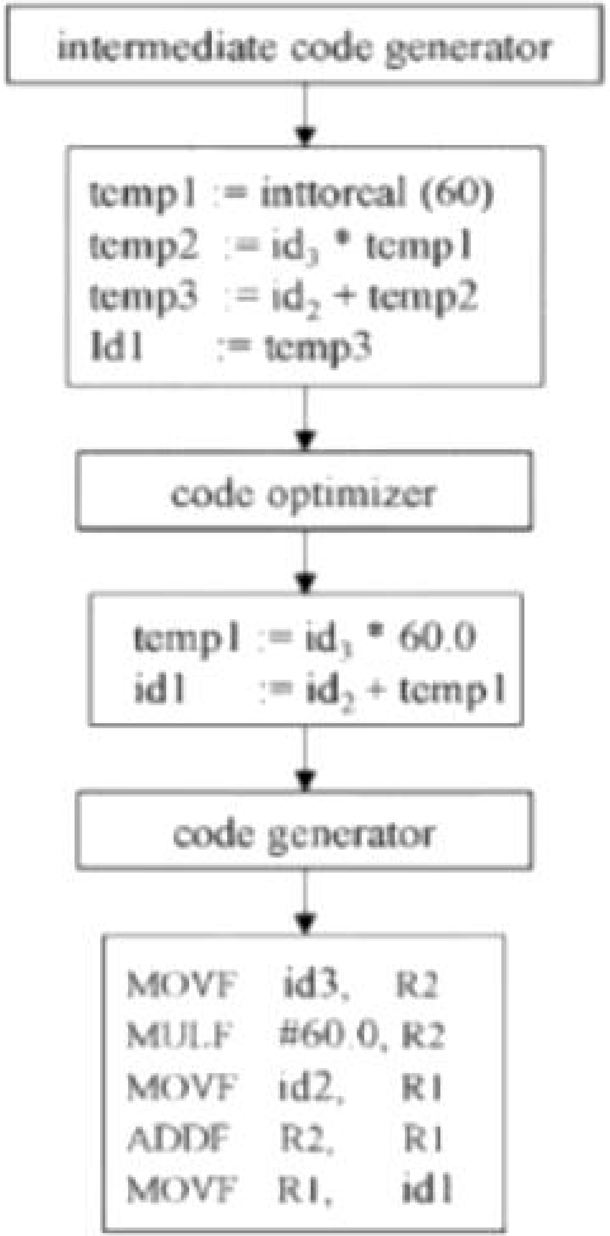


Phases of language processor (Toy compiler)



Typical Phases of a Compiler





Lexical Analysis / Scanning

Content

- Programming language grammar
- Classification of grammar
- Ambiguity in grammatic specification
- Scanning
- Language processor development tools (LEX)

Programming language grammar

Formal language

Graphical symbols

↓

{ a,b,c,d,e,f,g,h,i...........0,1,2,3............}

words

↓

{the, system, welcome, hello, computer, to, programming }

sentence

↓

"hello to system programming"

- A ***formal language*** is a collection of valid sentences, where each sentence is a sequence of words, and each word is a sequence of graphic symbols acceptable in a language.
- Set of rules that specify the construction of words and sentences is called ***formal language grammar***.

Formal language grammar

Write sentence of simple tense.

Sentence type	Rules	Example
Affirmative	subject + verb + object	She eats a fruit
Negative	subject + do/does + not + verb + object	She does not eat a fruit
Interrogative	do/does + subject + verb + object + ?	Does she eat a fruit?

Grammar

- A grammar G of a language L_G is a quadruple (Σ, SNT, S, P) where

Σ	is the alphabet of L_G, i.e. the set of terminal symbols
SNT	is the set of nonterminal symbols
S	is the start symbol
P	is the set of productions

Terminal symbol:

- A symbol in the alphabet.
- It is denoted by lower case letter and punctuation marks used in language.

<Noun Phrase> → <Article><Noun>

<Article> → a | an | the

<Noun> → boy | apple

Grammar

- A grammar G of a language L_G is a quadruple (Σ, SNT, S, P) where

Σ	is the alphabet of L_G, i.e. the set of terminal symbols
SNT	is the set of nonterminal symbols
S	is the start symbol
P	is the set of productions

Nonterminal symbol:

- The name of syntax category of a language, e.g., noun, verb, etc.
- It is written as a **single capital letter**, or as a **name enclosed between < … >,** e.g., A or <Noun>

<Noun Phrase> → <Article><Noun>

<Article> → a | an | the

<Noun> → boy | apple

Grammar

- A grammar G of a language L_G is a quadruple (Σ, SNT, S, P) where

Σ	is the alphabet of L_G, i.e. the set of terminal symbols
SNT	is the set of nonterminal symbols
S	is the start symbol
P	is the set of productions

Start symbol: First nonterminal symbol of the grammar is called start symbol.

<Noun Phrase> → <Article><Noun>
<Article> → a | an | the
<Noun> → boy | apple

Grammar

- A grammar G of a language L_G is a quadruple (Σ, SNT, S, P) where

Σ	is the alphabet of L_G, i.e. the set of terminal symbols
SNT	is the set of nonterminal symbols
S	is the start symbol
P	is the set of productions

Production: A production, also called a rewriting rule, is a rule of grammar. It has the form of

A nonterminal symbol → String of terminal and nonterminal symbols

<Noun Phrase> → <Article><Noun>
<Article> → a | an | the
<Noun> → boy | apple

Example: Grammar

Write terminals, non terminals, start symbol, and productions for following grammar.

E → E O E| (E) | -E | id

O → + | - | * | / | ↑

Terminals: **id + - * / ↑ ( )**

Non terminals: **E, O**

Start symbol: **E**

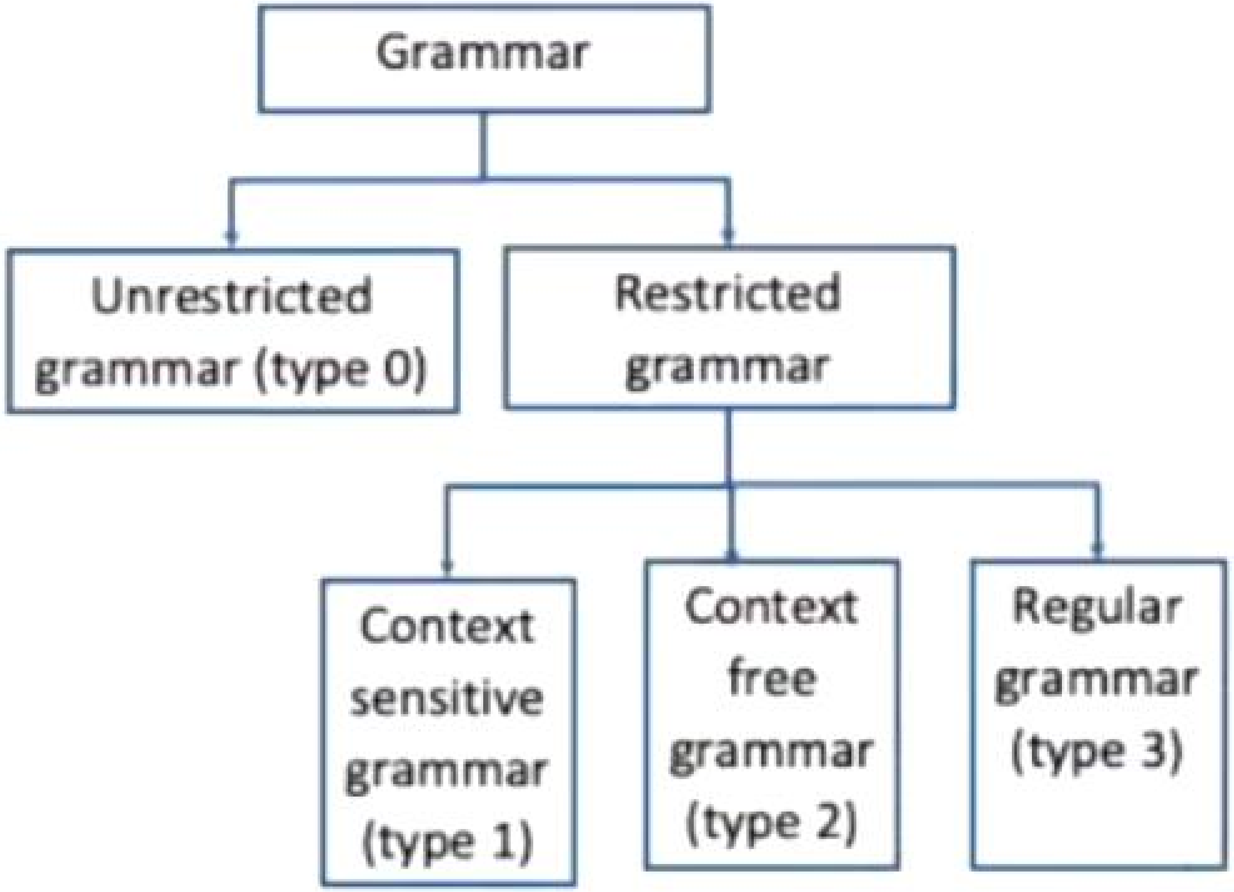
Productions: **E → E O E| (E) | -E | id**

O → + | - | * | /

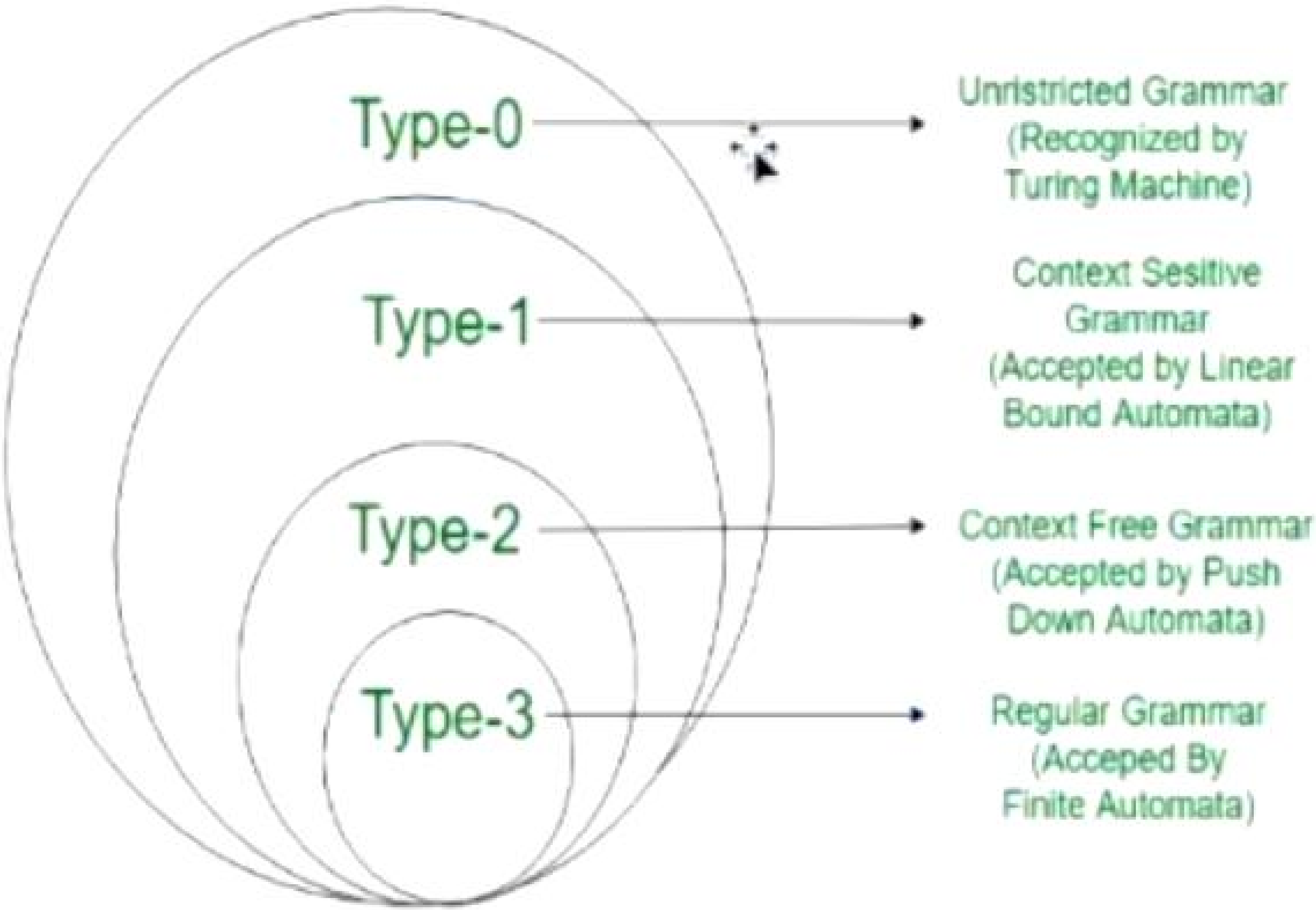
| ↑

Classification of grammar

Classification of grammar (Chomsky hierarchy)



Classification of grammar (Chomsky hierarchy)



Type 0 grammar (Phrase Structure Grammar)

- Their productions are of the form:

$$\alpha \rightarrow \beta$$

- where both α and β can be strings of terminal and nonterminal symbols.
- Such productions permit arbitrary substitution of strings during derivation or reduction, hence they are **not relevant to specification of programming languages.**
- Example: $S \rightarrow ACaB$

  $Bc \rightarrow acB$

  $CB \rightarrow DB$

  $aD \rightarrow Db$

Type 1 grammar (Context Sensitive Grammar)

- Their productions are of the form:

$$\alpha A\beta \rightarrow \alpha\pi\beta$$

- where **A** is non terminal and $\boldsymbol{\alpha}, \boldsymbol{\beta}, \boldsymbol{\pi}$ are strings of terminals and non terminals.
- The strings $\boldsymbol{\alpha}$ and $\boldsymbol{\beta}$ may be empty, but $\boldsymbol{\pi}$ must be non-empty.
- Here, a string π can be replaced by $'A'$ (or vice versa) only when it is enclosed by the strings α and β in a sentential form.
- Productions of Type-1 grammars specify that derivation or reduction of strings can take place **only in specific contexts**. Hence these grammars are also known as **context sensitive grammars**.
- These grammars are also **not relevant for programming language specification** since recognition of programming language constructs is not context sensitive in nature.
- Example: AB → AbBc

  A → bcA

  B → b

Type 2 grammar (Context Free Grammar)

- Their productions are of the form:

$$A \rightarrow \pi$$

- Where $\boldsymbol{A}$ is non terminal and $\boldsymbol{\pi}$ is string of terminals and non terminals.
- These grammars do not impose any context requirements on derivations or reductions which can be applied independent of its context.
- CFGs are **ideally suited for programming language specification**.
- Example: S → Xa

  X → a

  X → aX

  X → abc

Type 3 grammar (Linear or Regular grammar)

- Their productions are of the form:

$$A \rightarrow tB \mid t \qquad \text{or} \qquad A \rightarrow Bt \mid t$$

- Where A, B are non terminals and t is terminal.
- The specific form of the RHS alternatives - namely a single terminal symbol or a string containing a single terminal and a single nonterminal.
- However, the nature of the productions restricts the expressive power of these grammars, e.g., nesting of constructs or matching of parentheses cannot be specified using such productions.
- Hence the use of Type-3 productions is **restricted to the specification of lexical units, e.g., identifiers, constants, labels,** etc.
- Example: $X \rightarrow a \mid aY$

  $Y \rightarrow b$

Derivation

Grammar : <Noun Phrase> → <Article><Noun>
<Article> → a | an | the
<Noun> → boy | apple

<Noun Phrase>→<Article> <Noun>
→the <Noun>
→the boy

- Let production P_1 of grammar G be of the form P_1: $A \rightarrow \alpha$ and let β be a string such that $\beta = \gamma A \theta$, then replacement of A by α in string β constitutes a derivation according to production P_1.
- There are two types of derivation:
  1. Leftmost derivation
  2. Rightmost derivation

Leftmost derivation

- A derivation of a string W in a grammar G is a left most derivation if at every step the left most non terminal is replaced.
- Grammar: S→S+S | S-S | S*S | S/S | a   Output string: a*a-a

S

→S-S

→S*S-S

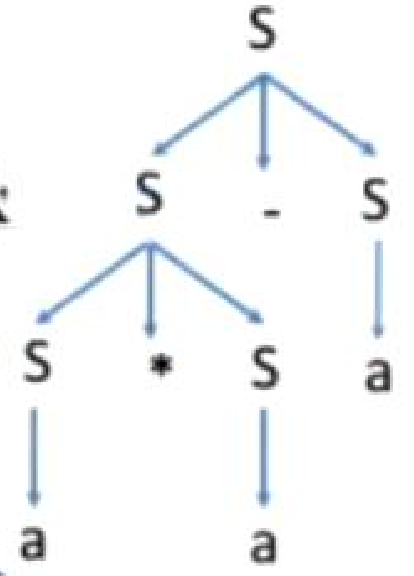
→a*S-S

→a*a-S

→a*a-a

Leftmost Derivation

Parse tree represents the structure of derivation



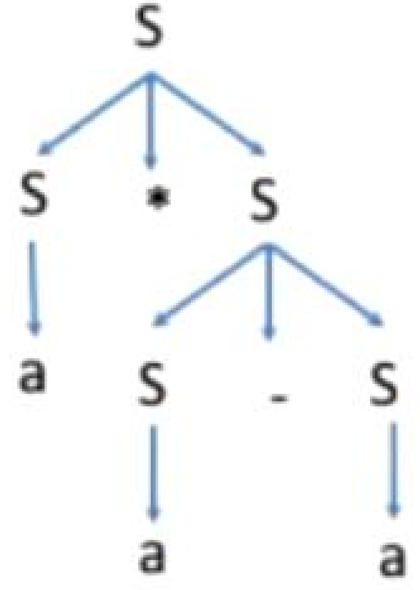
Parse tree

Rightmost derivation

- A derivation of a string W in a grammar G is a right most derivation if at every step the right most non terminal is replaced.
- It is all called canonical derivation.
- Grammar: $S \rightarrow S+S \mid S-S \mid S*S \mid S/S \mid a$ Output string: a*a-a

$$
\begin{aligned}
& S \\
& \rightarrow S*\underline{S} \\
& \rightarrow S*S-\underline{S} \\
& \rightarrow S*\underline{S}-a \\
& \rightarrow \underline{S}*a-a \\
& \rightarrow a*a-a
\end{aligned}
$$

Rightmost Derivation



Parse Tree

Reduction

Grammar : <Noun Phrase> → <Article><Noun>
<Article> → a | an | the
<Noun> → boy | apple

→the boy

→**<Article>** boy

→<Article> **<Noun>**

<Noun Phrase>

- Let production P_1 of grammar G be of the form P_1: $A \rightarrow \alpha$ and let σ be a string such that $\sigma \rightarrow \gamma\alpha\theta$, then replacement of α by A in string σ constitutes a reduction according to production P_1.

Current sentential form

- Current sentential form is any string derivable from start symbol.
- Grammar: S→S+S | S-S | S*S | S/S | a   Output string: a*a-a

S

→__S*S__

→S*__S-S__

→S*S-__a__

→S*__a__-a

→__a__*a-a

Current sentential forms (CSF)

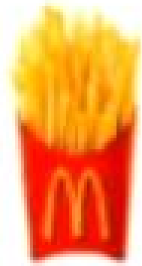
Ambiguity in grammatic specification

Ambiguity

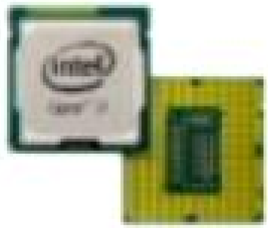
- Ambiguity, is a word, phrase, or statement which contains more than one meaning.

Chip

- A long thin piece of potato



- A small piece of silicon



Ambiguity

- In formal language grammar, ambiguity would arise if identical string can occur on the RHS of two or more productions.
- Grammar:

  $N_1 \rightarrow \alpha$

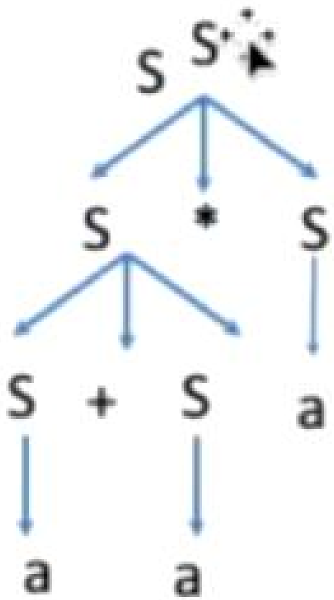
  $N_2 \rightarrow \alpha$

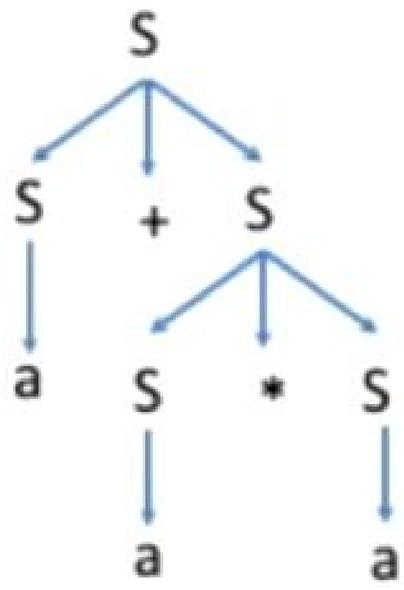
- α can be derived from either N_1 or N_2

Ambiguous grammar

- Ambiguous grammar is one that produces more than one leftmost or more then one rightmost derivation for the same sentence.
- Grammar: S→S+S | S*S | (S) | a   Output string: a+a*a

S	
→**S*S**	→**S+S**
→**S+S***S	→**a**+S
→**a**+S*S	→a+**S*S**
→a+**a***S	→a+**a***S
→a+a***a**	→a+a***a**





Here, ***Two leftmost derivation*** for string a+a*a is possible hence, above grammar is ambiguous.

Eliminating ambiguity

Grammar: S→ S+S | S*S | (S) | a

Equivalent unambiguous grammar is

S → S + T | T
T → T * F | F
F → (S) | a

Equivalent unambiguous grammar

Not possible????

Try for second leftmost derivation

Here, ***two left most derivation is not possible*** for string a+a*a hence, grammar is unambiguous.

Output string: a+a*a

S
→ **S+T**
→ **T**+T
→ **F**+T
→ **a**+T
→ a+**T*F**
→ a+**F***F
→ a+**a***F
→ a+a***a**

Regular expression

- A regular expression is a sequence of characters that define a pattern.
- Notational shorthand's
  1. One or more occurrences: +
  2. Zero or more occurrences: *
  3. Alphabets: Σ

- Regular Expression is mainly for use in pattern matching with strings, or string matching, i.e. "find and replace"-like operations.

Regular expression

L = Zero or More Occurrences of a = a*

*

ϵ
a
aa
aaa
aaaa
aaaaa.....

Infinite .....

Regular expression

L = One or More Occurrences of a = a^+

$$+$$

$$\left.\begin{array}{l} a \\ aa \\ aaa \\ aaaa \\ aaaaa..... \end{array}\right\}\text{Infinite .....}$$

Precedence and associativity of operators

Operator	Precedence	Associative
Kleene *	1	left
Concatenation	2	left
Union |	3	left

Regular expression examples

Strings: 0, 1 $\qquad$ $R.E. = 0 \mid 1$

Strings: 0, 11, 111 $\qquad$ $R.E. = 0 \mid 11 \mid 111$

Strings: $\epsilon, a, aa, aaa, aaaa \ldots$ $\qquad$ $R.E. = a^{*}$

Strings: $a, aa, aaa, aaaa \ldots\ldots$ $\qquad$ $R.E. = a^{+}$

Strings: abc, bca, bbb, cab, al $\qquad$ $R.E. = (a|b|c)\ (a|b|c)\ (a|b|c)$

Strings: 0, 11, 101, 10101, 11 $\qquad$ $R.E. = (0 \mid 1)^{+}$

Finite State Automata

Finite state automata

- A finite state automata is a triple (S, Σ, T) where

  S: is a finite set of states
  one of which is the initial state s_{init}
  one or more of which are the final states.

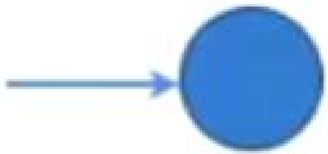
  Σ: is the input symbols.

  T: is a finite set of state transitions defining transitions out of states in S on encountering symbols in Σ.

Finite state automata notations

is a
state

is a transition



is a start state

is a final state

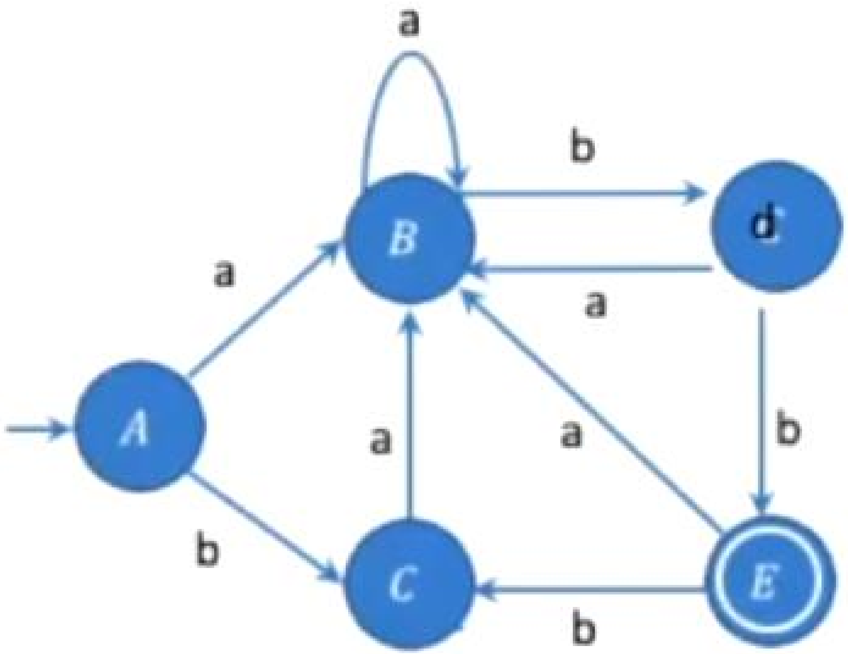
Finite state automata

- Deterministic finite state automaton:
  - It is a finite state automaton none of whose states has two or more transitions for the same source symbol.
  - The DFA has the property that it reaches a unique state for every source string input to it.
- Non Deterministic finite state automaton:
  - No restriction on edges leaving states.
  - There can be several with the same symbol as label and some edges can be labeled as ϵ.

DFA

States	a	b
A	B	C
B	B	D
C	B	C
D	B	E
E	B	C

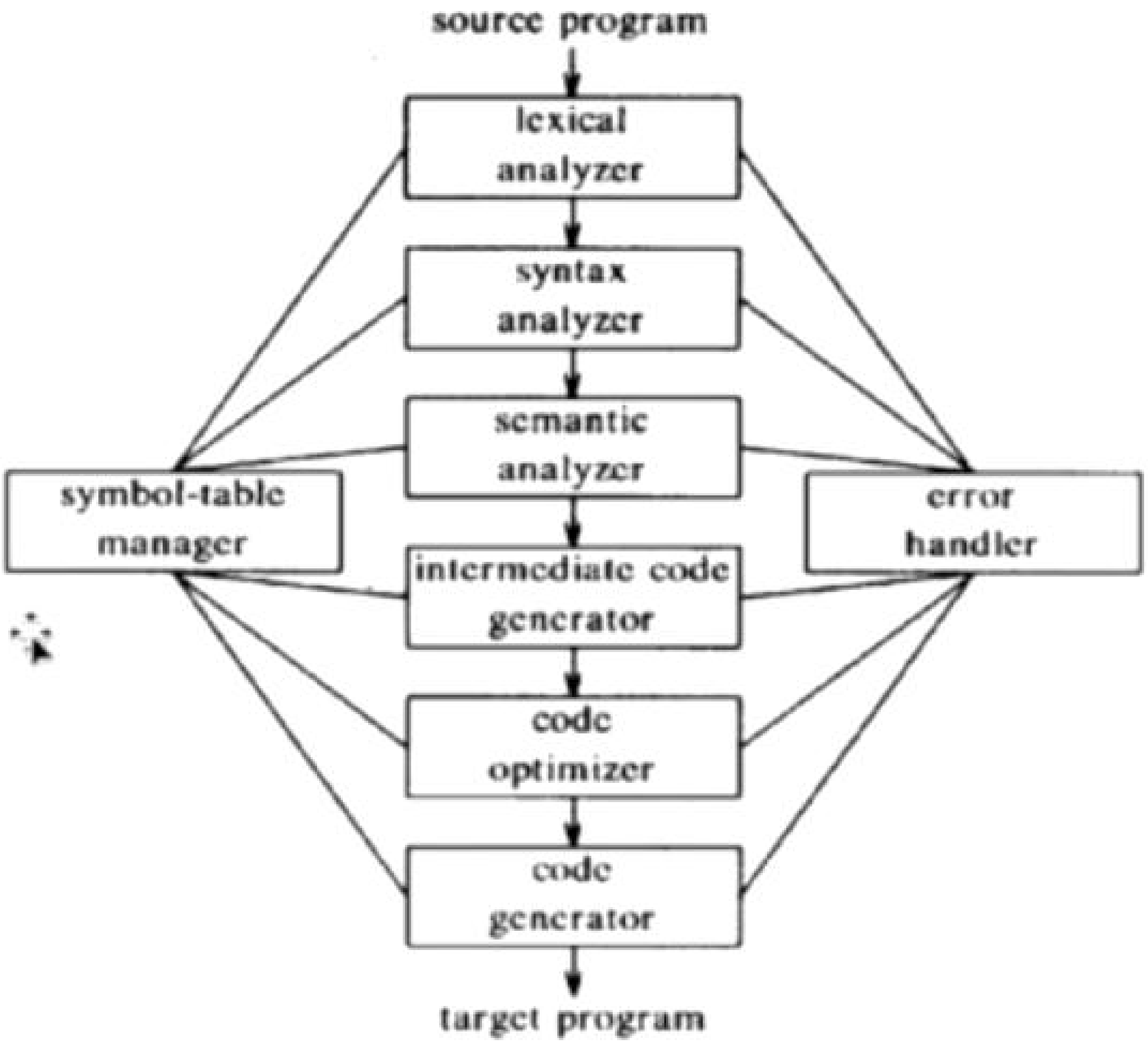
Transition Table



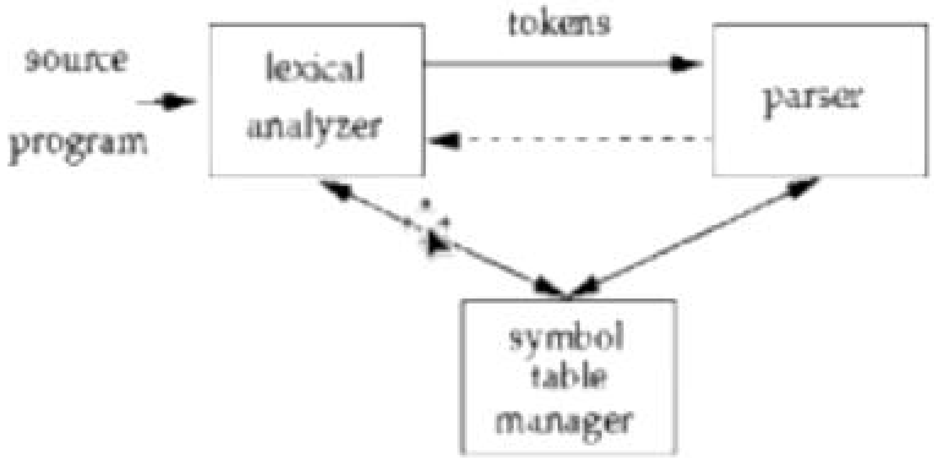
DFA

Lexical Analysis (Scanning)

Phases of a compiler



Overview



- *Main task*: to read input characters and group them into "*tokens*" – (Constants, identifiers, keywords etc.)
- *Secondary tasks*:
  - Skip comments and white space;
  - Correlate error messages with source program (e.g., line number of error).

Lexical Analysis: Terminology

- token: a name for a set of input strings with related structure.

  Example: "identifier," "integer constant"

- pattern: a rule describing the set of strings associated with a token.

  Example: "a letter followed by zero or more letters, digits, or underscores."

- lexeme: the actual input string that matches a pattern.

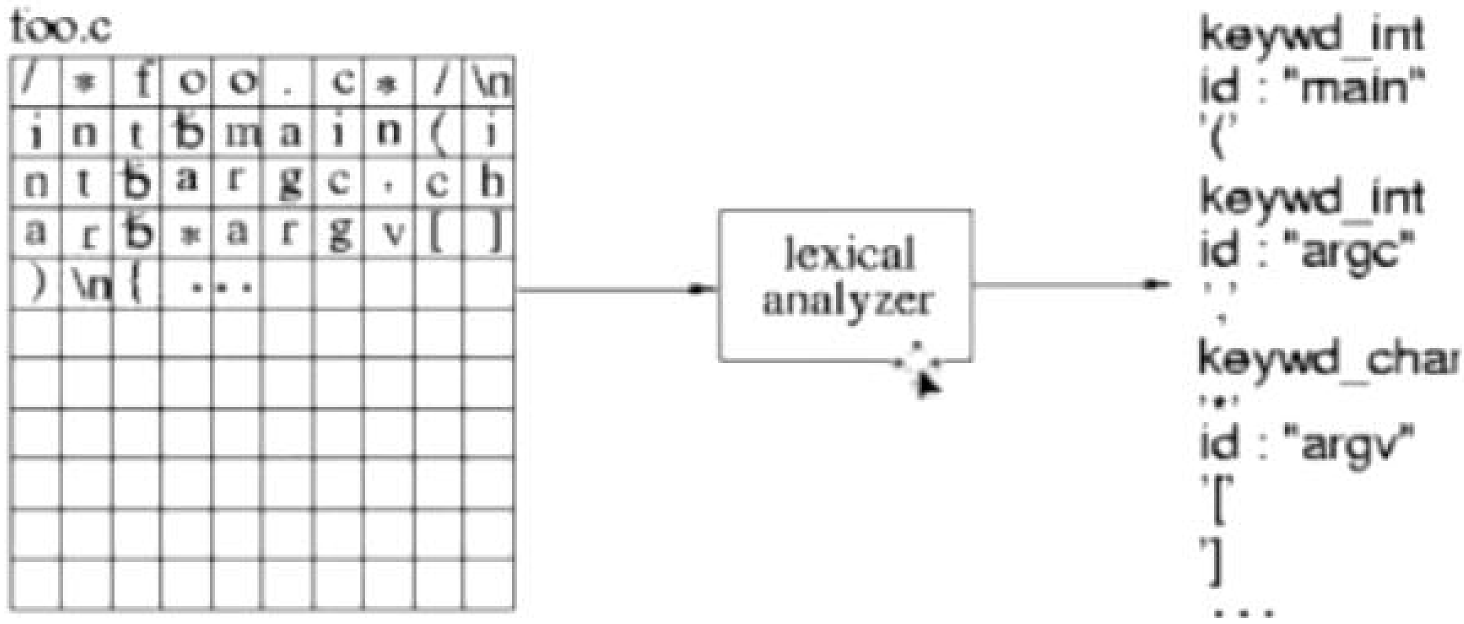
  Example: count

Lexical Analysis

Lexeme	Token	Attribute Value
White space..............		
Sequence of digits......	NUM	Numeric value of sequence
div..........................	DIV	
mod.........................	MOD	
Other sequence of a letter then letters and digits...	ID	Index into symtable
End – of – file char......	DONE	
Any other character.....	character	NONE

Description of tokens

Lexical Analysis



Examples

Input: `count = 123`

Tokens:

identifier : *Rule*: "letter followed by …"

Lexeme: count

assg_op : *Rule*: =

Lexeme: =

integer_const : *Rule*: "digit followed by …"

Lexeme: 123

Attributes for Tokens

- If more than one lexeme can match the pattern for a token, the scanner must indicate the actual lexeme that matched.

- This information is given using an ***attribute*** associated with the token.

  Example: The program statement

  ```
  count = 123
  ```

  yields the following token-attribute pairs:

  ⟨*identifier*, pointer to the string "**count**"⟩

  ⟨*assg_op*, ⟩

  ⟨*integer_const*, the integer value 123⟩

Implementing Lexical Analyzers

Different approaches:

- Using a scanner generator, e.g., **lex** or **flex**. This automatically generates a lexical analyzer from a high-level description of the tokens.

  (easiest to implement; least efficient)

- Programming it in a language such as C, using the I/O facilities of the language.

  (intermediate in ease, efficiency)

Implementing Lexical Analyzers

- Identify keywords, identifiers – id table
- Identify operators – operator table
- Identify Preprocessor Directives – Keyword # in id table
- Identify comments

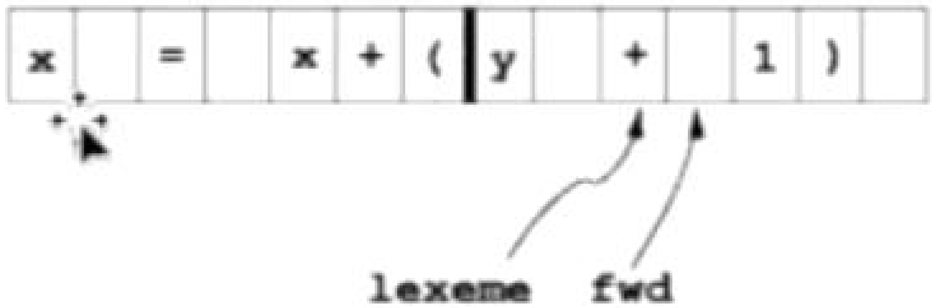
Program

- # include < stdio.h >
- int main (void)
- {
- Int a, b;
- float c, d;
- for (i=0; i<=10; i++) / * *can be any other loop* */
- d = a + b + c ;
- printf(“%f”,d); /* *can be any other function* */
- }

Input Buffering

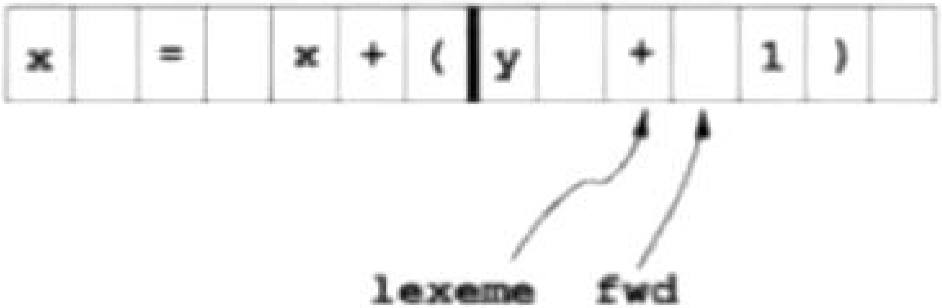
- Scanner performance is crucial:
  - This is the only part of the compiler that examines the entire input program one character at a time.
  - Disk input can be slow.
  - The scanner accounts for ~25-30% of total compile time.
- We need look ahead to determine when a match has been found.
- Scanners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

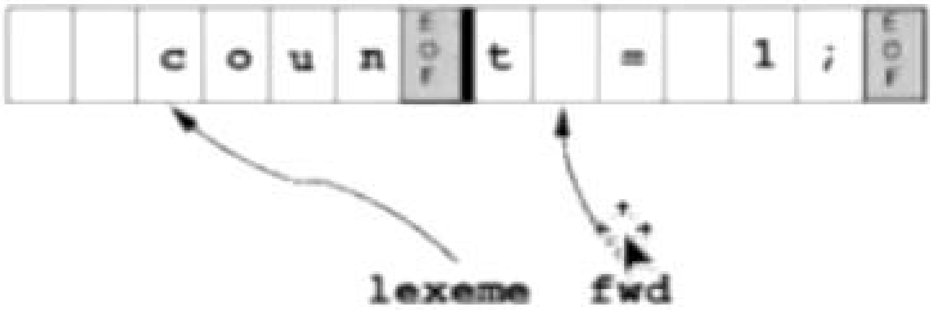


Code:

```
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



- *<u>Objective</u>*: Optimize the common case by reducing the number of tests to one per advance of fwd.
- *<u>Idea</u>*: 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).