Chapter 1

Introduction

✱ Before running a program, it first must be translated into a form in which it can be executed by a computer. The software systems that do this translation are called compilers.

1.1 Language Processors.

✱ compiler reports any errors in the source program that it detects during the translation process.

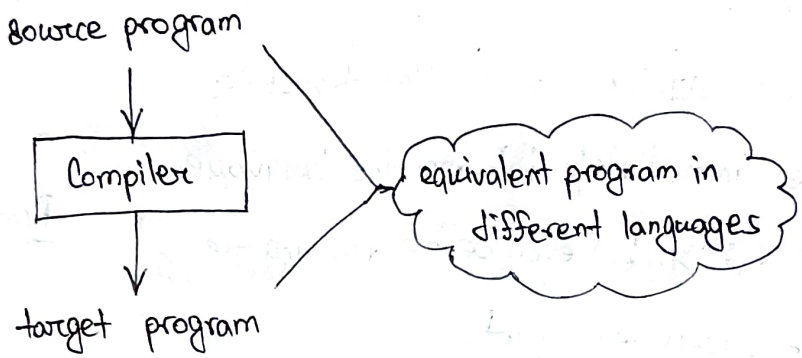


Figure 1.1: A compiler

✱ User can call the executable machine-language target program to process inputs and produce outputs;

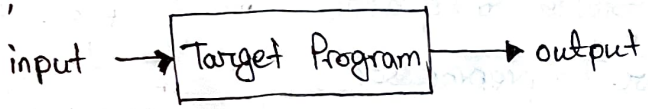


Figure 1.2: Running the target program.

✱ Interpreter directly executes the operations specified in the source program on inputs supplied by the user.

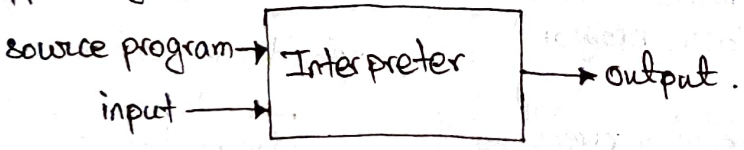


Figure 1.3: An Interpreter.

The machine language target program produced by a compiler is usually much faster than an interpreter at mapping inputs to outputs. An interpreter, however, can usually give better error diagnostics than a compiler, because it executes the source

program step-by-step. statement by statement.

Example 1.1

* Java language processors combine compilation and interpretation (Fig 1.4).
* Bytecodes ~~interpr~~ compiled on one machine can be interpreted on another machine (portable), perhaps across a network.
* Just-in-time (JIT) c Java compiler translates the bytecodes immediately into machine language, for faster processing, ~~bef~~ even before running the bytecodes to process the input.

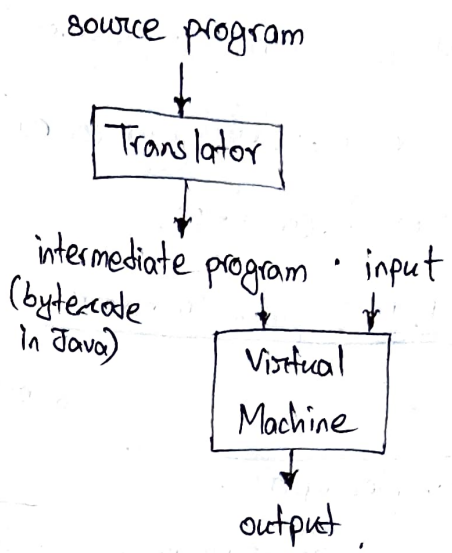


Figure 1.4: A hybrid compiler.

—o—

* A source program may be divided into modules stored in separate files. The task of collecting the source program is sometimes entrusted to a separate program, called a preprocessor. The preprocessor may also expand ~~#~~ shorthands, called macros, into source language statements. e.g.,

  #define m 100; (expand means replacing m with 100 in the source program)

* Large programs are often compiled in pieces, so the relocatable machine code may have to be linked together with other relocatable object files and library files into the code that actually runs on the machine.

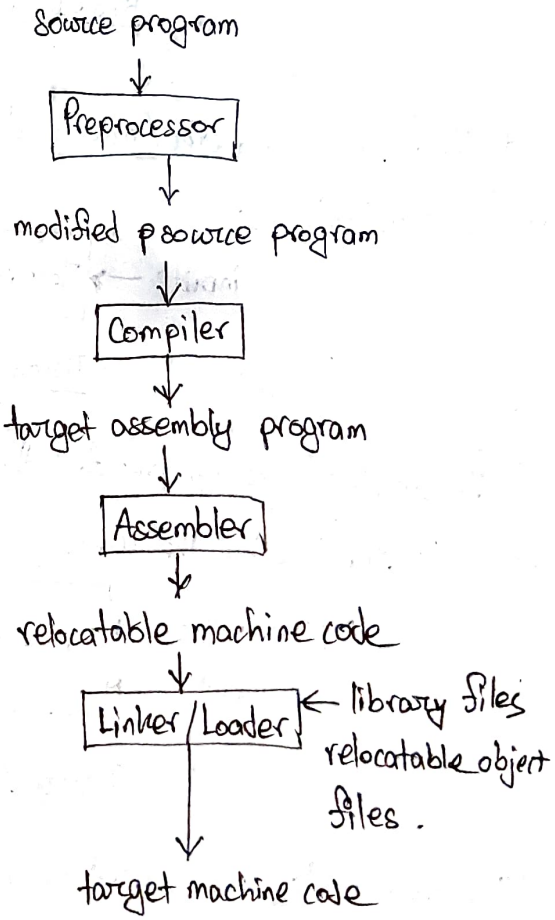


Figure 1.5: A language-processing system.

The linker resolves external memory addresses, where the code in one file may refer to a location in another file. The loader then ~~loads~~ puts together all of the executable object files into memory for execution.

* What is the difference between a compiler and an interpreter?

⇒

Aspect	Compiler	Interpreter.
Translation	Entire code at once.	Line-by-line or statement-by-statemer
Execution	Executes after full translation.	Executes directly as it transbates.
Error Handling	All errors at once, before execution.	Stops and reports errors during execution.
Speed (at runtime)	Typically faster.	Slower due to continuous transbation.
Output	Generates a separate machine code file	Does not generate machine code file.

A compiler is a program that can read a program in one language - the source language - and translate it into an equivalent program in another language - the ~~tag~~ target language and report any errors in the source program that it detects during the translation process.

Interpreter directly executes the operations specified in the source program on inputs supplied by the user.

* What are the advantages of (a) a compiler over an interpreter. ; (b) an interpreter over a compiler.?

⇒

(a)

i) Faster execution. as no need of translation during execution. (pre-compilati

ii) Compilers can apply advanced optimization techniques to produce optimized code. This increases performance and efficiency.

iii) Portability; after single translation, the executable can be distributed and ru multiple times without needing the source code or reinterpreting it.

iv) Catches all errors before execution.

v) Security; the original source code is not required for execution. So, no need of distribution allows protection for the source code.

vi) Can generate hardware-specific optimized code.

(b)

i) Immediate execution allows, for immediate feedback during development. Need of this, is in, interactive development, testing, rapid prototyping, etc.

ii) Easily cross-platform if interpreter exists.

iii) ~~Easir~~ Easier debugging. ~~Easir~~ Easier to spot runtime errors as they occur.

Compiler and interpreter each has strengths depending on the use case:

→ compilers are better for performance-critical applications.

→ interpreters are prefered for rapid development and scripting.

* What advantages are there to a language-processing system in which the compiler produces assembly language rather than machine language?

⇒

Advantages	Explanation
Human Readability	Easier to understand than machine code, making it more human-friendly.
Easier Debugging	Assembly is easier to debug with debuggers and diagnostic tools.
Platform-specific, Tool-portable	Assembly is platform specific but works with a wide range of assemblers.
Separation of concerns	Simplifies compiler design by offloading machine code generation to assembler.
Assembler optimizations	Assemblers can apply additional optimizations that the compiler doesn't.
Modular Design	Allows for modularity in the development toolchain.
Cross-compilation	Enables generation of code for different platforms using the same compiler.
Custom Assembly Tuning	Allows fine tuning of performance-critical code by modifying assembly.

so,
→ assembly language is easier to produce as output.
→ it is easier to debug.

* A compiler that translates a high-level language into another high-level language is called a source-to-source translator. What advantages are there to using C as a target language for a compiler?

⇒

For the C language there are many compilers available that compile to almost every hardware.

By targeting C, a compiler can gain significant advantages in terms of portability, optimization, simplicity, and interoperability, while also leveraging the broad and well-established ecosystem surrounding the C language. This makes it an attractive choice for many source-to-source compilers.

* Describe some of the tasks that an assembler needs to perform.

⇒

It translates from the assembly language to machine code (for a specific hardware platform). This machine code is relocatable.

① Symbol resolution: Resolves labels and symbols to memory addresses.

② Opcode translation: Converts mnemonics (e.g., MOV, ADD) into machine codes.

③ Address calculation: Computes memory addresses and instruction offsets.

④ Literal handling: Encodes constant values (literals) into binary form.

⑤ Data directives handling: Allocates memory and encodes variables and data.

⑥ Error checking: Ensures syntax and semantic correctness in assembly code.

⑦ Register and operand encoding: Translates register names and operands into binary.

⑧ Sections handling: manages code, data, and uninitialized data sections (e.g., .text, data, .bss).

⑨ Macro processing: expands macros into assembly instructions.

⑩ Relocation information: generates data for adjusting addresses during linki
-g.

⑪ Object file generation: produces object files containing machine code and symbo
data.

⑫ Assembler directives: handles commands like memory alignment, data inclusion,
and constants.

1.2 The Structure of a Compiler.

A compiler can be broken down into two key components: analysis and synthesis.

⇒ The analysis part, or the front end, processes the source program by breaking it into components, imposing grammatical structure, and creating an intermediate representation. It checks for syntactic and semantic errors, provide error messages if necessary and collects information in a symbol table.

* The synthesis part, or the back end, uses the intermediate representation (IR) and symbol table to generate the target program. The compilation process is divided into phases, each transforming the program's representation. Some compilers include a machine-independent optimization phase between the front and back ends, which improves the IR to produce a more efficient target program. Optimization is optional, and some phases may be grouped together without explicitly constructing IRs. The symbol table is shared across all phases.

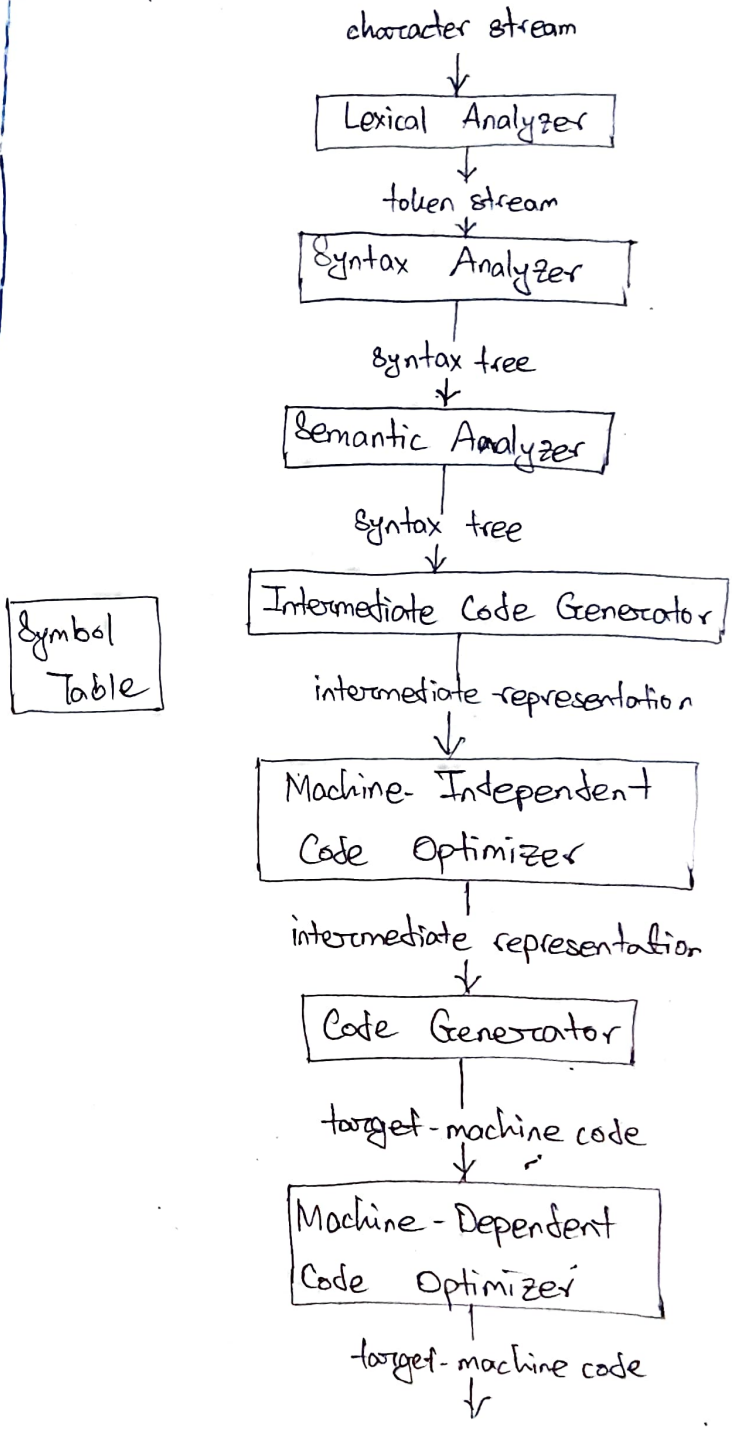


Figure: Phases of a compiler.

1.2.1 Lexical Analysis (Scanning)

* ⇒ First phase of a compiler is called lexical analysis or scanning. The lexical analyzer reads the stream of characters making up the source program and groups the characters into meaningful sequences called ~~lexemes~~ lexemes. For each lexeme, the lexical analyzer produces as output a token of the form,

<token-name, attribute-value> ; it is next passed to syntax analyzer.

- token-name: abstract symbol that is used during syntax analysis
- attribute-value: points to an entry in the symbol table for this token.

* ⇒ Example:

position = initial + rate * 60 ⇒ [Lexical Analyzer] ⇒ <id,1> <=> <id,2> <+> <id,3> <*> <60>

just <=>, because this token needs no attribute value, so the second component is omitted. We could have used "assign" as token name, but didn't for notational convenience.

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

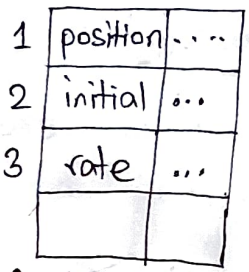
Symbol Table

* ⇒ Blanks separating the lexemes could be discarded by the lexical analyzer.

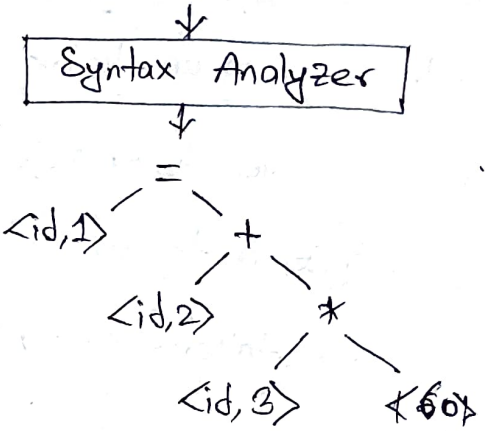
1.2.2 Syntax Analysis.

* ⇒ The second phase of the compiler is syntax analysis or parsing.

<id,1> <=> <id,2> <+> <id,3> <*> <60>



Symbol Table



* ⇒ The parser uses the first components of the tokens produced by the lexical analyzer to create a tree-like IR that depicts the grammatical structure of the token stream, the tree is called syntax tree.

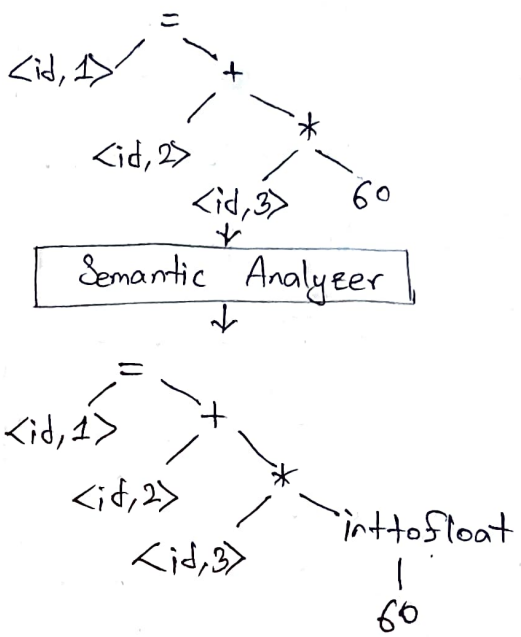
* ⇒ In syntax tree,

interior node = operation

children of " = arguments of the operation.

1.2.3 Semantic Analysis

The semantic analyzer ensures that a program is semantically consistent with the language's rules by using the syntax tree and the symbol table. It checks for type consistency, gathers type information, and stores it for later use in intermediate-code generation. A key part of this process is type checking, where the analyzer verifies that operators are applied to operands of matching types. For example, an array index must be an integer, and the analyzer will report an error if a floating-point number is used instead.



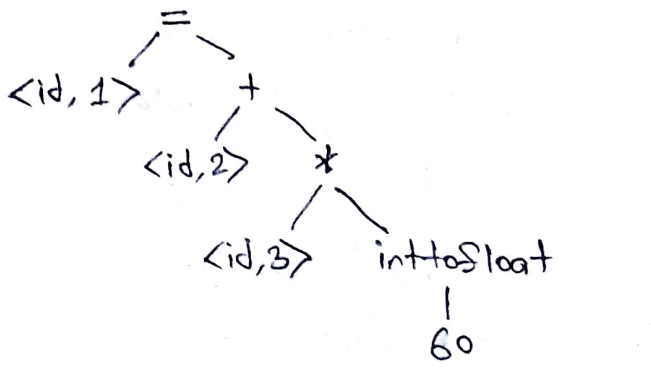
The language may allow type coercion, where the compiler converts data types to ensure compatibility. For instance, if a float number is multiplied by an integer, the compiler may automatically convert the integer to a floating-point number. The semantic analyzer would add an explicit conversion node (e.g., inttofloat) in the syntax tree to represent this conversion. This process helps maintain the program's type integrity during execution.

1.2.4 Intermediate Code Generation

In intermediate code generation phase many compilers generate an explicit low-level or machine-like intermediate representation (IR) which functions like a program for an abstract machine. This IR should have 2 important properties.

① Easy to produce, &

② Should be easy to translate into the target machine.



Intermediate Code Generation

$t1 = \text{inttofloat}(60)$

$t2 = id3 * t1$

$t3 = id2 + t2$

$id1 = t3$

(Three-address code ⇒ assembly-like instructions).

1.2.5 Code Optimization

⇒ This phase attempts to improve the intermediate code to get better target code.

- time (faster)
- space (shorter and less memory)

⇒ Here, optimizer replaced the integer 60 by the floating-point number 60.0.

$t1 = \text{inttofloat}(60)$

$t2 = id3 * t1$

$t3 = id2 + t2$

$id1 = t3$

↓

Code Optimizer (Machine Independent)

↓

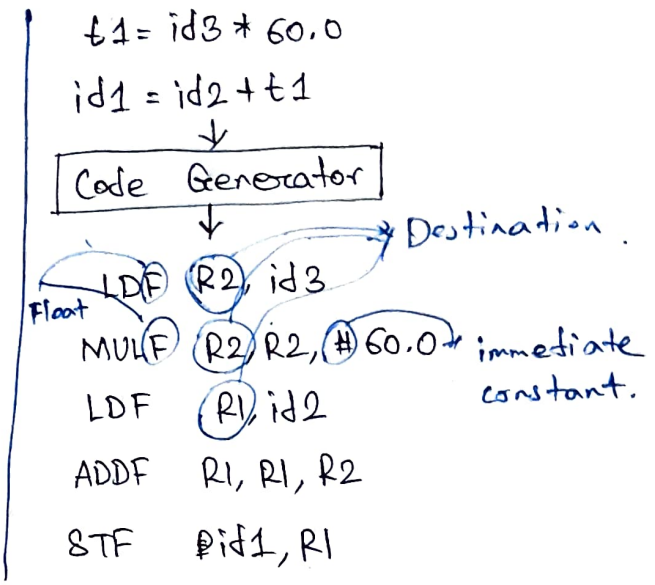
$t1 = id3 * 60.0$

$id1 = id2 + t1$.

1.2.6 Code Generation :

⇒ The code generator takes an intermediate representation (IR) of the source program and maps / translates it into the target language.

⇒ The organization of storage at run-time depends on the language being compiled. Storage allocation decisions are made either during intermediate ~~immediate~~ code generation or during code generation.



1.2.7 Symbol-table Management

⇒ An essential function of a compiler is to record the variable names used in the source program and collect information about various attributes of each name.

- storage allocation
- type
- scope
- in case of procedure
  - number and types of arguments
  - argument passing method → value / reference
  - type returned.

⇒ The symbol table is a data structure containing a record for each variable name, with fields for the attributes of the name.

Chapter 3

Lexical Analysis

3.1 The Role of the Lexical Analyzer.

* The main task of the lexical analyzer is to
  - read the input characters of the source program.
  - group them into lexemes, and
  - output: a sequence of tokens for each lexeme in the source program, these are sent to syntax analyzer.

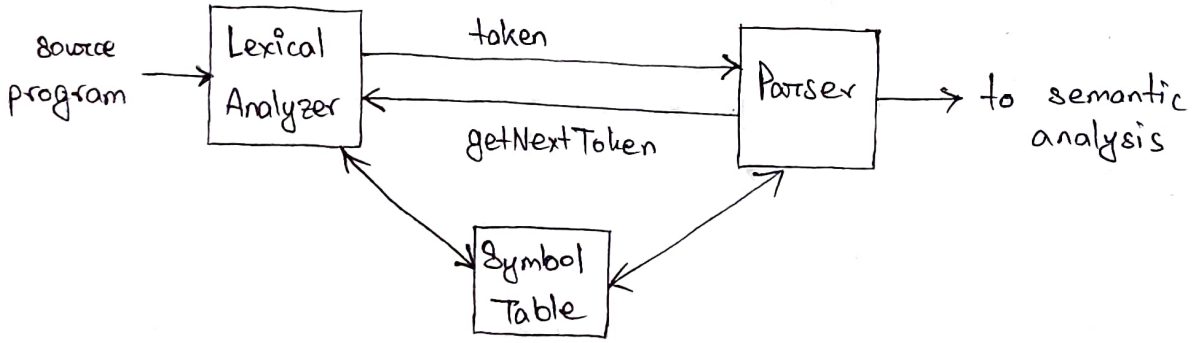


Figure: Interactions between the lexical analyzer and the parser.

* Side-tasks of a lexical analyzer as a part of the compiler,
  - Stripping out comments and whitespace (blank, newline, tab, and perhaps other characters that are used to separate tokens in the input.)
  - Correlating error messages generated by the compiler with the source program.
  - Expansion of macros may also be performed by the lexical analyzer.

Sometimes, lexical analyzers are divided into a cascade of two processes:

(a) Scanning; e.g., deletion of comments and compaction of consecutive whitespace characters into one.

(b) Lexical analysis; produces tokens from the output of the scanner.

* Why separate lexical analysis and parsing?

⇒

① Simplicity of the design. When designing a new language, separation of lexical and syntactic concerns can lead to a ~~clear~~ cleaner overall language design.

② Improvement of compiler efficiency.

③ Compiler portability is enhanced.

3.1.2 · Tokens, Patterns, and Lexemes.

- Token: A token is a pair consisting of a token name and an optional attribute value. The token name is an abstract ± symbol representing a kind of lexical unit, e.g., a particular keyword, or a sequence of input characters denoting an identifier. The token names are the input symbols that the parser processes.

- Pattern: A pattern is a description of the form that the lexemes of a token may take.

- Lexeme: A lexeme is a sequence of characters in the source program that matches the pattern for a token and is identified by the lexical analyzer as an instance of that token.

Example:

printf("Total= %d\n", score);

Tokens	Informal descriptions (patterns)	Sample lexemes
if	characters i, f	if
else	characters e, l, s, e	else
comparison	< or > or <= or >= or == or !=	<=, !=
id	letter followed by letters and digits	pi, score, D2
number	any numeric constant	3.14159, 0, 6.02e23
literal	anything but ", surrounded by "'s	"core dumped

Here, both printf and score are lexemes matching the pattern for token 'id', and "Total= %d\n" is a lexeme matching "literal".

—o—

* Following classes cover most or all of the tokens:

① One token for each keyword. The pattern for a keyword is the keyword itself. (e.g. if, else.)

② Tokens for the operators, either individually or in classes such as the token "comparison" mentioned in the table above.

③ One token representing all identifiers.

④ One or more tokens representing constants, such as numbers and literal strings.

⑤ Tokens for each punctuation symbol, such as left and right parentheses, comma, and semicolon.

.1.3 Attributes for Tokens.

→ Attribute value describes the lexeme represented by the token; the token name ...ences parsing decisions, while the attribute value influences ~~th~~ translation of tokens ... e after the parse.

→ Normally, attribute information is kept in the symbol table. So, attribute value ...n be a pointer to the symbol-table entry.

Example: Write the token names and associated attribute values for the Fortran statement: E = M* C ** 2 ..

⇨

<id, pointer to symbol-table entry for E>
<assign_op>
<id, pointer to symbol-table entry for M>
<mult_op>
<id, pointer to symbol-table entry for C>
<exp_op>
<number, integer value 2> | <constant, pointer to symbol-table entry for 2>

* The sentinel is a special character that cannot be part of the source program, and a natural choice is the character $ eof.

3.3 Specifications of Tokens.

3.3.1 Strings and Languages.

* An alphabet (Σ) is any finite set of symbols. (letters, digits, and punctuation etc.)
  - → The set $\{0,1\}$ is the binary alphabet.
  - → ASCII is an important example of an alphabet.
  - → UNICODE too.

* A string over an alphabet is a finite sequence of symbols drawn from that alphabet.
  - → The length of a string s, usually written $|s|$.
  - → The empty string, denoted ϵ, is the string of length zero.

* A language is any countable set of strings over some fixed alphabet.
  - → Abstract languages like $\emptyset$, the empty set, or $\{\epsilon\}$, is the set containing only the empty string, are languages under this definition.

* Terms for parts of Strings.
  - → Suppose a string s = "banana".
  - → Prefix of s is any string obtained by removing zero or more symbols from the end of s. e.g., ban, banana, and ϵ.
  - → Suffix of s is similar to prefix, but from beginning of s. e.g., nana, banana, ϵ.
  - → ϵ can be a substring of s.
  - → Proper prefixes, suffixes, and substrings of s are not ϵ or s itself.

→ A subsequence of s is any string formed by deleting zero, or more not necessarily consecutive positions of s. For example, baan is a subsequence of banana. However, order of the existing positions will remain same.

(*) → If x = dog and y = house, the ~~conta~~ concatenation of x and y, denoted as xy is -

$$xy = doghouse$$

→ For any string s,

$$\epsilon s = s\epsilon = s.$$

(*) $s^0 = \epsilon \mid s^{i-1}.s = s^i \mid s^1 = s,\ s^2 = ss,\ s^3 = sss, \ldots$, and $s^n = sss\ldots s.$

3.3.2 Operations on Languages.

- Union
- Concatenation
- Closure; Kleen closure ~~of a lang~~.

Concatenation: All strings formed by taking a string from the first language and a string from the second language, in all possible ways, and concatenating them.

Closure: (*) The (Kleene) closure of a language L, denoted L^*, is the set of strings got by ~~cont~~ concatenating L zero (L^0) or more times ($L^1, L^2, \ldots$).
↳ $\{\epsilon\}$

(*) The positive closure, denoted L^+, is the same as the Kleene closure, but without the term L^0. That is, ϵ will not be in L^+ unless it is in L ~~itself~~ itself.

Operation	Definition & notation
Union of L and M	$L \cup M = \{s \mid s \text{ is in } L \text{ or } s \text{ is in } M\}$
Concatenation of L and M	$LM = \{st \mid s \text{ is in } L \text{ and } t \text{ is in } M\}$
Kleene closure of L	$L^* = \bigcup_{i=0}^{\infty} L^i$
Positive closure of L	$L^+ = \bigcup_{i=1}^{\infty} L^i$

Example

$L = \{A, B, \ldots, Z, a, b, \ldots, z\}$

$D = \{0, 1, \ldots, 9\}$

Here, L and D

- can be thought of as alphabets. Also
- as languages whose all strings are of length one.

Let's construct languages using L and D with some operations.

① LUD - a language with 62 strings of length one.

$LUD = \{A, B, \ldots, Z, a, b, \ldots, z, 0, 1, \ldots, 9\}$.

② LD is the set of 520 strings of length two, each consisting of one letter followed by one digit.

$LD = \{A0, A1, A2, \ldots, Z9, \ldots, Z9\}$.

③ L^4 is the set of all 4-letter strings.

④ L^* " " " " " strings of letters, including ϵ, the empty string.

$L^* = \{\epsilon, A, \ldots, Z, \ldots, AaaA, \ldots, ZZZ, \ldots, ZZZ\}$.

⑤ $L(LUD)^*$ is the set of all strings of letters and digits beginning with a letter.

⑥ D^+ is the set of all strings of one or more digits.

$L(LUD)^* = \{a, A, \ldots, Z, \ldots, aaAA, \ldots, a1a2ab9, \ldots\}$.

$D^+ = \{0, 1, \ldots, 9, 00, 11, \ldots, 0123, \ldots\}$

3.3.3 Regular Expressions.

* The language of C identifiers can be described as:

letter_ (letter_ | digit)* [vertical line means union (bar).]

* '*' means zero or more times.
* Each regular expression r denotes a language L(r).
* There are two rules that form the basis:

(1) ε is a regular expression, and L(ε) is {ε}, that is, the language whose sole member is the empty string.

(2) If 'a' is a symbol in Σ, then 'a' is a regular expression, and L(a) = {a}, that is, the language with one string of length 1, with 'a' in its one position.

* Induction: larger regular expressions are built from smaller ones. Suppose, r and s are regular expressions denoting languages L(r) and L(s) respectively.

(1) (r)|(s) is a regular expression denoting the language L(r) ∪ L(s)

(2) (r)(s) " " " " " " " L(r)L(s)

(3) (r)* " " " " " " (L(r))*

(4) (r) " " " " " " L(r). ; we can add additional pairs of parenthesis around expressions without changing the language they denote.

* In regular expressions,

a) The unary operator * has highest precedence and is left associative.

(b) Concatenation has second highest precedence and " " ".

(c) | has lowest precedence and is left associative.

(a)|((b)*(c)) => a | b*c

(Denotes the set of strings that are either a single 'a' or are zero or more b's followed by a c.)

Example:

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

① Regular expression a|b denotes the language $\{a, b\}$.

② Regular expression (a|b)(a|b) or aa|ab|ba|bb denotes $\{aa, ab, ba, bb\}$, the language of all strings of length 2 over the alphabet Σ.

③ a* denotes $\{\epsilon, a, aa, aaa, \ldots\}$.

④ (a|b)* denotes $\{\epsilon, a, b, ab, ba, aaa, bbbb, \ldots\}$ - set of all strings consisting of zero or more instances of 'a' or 'b'.

⑤ a|a*b denotes the language $\{a, b, ab, aab, aaab, \ldots\}$.

A language that can be defined by a regular expression is called a regular set

If two regular expressions r and s denote the same regular set, they are equivalent and r = s, e.g., (a|b) = (b|a)

Law	Description
$r\|s = s\|r$	$\|$ is commutative
$r\|(s\|t) = (r\|s)\|t$	$\|$ is associative
$r(st) = (rs)t$	Concatenation is associative
$r(s\|t) = rs\|rt ; (s\|t)r = sr\|tr$	Concatenation distributes over $\|$.
$\epsilon r = r\epsilon = r$	ϵ is the identity for concatenation.
$r^* = (r\|\epsilon)^*$	ϵ is guaranteed in a closure.
$r^{**} = r^*$	$*$ is idempotent.

3.3.4 Regular Definitions.

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

$d_1 \rightarrow r_1$
$d_2 \rightarrow r_2$
$d_3 \rightarrow r_3$
$\vdots$
$d_n \rightarrow r_n$.

where,

① Each d_i is a new symbol, not in Σ and not the same as any other of ~~ds~~ the d's, and

② Each r_i is a regular expression over the alphabet $\Sigma \cup \{d_1, d_2, \ldots, d_{i-1}\}$.

Example 3.5
C identifiers are strings of letters, digits, and underscores. Here is a regular definition for the language of C identifiers.

$$\underset{d_1}{\text{letter\_}} \rightarrow \underset{r_1}{A|B|\ldots|Z|a|b|\ldots|z|\_}$$

$$\underset{d_2}{\text{digit}} \rightarrow \underset{r_2}{0|1|\ldots|9}$$

$$\underset{d_3}{\text{id}} \rightarrow \underbrace{\text{letter\_}(\text{letter\_}|\text{digit})^*}_{r_3} \quad [r_i \in \Sigma \cup \{d_1, d_2, \ldots d_{i-1}\}]$$

Example 3.6

Unsigned numbers (integer or floating point) are strings such as 5280, 0.01234, 6.336E4, or 1.89E-4. The regular definition,

$$digit \rightarrow 0 \mid 1 \mid \dots \mid 9$$

$$digits \rightarrow digit\ digit^*$$ [if, digits → digit* then ε can come many times. And it won't be valid.]

ε/ .1/ .005/ .20/ .11845 etc. ↖

$$optionalFraction \rightarrow .\ digits \mid \epsilon$$ [~~it could've been . digit*~~] . followed by digits not ε.]

$$optionalExponent \rightarrow (E(+ \mid - \mid \epsilon)\ digits) \mid \epsilon.$$ [E+4, E-4, E2, ε]

$$number \rightarrow digits\ optionalFraction\ optionalExponent.$$ (1.0E5)

is a precise ~~definit~~ specification for this set of strings.

[At least a digit must follow the dot.]

3.3.5 Extensions of Regular Expressions.

① One or more instances: If r is a regular expression, then $(r)^+$ denotes the language $(L(r))^+$. + has the same precedence as *. (left associative).

$$r^* = r^+ \mid \epsilon.$$

$$r^+ = rr^* = r^*r.$$ [ε এর সাথে যা concatenate করি তাই হবে final, $\epsilon r = r\epsilon = r$.]

② Zero or one instance: The unary postfix operator ? means "zero or one occurrence."

$r?$ is equivalent to $r \mid \epsilon$.

So,

$$L(r?) = L(r) \cup \{\epsilon\}.$$

'?' has the same precedence and associativity as * and +.

③ Character classes: When $a_1, a_2, \ldots, a_n$ form a logical sequence, e.g., consecutive uppercase letters, lowercase letters, or digits, we can replace them by a_1-a_n. that is, just the ~~last and~~ first ~~seq~~ and last separated by a hyphen.

$a_1|a_2|\ldots|a_n$ can be replaced by $[a_1\text{-}a_n]$

$a|b|\ldots|z$ has the shorthand $[a-z]$

Example 3.7

⇒ letter_ → a A|B|...|Z|a|b|...|z|_ | letter → [A-Za-z_]

digit → 0|1|...|9 | digit → [0-9]

digits → digit digit* | digits → digit+

Ex 3.6 number → digits optionalFraction optionalExponent | ~~number → digits (.digits)? (E[+-]?~~

number → digits (.digits)? (E[+-]? digits)?

Exercise 3.3.2

Describe the languages denoted by the following regular expressions:

(a) $a(a|b)^*a$

(b) $(\epsilon|a)b^*)^*$

(c) $(a|b)^*a(a|b)(a|b)$

(d) $a^*ba^*ba^*ba^*$

(e) $(aa|bb)^*((ab|ba)(aa|bb)^*(ab|ba)(aa|bb)^*)^*$

(a) String of two or more a and zero or more b.; all of which, starts and end with a.

e.g., {aa, aaaba, abbba, aaaaa, ...}

(b) The language can have empty string, and strings of a's and b's. e.g.,

$L = \{\epsilon, a, aa, b, bbb, ab, abab, abbbabbb, \ldots\}$.

(c) Strings of a's and b's where the third character from the last is a.

(d) Strings of a's and b's with only three b's.

(e) String of as and bs ~~with~~ with even number of a and b.

Substring defined as a contiguous part of a string.

Exercise 3.3.3

In a string of length n, how many of the following are there?

⇒

(a) Prefixes → (n+1) [1 added because of ϵ]

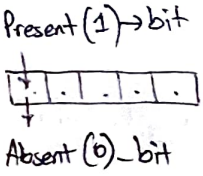
(b) Suffixes → (n+1)

(c) Proper prefixes → (n-1) [Excluding string itself and ϵ]

(d) # Substrings → $\frac{n(n+1)}{2} + 1$ [Including ϵ]

For a string of size n, there are $\frac{n(n+1)}{2}$ non-empty substrings.

(e)



⇒ 2^5 combinations. (including string itself and ϵ).

Subsequences → 2^n or, $\sum_{i=0}^{n} C(n,i)$

Exercise 3.3.4

Show how to write a regular expression for a keyword in case-insensitive language by writing the expression for "select" in SQL.

⇒

select → [Ss][Ee][Ll][Ee][Cc][Tt]

Exercise 3.3.5

Write regular expressions definitions for the following languages:

(g) All strings of lowercase letters that contain the five vowels in order.

⇒

other → [b-df-hj-np-tvwxyz] aeiou

want → other* a (other|a)* e (other|e)* i (other|i)* o (other|o)* u (other|u)*

All strings of lowercase letters in which the letters are in ascending lexicographic order

want → a* b* ... z*

) Comments, consisting of a string surrounded by /* and */, without an intervening

/, unless it is inside double-quotes (").

⇒

want → \/*([^*"]*|".*")

want → \/*([^*"]*|".*"|*+[^/])**\/

matches one or more * character followed by a character that is not /.

ensures */.

repeat zero or more times everything.

→ \/ : escapes the character (/), ensuring it matches a literal /.

* : escapes the * character, ensuring it matches a literal *.

Together, \/* matches the exact sequence /* ; the start of a comment.

→ [^*"]* : matches any sequence of characters that are neither * or ". This avoids accidently forming the sequence */ or interfering with quoted strings.

→ ".*" : matches a string enclosed in double quotes. (*/ can apper in double quotes).

Or,

want → \/*([^*]|*+[^*/])|"([^"\]|\(.))*"**\/

allowing any characters in the string.

anything except *

many * however no */

if not in " "

in double quotes ", matches any character except " or \. Avoiding unescaped " or \. As \ used to escape sequence.

(d) All strings of digits with no repeated digits. $\Sigma = \{0,1,2\}$.

first → [0-2]

second → [^first]

third → [^first][^second]

want → first second third.

(Do it, and the rest)

Exercise 3.3.6. Write the character classes for the following sets of characters:

(a) The first ten letters (up to "j") in either upper or lower case.

→ [A-Ja-j]

(b) The lowercase consonants

→ [bcdf-hj-np-tv-z]

(c) The "digits" in a hexadecimal number (choose either upper or lower case for the "digits" above 9).

→ [0-9a-fA-F]

(d) The characters that can appear at the end of a legitimate English sentence (e.g., exclamation point).

→ [.!?]

Exercise 3.3.7. Write a regular expression that matches the string "\.

→ \"\\.

Exercise 8.3.8 Complemented character class.

[^A-Za-z] matches any character that is not any uppercase or lowercase letter.

[^\n] represents any character but the caret (or newline, since newline cannot be in any character class).

— o —

* Lex regular expressions :

Expression	Matches
c	the one non-operator character c
\c	character c literally.
"s"	string s literally.
.	any character but newline.
^	beginning of a line
$	end of a line
[s]	any one of the characters in string s
[^s]	any one character not in string s.
r*	zero or more strings matching r.
r+	one or more strings matching r.
r?	zero or one r
r{m,n}	between m and n occurrences of r.
$r_1 r_2$	an r_1 followed by an r_2.
$r_1 \mid r_2$	an r_1 or an r_2.
(r)	same as r
r_1 / r_2	r_1 when followed by r_2.

* a{1,5} matches a string of one to five a's.

(Do the rest).

3.4 Recognition of Tokens.

digit → [0-9]

digits → digit+

number → digits(. digits)? (E [+ -]? digits)?

letter → [A-Z a-z]

id → letter (letter | digit)*

reserved words are not identifier, though their pattern matches that of id.

if → if

then → then

else → else

comparison operator

relop → < | > | <= | >= | = | <>

not equal in Fortran and its like

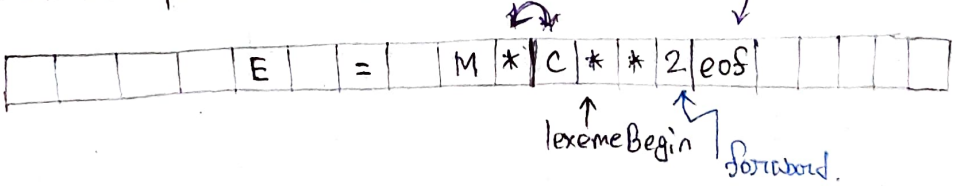
3.2 Input Buffering

* Sometimes we are needed to look ahead to be sure about the lexemes pattern.

* Buffer can ~~not~~ handle this lookaheads safely.

3.2.1

Two pointers to the input are maintained:



① Pointer lexemeBegin, marks the beginning of the current lexeme, whose extent we are attempting to determine.

② Pointer forward scans ahead until a pattern match is found; ~~the exact strategy~~ Once the next lexeme is determined, forward is set to the character at its right end. Then, after the lexeme is recorded as an

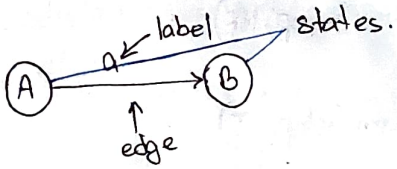
attribute value of a token returned to the parser, lexemeBegin is set to the character immediately after the lexeme just found.

3.4.1 Transition Diagrams.

↳ an intermediate step in the construction of a lexical analyzer. where we convert patterns into stylized flowcharts, called "transition diagrams".

↳ Transition diagrams have a collection of nodes or circles, called states. which represents a condition that could occur during the process of input scanning.

↳ Edges are directed from one state of the transition diagram to another and are labeled by a symbol or set of symbols.



↳ Deterministic transition diagram means that there is never more than one edge out of a given state with a given symbol among its labels.

↳ Some important conventions about transition diagrams are:

① Certain states are said to be accepting, or final. These states indicate that a lexeme has been found, although the actual lexeme may not consists of all positions between the lexemeBegin and forward pointers (in buffer). We always indicate an accepting state by a double circle (◎), and if there is an action to be taken – typically returning a token and an attribute value to the parser – we shall attach that action to the accepting state.

② In addition, if it is necessary to retract the forward pointer one position (i.e., the lexeme does not include the symbol that got us to the accepting state), then we shall additionally place an * near that accepting state. If it is necessary to retract more than one position, we could attach any number of *'s to the accepting state.

③ One state is designed the start state, or initial state; it is indicated by an edge, labeled "start", entering from nowhere. Transition diagram starts from here before any reading.

Example 3.9

Draw a transition diagram that recognizes the lexemes matching the token relop.

Lexemes	Token Name	Attribute Value
<	relop	LT
<=	relop	LE
=	relop	EQ
<>	relop	NE
>	relop	GT
>=	relop	GE

⇒

In figure, we begin in state 0, the start state. If we see < as the 1st symbol input symbol, then the lexemes that match the pattern are <, <>, or <=. We therefore go to state 1,

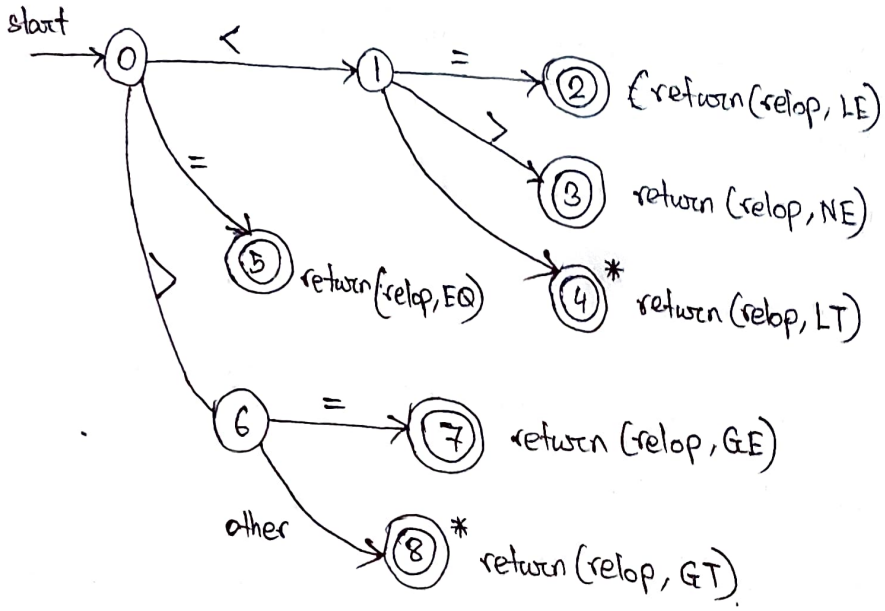


Figure: Transition diagram for relop.

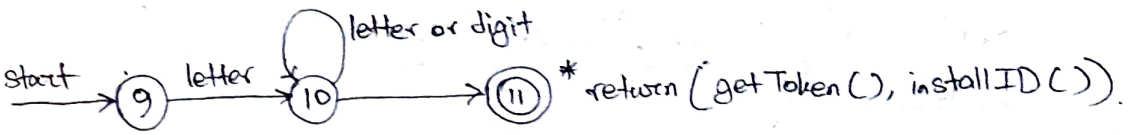
and look at the next input character. If it is = then we recognize lexeme <=, enter state 2, and return the token relop with attribute LE, the symbolic constant representing this particular comparison operator. Similarly, when the next input character is > when we were in step 1. On any other input character, the lexeme is <, and we enter state 4 to return that information. Note, however, that state 4 has an * to indicate that we must retract the input one position. ▯▯▯▯ ← retract.

Similary, all other lexemes are recognized. If in state 0, we see any character besides <, =, or >, we cannot possibly be seeing a relop lexeme, so this transition diagram will be not be used. □

3.4.2 Recognition of reserved words and identifiers.

↳ Recognizing ~~keyw~~ keywords and identifiers presents a problem as they look same.

For example, the following transition diagram reconizes keywords as well as identifiers (id).

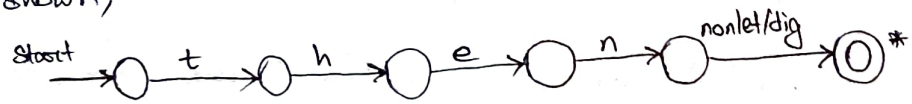


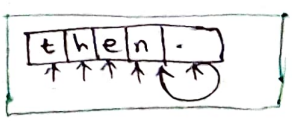
- Now we want distinction when recognizing keywords and identifiers.

↳ There are 2 ways that we can handle reserved words (if, else, etc.) that look like identifiers:

① Install the reserved words in the symbol table initially. A field of the symbol table entry indicates that these strings are never ordinary identifiers, and tells which token they represent. When we find an identifier, a call to installID places it in the symbol table if it is not already there and returns a pointer to the symbol-table entry for the lexeme found. Of course, any identifier not in the symbol table during lexical analysis cannot be a reserved word, so its token is id. The function getToken examines the symbol table entry for the lexeme found, and returns whatever token name the symbol-table says this lexemes represents- either id or one of the keyword tokens that was initially installed in the table.

② Create separate transition diagrams for each keyword; an example for the keyword 'then' is shown,





Such transition diagram consists of states representing the situation after each successive letter of the keyword seen, followed by a test for a "nonletter-or-digit", i.e., any character that cannot be the continuation of an identifier.

It is necessary to check that the identifier has ended, or else we would return token 'then' in situations where the correct token was 'id', with a lexeme like 'thennextvalue' that has 'then' as a proper prefix.

3.4.3 Completion of the running example

↳ The transition diagram for a token 'number' is shown,

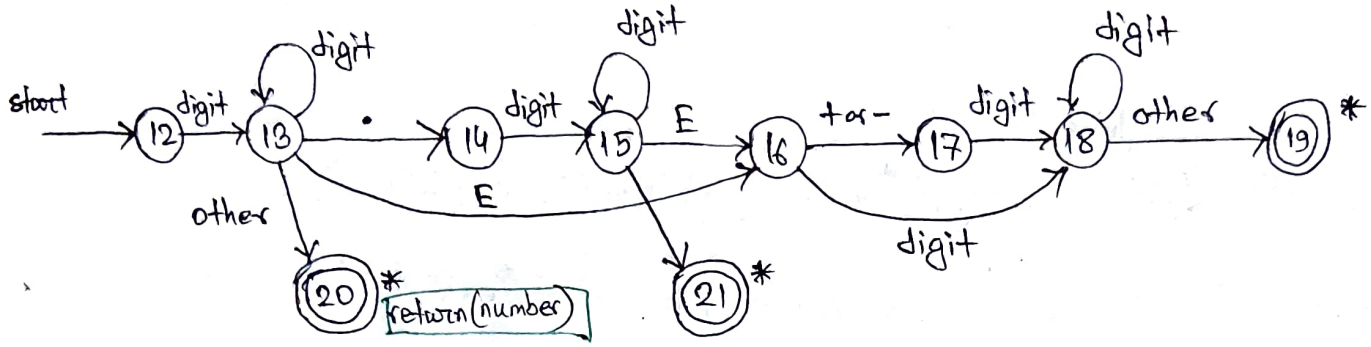


Figure: A transition diagram for unsigned numbers.

In state 20, we return a token 'number' and a pointer to a table of constants where the found lexeme is entered.

123E8
123.78E-3
etc.

↳ A transition diagram for whitespace is shown,

ws → (blank | tab | newline)+

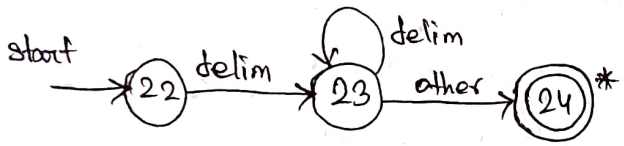


Fig: A transition diagram for whitespace.

In the diagram, we look for one or more "whitespace" characters, represented by delim in that diagram - typically these characters would be blank, tab, newline, and perhaps other characters that are not considered by the language design to be part of any token.

In state 24, we retract the input to begin at the nonwhitespace, but we do not return to the parser. Rather, we must restart the process of lexical analysis after the whitespace.

4 Architecture of a Transition-Diagram-Based Lexical Analyzer.

collection of transition diagrams can be used to build a lexical analyzer.

Transition diagram:

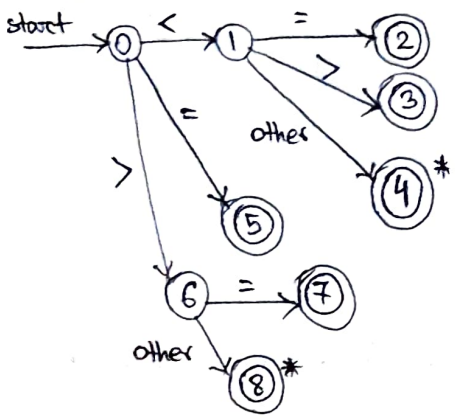


Figure: Transition diagram for relop.

Its sketch implementation in C++,

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

TOKEN is a pair consisting of the token name (which must be relop in this case) and an attribute value (the code for one of the six comparison operators in this case).

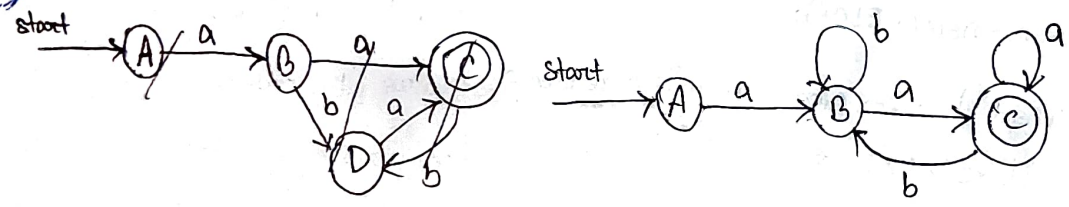
* Take the longest prefix of the input that matches any pattern. That rule allows us to prefer identifier thenext to keyword then, or the operator -> to -, for example.

Exercise 8.4.1 Provide transition diagrams to recognize the same languages as each of the regular expressions :-

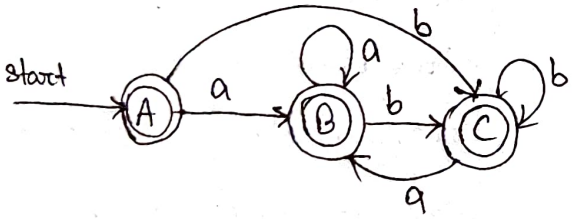
(a) $a(a|b)^*a$ | (b) $((\epsilon|a)b^*)^*$ | (c) $(a|b)^*a(a|b)(a|b)$ | (d) $a^*ba^*ba^*ba^*$

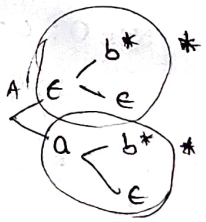
~~(b)~~

(a)

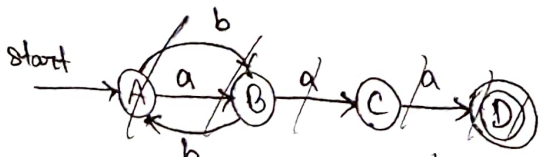


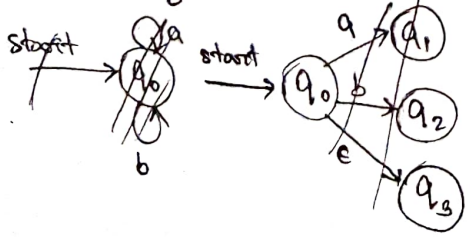
(b) $((\epsilon|a)b^*)^*$

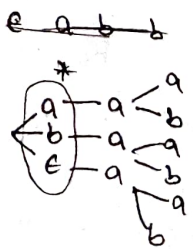




(c) $(a|b)^*a(a|b)(a|b)$







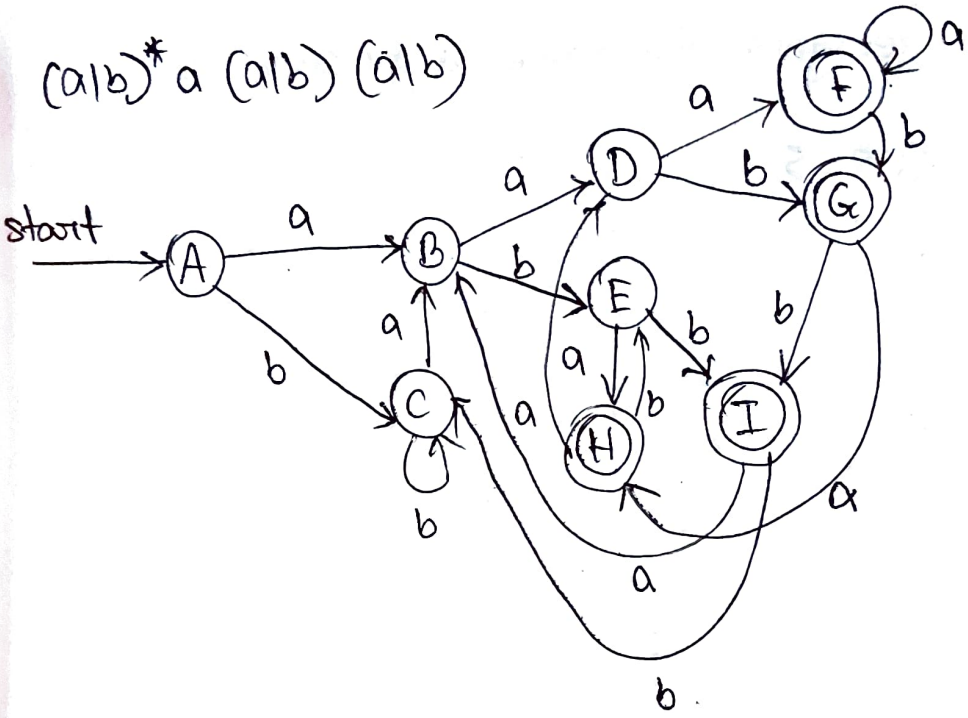
abb

aaa

aaaa

abb

$(a|b)^* a\ (a|b)\ (a|b)$



Chapter 4

Syntax Analysis

* A grammar gives a precise, yet easy-to-understand, syntactic specification of a programming language.

4.1 Introduction

4.1.1 The Role of the Parser

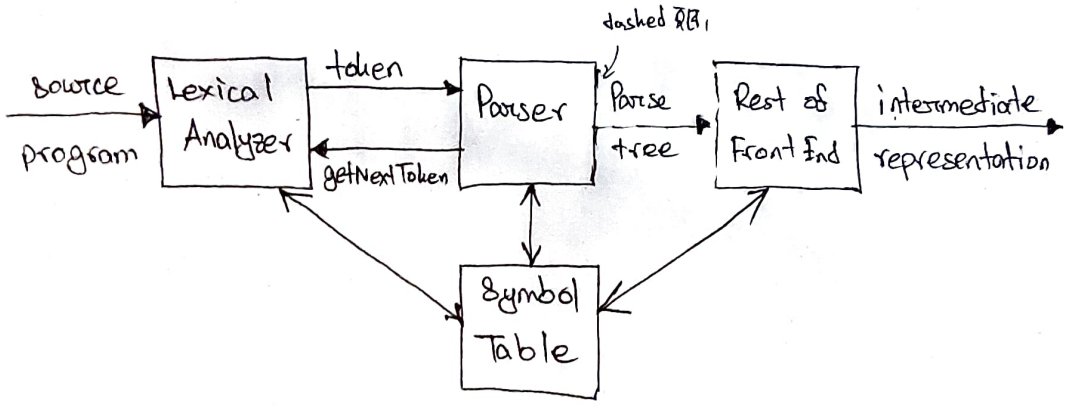


Figure: Position of parser in compiler model.

* The parser obtains a string of tokens from the lexical analyzer and verifies that the string of token names can be generated by the grammar for the source ~~program~~ language.

- reports any syntax errors in an intelligible fashion and recover from commonly occuring errors to continue processing the remainder of the program.
- Constructs a parse tree and passes it to the rest of the compiler for further processing.

* There are 3 general types of parsers for grammars: ~~universal~~

i) universal,

ii) top-down, and

iii) bottom-up.

→ Commonly used in compilers.

↳ Universal parsing methods can parse any grammar. However, they are too inefficient to use in production compilers.

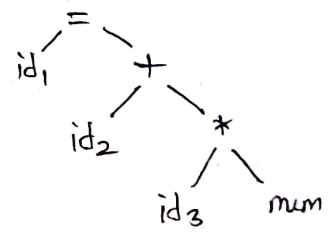
↳ Top-down methods build parse trees from the top (root) to the bottom (leaves)

↳ Bottom-up " " " " " " leaves and work their way up to the root.

In both top-down and bottom-up, the input to the parser is scanned from left to right, one symbol at a time.

position = initial + rate * 60;

$id_1 = id_2 + id_3 * num$;



4.1.2 Representative Grammars

$E \to E+T \mid T$

$T \to T*F \mid F$

$F \to (E) \mid id$ (4.1)

⇒ • LR grammars are suitable for bottom-up parsing.

↳ not for top-down parsing because it is left recursive.

$E \to TE'$

$E' \to +TE' \mid \epsilon$

$T' \to FT'$

$T' \to *FT'$

$F \to (E) \mid id$ (4.2)

⇒ • Non-recursive variant of (4.1)

• Will be used for top-down parsing.

4.1.3 Syntax Error Handling

Programming errors occur at various levels:

↳ Lexical errors: Misspelled identifiers/keywords or missing quotes (e.g., elipsesize instead of ellipsesize)

↳ Syntactic errors: Misplaced semicolons or braces or or a 'case' outside a 'switch'.

↳ Semantic errors: Type mismatches (e.g., return in a void method)

↳ Logical errors: Incorrect logic, like using = instead of == in C.

* Parsers have the viable-prefix property, meaning that they detect an error has occurred as soon as they see a prefix of the input that cannot be completed to form a string in the language.

* The error handler in a parser has goals that are simple to state but challenging to realize:

i) Report the presence of errors clearly and accurately.

ii) Recover from each error quickly enough to detect subsequent errors.

iii) Add minimal overhead to the processing of correct programs.

4.1.4 Error-recovery strategies

* No universally acceptable strategy.

* In general,

- Parsers may stop with an error message or try to recover for further diagnostics.
- Limit excessive errors make the compiler give up to avoid overwhelming outputs.

Recovery Strategies:

① Panic-Mode recovery: Discards input symbols until a synchronizing token (like ; or }) is found. It's simple, prevent infinite loops, but may skip large sections of input.

② Phrase-level recovery: Makes local corrections by replacing, deleting, or inserting tokens (e.g., replacing , with ;). It's effective but struggles when errors occur far from detection.

③ Error productions: Anticipates common errors by augmenting the grammar with rules for invalid constructs, allowing for specific error ~~mes~~ messages when such rules are applied.

④ Global correction: Attempts minimal changes to transform an incorrect input string into a valid one. Though theoretically optimal, it's computationally expensive and impractical for real compilers.

These techniques aim to strike a balance between error detection and meaningful diagnostics while avoiding overwhelming the programmer with unnecessary error messages.

4.2 Context-Free Grammars

* A context-free grammar, or CFG, consists of set of terminals, a set of nonterminals, a start symbol, and a set of productions.

① Terminals: Basic symbols (e.g., tokens like if, else, (,)), used to form strings. (Tokens)

② Nonterminals: Syntactic variables (e.g., stmt, expr) representing sets of strings and imposing hierarchical structure. Will eventually be "replaced" by terminals (tokens) matching a grammatical pattern.

③ Start symbol: A distinguished nonterminal whose strings define the grammar's language. Conventionally, the productions for the start symbol are listed first.

④ Productions: Rules specifying how terminals and nonterminals combine, ~~conbi~~ consisting of:

- Head: A nonterminal being defined. (On left)
- Arrow (→): Separates head and body. (→ or, :: = ~~are~~ is used too).
- Body: A combination of terminals and nonterminals forming valid strings. Empty string also included. (On right of arrow). The components of the body describe one way in which strings of the nonterminals at the head can be constructed.

If terminal symbols are,

id, + - * / ( )

The nonterminal symbols are E, T, and F, and E is the start symbol, then productions are-

$E \rightarrow E+T$

$E \rightarrow E-T$

$E \rightarrow T$

$T \rightarrow T*F$

$T \rightarrow F$

$F \rightarrow (E)$

$F \rightarrow id.$

(Grammar for simple arithmetic expressions).

4.2.2 Notational conventions

(1) Terminals:

a) Lowercase letters.

b) Operators

c) Punctuations.

d) Digits

e) Boldface strings such as **id.** or **if**, each of which represents a single terminal.

(2) Nonterminals:

(a) Uppercase letters

(b) Letter S is usually the start symbol

(c) Italic lowercase names.

(3) Late uppercase letters (e.g., X, Y, Z) represent grammar symbols (nonterminals or terminals)

(4) Lowercase letter late in alphabet (u e.g., u, v, ..., z) represent strings of terminals. (possibly empty).

(5) α, β, γ represents (possibly empty) strings of grammar symbols.

(6) $A \to \alpha_1, A \to \alpha_2, \ldots, A \to \alpha_k$ can be represented as:

$A \to \alpha_1 | \alpha_2 | \alpha_3 | \ldots | \alpha_k$.

(7) Unless stated otherwise, the head of the first production is the start symbol.

4.2.3 Derivations

* $E \to E+E \mid E*E \mid -E \mid (E) \mid id$

$E \to -E$
$\to -(E)$
$\to -(id)$

derivation in zero or more steps.

$E \xrightarrow{*} -(id)$

⇒ We call such a sequence of replacements a derivation of $-(id)$ from E. ⇒ The string $-(id)$ is one particular instance of an E.

Here, • A string of terminals ω is in $L(G)$ (grammar), the language generated by G, if and only if ω is a sentence of G (or $S \overset{*}{\Rightarrow} \omega$). A language that can be generated by a grammar is said to be a context-free grammar language. If two grammars generate the same language, the grammars are said to be equivalent.

For example, $-(id)$ is a sentence of grammar because there is a derivation.

* Here, we are deriving strings of terminals by • beginning with the start symbol,
• repeatedly replacing a nonterminal with the body of a production of which that nonterminal is the head, until the resulting string consists of only terminals.

* To understand how parsers work, we shall consider derivations in which the nonterminal to be replaced at each step is chosen as follows:

① In leftmost derivations, the leftmost nonterminal in each ~~sentential~~ sentential is always ~~choosen~~ chosen. If $\alpha \Rightarrow \beta$ is a step in which the leftmost nonterminal is replaced, we write $\alpha \underset{lm}{\Rightarrow} \beta$.

For, $E \to E+E \mid E*E \mid -E \mid (E) \mid id$, a leftmost derivation,

$$E \underset{lm}{\Rightarrow} -E \underset{lm}{\to} -(E) \underset{lm}{\to} -(E+E) \underset{lm}{\to} -(id+E) \underset{lm}{\to} -(id+id)$$

② In rightmost derivations, the rightmost nonterminal is always chosen; we write $\alpha \underset{rm}{\Rightarrow} \beta$ in this case.

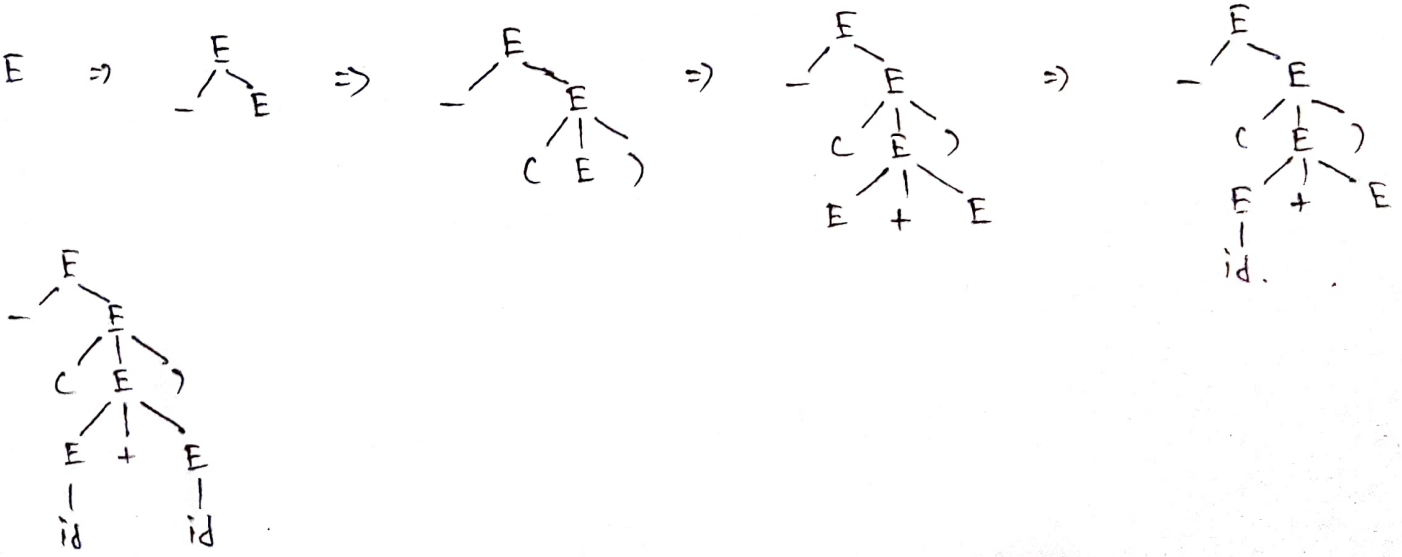
Example,

$$E \underset{rm}{\to} -E \underset{rm}{\to} -(E) \underset{rm}{\to} -(E+E) \underset{rm}{\to} -(E+id) \underset{rm}{\to} -(id+id)$$

Rightmost derivations are sometimes called ~~canno~~ canonical derivations.

4.2.4 Parse trees and derivations

- Sequence of parse trees for $E \underset{lm}{\overset{*}{\Rightarrow}} -(id+id)$,



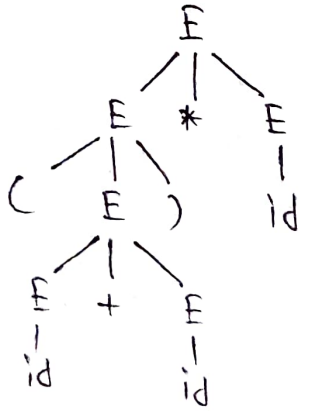
P.T.O

* A parse tree is a tree structure that represents a derivation.

- The root node is the start symbol.
- Every interior node is a nonterminal
- Every leaf node is a terminal.
- The children of each nonterminal node are the nonterminals and terminals of the body of a production for that ~~terminal~~ nonterminal.

For, productions,

E → E+E|E*E|(E)|id. A parse tree for the derivation of the string: (id+id)*id



4.2.5 Ambiguity

* A grammar that produces more than one parse tree for some sentence is said to be ambiguous. Or to say, an ambiguous grammar is one that produces more than one left-most derivation or more than one rightmost derivation for the same sentence. So, in this grammar, a particular string can be derived in more than one way.

For example, using the rules,

$$E \rightarrow E+E \mid E*E \mid (E) \mid id$$

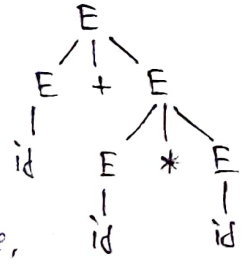
id+id*id can be derived as,

(a)	(b)
$E \rightarrow E+E$	$E \rightarrow E*E$
$\rightarrow id+E$	$E \rightarrow E+E*E$
$\rightarrow id+E*E$	$\rightarrow id+E*E$
$\rightarrow id+id*E$	$\rightarrow id+id*E$
$\rightarrow id+id*id$	$\rightarrow id+id*id$

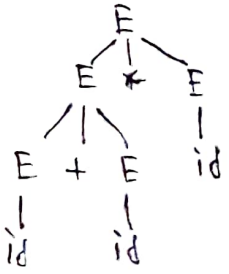
in both, using the leftmost derivations.

And their parse trees are,

When evaluating,
a+b*c
(a) is like,
a+(b*c)
while (b) is like,
(a+b)*c



(a)



(b)

* If grammar is ambiguous, parser cannot uniquely determine which parse tree to select for a sentence.

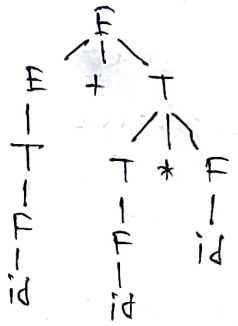
* A language is inherently ambiguous if every grammar for that language is ambiguous.

* Unambiguous form of, $E \to E+E \mid E*E \mid (E) \mid id$ is

$$E \to E+T \mid T$$
$$T \to T*F \mid F$$
$$F \to (E) \mid id.$$

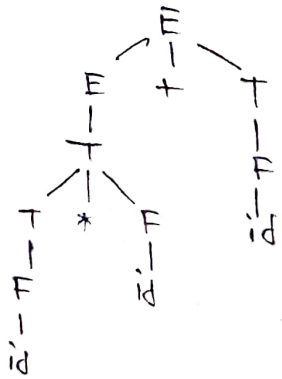
- Here, a leftmost derivation of id+id*id is,

$E \to E+T$
$\to T+T$
$\to F+T$
$\to id+T$
$\to id+T*F$
$\to id+F*F$
$\to id+id*F$
$\to id+id*id$



And a leftmost derivation of id*id+id is,

$E \to E+T$
$E \to T+T$
$\to T*F+T$
$\to F*F+T$
$\to id*F+T$
$\to id*id+T$
$\to id*id+F$
$\to id*id+id$



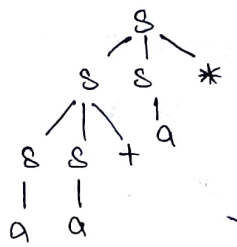
Exercise 4.2.1

Consider the context free grammar: $S \to SS+ \mid SS* \mid a$ and the string $aa+a*$.

(a) Give a leftmost derivation for the string.
(b) " " rightmost " " " ".
(c) Give a parse tree for the string.

(a)

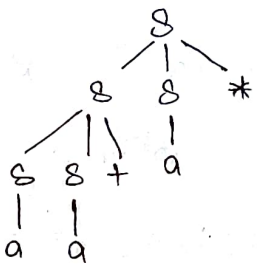
$S \to SS*$
$\to SS+S*$
$\to aS+S*$
$\to aa+S*$
$\to aa+a*$



Same; ∴ unambiguous.

(b)

$S \to SS*$
$\to Sa*$
$\to SS+a*$
$\to Sa+a*$
$\to aa+a*$



Exercise 4.2.2:

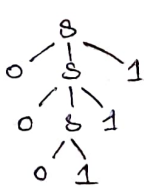
Repeat exercise 4.2.1 for each of the following grammars and strings:

(a) $S \to 0S1 \mid 01$ with string 000111.

(b) $S \to +SS \mid *SS \mid a$ with string $+*aaa$.

(a) **(a.a)**

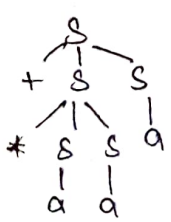
$S \to 0S1$
$\to 00S11$
$\to 000111$



(a.b) Same.

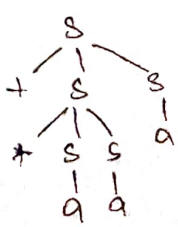
(b) **(b.a)**

$S \to +SS$
$\to +*SSS$
$\to +*aSS$
$\to +*aaS$
$\to +*aaa$.



(b.b)

$S \to +SS$
$\to +Sa$
$\to +*SSa$
$\to +*Saa$
$\to +*aaa$



4.3 Writing a Grammar

* Techniques to get a grammar more suitable for parsing:
  - → eliminate ambiguity in the grammar.
  - → left-recursion elimination, and
  - → left factoring.

* Everything described by a regular expression, can also be described by a grammar. However, nested structures like if-then-'else's can be described by grammar but not by regular expressions.

* To remove ambiguity, we need to ~~ensure~~ enforce operator precedence and associativity rules:
  - Multiplication (*) has higher precedence than addition (+).
  - Both operators are left-associative.

Rewriting the grammar: $E \to E+E \mid E*E \mid (E) \mid id$ to separate addition and multiplication:

$$E \to E+T \mid T$$
$$T \to T*F \mid F$$
$$F \to (E) \mid id.$$

3.3 Elimination of left recursion

A grammar is left-recursive if it has a nonterminal A such that there is a derivation $A \xrightarrow{+} A\alpha$ for some string α.

Top-down parsing methods cannot handle left-recursive grammar, so elimination of left recursion is necessary. The left-recursive pair of productions $A \to A\alpha | \beta$ could be replaced by the non left-recursive productions:

$$A \to \beta A'$$
$$A' \to \alpha A' | \epsilon$$

without changing the strings derivable from A.

For example,

- $E \to E+T | T$
  $T \to T*F | F$
  $F \to (E) | id.$

$A \to A\alpha|\beta$

Here, $E \to E+T|T \Rightarrow$ $E \to TE'$, $E' \to +TE'|\epsilon$

$T \to T*F|F \Rightarrow$ $T \to FT'$, $T' \to *FT'|\epsilon$

Therefore, the obtained non-left-recursive expression grammar,

parse tree for id + id * id =>

$E \to TE'$.
$E' \to +TE'|\epsilon$
$T \to FT'$
$T' \to *FT'|\epsilon$
$F \to (E)|id.$

$A \to A\alpha|\beta$

- Under the original production, a derivation of $\beta\alpha\alpha\alpha$ is,

$A \to A\alpha$
$\to A\alpha\alpha$
$\to A\alpha\alpha\alpha$
$\to \beta\alpha\alpha\alpha$

→ Under new productions, the derivation of $\beta\alpha\alpha\alpha$ is,
$A \to \beta A' \Rightarrow A \to \beta A'$
$A' \to \alpha A'|\epsilon \Rightarrow A' \to \alpha A'|\epsilon$

$\Rightarrow$

$A \to \beta A'$
$\to \beta\alpha A'$
$\to \beta\alpha\alpha A'$
$\to \beta\alpha\alpha\alpha A'$
$\to \beta\alpha\alpha\alpha$

Here, left recursive productions

$$A \to A\alpha_1 \mid A\alpha_2 \mid A\alpha_3 \mid \ldots \mid A\alpha_m \mid \beta_1 \mid \beta_2 \mid \beta_3 \mid \ldots \mid \beta_m$$

where no β_i begins with an A, productions can be replaced by,

$$A \to \beta_1 A' \mid \beta_2 A' \mid \beta_3 A' \mid \ldots \mid \beta_m A'$$

$$A' \to \alpha_1 A' \mid \alpha_2 A' \mid \alpha_3 A' \mid \ldots \mid \alpha_m A' \mid \epsilon.$$

⇒ non-left-recursive

- its all about immediate left recursive.

$S \to Aa \mid b$

$A \to Ac \mid Sd \mid \epsilon$ ⇒ Here, $S \to Aa$ → not immediate left-recursive.

$\to Sda$

→ Lets eliminate left-recursion here.

① Oreder non-terminal S, A. (S, A)

②

(i) No immediate left recursion among the S-productions.

(ii) Substitute S in A, we obtain the following productions,

$$A \to Ac \mid Aad \mid bd \mid \epsilon$$

(iii) Eliminating the immediate left recursion among these A-productions yields the following grammar.

$$S \to Aa \mid b.$$

$$A \to bdA' \mid \epsilon A'$$

$$A' \to cA' \mid adA' \mid \epsilon$$

$A \to A\alpha \mid \beta$

$A \to \beta A'$

$A' \to \alpha A' \mid \epsilon.$

4.3.4 Left Factoring

* A grammar transformation that is useful for producing a grammar suitable for predictive, or top down, parsing.

* If we have the two productions

stmt → if expr then stmt else stmt
| if expr then stmt

or, $A \rightarrow \alpha\beta_1 | \alpha\beta_2$.

On seeing if we cannot decide which production to choose to expand stmt.

Similarly, in two A-productions. when the input begins with a non-empty string derived from α.

* So, we do,

$A \rightarrow \alpha\beta_1 | \alpha\beta_2$ —left-factoring→ $A \rightarrow \alpha A'$, $A' \rightarrow \beta_1 | \beta_2$

After seeing the input derived from α, we expand A' to β_1 or to β_2.

* Method:

For each non-terminal A, find the longest prefix α common to ~~all~~ two or more of its alternatives. If $\alpha \neq \epsilon$, - i.e, there is a nontrivial common prefix - replace all of the A-productions $A \rightarrow \alpha\beta_1 | \alpha\beta_2 | \ldots | \alpha\beta_n | \gamma$, where γ represents all ~~all~~ alternati-s that do not begin with α, by

$A \rightarrow \alpha\beta_1 | \alpha\beta_2 | \ldots | \alpha\beta_n | \gamma$ ⟹ $A \rightarrow \alpha A' | \gamma$, $A' \rightarrow \beta_1 | \beta_2 | \ldots | \beta_n$

Repeatedly apply this transformation until no two alternatives for a nonterminal have a common prefix.

Ex: The following grammar abstracts the "dangling-else" problem :-

$S \rightarrow iEtS \mid iEtSeS \mid a$

$E \rightarrow b$

Here, i, t, and e stand for if, then, and else; E and S stand for "conditional expression" and "statement". Left-factored, this grammar becomes:

$S \rightarrow iEtSS' \mid a$ — যেহেতু এর পরে আবার S -production এর RHS, S' এ আছে ε থাকবে।

$S' \rightarrow eS \mid \epsilon$

$E \rightarrow b.$

$A \rightarrow \alpha \mid \alpha\beta_1 \mid \alpha\beta_2 \mid \ldots \mid \alpha\beta_n \mid b$

↳ $A \rightarrow \alpha A' \mid b.$

$A' \rightarrow \beta_1 \mid \beta_2 \mid \ldots \mid \beta_n \mid \epsilon$

$A \rightarrow \alpha\beta_1 \mid \alpha\beta_2 \mid \ldots \mid \alpha\beta_n \mid b$
$A \rightarrow \alpha A' \mid b$
$A' \rightarrow \beta_1 \mid \beta_2 \mid \ldots \mid \beta_n.$

* We process the part productions have in common, before deciding ~~actu~~ the actual production.

* Left-factor: (i) $C \rightarrow id == num \mid id\ != num \mid id < num.$

⇒

$C \rightarrow id\, C'$

$C' \rightarrow\ == num \mid\ != num \mid < num.$

(ii) $S \rightarrow$ if C then S else S

| if C then S

⇒ $S \rightarrow$ if C then S S'

$S' \rightarrow$ else S | ε.

Exercises for 4.3

Exercise 4.3.1

The following is a grammar for regular expressions over symbols a and b only, using + in place of | for union, to avoid conflict with the use of vertical bar as a metasymbol in grammars:

rexpr ⟶ rexpr + rterm | rterm

rterm ⟶ rterm rfactor | rfactor.

rfactor → rfactor * | rprimary

rprimary → a|b

Solution to (a)

(a) Left factor this grammar.

(b) Does left factoring make the grammar suitable for top-down parsing?

(c) In addition to left-factoring, eliminate left-recursion from the original grammar.

(d) Is the resulting grammar suitable for top-down parsing?

Sol^n

(a) The grammar doesn't have left-factors to extract as each production's prefixes are unique.

(b) This grammar is not suitable for top-down parsing due to left-recursion (e.g., rexpr → rexpr + rterm).

Q4 Grammar -

rexpr → rexpr + rterm | rterm

rterm → rterm rfactor | rfactor

rfactor → rfactor * | rprimary.

rprimary → a|b.

We know,

$A \to A\alpha | \beta \Rightarrow A \to \beta A'$, $A' \to \alpha A' | \epsilon$

Now, here,

- rexpr → rexpr + rterm | rterm

  becomes,

  rexpr → rterm A

  A → + rterm A | ϵ

- rterm → rterm rfactor | rfactor.

  becomes,

  rterm → rfactor B

  B → rfactor B | ϵ

- rfactor → rfactor * | rprimary.

  becomes,

  rfactor → rprimary C

  C → * C | ϵ.

- rprimary → a|b.

(d)

After removing left-recursion, the grammar is now suitable for top-down parsing techniques.

Exercise 4.3.2

Same as 4.3.1, For,

(a) 4.2.1 => $S \to SS+ \mid SS* \mid a$

(b) 4.2.2 *(a)=> $S \to OS1 \mid 01$

(c) 4.2.2 (c)=> $S \to S(S)S \mid \epsilon$

(d) 4.2.2 (e)=> $S \to (L) \mid a$ and $L \to L, S \mid S$

(e) 4.2.2 (a) =>

bexpr → bexpr or bterm | bterm

bterm → bterm and bfactor | bfactor

bfactor → not bfactor | (bexpr) | true | false

4.4 Top-Down Parsing.

↳ nodes of the parse tree are created in preorder (depth-first).

$A \to A\alpha \mid \beta$
$\Rightarrow A \to \beta A',\ A' \to \alpha A' \mid \epsilon$

$E \to E+E \mid E*E \mid (E) \mid id$

Removing ambiguity ⇒

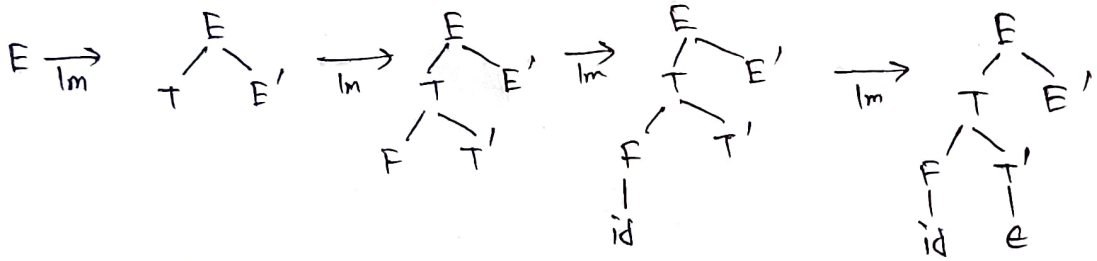
$E \to E+T \mid T$
$T \to T*F \mid F$
$F \to (E) \mid id$

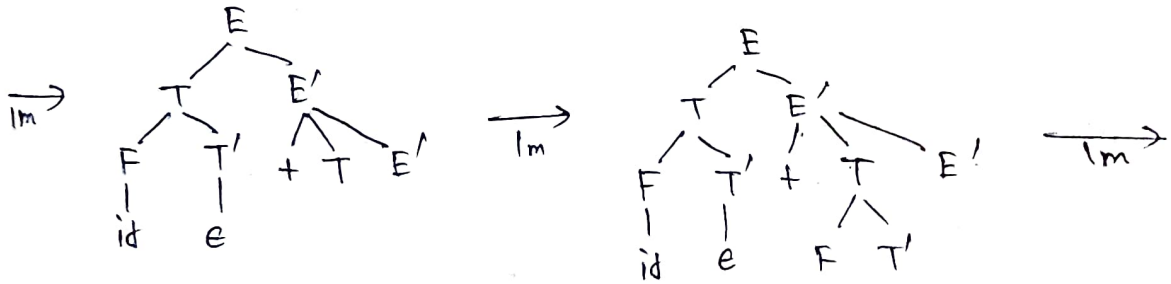
removing left-recursion →

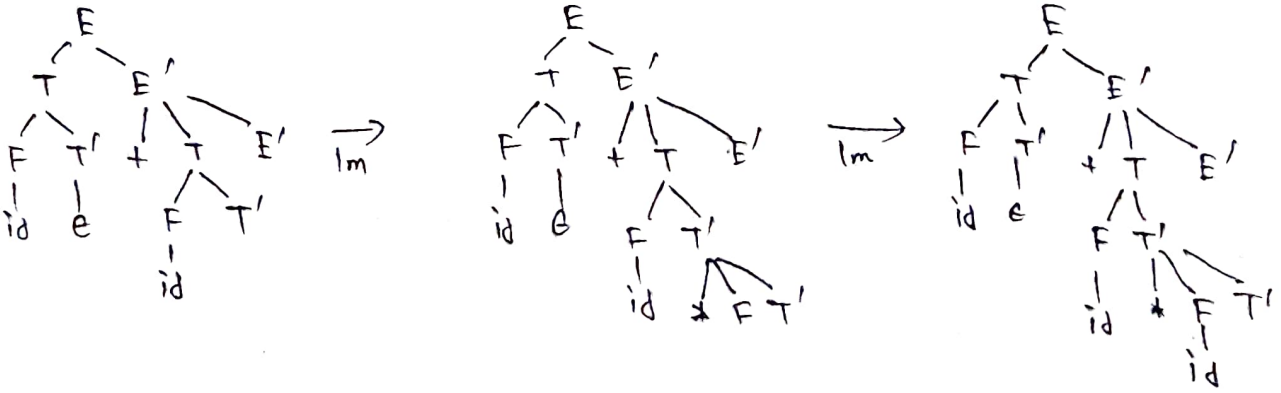
$E \to TE'$
$E' \to +TE' \mid \epsilon$
$T \to FT'$
$T' \to *FT' \mid \epsilon$
$F \to (E) \mid id.$

(a)

Parse tree for id+id*id using (a) [A top down parse],







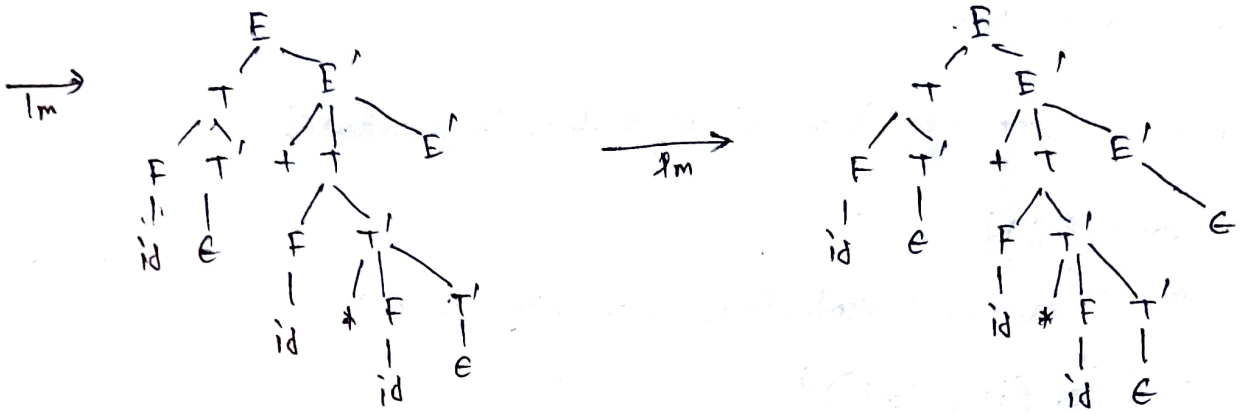


Figure: Top-down parse for id + id * id.

* The key problem is that of determining the production to be applied for a non-terminal, at each step of top-down parse.

The class of grammars for which we can construct predictive parsers looking k symbols ahead in the input is sometimes called the LL(k) class.

4.4.1 Recursive-Descent Parsing

* A typical procedure for a nonterminal in a top-down parser;

```
void A(){
① Choose an A-production, A → X1, X2, ---, Xk;
② for (i=1 to k){
③     if (Xi is a nonterminal)
④         call procedure Xi();
⑤     else if (Xi equals the current input symbol a)
⑥         advance the input to the next symbol;
⑦     else /* an error has occurred */
    }
}
```

* General recursive-descent parser may require backtracking. To allow backtracking in the previous procedure-

i) At ⊖ ①, cannot choose a unique a A-production. Try each of several productions in some order.

ii) At ⑦, failure is not ultimate. Suggest, we need to return to line ①. to try another production.

iii) If no more A-productions to try, an input error has been found.

iv) In each try, pointers need to be reset.

Consider the grammar:

$S \rightarrow cAd$

$A \rightarrow ab \mid a$

input string $w = cad$, construct a parse tree top-down.

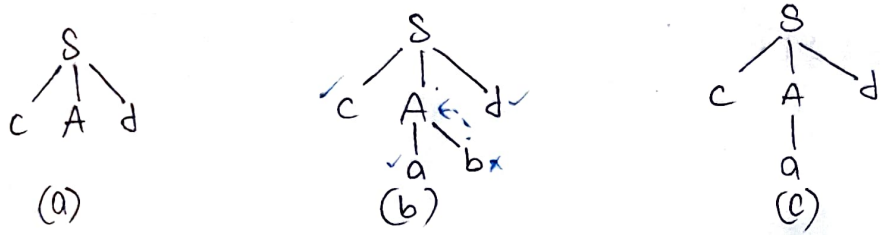
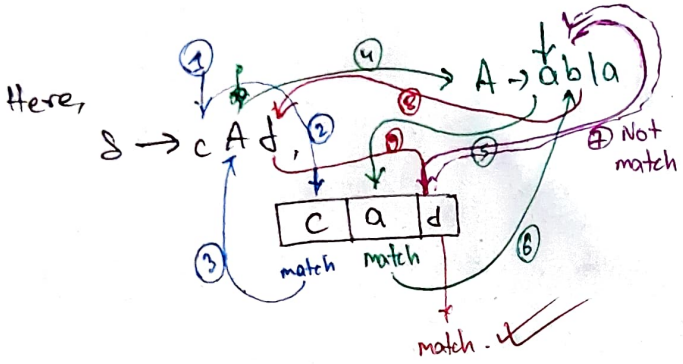
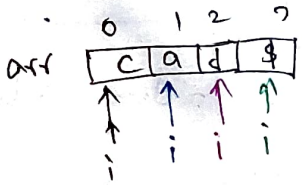


Figure: Steps in a top down parse.






```
S(){
    S → c A d
    if(c == match input symbol){
        i++ (1)  (i=1)
    }
i=1 A(i=1)
    i=2
    d == d
        i++; (3)
}
arr[3] = '$';
print("Success");
}
```

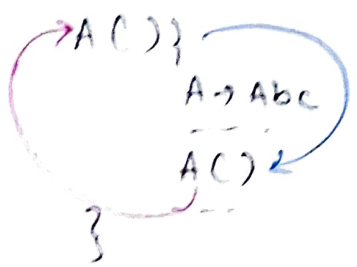
```
A(1){
    A → ab | a
    a == a
        i++; (2)  i=2
    b != d
        i++  i=3 No match; try A → a}
    A → a
    a != d.   Reset i=1, A,(1)}
    A → a
    a == a,
        i++  2
    ret 2;
}
```

* A left-recursive grammar can cause a recursive-descent parser, even one with backtracking, to go into an infinite loop.

e.g.,

$A \to Abc \mid d \mid c$

$c \to ab$



4.4.2 FIRST and FOLLOW

* FIRST and FOLLOW allow us to choose which production to apply based on the next input symbol.

* During panic-mode error recovery, sets of tokens produced by FOLLOW can be used as synchronizing tokens.

* Example:

$E \to TE'$

$E' \to +TE' \mid \epsilon$

$T \to FT'$

$T' \to *FT' \mid \epsilon$

$F \to (E) \mid id.$

① FIRST(F) = {(, id}

② FIRST(T') = {*, ε}

③ FIRST(T) = FIRST(F) = {(, id}

④ FIRST(E') = {+, ε}.

⑤ FIRST(E) = FIRST(T) = {(, id}.

FIRST(α), where α is any string of grammar symbols, to be the set of terminals that begin strings derived from α. If $\alpha \xrightarrow{*} \epsilon$, then ε is also in FIRST(α).

• FIRST এ nonterminal হলো যা প্রথমে আসে তা নেই, আর ϵ ও নে।

*** Example:**

$E \to TE'$

$E' \to +TE' \mid \epsilon$

$T \to FT'$

$T' \to *FT' \mid \epsilon$

$F \to (E) \mid id$

FOLLOW(E) = {), $ } → input string-এ শেষে থাকে। Always source এর Follow হবে।

FOLLOW(E') = FOLLOW(E) = {), $ }

FOLLOW(T) = ~~FIRR~~ FIRST(E') = { +, ϵ → FOLLOW(E') = {), $ } }

= { +,), $ }.

FOLLOW(T') = FOLLOW(T) = { +,), $ }.

FOLLOW(F) = FIRST(T') = { *, ϵ → FOLLOW(T), FOLLOW(T') { +,), $ } FOLLOW(T) }

= { *, +,), $ }.

FOLLOW(A), for nonterminal A, to be the set of terminals a that can appear immediately to the right of A in some sentential form; that is, the set of terminals a such that there exists a derivation of the form $S \overset{*}{\Rightarrow} \alpha A a \beta$, for some α and β.

First নিচ থেকে, FOLLOW উপর থেকে করা সুবিধা। (Neso দেখাতে পারেনি।) ☺

—o—

(*) For FIRST(X), apply rules until no more terminals or ϵ can be added to any FIRST set. (যত production দেখবে, যাদের head এ X আছে।)

A nonterminal A is nullable, if $A \overset{*}{\Rightarrow} \epsilon$
$A \to B_1 B_2 \dots B_n$
→ সব nullable হলেও A nullable.

① If X is a terminal (token), then FIRST(X) = {X}.

② If X is nonterminal, and $X \to Y_1 Y_2 Y_3 \dots Y_k$ for some $k \geq 1$,

$A \to BCD$, FIRST(A) = FIRST(D), if B, and C are nullable.

FIRST(X) = FIRST(Y_1) অথবা, ϵ থাকলে পরেরটা সেটা আসবে।
→ Y_1 যদি ϵ derive করে অর্থাৎ কোনোভাবে তবে তার পরবর্তী ~~nonterminal~~ nonterminal Y_2 নিতে হবে।

(*) FOLLOW এর ক্ষেত্রে, যতক্ষণ FOLLOW set এ কিছু add করা যায় ততক্ষণ পর্যন্ত। (যত production দেখবে যাদের body তে A আছে। → FOLLOW(A) এর জন্য।)

① S start symbol হলে $ FOLLOW(S) এ আসবে।

② যদি $A \to \alpha B \beta$ production থাকে, তবে FOLLOW(B)-তে FIRST(β) থাকবে ~~exp~~ ϵ ছাড়া।

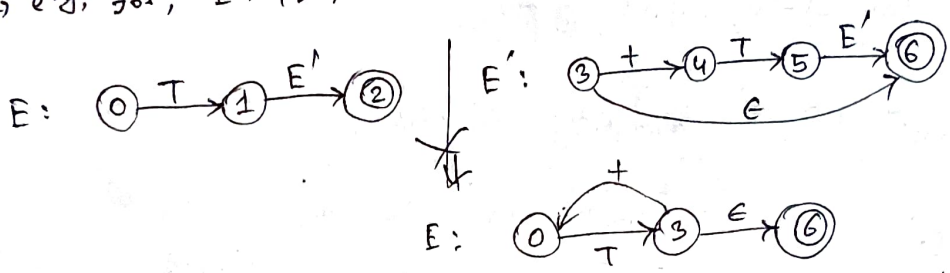
③ যদি $A \to \alpha B$ production অথবা, $A \to \alpha B \beta$ হয় যেখানে FIRST(β) এ ϵ থাকে তবে, FOLLOW(A) এর সব FOLLOW(B) তে আসবে।

দুই জায়গায় nonterminal থাকলে দুটিতেই আসে।

4.4.3 LL(1) Grammars

↳ The first 'L' stands for scanning the input from left to right. The second 'L' for producing a leftmost derivation. and the '1' for using one input symbol of lookahead at each step to make parsing action decisions.

* Transition diagrams for predictive parsers.:
  - ↳ eliminate left-recursion
  - ↳ left-factor the grammar.
  - ↳ e.g. for, $E \to TE'$, $E' \to +TE' \mid \epsilon$:



Predictive Parsing Table

↳ M[A, a]

↳ 2-D array, → A (nonterminal)
↳ a is terminal or $ (the input endmarker)

M[A, a] table এ,

$A \to \alpha$ add করো; সব terminal যারা FIRST(α) তে আছে, তাদের জন্য,

যদি ϵ, FIRST(α) তে থাকে, তাহলে FOLLOW(A) এর সব terminal ~~এর~~ জন্য (b)-র জন্য $A \to \alpha$, M[A, b] তে বসাও।

সব বসানোর শেষে বাকি খালি entry গুলো error.

$E \to TE'$
$E' \to +TE' \mid \epsilon$
$T \to FT'$
$T' \to *FT' \mid \epsilon$
$F \to (E) \mid id$.

Nonterminal	Nullable	FIRST	FOLLOW
E	No	{(, id}	{), $}
E'	Yes	{+, ε}	{$,)}
T	No	{(, id}	{+,), $}
T'	Yes	{*, ε}	{+,), $} → Follow(T)
F	No	{(, id}	{*, +,), $}

Now, using the procedure, our predictive parsing table,

Nonterminal	Input symbol (terminal)					
	id	+	*	(	)	$
E	$E \to TE'$			$E \to TE'$		
E'		$E' \to +TE'$		$E' \to \epsilon$		$E' \to \epsilon$
T	$T \to FT'$			$T \to FT'$		
T'		$T' \to \epsilon$	$T' \to *FT'$		$T' \to \epsilon$	$T' \to \epsilon$
F	$F \to id$			$F \to (E)$		

Table: The corresponding parsing table.

* In table, blanks are error entries; nonblanks indicate a production with which to expand a nonterminal. Each parsing table entry uniquely identifies a production or signals an error.

* For some grammars, table has two productions for one entry. Those languages have no LL(1) grammar at all. For example, a grammar is,

$S \rightarrow iEtSS' \mid a$

$S' \rightarrow eS \mid \epsilon$

$E \rightarrow b$

Nonterminal	Nullable	FIRST	FOLLOW
S	No	{i, a}	{e, $}
S'	Yes	{e, ε}	{e, $}.
E	No	{b}	{t}

Nonterminal	Input Symbol a	b	e	i	t	$
S	S→iEtSS' / S→a			S→iEtSS'		
S'			S'→eS, S'→ε			S'→ε
E		E→b				

Table: Parsing table

Here, the grammar is ambigious and the ambiguity is manifested by a choice in what production to use when an e (else) is seen. We can resolve this ambiguity by choosing $S' \rightarrow eS$. This choice corresponds to associating an 'else' with closest previous then. Note that the choice of $S' \rightarrow \epsilon$ would prevent e from ever being put on the stack or removed from the input, and is surely wrong.

4.4.4 Nonrecursive Predictive Parsing

A nonrecursive predictive parser can be built by maintaining an explicit stack, rather than implicitly via recursive calls. The parser mimics a leftmost derivation.

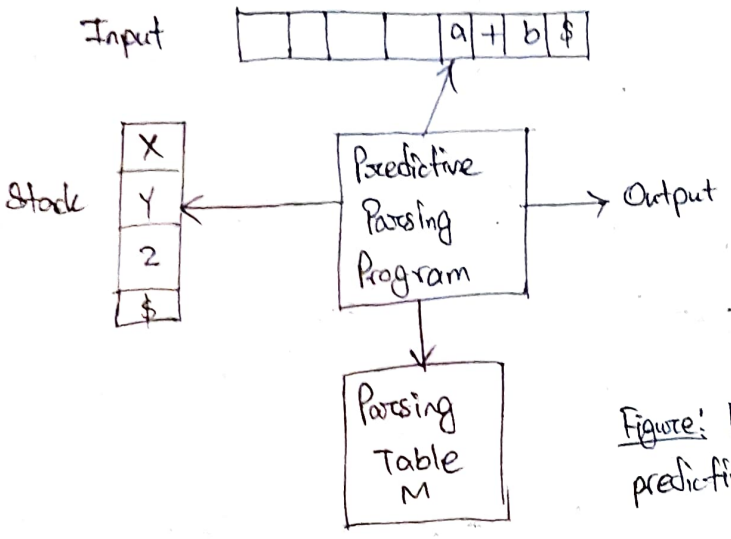


Figure: Model of a table-driven predictive parser.

Stack contains sequence of grammar symbols. In table-driven predictive parsing, if a string w is in $L(G)$, a leftmost derivation of w exists. Otherwise, an error. Method,

Initially: $w\$$, S → start symbol, $

Now:

```
set ip to the first symbol of w;        // ip → input pointer
set X to the top stack symbol;
while (X ≠ $) { /* stack is not empty */
    if (X is a) pop the stack and advance ip;
    else if (X is a terminal) error();
    else if (M[X,a] is an error entry) error();
    else if (M[X,a] = X → Y1 Y2 ... Yk) {
        output the production X → Y1 Y2 ... Yk;
        pop the ~~stat~~ stack;
        push Yk, Yk-1, ..., Y1 onto the stack, with Y1 on top;
    }
    set X to the top stack symbol;
}
```

Example:

G =>

$E \to TE'$

$E' \to +TE' \mid \epsilon$

$T \to FT'$

$T' \to *FT' \mid \epsilon$

$F \to (E) \mid id$

Nonterminal	Nullable	FIRST	FOLLOW
E	No	{(, id}	{$,)}
E'	Yes	{+, ε}	{$,)}
T	No	{(, id}	{+, $,)}
T'	Yes	{*, ε}	{+, $,)}
F	No	{(, id}	{*, +, $,)}

Nonterminal	Input symbols					
	id	+	*	(	)	$
E	$E \to TE'$			$E \to TE'$		
E'		$E' \to +TE'$			$E' \to \epsilon$	$E' \to \epsilon$
T	$T \to FT'$			$T \to FT'$		
T'		$T' \to \epsilon$	$T' \to *FT'$		$T' \to \epsilon$	$T' \to \epsilon$
F	$F \to id$			$F \to (E)$		

On input id+id*id, the nonrecursive predictive parser makes the following sequence of moves: (F→T→ε)

Id Matched	STACK	INPUT	ACTION
	$E\$$	id+id*id$	
	$TE'\$$	id+id*id$	output $E \to TE'$
	$FT'E'\$$	id+id*id$	output $T \to FT'$
	$idT'E'\$$	id+id*id$	output $F \to id$
id	$T'E'\$$	+id*id$	eat match id
id	$E'\$$	+id*id$	output $T' \to \epsilon$
id	$+TE'\$$	+id*id$	output $E' \to +TE'$
id+	$TE'\$$	id*id$	match +
id+	$FT'E'\$$	id*id$	output $T \to FT'$
id+	$idT'E'\$$	id*id$	output $F \to id$
id+id	$T'E'\$$	*id$	match id
id+id	$*FT'E'\$$	*id$	output $T' \to *FT'$

MATCHED	STACK	INPUT	ACTION
id+id *	FT'E'$	id $	match *
id+id *	idT'E'$	id $	output F→id
id+id*id	T'E'$	$	match id
id+id*id	E'$	$	output T'→ε
id+id*id	$	$	output E'→ε

↑ empty stack.

4.4.5 Error recovery in predictive parsing

* An error is detected during predictive parsing when terminal on top of the stack does not match the next input symbol or when nonterminal A is on top of the stack, a is the next input symbol, and M[A,a] is error (i.e., the parsing-table entry is empty.)

Panic Mode

Panic-mode error recovery is based on the idea of skipping over symbols on the input until a token in a selected set of synchronizing tokens appears. Its effectiveness depends on the choice of synchronizing set. Key to choose the set:

① As a starting point, place all symbols in ~~fo~~ Follow(A) into the synchronizing set for nonterminal A. If we skip tokens until an element of FOLLOW(A) is seen and pop A from the stack, it is likely that parsing can continue.

② Add keywords that begin statements to the synchronizing sets for the nonterminals generating expressions.

③ If we add symbols in FIRST(A) to the synchronizing set for nonterminal A, then it may be possible to resume parsing according to A if a symbol in FIRST(A) appears in the input.

④ If a nonterminal can generate the empty string, then the production deriving ϵ can be used as a default. Doing so may postpone some error detection, but cannot cause an error to be missed. This approach reduces the number of nonterminals that have to be considered during err. recovery.

⑤ If a terminal on top of the stack can't be matched, a simple idea is to pop the terminal, issue a message saying that the terminal was inserted, and continue parsing. In effect, this approach takes the synchronizing set of a token to consist of all other tokens.

① and ② usually works reasonably well.

	FIRST	FOLLOW
E	{(, id}	{$,)}
E'	{+, ε}	{$,)}
T	{(, id}	{+, $,)}
T'	{*, ε}	{+, $,)}
F	{(, id}	{*, +, $,)}

Nonter-minal	Input symbol					
	id	+	*	(	)	$
E	E → TE'			E → TE'	synch	synch
E'		E' → +TE'			E' → ε	E' → ε
T	T → FT'	synch		T → FT'	synch	synch
T'		T' → ε	T' → *FT'		T' → ε	T' → ε
F	F → id	synch	synch	F → (E)	synch	synch

Figure: Synchronizing tokens added to the parsing table.

If the parser looks up the entry ~~table~~ M[A, a] and finds that it is blank, then the input symbol a is skipped. If it is "synch", then the

nonterminal on top of the stack is popped in an attempt to resume parsing. If a token on top of the stack does not match the input symbol, then we pop the token from the stack.

On the ~~erroneous~~ erroneous input)id * + id, the parser and error recovery mechanism -

STACK	INPUT	REMARK
E $	)id * + id $	error, skip)
E $	id * + id $	id is in FIRST(E)
TE' $	id * + id $	E → TE'
FT'E' $	id * + id $	T → FT'
idT'E' $	id * + id $	F → id
T'E' $	* + id $	match id
* FT'E' $	* + id $	T' → * FT'
FT'E' $	+ id $	* == *
T'E' $	+ id $	F has been popped
E' $	+ id $	T' → ε
+ TE' $	+ id $	E' → + TE'
TE' $	id $	+ == +
FT'E' $	id $	T → FT'
idT'E' $	id $	F → id
T'E' $	$	id == id
E' $	$	T' → ε
$	$	E' → ε

* Error message — describe error.
                 — draw attention to where the err. was discovered.

Phrase-level Parsing

Phrase-level error recovery is implemented by filling in the blank entries in the the predictive parsing table with pointers to error routines. These routines take appropriate predefined actions.

Exercise 4.4.1

Assignment

For the grammar

$S \to (L) \mid id$

$L \to S, L, S \mid S.$

construct a parse table for this grammar.

And parse the expression (id, id).

$A \to A\alpha \mid \beta$
$A \to \beta A'$
$A' \to \alpha A' \mid \epsilon$

=> Here,

G,

$S \to (L) \mid id$

$L \to L, S \mid S$ ← left recursion

↳ $L \to SL'$

$L' \to , SL' \mid \epsilon .$

=>

$S \to (L) \mid id$

$L \to SL'$

$L' \to , SL' \mid \epsilon .$

Nonterminal	FIRST	FOLLOW
S	{(, id}	{$, , ,)}
L	{(, id}	{)}
L'	{, , ε}	{)}

Non terminal	Input symbols: (	)	id	,	$
S	S→(L)		S→id		
L	L→SL'		L→SL'		
L'		L'→ε		L'→,SL'	

* (id, id)

Stack	Input	Remark
S$	(id,id)$	
(L)$	(id,id)$	S→(L)
L)$	id,id)$	(== (
SL')$	id,id)$	L→SL'
idL')$	id,id)$	S→id
L')$	,id)$	id==id
,SL')$	,id)$	ε ⇒ L'→,SL'
SL')$	id)$	,==,
idL')$	id)$	S→id

Stack	Input	Action
L')$	)$	id==id
)$	)$	L'→ε
$	$	)==).

↑ end of stack

(id, id) is ~~passed~~ parsed successfully by the grammar.