

Lexical Analysis

- Recognize tokens and ignore white spaces, comments

i	f		(x	1		*	x	2		<	1	.	0)	{	
---	---	--	---	---	---	--	---	---	---	--	---	---	---	---	---	---	--

Generates token stream

if	(x1	*	x2	<	1.0)	{	
----	---	----	---	----	---	-----	---	---	--

- Error reporting
- Model using regular expressions
- Recognize using Finite State Automata₁

Lexical Analysis

- Sentences consist of string of **tokens** (a syntactic category)
For example, number, identifier, keyword, string
- Sequences of characters in a token is a **lexeme**
for example, 100.01, counter, const, "How are you?"
- Rule of description is a **pattern**
for example, `letter (letter | digit)*`
- Task: Identify Tokens and corresponding Lexemes

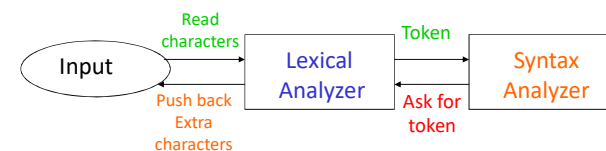
2

Lexical Analysis

- Examples
- Construct constants: for example, convert a number to token num and pass the value as its attribute,
- 31 becomes **<num, 31>**
- Recognize keyword and identifiers
- **counter = counter + increment** becomes **id = id + id**
- check that **id** here is not a keyword
- Discard whatever does not contribute to parsing
- white spaces (blanks, tabs, newlines) and comments

3

Interface to other phases



- Why do we need Push back?
- Required due to look-ahead
for example, to recognize `>=` and `>`
- Typically implemented through a buffer
 - Keep input in a buffer
 - Move pointers over the input

4

Approaches to implementation

- Use assembly language
Most efficient but most difficult to implement
- Use high level languages like C
Efficient but difficult to implement
- Use tools like lex, flex
Easy to implement but not as efficient as the first two cases

5

Construct a lexical analyzer

- Allow white spaces, numbers and arithmetic operators in an expression
- Return tokens and attributes to the syntax analyzer
- A global variable **tokenval** is set to the value of the number
- Design requires that
 - A finite set of tokens be defined
 - Describe strings belonging to each token

6

```
#include <stdio.h>
#include <ctype.h>
int lineno = 1;
int tokenval = NONE;
int lex() {
    int t;
    while (1) {
        t = getchar ();
        if (t == ' ' || t == '\t');
        else if (t == '\n') lineno = lineno + 1;
        else if (isdigit (t) ) {
            tokenval = t - '0' ;
            t = getchar ();
            while (isdigit(t)) {
                tokenval = tokenval * 10 + t - '0' ;
                t = getchar();
            }
            ungetc(t,stdin);
            return num;
        }
        else { tokenval = NONE; return t; }
    }
}
```

7

Problems

- Scans text character by character
- Look ahead character determines what kind of token to read and when the current token ends
- First character cannot determine what kind of token we are going to read

8

Symbol Table

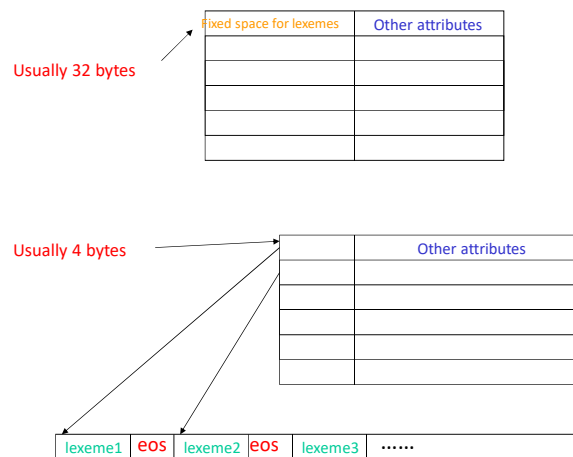
- Stores information for subsequent phases
- Interface to the symbol table
 - **Insert(s,t)**: save lexeme s and token t and return pointer
 - **Lookup(s)**: return index of entry for lexeme s or 0 if s is not found

9

Implementation of Symbol Table

- Fixed amount of space to store lexemes.
 - Not advisable as it waste space.
- Store lexemes in a separate array.
 - Each lexeme is separated by eos.
 - Symbol table has pointers to lexemes.

10



11

How to handle keywords?

- Consider token **DIV** and **MOD** with lexemes **div** and **mod**.
- Initialize symbol table with **insert("div" , DIV)** and **insert("mod" , MOD)**.
- Any subsequent **insert** fails (unguarded insert)
- Any subsequent **lookup** returns the *keyword* value, therefore, these cannot be used as an identifier.

12

Difficulties in the design of lexical analyzers

Is it as simple as it sounds?

13

Lexical analyzer: Challenges

- Lexemes in a fixed position. Fixed format vs. free format languages
- FORTRAN Fixed Format
 - 80 columns per line
 - Column 1-5 for the statement number/label column
 - Column 6 for continuation mark (?)
 - Column 7-72 for the program statements
 - Column 73-80 Ignored (Used for other purpose)
 - Letter C in Column 1 meant the current line is a comment

14

Lexical analyzer: Challenges

- Handling of blanks
 - in C, blanks separate identifiers
 - in FORTRAN, blanks are important *only* in literal strings
 - variable **counter** is same as **count er**
 - Another example
- | | |
|-----------------------|-------------------|
| DO 10 I = 1.25 | DO10I=1.25 |
| DO 10 I = 1,25 | DO10I=1,25 |

15

- The first line is a variable assignment
DO10I=1.25
- The second line is beginning of a
Do loop
- Reading from left to right one can not distinguish between the two until the “,” or “.” is reached

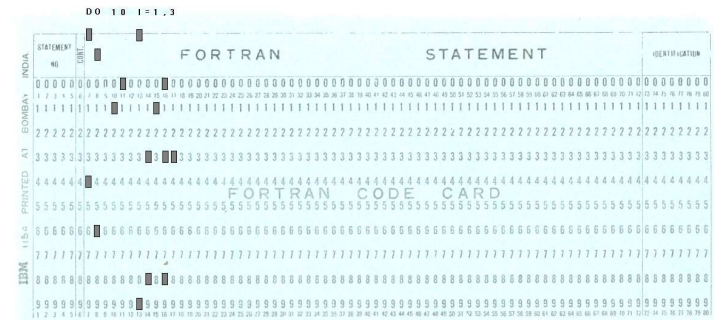
16

Fortran white space and fixed format rules came into force due to punch cards and errors in punching



17

Fortran white space and fixed format rules came into force due to punch cards and errors in punching



18

PL/1 Problems

- Keywords are not reserved in PL/1
if then then then = else else else = then
if if then then = then + 1
- PL/1 declarations
Declare(arg₁,arg₂,arg₃,.....,arg_n)
- Cannot tell whether **Declare** is a keyword or array reference until after “)”
- Requires arbitrary lookahead and very large buffers.

19

Problem continues even today!!

- C++ template syntax: Foo<Bar>
- C++ stream syntax: cin >> var;
- Nested templates:
Foo<Bar<Bazz>>
- Can these problems be resolved by lexical analyzers alone?

20

How to specify tokens?

- How to describe tokens
2.e0 20.e-01 2.000
- How to break text into token
if (x==0) a = x << 1;
if (x==0) a = x < 1;
- How to break input into tokens efficiently
 - Tokens may have similar prefixes
 - Each character should be looked at only once

21

How to describe tokens?

- Programming language tokens can be described by regular languages
- Regular languages
 - Are easy to understand
 - There is a well understood and useful theory
 - They have efficient implementation
- Regular languages have been discussed in great detail in the “Theory of Computation” course

22

Operations on languages

- $L \cup M = \{s \mid s \text{ is in } L \text{ or } s \text{ is in } M\}$
- $LM = \{st \mid s \text{ is in } L \text{ and } t \text{ is in } M\}$
- $L^* = \text{Union of } L^i \text{ such that } 0 \leq i \leq \infty$
Where $L^0 = \epsilon$ and $L^i = L^{i-1} L$

23

Example

- Let $L = \{a, b, \dots, z\}$ and $D = \{0, 1, 2, \dots, 9\}$ then
- LUD is a set of letters and digits
- LD is a set of strings consisting of a letter followed by a digit
- L^* is a set of all strings of letters including ϵ
- $L(LUD)^*$ is a set of all strings of letters and digits beginning with a letter
- D^* is a set of strings of one or more digits

24

Notation

- Let Σ be a set of characters. A language over Σ is a set of strings of characters belonging to Σ
- A regular expression r denotes a language $L(r)$
- Rules that define the regular expressions over Σ
 - ϵ is a regular expression that denotes $\{\epsilon\}$ the set containing the empty string
 - If a is a symbol in Σ then a is a regular expression that denotes $\{a\}$

25

- If r and s are regular expressions denoting the languages $L(r)$ and $L(s)$ then
- $(r)|(s)$ is a regular expression denoting $L(r) \cup L(s)$
- $(r)(s)$ is a regular expression denoting $L(r)L(s)$
- $(r)^*$ is a regular expression denoting $(L(r))^*$
- (r) is a regular expression denoting $L(r)$

26

- Let $\Sigma = \{a, b\}$
- The regular expression $a|b$ denotes the set $\{a, b\}$
- The regular expression $(a|b)(a|b)$ denotes $\{aa, ab, ba, bb\}$
- The regular expression a^* denotes the set of all strings $\{\epsilon, a, aa, aaa, \dots\}$
- The regular expression $(a|b)^*$ denotes the set of all strings containing ϵ and all strings of a 's and b 's
- The regular expression $a|a^*b$ denotes the set containing the string a and all strings consisting of zero or more a 's followed by a character b

27

- Precedence and associativity
- $*$, concatenation, and $|$ are left associative
- $*$ has the highest precedence
- Concatenation has the second highest precedence
- $|$ has the lowest precedence

28

How to specify tokens

- Regular definitions
 - Let r_i be a regular expression and d_i be a distinct name
 - Regular definition is a sequence of definitions of the form
$$d_1 \rightarrow r_1$$
$$d_2 \rightarrow r_2$$
$$\dots$$
$$d_n \rightarrow r_n$$
 - Where each r_i is a regular expression over $\Sigma \cup \{d_1, d_2, \dots, d_{i-1}\}$

29

Examples

- My fax number
91-(512)-259-7586
- $\Sigma = \text{digit} \cup \{-, (,)\}$
- Country $\rightarrow \text{digit}^+$ digit²
- Area $\rightarrow '(\text{digit}^+)'$ digit³
- Exchange $\rightarrow \text{digit}^+$ digit³
- Phone $\rightarrow \text{digit}^+$ digit⁴
- Number $\rightarrow \text{country '-' area '-' exchange '-' phone}$

30

Examples ...

- My email address
karkare@iitk.ac.in
- $\Sigma = \text{letter} \cup \{ @, . \}$
- letter $\rightarrow a | b | \dots | z | A | B | \dots | Z$
- name $\rightarrow \text{letter}^+$
- address $\rightarrow \text{name '@' name '.' name '.' name}$

31

Examples ...

- Identifier
 - letter $\rightarrow a | b | \dots | z | A | B | \dots | Z$
 - digit $\rightarrow 0 | 1 | \dots | 9$
 - identifier $\rightarrow \text{letter}(\text{letter} | \text{digit})^*$
- Unsigned number in C
 - digit $\rightarrow 0 | 1 | \dots | 9$
 - digits $\rightarrow \text{digit}^+$
 - fraction $\rightarrow \text{'.' digits} | \epsilon$
 - exponent $\rightarrow (E ('+' | '-' | \epsilon) \text{digits}) | \epsilon$
 - number $\rightarrow \text{digits fraction exponent}$

32

Regular expressions in specifications

- Regular expressions describe many useful languages
- Regular expressions are only specifications; implementation is still required
- Given a string s and a regular expression R , does $s \in L(R)$?
- Solution to this problem is the basis of the lexical analyzers
- However, just the yes/no answer is not sufficient
- Goal: Partition the input into tokens

33

1. Write a regular expression for lexemes of each token
 - number $\rightarrow \text{digit}^+$
 - identifier $\rightarrow \text{letter}(\text{letter}|\text{digit})^+$
2. Construct R matching all lexemes of all tokens
 - $R = R_1 + R_2 + R_3 + \dots$
3. Let input be $x_1 \dots x_n$
 - for $1 \leq i \leq n$ check $x_1 \dots x_i \in L(R)$
4. $x_1 \dots x_i \in L(R) \Leftrightarrow x_1 \dots x_i \in L(R_j)$ for some j
 - smallest such j is token class of $x_1 \dots x_i$
5. Remove $x_1 \dots x_i$ from input; go to (3)

34

- The algorithm gives priority to tokens listed earlier
 - Treats “if” as keyword and not identifier
- How much input is used? What if
 - $x_1 \dots x_i \in L(R)$
 - $x_1 \dots x_j \in L(R)$
 - Pick up the longest possible string in $L(R)$
 - The principle of “maximal munch”
- Regular expressions provide a concise and useful notation for string patterns
- Good algorithms require a single pass over the input

35

How to break up text

- Elsex=0

else	x	=	0
------	---	---	---

elsex	=	0
-------	---	---
- Regular expressions alone are not enough
- Normally the longest match wins
- Ties are resolved by prioritizing tokens
- Lexical definitions consist of regular definitions, priority rules and maximal munch principle




36

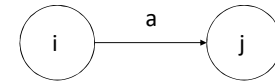
Transition Diagrams

- Regular expressions are declarative specifications
- Transition diagram is an implementation
- A transition diagram consists of
 - An input alphabet belonging to Σ
 - A set of states S
 - A set of transitions $\text{state}_i \xrightarrow{\text{input}} \text{state}_j$
 - A set of final states F
 - A start state n
- Transition $s_1 \xrightarrow{a} s_2$ is read:
 - in state s_1 on input a go to state s_2
- If end of input is reached in a final state then accept, Otherwise, reject

37

Pictorial notation

- A state 
- A final state 
- Transition 
- Transition from state i to state j on an input a



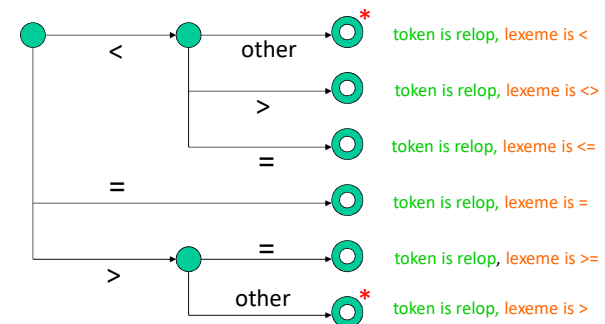
38

How to recognize tokens

- Consider
 - $\text{relop} \rightarrow < \mid <= \mid = \mid <> \mid >= \mid >$
 - $\text{id} \rightarrow \text{letter}(\text{letter} \mid \text{digit})^*$
 - $\text{num} \rightarrow \text{digit}^+ (\text{'.' digit}^+)? (\text{E}(\text{'+'} \mid \text{'-'})? \text{digit}^+)?$
 - $\text{delim} \rightarrow \text{blank} \mid \text{tab} \mid \text{newline}$
 - $\text{ws} \rightarrow \text{delim}^+$
- Construct an analyzer that will return $\langle \text{token}, \text{attribute} \rangle$ pairs

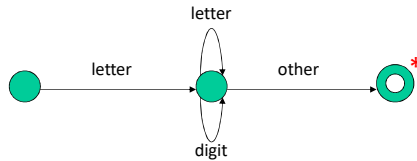
39

Transition diagram for relops

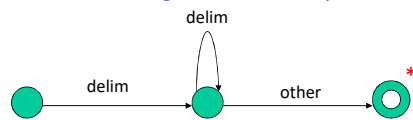


40

Transition diagram for identifier

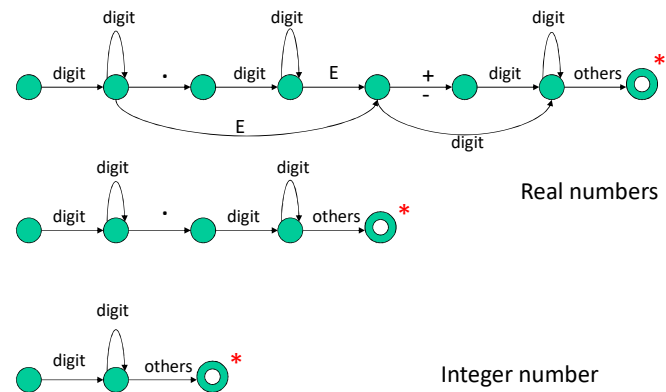


Transition diagram for white spaces



41

Transition diagram for unsigned numbers



42

- The lexeme for a given token must be the longest possible
- Assume input to be 12.34E56
- Starting in the third diagram the accept state will be reached after 12
- Therefore, the matching should always start with the first transition diagram
- If failure occurs in one transition diagram then retract the forward pointer to the start state and activate the next diagram
- If failure occurs in all diagrams then a lexical error has occurred

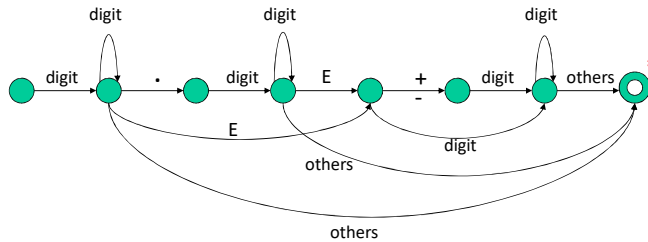
43

Implementation of transition diagrams

```
Token nexttoken() {
    while(1) {
        switch (state) {
            .....
            case 10: c=nextchar();
                    if(isletter(c)) state=10;
                    elseif (isdigit(c)) state=10;
                    else state=11;
                    break;
            .....
        }
    }
}
```

44

Another transition diagram for unsigned numbers



A more complex transition diagram is difficult to implement and may give rise to errors during coding, however, there are ways to better implementation

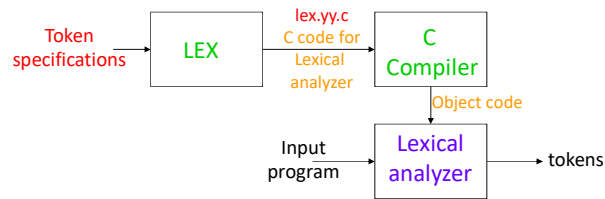
45

Lexical analyzer generator

- Input to the generator
 - List of regular expressions in priority order
 - Associated actions for each of regular expression (generates kind of token and other book keeping information)
- Output of the generator
 - Program that reads input character stream and breaks that into tokens
 - Reports lexical errors (unexpected characters), if any

46

LEX: A lexical analyzer generator



Refer to LEX User's Manual

47

How does LEX work?

- Regular expressions describe the languages that can be recognized by finite automata
- Translate each token regular expression into a non deterministic finite automaton (NFA)
- Convert the NFA into an equivalent DFA
- Minimize the DFA to reduce number of states
- Emit code driven by the DFA tables

48