Assignment No.4

AIM:

Implement lexical analyzer for C-language

THEORY:

Role of Lexical Analyzer

The lexical analyzer is the first phase of a compiler. Its main task is to read the input characters and produce as output a sequence of tokens that the parser uses for syntax analysis. This interaction, summarized schematically in Figure, is commonly implemented by making the lexical analyzer be a subroutine or a coroutine of the parser. Upon receiving a "get next token" command from the parser, the lexical analyzer reads input characters until it can identify the next token.

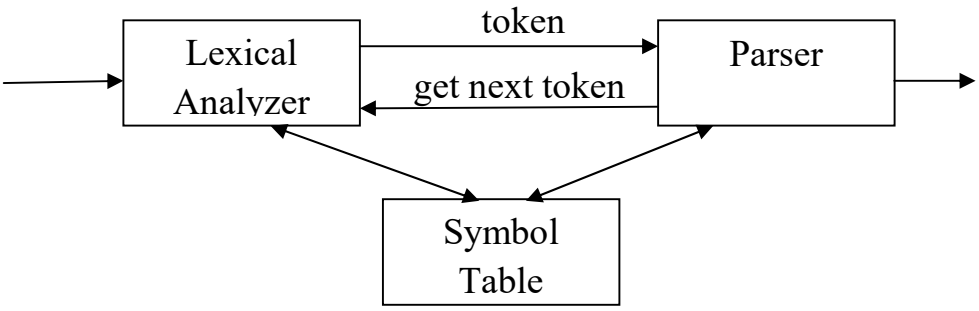

Fig: Interaction of lexical anlyzer with parser

Since the lexical analyzer is the part of the compiler that reads the source text, it may also perform certain secondary tasks at the user interface. One such task is stripping out from the source program comments and white space in the form of blank, cab, and newline characters. Another is correlating error messages from the

compiler with the source program. For example, the lexical analyzer may keep track of the number of newline characters seen, so that a line number can be associated with an error message. In some compilers, the lexical analyzer is in charge of making a copy of the source program with the error messages marked in it. If the source language supports some macro preprocessor functions, then these preprocessor functions may also be implemented as lexical analysis takes place,

Issues in lexical analyzer

There are several reasons for separating the analysis phase of compiling into lexical analysis and parsing.

- Simpler design is perhaps the most important consideration. The separation of lexical analysis from syntax analysis often allows us to simplify one or the other of these phases. For example, a parser embodying the conventions for comments and white space is significantly more complex than one that can assume comments and white space have already been removed by a lexical analyzer. If we are designing a new language, separating the lexical and syntactic conventions can lead to a cleaner overall language design.
- Compiler efficiency is improved. A separate lexical analyzer allows us to construct a specialized and potentially more efficient processor for the task. A large amount of time is spent reading the source program and partitioning it into tokens. Specialized buffering techniques for reading input characters and processing tokens can significantly speed up the performance of a compiler,
- Compiler portability is enhanced. Input alphabet peculiarities and other device-specific anomalies can be restricted to the lexical analyzer. The representation of special or non-standard symbols, such as in Pascal, can be isolated in the lexical analyzer.

Token, patterns & lexemes

TOKEN

- A classification for a common set of strings
- Examples Include <Identifier>, <number>, etc.

PATTERN

- The rules which characterize the set of strings for a token
- File and OS Wildcards ([A-Z]*.*)

LEXEME

- Actual sequence of characters that matches pattern and is classified by a token
- Identifiers: x, count, name, etc…

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 "

Classifies Pattern

Actual values are critical.

Info is :

1. Stored in symbol table
2. Returned to parser

Attributes of tokens

When more than cine pattern matches a lexeme, the lexical analyzer must provide additional information about the particular lexeme that matched to the subsequent phases of the compiler. For example, the pattern num matches both the strings 0 and 7, but it is essential for the code generator to know what string was actually matched.

The lexical analyzer collects information about tokens into their associated attributes, The tokens influence parsing decisions; the attributes influence the translation of tokens. As a practical matter, a token has usually only a single attribute - a pointer to the symbol-table entry in which the information about the token is kept; the pointer becomes the attribute for the token.

Example: 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, >

<num, integer value 2>

Lexical Error

Few errors are discernible at the lexical level alone, because a lexical analyzer has a very localized view of a source program. If the string f i is encountered in a C program for the first time in the context

fi((a == f(x))…..

a lexical analyzer cannot tell whether fi is a misspelling of the keyword if or an undeclared function identifier. Since fi is a valid identifier, the lexical analyzer

must return the token for an identifier and let some other phase of the compiler handle any error.

But, suppose a situation does arise in which the lexical analyzer is unable to proceed because none of the patterns for tokens matches a prefix of the remaining input. Perhaps the simplest recovery strategy is "panic mode" recovery. We delete successive characters from the remaining input until the lexical analyzer can find a well-formed token. This recovery technique may occasionally confuse the parser, but in an interactive computing environment it may be quite adequate.

Other possible error-recovery actions are:

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

INPUT:

TRIAL.CPP

```
# include< stdio.h>
# include< conio.h>

void main()
{
    int num1= 5 , count= 1 , ab= 10 ;
    char ch;
    printf ( "This is a trial program" ) ;
    while ( count ! = num1 )
    {
```

```
                ab = ab* count/ 2 ;
                if ( count== 3 )
                count= count+ 1 ;
                printf ( "AB %d" , ab);
        }
}
```

OUTPUT:

Token ID=0	#	Special Character
Token ID=1	include	Keyword type
Token ID=2	<	Special Character
Token ID=3	stdio	Keyword type
Token ID=4	.	Special Character
Token ID=5	h	Identifier type
Token ID=6	>	Special Character
Token ID=7	#	Special Character
Token ID=8	include	Keyword type
Token ID=9	<	Special Character
Token ID=10	conio	Keyword type
Token ID=11	.	Special Character
Token ID=12	h	Identifier type
Token ID=13	>	Special Character
Token ID=14	void	Keyword type
Token ID=15	main	Keyword type
Token ID=16	(	Special Character
Token ID=17	)	Special Character
Token ID=18	{	Special Character

Token ID=19	int	Keyword type
Token ID=20	num1	Identifier type
Token ID=21	=	Operator type
Token ID=22	5	Numeric type
Token ID=23	,	Special Character
Token ID=24	count	Identifier type
Token ID=25	=	Operator type
Token ID=26	1	Numeric type
Token ID=27	,	Special Character
Token ID=28	ab	Identifier type
Token ID=29	=	Operator type
Token ID=30	10	Numeric type
Token ID=31	;	Special Character
Token ID=32	char	Keyword type
Token ID=33	ch	Identifier type
Token ID=34	;	Special Character
Token ID=35	printf	Keyword type
Token ID=36	(	Special Character
Token ID=37	This is a trial program	Literal type
Token ID=38	)	Special Character
Token ID=39	;	Special Character
Token ID=40	while	Keyword type
Token ID=41	(	Special Character
Token ID=42	count	Identifier type
Token ID=43	!	Special Character
Token ID=44	=	Operator type
Token ID=45	num1	Identifier type

Token ID=46	)	Special Character
Token ID=47	{	Special Character
Token ID=48	ab	Identifier type
Token ID=49	=	Operator type
Token ID=50	ab	Identifier type
Token ID=51	*	Operator type
Token ID=52	count	Identifier type
Token ID=53	/	Operator type
Token ID=54	2	Numeric type
Token ID=55	;	Special Character
Token ID=56	if	Keyword type
Token ID=57	(	Special Character
Token ID=58	count	Identifier type
Token ID=59	=	Operator type
Token ID=60	=	Operator type
Token ID=61	3	Numeric type
Token ID=62	)	Special Character
Token ID=63	count	Identifier type
Token ID=64	=	Operator type
Token ID=65	count	Identifier type
Token ID=66	+	Operator type
Token ID=67	1	Numeric type
Token ID=68	;	Special Character
Token ID=69	printf	Keyword type
Token ID=70	(	Special Character
Token ID=71	AB %d	Literal type
Token ID=72	,	Special Character

Token ID=73	ab	Identifier type
Token ID=74	)	Special Character
Token ID=75	;	Special Character
Token ID=76	}	Special Character
Token ID=77	}	Special Character

CONCLUSION:

REFERENCES:

- Compilers - Principles, Techniques and Tools - A.V. Aho, R. Shethi and J. D. Ullman (Pearson Education)