LEXICAL ANALYSIS & ITS ROLE

Lexical analysis

- The scanning/lexical analysis phase of a compiler performs the task of **reading the source program** as a file of characters and **dividing up into tokens.**
- Usually implemented as **subroutine or co-routine of parser.**
- **Front end of compiler.**

tokens

- Each token is a sequence of characters that represents a unit of information in the source program.

Example-tokens

- **Keywords** which are fixed string of letters .eg: “if”, “while”.
- **Identifiers** which are user-defined strings composed of letters and numbers.
- **Special symbols** like arithmetic symbols.

Applications

- Scanners perform **pattern matching process**.
- The techniques used to implement lexical analyzers can also be applied to other areas such as **query languages and information retrieval systems**.
- Since pattern directed programming is widely useful, pattern action language called **Lex** for specifying lexical analyzers.
- In lex , patterns are specified by regular expressions, and a compiler for lex can generate an **efficient finite-automaton** recognizer for the regular expression.

- A software tool that automates the construction of lexical **analyzers allows people with different backgrounds to use pattern matching in their own areas.**
- Jarvis[1976] Lexical analyzer generator to create a program **that recognizes imperfections in printed circuit boards**.
- ***The circuits are digitally scanned and converted into "strings" of line segments at different angles***.
- The "lexical analyzer" looked for patterns corresponding to imperfections in the string of line segments.

Advantage-lexical analyzer generator

- It can utilize the best-known pattern-matching algorithms and thereby create efficient lexical analyzers for people who are not experts in pattern-matching techniques.

The Role of Lexical Analyzer

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

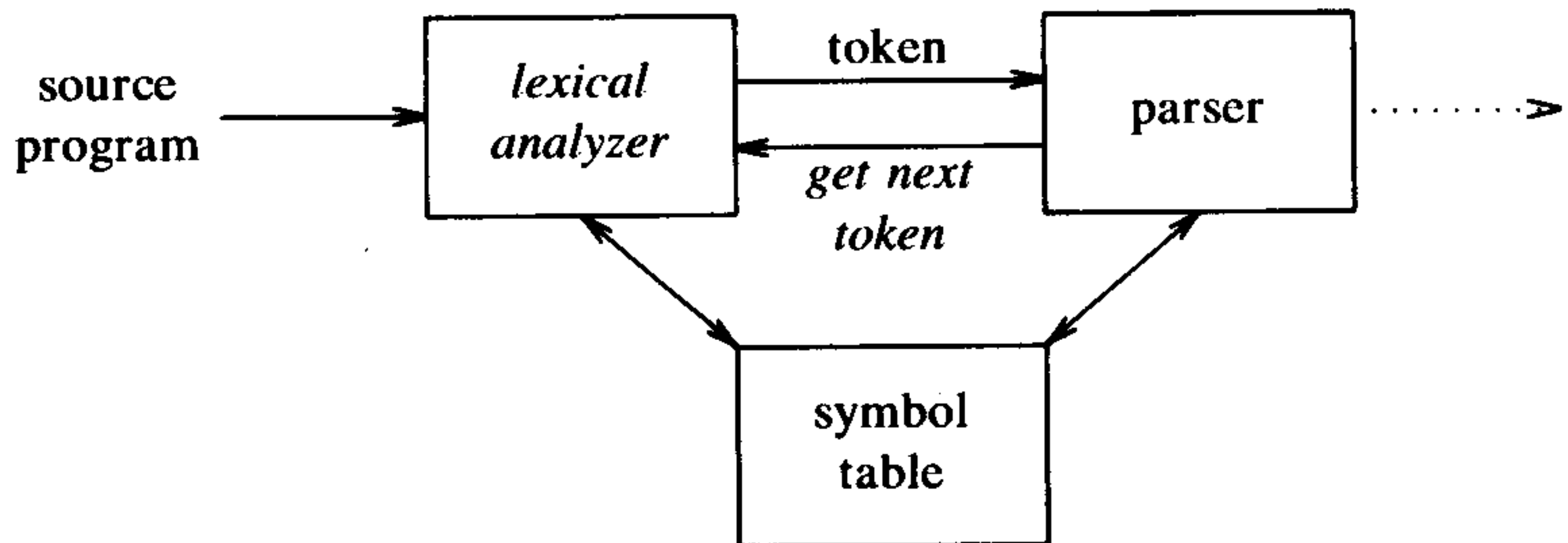


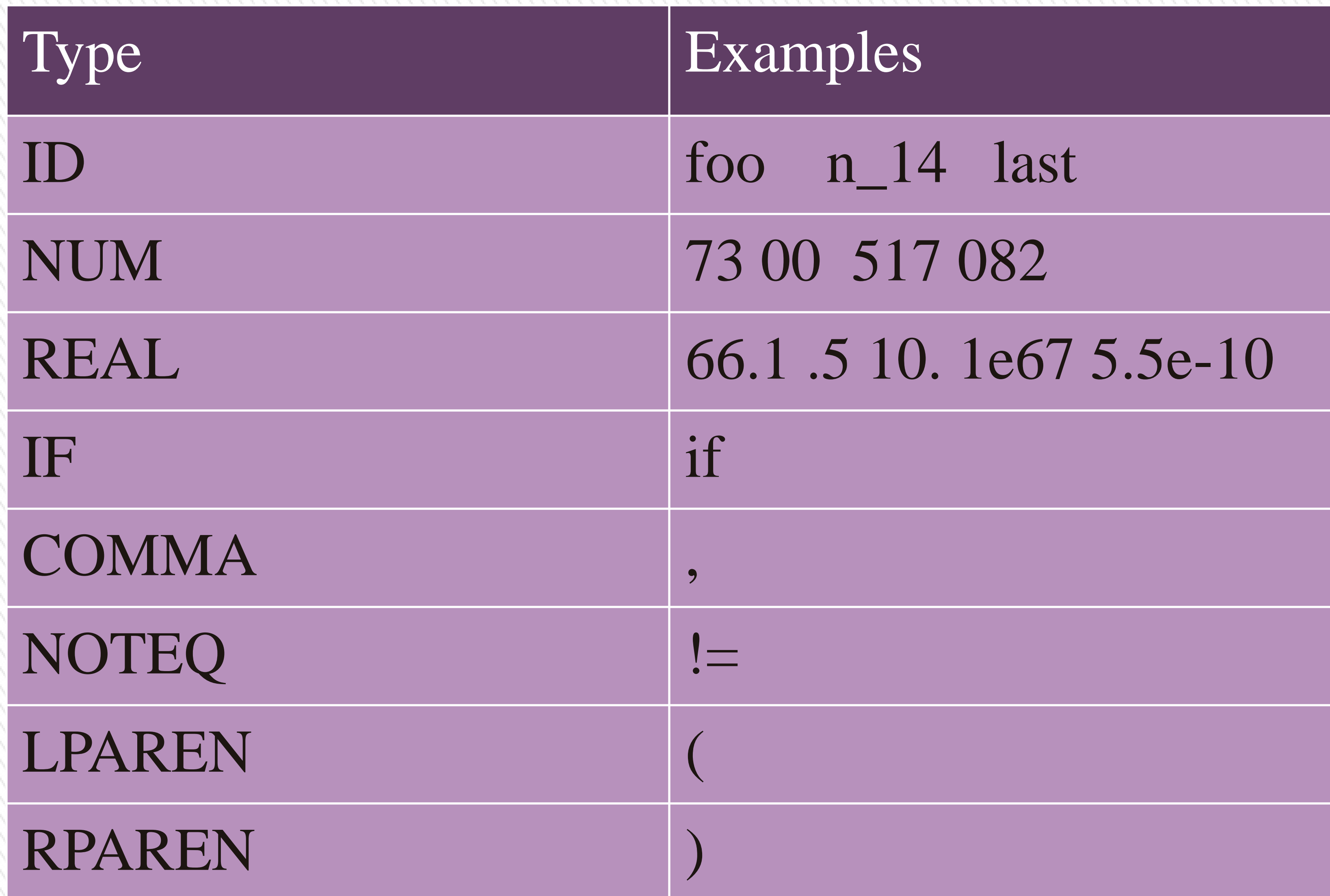
Fig. 3.1. Interaction of lexical analyzer with parser.

A Simple Lexical Analyzer

E=M*C**2 → Keyword? Identifier? Operators? ??? → Identifier

Identifier pattern matching

Example Tokens



Type	Examples
ID	foo n_14 last
NUM	73 00 517 082
REAL	66.1 .5 10. 1e67 5.5e-10
IF	if
COMMA	,
NOTEQ	!=
LPAREN	(
RPAREN	)

Example NonTokens

Type	Examples
comment	/* ignored */
preprocessor directive	#include <foo.h>
	#define NUMS 5, 6
macro	NUMS
whitespace	\t \n \b

Tasks –lexical analyzer

- **Separation of the input source code into tokens**.
- **Stripping out the unnecessary white spaces** from the source code.
- **Removing the comments** from the source text.
- **Keeping track of line numbers** while scanning the new line characters. These line numbers are used by the error handler to print the error messages.
- **Preprocessing of macros.**

Issues in Lexical Analysis

- **There are several reasons for separating the analysis phase of compiling into lexical analysis and parsing:**
- It leads to ***simpler design*** **of the parser** as the unnecessary tokens can be eliminated by scanner.
- **Efficiency** of the process of compilation is improved. The lexical analysis phase is most time consuming phase in compilation. Using ***specialized buffering*** to improve the speed of compilation.
- **Portability** of the compiler is enhanced as the specialized symbols and characters(language and machine specific) are isolated during this phase.

Tokens, Patterns, Lexemes

- Connected with lexical analysis are three important terms with similar meaning.
- Lexeme
- Token
- Patterns

Tokens, Patterns, Lexemes

- A ***token*** is a pair consisting of a **token name and an optional attribute value**. Token name: Keywords, operators, identifiers, constants, literal strings, punctuation symbols(such as commas,semicolons)
- 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**. E.g.**Relation** {<.<=,>,>=,==,<>}

- A ***pattern*** is a ***description of the form that the lexemes of token may take***.
- It gives an **informal or formal description of a token.**
- ***Eg: identifier***
- **2 purposes**
- Gives a precise description/ specification of tokens.
- Used to automatically generate a lexical analyzer

Example of tokens

» `const pi = 3.1416;`

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

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

Identify tokens and lexemes?

- x=x*(acc+123)

Lexical Analysis

input: x = x * (acc+123)

token	lexemes
identifier	x
equal	=
identifier	x
star	*
left-paren	(
identifier	acc
plus	+
integer	123
right-paren	)

Lexical errors

- **1.)** let us consider a statement **"fi(a==f)".** Here "fi" is a **misspelled keyword**. This error **is not detected in lexical analysis** as **"fi" is taken as an identifier.** This error is then detected in other phases of compilation.
- **2.)** in case the lexical analyzer is not able to continue with the process of compilation, it resorts to **panic mode of error recovery**.
- ***Deleting the successive characters*** from the remaining input until a token is detected.
- ***Deleting extraneous characters.***

- *Inserting missing characters*
- *Replacing an incorrect character by a correct character.*
- *Transposing two adjacent characters*

Minimum distance error correction

- Is the strategy generally followed by the **lexical analyzer to correct the errors in the lexemes.**
- It is nothing but the **minimum number of the corrections to be made to convert an invalid lexeme to a valid lexeme.**
- But it is ***not generally used in practice*** because it is ***too costly*** to implement.

SPECIFICATION OF T0KENS USING REGULAR EXPRESSION

Specification of tokens

- **Scanners** are special **pattern matching processors.**
- For representing patterns of strings of characters, **Regular Expressions(RE**) are used.
- *A regular expression (r) is defined by set of strings that matches it.*
- This set is called as the **language** generated by the regular expression and is represented as **L(r).**
- The **set of symbols** in the language is called the **alphabet** of the language is represented as ∑.

- An alphabet is a finite set of symbols.
- <u>Example</u>
- A set of alphabetic characters is represented as L={A,…,Z,a,…,z} and set of digits is represented as D={0,1,…,9}.
- LUD is a language.
- *Strings over LUD*- Begin,Max1, max1, 123, €…

Operations on Languages

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

Regular expression operations

- Choice among alternates
- Concatenation
- Repetition

1. CHOICE AMONG ALTERNATES

- Indicated by metacharacter **‘|’(vertical bar)**
- **r|s**
- R.E that matches any string that is matched **either by r or s.**
- **L(r|s)= L(r) U L(s)**

2. CONCATENATION

- **rs**
- It matches any string that is a concatenation of 2 strings, **the first of which matches r and second of which matches s.**
- **L(rs) = L(r) L(s)**

3. REPETITION

- Also called **Kleene closure**
- Represents **any finite concatenation of strings each matches strings from L(r).**
- **r***
- Let S={a}, then L(a*)={€, a, aa, aaa,…}
- S*={€}USUSSUSSSU….= $\bigcup_{n=0}^{\infty} S^n$
-