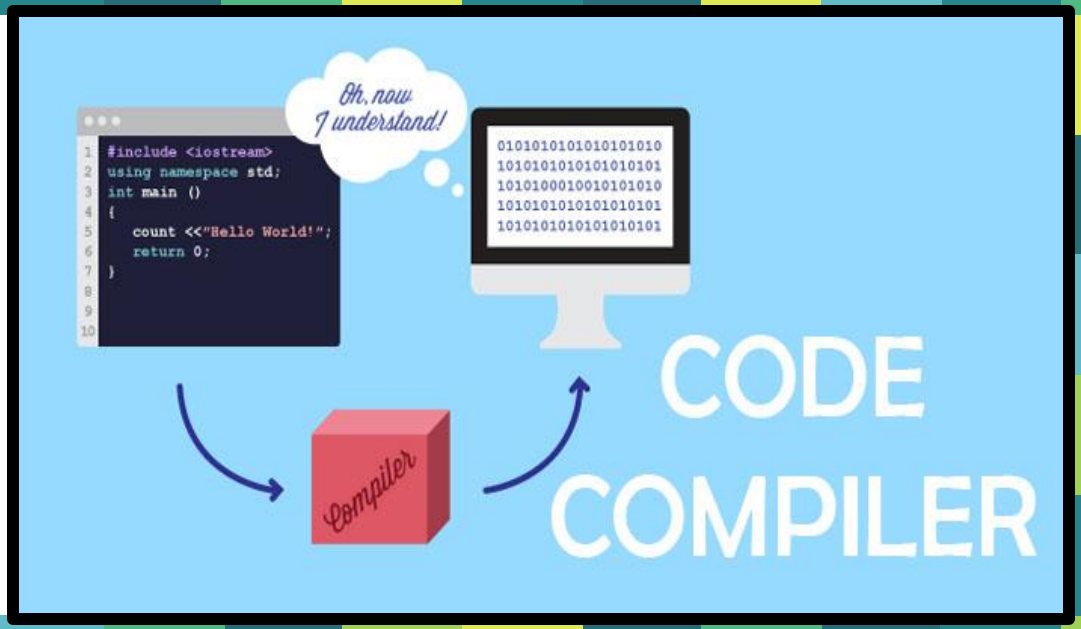
CSE- 303

Compiler

Chapter - 3



The Role of the Lexical Analyzer

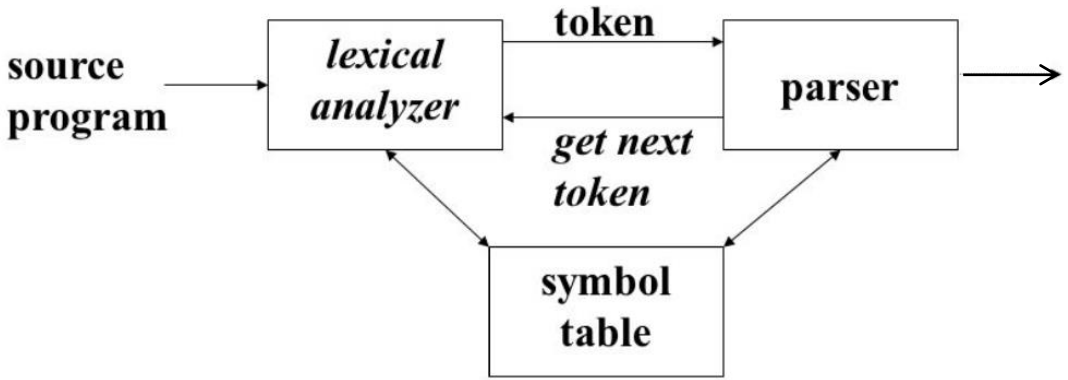
- As the first phase of a compiler, the main task of the lexical analyzer is to,

1) read the input characters of the source program,
2) group them into lexemes,
3) and produce as output a sequence of tokens for each lexeme in the source program.

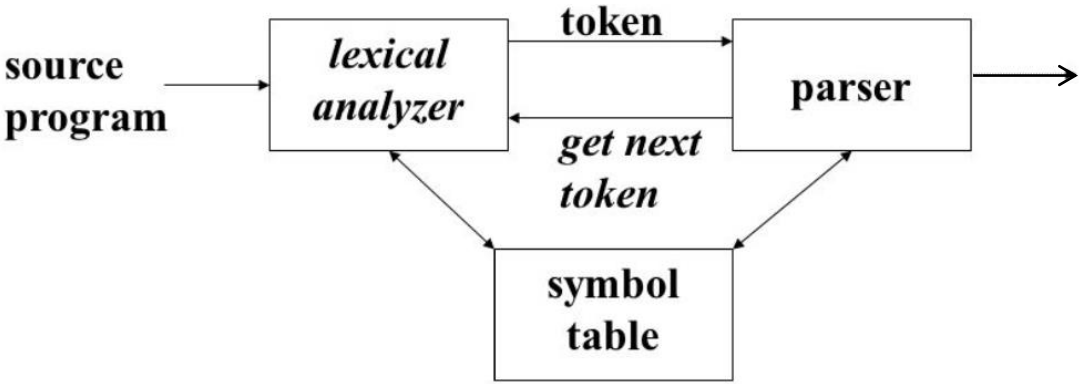
- The stream of tokens is sent to the parser for syntax analysis.
- It is common for the lexical analyzer to interact with the symbol table as well.

The Role of the Lexical Analyzer

- When the lexical analyzer discovers a lexeme constituting an identifier, it needs to enter that lexeme into the symbol table.
- In some cases, information regarding the kind of identifier may be read from the symbol table by the lexical analyzer to assist it in determining the proper token it must pass to the parser.



The Role of the Lexical Analyzer



- The interaction is implemented by having the parser call the lexical analyzer.
- The call, suggested by the getNextToken command, causes the lexical analyzer to read characters from its input until it can identify the next lexeme and produce for it the next token, which it returns to the parser.

The Role of the Lexical Analyzer

- It may perform certain other tasks such as:
  - stripping out comments and whitespace
  - correlating error messages generated by the compiler with the source program. For instance, the lexical analyzer may keep track of the number of newline characters seen, so it can associate a line number with each error message.
  - In some compilers, the lexical analyzer makes a copy of the source program with the error messages inserted at the appropriate positions.
  - If the source program uses a macro-preprocessor, the expansion of macros may also be performed by the lexical analyzer.

The Role of the Lexical Analyzer

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

a) Scanning consists of the simple processes that do not require tokenization of the input, such as deletion of comments and compaction of consecutive whitespace characters into one.

b) Lexical analysis proper is the more complex portion, where the scanner produces the sequence of tokens as output.

Lexical Analysis Versus Parsing

- Reasons behind separating the analysis portion of a compiler into lexical analysis and parsing (syntax analysis) phases are:

1. Simplicity of design is the most important consideration. Lexical analysis
2. Compiler efficiency is improved.
3. Compiler portability is enhanced.

Tokens, Patterns, and Lexemes

- Token: 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.
  - We shall generally write the name of a token in boldface.
  - We will often refer to a token by its token name.

Tokens, Patterns, and Lexemes

- Pattern: is a description of the form that the lexemes of a token may take.
  - In the case of a keyword as a token, the pattern is just the sequence of characters that form the keyword.
  - For identifiers and some other tokens, the pattern is a more complex structure that is matched by many strings.

Tokens, Patterns, and Lexemes

- 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.

TOKEN	INFORMAL DESCRIPTION	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"`

Examples of tokens

Tokens, Patterns, and Lexemes

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

- printf, score → matches pattern of id
- "Total = %d\n" → matches pattern of literal

- In many programming languages, the following classes cover most or all of the tokens:

1) One token for each keyword. The pattern for a keyword is the same as the keyword itself.
2) Tokens for the operators, either individually or in classes such as the token comparison mentioned.

Tokens, Patterns, and Lexemes

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.

Attributes for Tokens

- When more than one lexeme can match a pattern, the lexical analyzer must provide the subsequent compiler phases additional information about the particular lexeme that matched.
- For example, the pattern for token number matches both 0 and 1, but it is extremely important for the code generator to know which lexeme was found in the source program.
- Thus, in many cases the lexical analyzer returns to the parser not only a token name, but an attribute value that describes the lexeme represented by the token.

Attributes for Tokens

- We shall assume that tokens have at most one associated attribute, although this attribute may have a structure that combines several pieces of information.
- The most important example is the token id, where we need to associate with the token a great deal of information.
- Normally, information about an identifier is its lexeme, its type, and the location at which it is first found (in case an error message about that identifier must be issued) — is kept in the symbol table.
- Thus, the appropriate attribute value for an identifier is a pointer to the symbol-table entry for that identifier.

Attributes for Tokens

- Example: 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>

- In certain pairs, especially operators, punctuation, and keywords, there is no need for an attribute value.

Lexical Errors

- It is hard for a lexical analyzer to tell, without the aid of other components, that there is a source-code error.
- Example:

  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 lexeme for the token id, the lexical analyzer must return the token id to the parser and let some other phase of the compiler — probably the parser in this case — handle an error due to transposition of the letters.

Lexical Errors

- 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.

- Typical lexical errors are
  - Exceeding length of identifier or numeric constants.
  - Appearance of illegal characters
  - Unmatched string

- The simplest recovery strategy is “panic mode” recovery.

Lexical Errors

- We delete successive characters from the remaining input until the lexical analyzer can find a well-formed token.
- **Advantage:** it is easy to implement and guarantees not to go to infinite loop
- **Disadvantage:** a considerable amount of input is skipped without checking it for additional errors

- Other possible error-recovery actions are:

1. deleting an extraneous character,
2. inserting a missing character,
3. replacing an incorrect character by a correct character,
4. transposing two adjacent characters.

Lexical Errors

lexerror1.cpp

```
#include <iostream>
int main()
{
   `int i, j, k;
   return 0;
}
```

Response from gcc

```
lexerror1.cpp:4: error: stray ` in program
```

Lexical Errors

lexerror2.cpp

```
int main()
{
  int 5test;

  return 0;
}
```

Response from gcc

```
lexerror2.cpp:3:7: error: invalid suffix
"test" on integer constant
```

Lexical Errors

- Transformations like these may be tried in an attempt to repair the input.
- The simplest such strategy is to see whether a prefix of the remaining input can be transformed into a valid lexeme by a single transformation.
- This strategy makes sense, since in practice most lexical errors involve a single character.
- A more general correction strategy is to find the smallest number of transformations needed to convert the source program into one that consists only of valid lexemes.
- But this approach is considered too expensive in practice to be worth the effort.

Lexical Errors

- Exercise:

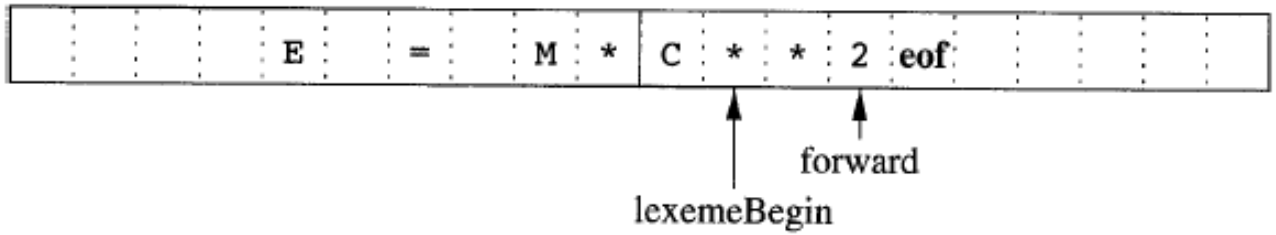
```
float limitedSquare(x)
{
float x;
/* returns x-squared, but never more than 100 */
return (x <= -10.0 || x >= 10.0) ? 100 : x*x;
}
```

Input Buffering

- Let us examine some ways that the simple but important task of reading the source program can be speeded.
- This task is made difficult by the fact that we often have to look one or more characters beyond the next lexeme before we can be sure we have the right lexeme.
- There are many situations where we need to look at least one additional character ahead.
- For instance, we cannot be sure we've seen the end of an identifier until we see a character that is not a letter or digit, and therefore is not part of the lexeme for id.
- In C, single-character operators like −, =, or < could also be the beginning of a two-character operator like − >, ==, or <=.

Input Buffering

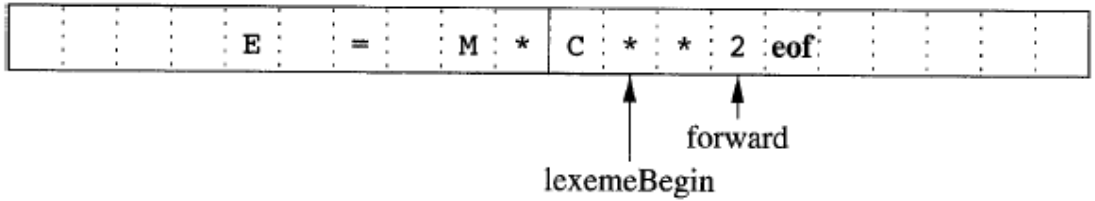
- Thus, we shall introduce a two-buffer scheme that handles large lookaheads safely.
- We then consider an improvement involving “sentinels” that saves time checking for the ends of buffers.



Using a pair of input buffers

Buffer Pairs

- Specialized buffering techniques have been developed to reduce the amount of overhead required to process a single input character.
- An important scheme involves two buffers that are alternately reloaded.

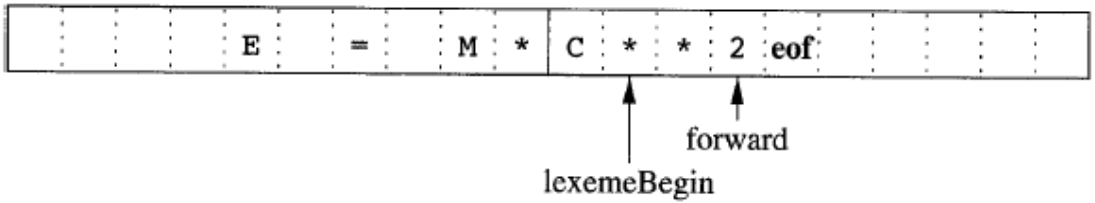


Using a pair of input buffers

- Each buffer is of the same size **N**.
- **N** is usually the size of a disk block, e.g., 4096 bytes.
- Using one system read command we can read **N** characters into a buffer, rather than using one system call per character.

Buffer Pairs

- If fewer than N characters remain in the input file, then a special character, represented by eof, marks the end of the source file.
- This eof is different from any possible character of the source program.

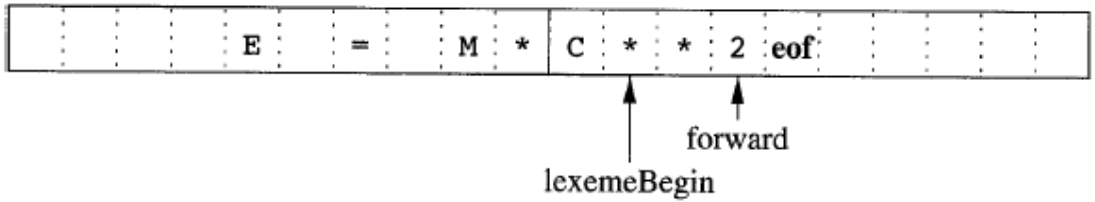


Using a pair of input buffers

- 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.

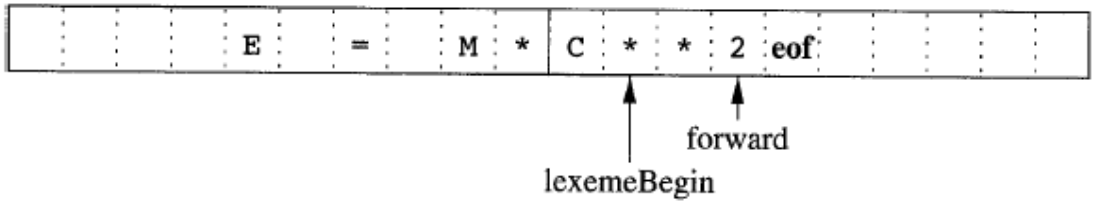
Buffer Pairs



Using a pair of input buffers

- 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.
- In figure, we see forward has passed the end of the next lexeme, ** (the Fortran exponentiation operator), and must be retracted one position to its left.

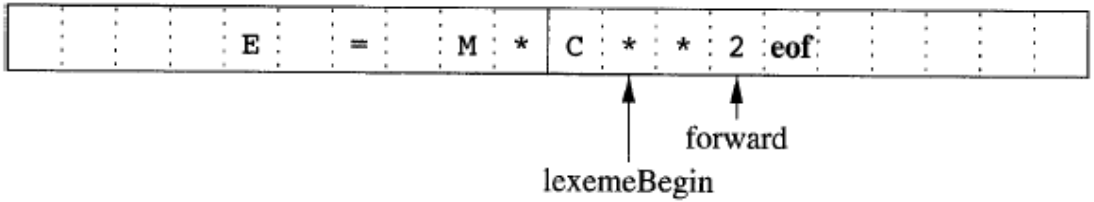
Buffer Pairs



Using a pair of input buffers

- Advancing forward requires that we first test whether we have reached the end of one of the buffers.
- If so, we must reload the other buffer from the input.
- And move forward to the beginning of the newly loaded buffer.
- As long as we never need to look so far ahead of the actual lexeme that the sum of the lexeme's length plus the distance we look ahead is greater than **N**, we shall never overwrite the lexeme in its buffer before determining it.

Sentinels



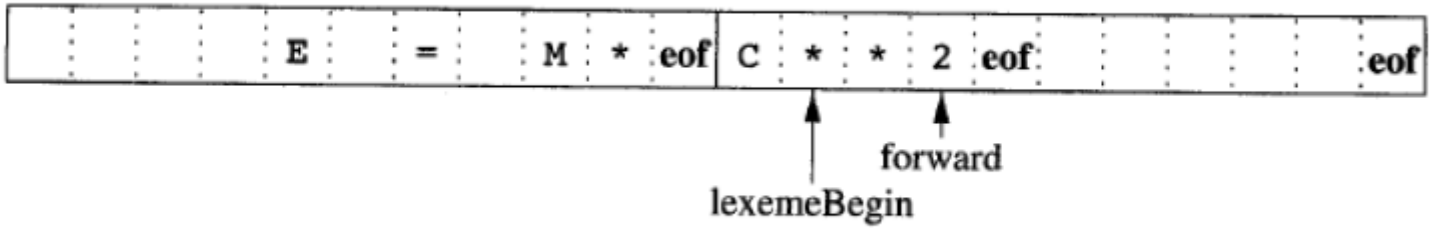
Using a pair of input buffers

- If we use the previous scheme as described, we must check, each time we advance forward, that we have not moved off one of the buffers.
- If we do, then we must also reload the other buffer.
- Thus, for each character read, we make two tests:

1. One for the end of the buffer.
2. And one to determine what character is read

Sentinels

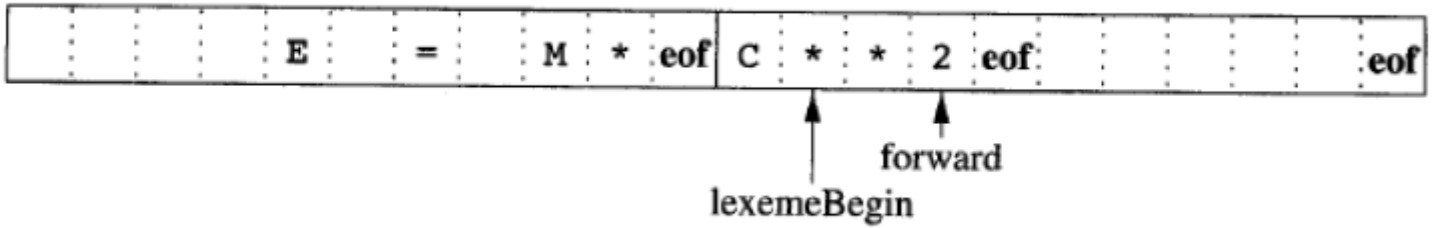
- We can combine the buffer-end test with the test for the current character if we extend each buffer to hold a sentinel character at the end.
- The sentinel is a special character that cannot be part of the source program, and a natural choice is the character eof.



Sentinels at the end of each buffer

Sentinels

- Figure shows the same arrangement as previous, but with the sentinels added.
- Note that eof retains its use as a marker for the end of the entire input.
- Any eof that appears other than at the end of a buffer means that the input is at an end.



Sentinels at the end of each buffer

Sentinels

- Notice how the first test, which can be part of a multiway branch based on the character pointed to by forward, is the only test we make, except in the case where we actually are at the end of a buffer or the end of the input.

```
switch ( *forward++ ) {
    case eof:
        if (forward is at end of first buffer ) {
            reload second buffer;
            forward = beginning of second buffer;
        }
        else if (forward is at end of second buffer ) {
            reload first buffer;
            forward = beginning of first buffer;
        }
        else /* eof within a buffer marks the end of input */
            terminate lexical analysis;
        break;
    Cases for the other characters
}
```

Lookahead code with sentinels

Recognition of Tokens

- We can express patterns using regular expressions.
- Our discussion will make use of the following running
- The grammar fragment describes a simple form of branching statements and conditional expressions.

stmt	→	**if** *expr* **then** *stmt*
	|	**if** *expr* **then** *stmt* **else** *stmt*
	|	ϵ
expr	→	*term* **relop** *term*
	|	*term*
term	→	**id**
	|	**number**

A grammar for branching statements

Recognition of Tokens

- The terminals of the grammar, which are **if**, **then**, **else**, **relop**, **id**, and **number**, are the names of tokens as far as the lexical analyzer is concerned.

stmt	→	**if** *expr* **then** *stmt*
	|	**if** *expr* **then** *stmt* **else** *stmt*
	|	ϵ
expr	→	*term* **relop** *term*
	|	*term*
term	→	**id**
	|	**number**

A grammar for branching statements

Recognition of Tokens

- The patterns for these tokens are described using regular definitions.

digit	→	`[0-9]`
digits	→	*digit*$^+$
number	→	*digits* (`.` *digits*)? (`E` `[+-]`? *digits*)?
letter	→	`[A-Za-z]`
id	→	*letter* (*letter* | *digit*)*
if	→	`if`
then	→	`then`
else	→	`else`
relop	→	`<` | `>` | `<=` | `>=` | `=` | `<>`

Patterns for tokens of Example

Recognition of Tokens

- For this language, the lexical analyzer will recognize the keywords **if**, **then**, and **else**, as well as lexemes that match the patterns for **relop**, **id**, and **number**.

digit	→	`[0-9]`
digits	→	*digit*$^+$
number	→	*digits* (. *digits*)? (`E` `[+-]`? *digits*)?
letter	→	`[A-Za-z]`
id	→	*letter* (*letter* | *digit*)*
if	→	`if`
then	→	`then`
else	→	`else`
relop	→	`<` | `>` | `<=` | `>=` | `=` | `<>`

Patterns for tokens of Example

Recognition of Tokens

- To simplify matters, we make the common assumption that keywords are also reserved words.
- They are not identifiers, even though their lexemes match the pattern for identifiers.
- In addition, we assign the lexical analyzer the job of stripping out white-space, by recognizing the “token” ws defined by:

$$ws \rightarrow (\textbf{blank} \mid \textbf{tab} \mid \textbf{newline})^{+}$$

- Here, blank, tab, and newline are abstract symbols that we use to express the ASCII characters of the same names.

Recognition of Tokens

- Token ws is different from the other tokens in that, when we recognize it, we do not return it to the parser.
- We rather restart the lexical analysis from the character that follows the whitespace.
- It is the following token that gets returned to the parser.

Recognition of Tokens

LEXEMES	TOKEN NAME	ATTRIBUTE VALUE
Any *ws*	–	–
`if`	**if**	–
`then`	**then**	–
`else`	**else**	–
Any *id*	**id**	Pointer to table entry
Any *number*	**number**	Pointer to table entry
`<`	**relop**	LT
`<=`	**relop**	LE
`=`	**relop**	EQ
`<>`	**relop**	NE
`>`	**relop**	GT
`>=`	**relop**	GE

Tokens, their patterns, and attribute values

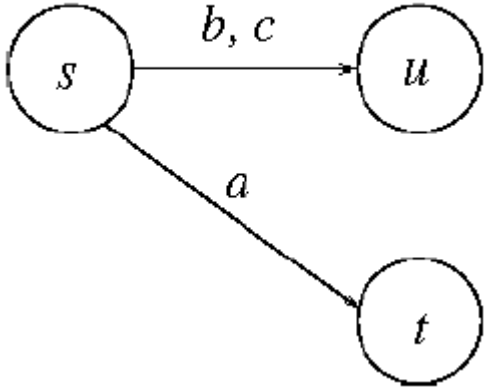
- Our goal for the lexical analyzer is summarized in figure.

Transition Diagrams

- As an intermediate step in the construction of a lexical analyzer, we first convert patterns into stylized flowcharts, called “transition diagrams.”
- In this section, we perform the conversion from regular-expression patterns to transition diagrams by hand.
- Later we shall see that there is a mechanical way to construct these diagrams from collections of regular expressions.
- Transition diagrams have a collection of nodes or circles, called states.
- We may think of a state as summarizing all we need to know about what characters we have seen between the lexemeBegin pointer and the forward pointer.

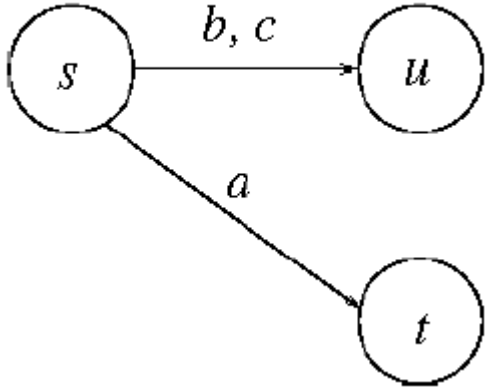
Transition Diagram Rules

- Edges are directed from one state of the transition diagram to another.
- Each edge is labeled by a symbol or set of symbols.
- If we are in some state s, and the next input symbol is a, we look for an edge out of state s labeled by a.
- If we find such an edge, we advance the forward pointer and enter the state of the transition diagram to which that edge leads.



Transition Diagram Rules

- We shall assume that all our transition diagrams are deterministic, meaning that there is never more than one edge out of a given state with a given symbol among its labels.
- Later we shall relax the condition of determinism, making life much easier for the designer of a lexical analyzer, although trickier for the implementer.



Transition Diagram Rules

- 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 consist of all positions between the lexemeBegin and forward pointers.
   - We always indicate an accepting state by a double circle.
   - 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.

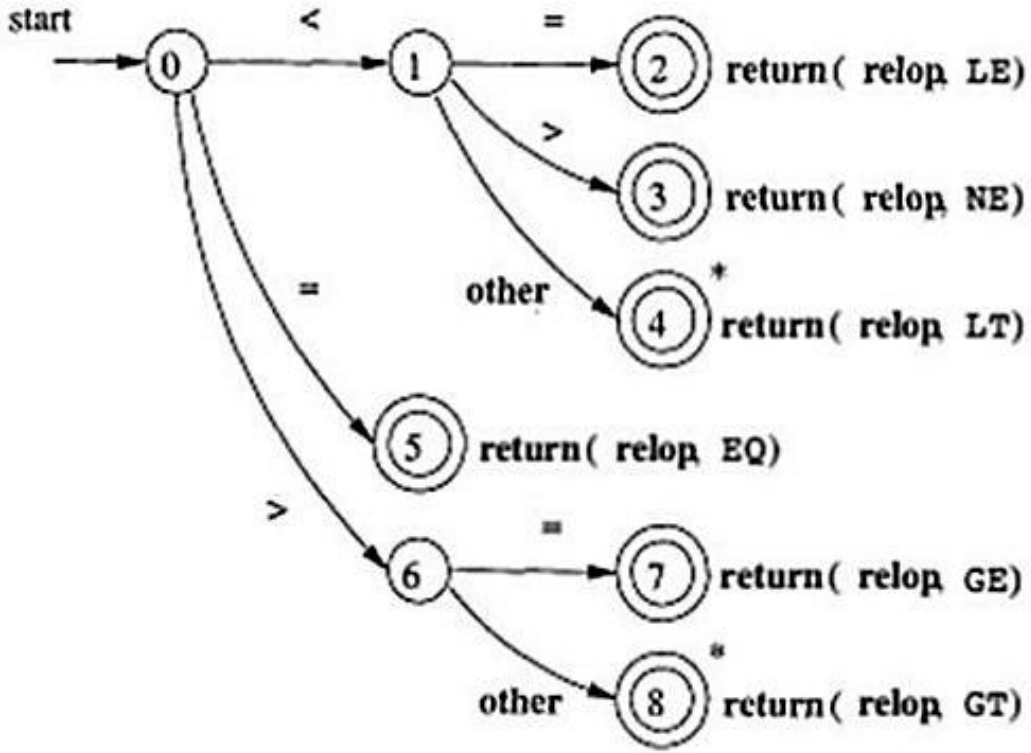
Transition Diagram Rules

- Some important conventions about transition diagrams are:

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 a * near that accepting state.
   - We can attach any number of *'s to the accepting state.

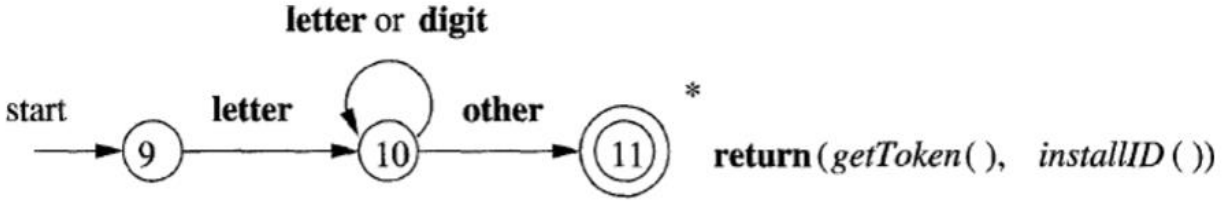
3. One state is designated the start state, or initial state it is indicated by an edge, labeled "start ," entering from nowhere.
   - The transition diagram always begins in the start state before any input symbols have been read.

Transition Diagram Example



Recognition of Reserved Words and Identifiers

- Recognizing keywords and identifiers presents a problem.
- We typically use a transition diagram like the following to search for identifier lexemes.

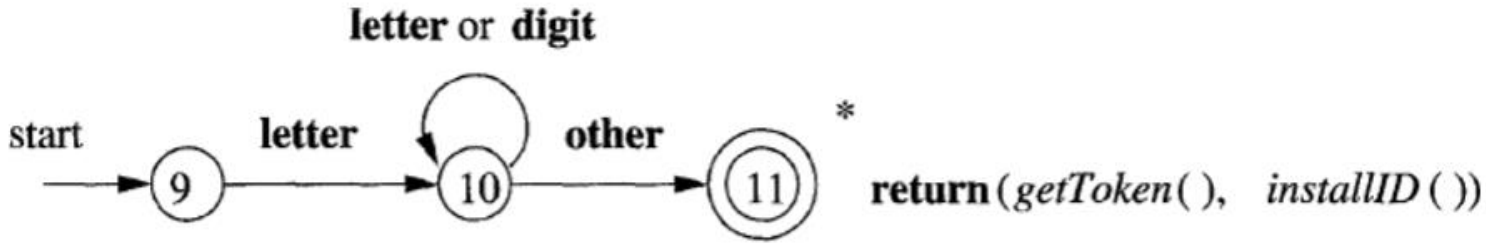


- But, this diagram will also recognize the keywords if, then, and else.
- But usually, keywords like if or then are reserved, so they are not identifiers even though they look like identifiers.

Recognition of Reserved Words and Identifiers

- There are two ways that we can handle reserved words 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.



Recognition of Reserved Words and Identifiers

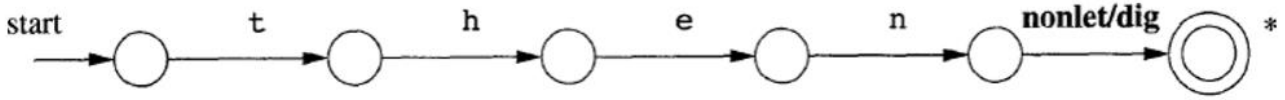
Steps:

- 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 lexeme represents - either id or one of the keyword tokens that was initially installed in the table.

Recognition of Reserved Words and Identifiers

- There are two ways that we can handle reserved words that look like identifiers.

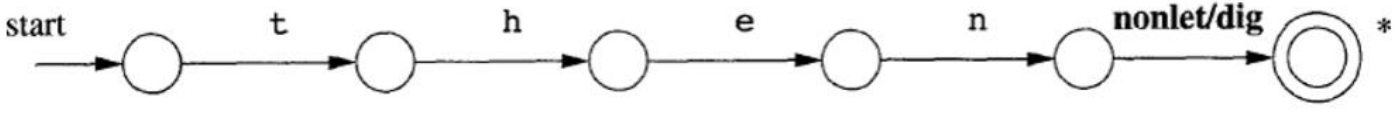
2. Create separate transition diagrams for each keyword: An example for the keyword then is shown in figure.



Hypothetical transition diagram for the keyword then

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

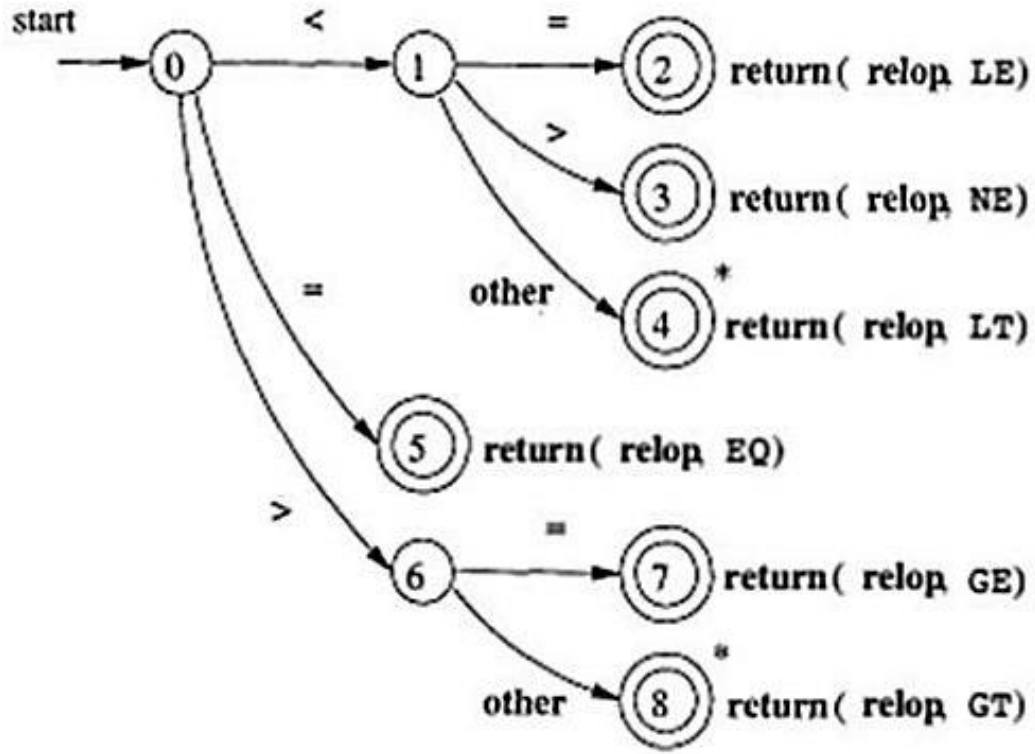
Recognition of Reserved Words and Identifiers



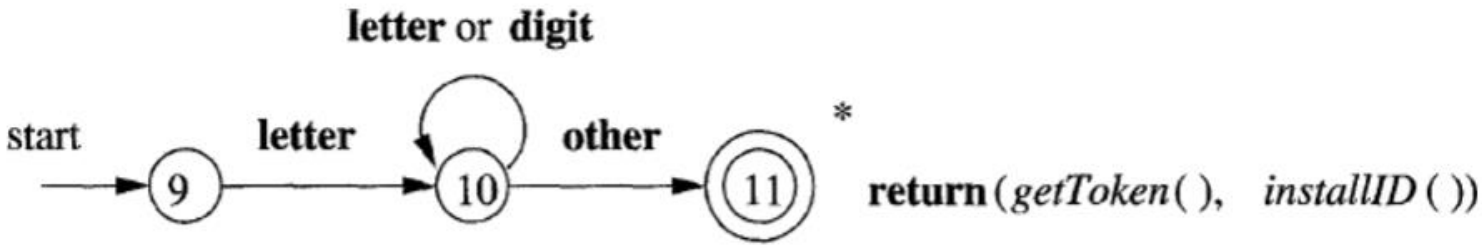
Hypothetical transition diagram for the keyword then

- 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 thenextvalue that has then as a proper prefix.
- If we adopt this approach, then we must prioritize the tokens so that the reserved-word tokens are recognized in preference to id, when the lexeme matches both patterns.

Transition Diagram Example (Relop)



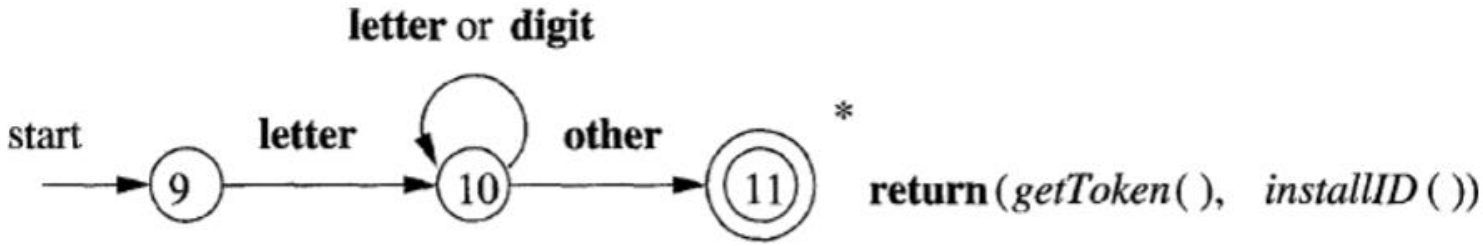
Transition Diagram Example (Id)



Steps:

- Starting in state 9, it checks that the lexeme begins with a letter and goes to state 10 if so.
- We stay in state 10 as long as the input contains letters and digits.
- When we first encounter anything but a letter or digit, we go to state 11 and accept the lexeme found.

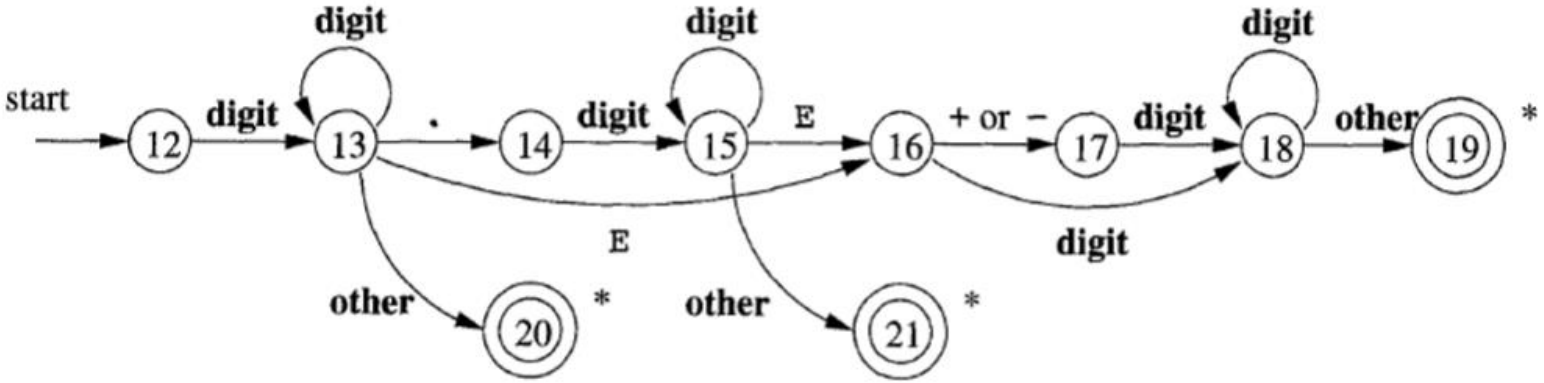
Transition Diagram Example (Id)



Steps:

- Since the last character is not part of the identifier, we must retract the input one position.
- We enter what we have found in the symbol table and determine whether we have a keyword or a true identifier.

Transition Diagram Example (Number)

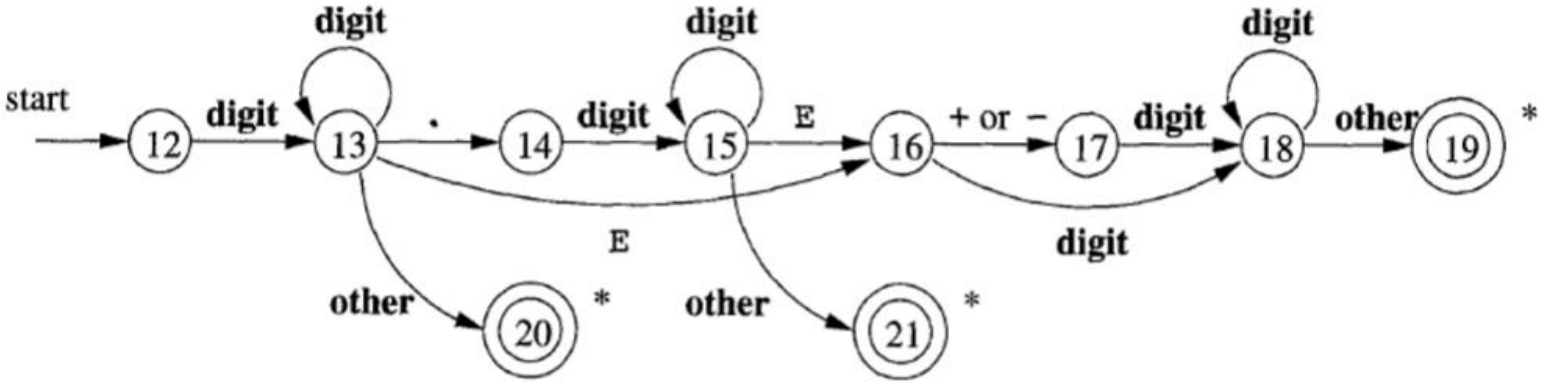


A transition diagram for unsigned numbers

Steps for Number 123:

- Beginning in state 12, if we see a digit, we go to state 13.
- In that state, we can read any number of additional digits.
- However, if we see anything but a digit or a dot, we have seen a number in the form of an integer.

Transition Diagram Example (Number)

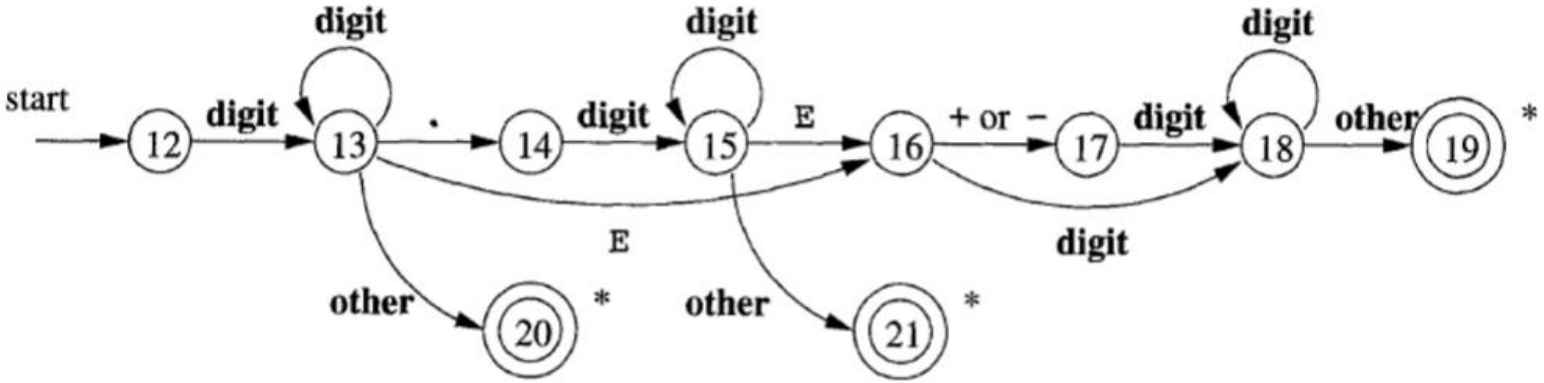


A transition diagram for unsigned numbers

Steps for Number 123:

- That case is handled by entering state 20, where we return token number and a pointer to a table of constants where the found lexeme is entered.
- These mechanics are not shown on the diagram but are analogous to the way we handled identifiers.

Transition Diagram Example (Number)

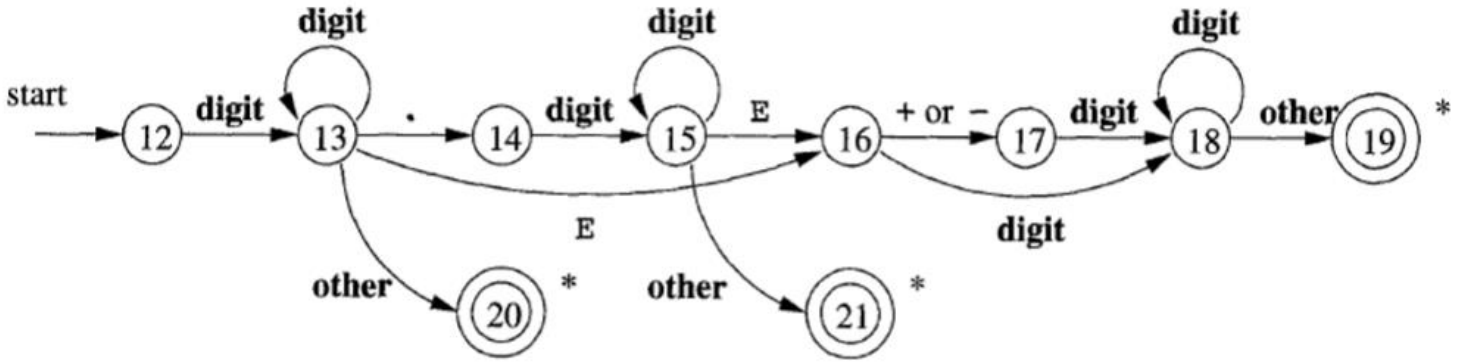


A transition diagram for unsigned numbers

Steps for Number 123.456 :

- If we instead see a dot in state 13, then we have an “optional fraction.”
- State 14 is entered, and we look for one or more additional digits; state 15 is used for that purpose.

Transition Diagram Example (Number)

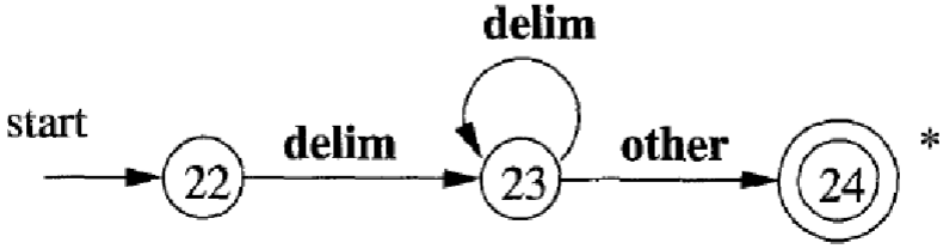


A transition diagram for unsigned numbers

Steps for Number 123.456E789, 123.456E+789, 123.456E-789 :

- If we see an E, then we have an “optional exponent,” whose recognition is the job of states 16 through 19.
- Should we, in state 15, instead see anything but E or a digit, then we have come to the end of the fraction, there is no exponent, and we return the lexeme found, via state 21.

Transition Diagram Example (Whitespace)

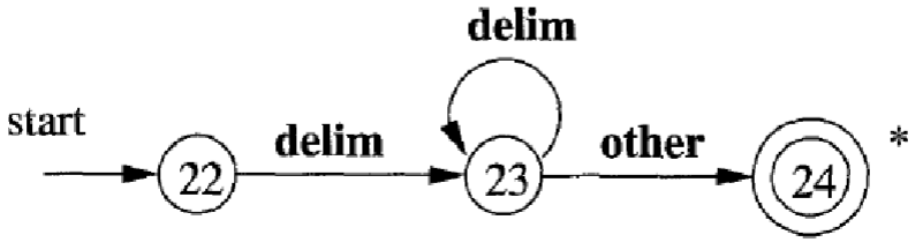


A transition diagram for whitespace

Steps:

- In that 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.

Transition Diagram Example (Whitespace)



A transition diagram for whitespace

Steps:

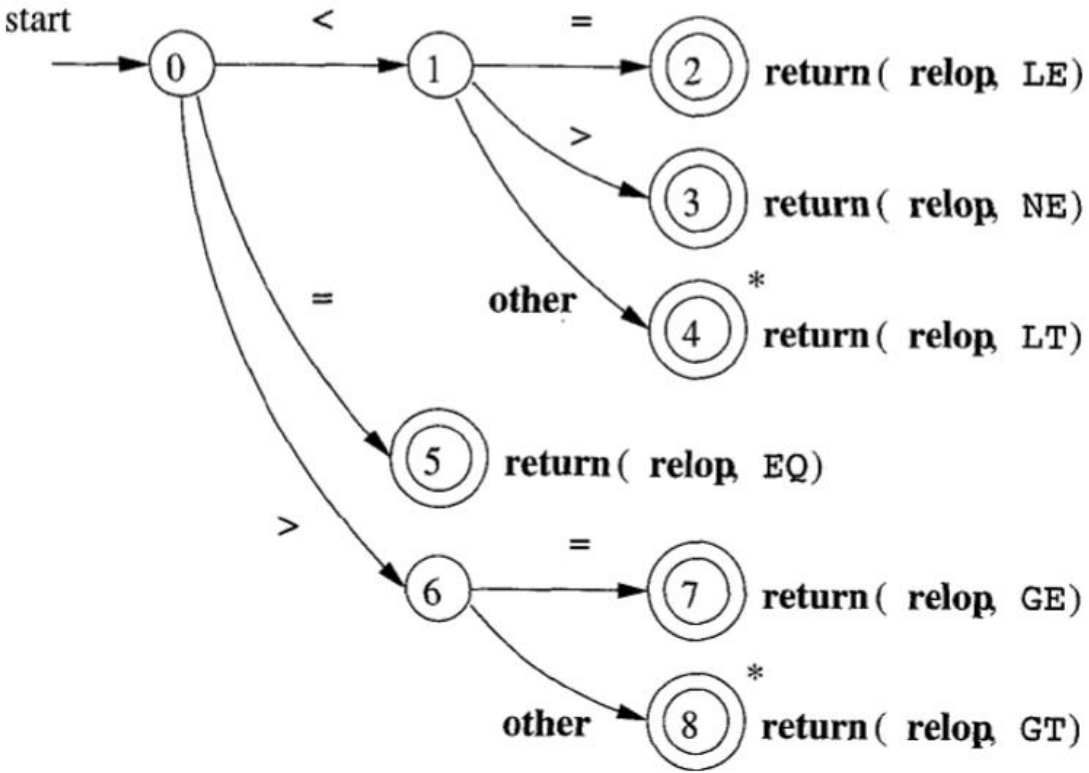
- Note that in state 24, we have found a block of consecutive whitespace characters, followed by a nonwhitespace character.
- We retract the input to begin at the nonwhitespace, but we do not return to the parser.
- Rather, we must restart the process of lexical analysis after the whitespace.

Architecture of a Transition-Diagram Based Lexical Analyzer

- There are several ways that a collection of transition diagrams can be used to build a lexical analyzer.
- Regardless of the overall strategy, each state is represented by a piece of code.
- We may imagine a variable state holding the number of the current state for a transition diagram.
- A switch based on the value of state takes us to code for each of the possible states, where we find the action of that state.
- Often, the code for a state is itself a switch statement or multiway branch that determines the next state by reading and examining the next input character.

Example

In figure we see a sketch of getRelop(), a C++ function whose job is to simulate the transition diagram for relop.



Example

In figure we see a sketch of getRelop(), a C++ function whose job is to simulate the transition diagram for relop.

```
TOKEN getRelop()
{
    TOKEN retToken = new(RELOP);
    while(1) { /* repeat character processing until a return
                  or failure occurs */
        switch(state) {
            case 0: c = nextChar();
                    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);
        }
    }
}
```

Example

- Function getRelop() returns an object of type TOKEN, that is, a pair consisting of the token name **relop** and an attribute value (the code for one of the six comparison operators in this case).
- getRelop() first creates a new object retToken and initializes its first component to RELOP, the symbolic code for token **relop**.
- We see the typical behavior of a state in case 0, the case where the current state is 0.
- A function nextchar() obtains the next character from the input and assigns it to local variable c.
- We then check c for the three characters we expect to find, making the state transition dictated by the transition diagram in each case.

Example

- For example, if the next input character is =, we go to state 5.
- If the next input character is not one that can begin a comparison operator, then a function fail() is called.
- What fail() does depends on the global error recovery strategy of the lexical analyzer.
- It should reset the forward pointer to lexemeBegin, in order to allow another transition diagram to be applied to the true beginning of the unprocessed input.
- It might then change the value of state to be the start state for another transition diagram, which will search for another token.

Example

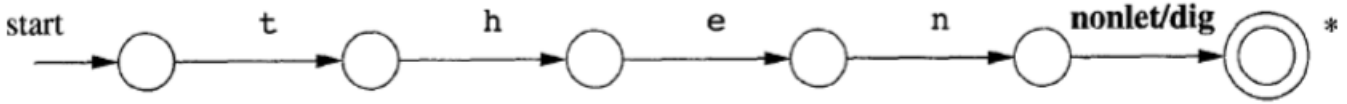
- Alternatively, if there is no other transition diagram that remains unused, fail() could initiate an error-correction phase that will try to repair the input and find a lexeme.
- We also show the action for state 8.
- Because state 8 bears a $*$, we must retract the input pointer one position (i.e., put c back on the input stream).
- That task is accomplished by the function retract().
- Since state 8 represents the recognition of lexeme >=, we set the second component of the returned object, which we suppose is named attribute, to GT, the code for this operator.

Architecture of a Transition-Diagram Based Lexical Analyzer

- Let us consider the ways code could fit into the entire lexical analyzer.

1. Arrange for the transition diagrams for each token to be tried sequentially:

- Then, the function fail() resets the pointer forward and starts the next transition diagram, each time it is called.
- This method allows us to use transition diagrams for the individual keywords.



- We have only to use these before we use the diagram for id, in order for the keywords to be reserved words.

Architecture of a Transition-Diagram Based Lexical Analyzer

2. We could run the various transition diagrams “in parallel,” feeding the next input character to all of them and allowing each one to make whatever transitions it required:

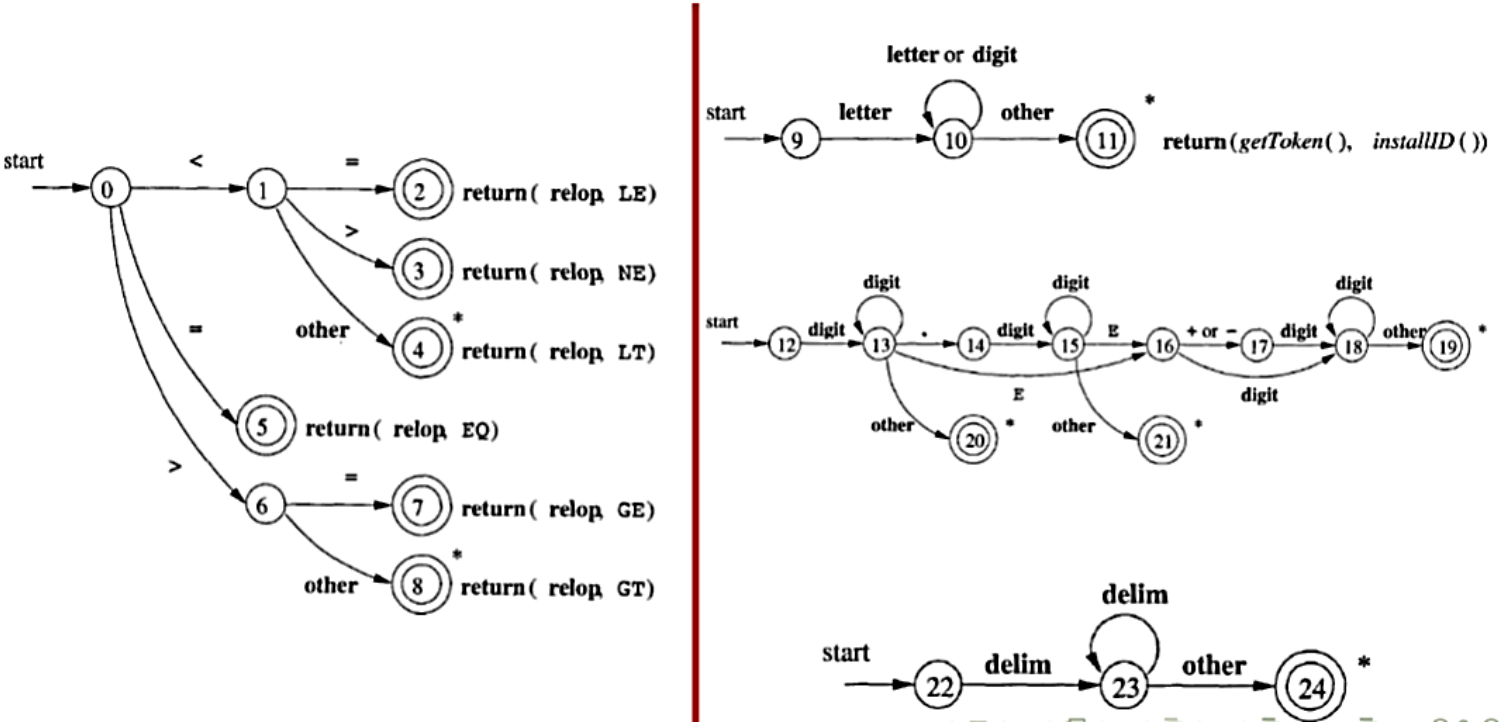
- If we use this strategy, we must be careful to resolve the case where one diagram finds a lexeme that matches its pattern, while one or more other diagrams are still able to process input.
- The normal strategy is to take the longest prefix of the input that matches any pattern.
- That rule allows us to prefer identifier thenext to keyword then, or the operator -> to -, for example.

Architecture of a Transition-Diagram Based Lexical Analyzer

3. The preferred approach, is to combine all the transition diagrams into one:

- We allow the transition diagram to read input until there is no possible next state. And then take the longest lexeme that matched any pattern.
- In our running example, this combination is easy, because no two tokens can start with the same character.
- The first character immediately tells us which token we are looking for.
- Thus, we could simply combine states 0, 9, 12, and 22 into one start state, leaving other transitions intact.
- However, in general, the problem of combining transition diagrams for several tokens is more complex.

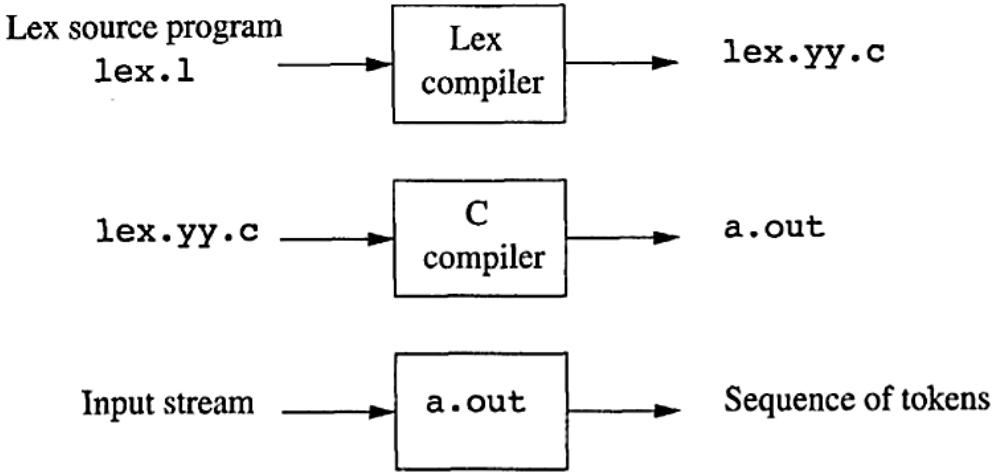
Architecture of a Transition-Diagram Based Lexical Analyzer



The Lexical- Analyzer Generator Lex

- We introduce a tool called Lex, or in a more recent implementation Flex.
- This allows one to specify a lexical analyzer by specifying regular expressions to describe patterns for tokens.
- The input notation for the Lex tool is referred to as the Lex language and the tool itself is the Lex compiler.
- Behind the scenes, the Lex compiler transforms the input patterns into a transition diagram and generates code, in a file called lex.yy.c, that simulates this transition diagram.

Use of Lex



- The attribute value is placed in a global variable yylval which is shared between the lexical analyzer and parser, thereby making it simple to return both the name and an attribute value of a token.

Structure of Lex Programs

- A Lex program has the following form:

```
declarations
%%
translation rules
%%
auxiliary functions
```

- The declarations section includes declarations of variables, manifest constants (identifiers declared to stand for a constant, e.g., the name of a token), and regular definitions.

Structure of Lex Programs

- The translation rules each have the form

```
Pattern { Action }
```

  - Each pattern is a regular expression, which may use the regular definitions of the declaration section.
  - The actions are fragments of code, typically written in C, although many variants of Lex using other languages have been created.

- The auxiliary functions holds whatever additional functions are used in the actions. Alternatively, these functions can be compiled separately and loaded with the lexical analyzer.

Structure of Lex Programs

- The lexical analyzer created by Lex behaves in concert with the parser as follows:

1. When called by the parser, the lexical analyzer begins reading its remaining input, one character at a time, until it finds the longest prefix of the input that matches one of the patterns *Pi*.
2. It then executes the associated action *Ai*.
3. Typically, *Ai* will return to the parser.
4. But if it does not (e.g., because *Pi* describes whitespace or comments), then the lexical analyzer proceeds to find additional lexemes, until one of the corresponding actions causes a return to the parser.
5. The lexical analyzer returns a single value, the token name, to the parser, but uses the shared, integer variable yylval to pass additional information about the lexeme found, if needed.

Example

- Figure is a Lex program that recognizes the tokens of the definitions and returns the token found.

Declarations

```
%{
    /* definitions of manifest constants
    LT, LE, EQ, NE, GT, GE,
    IF, THEN, ELSE, ID, NUMBER, RELOP */
%}

/* regular definitions */
delim     [ \t\n]
ws        {delim}+
letter    [A-Za-z]
digit     [0-9]
id        {letter}({letter}|{digit})*
number    {digit}+(\.{digit}+)?(E[+-]?{digit}+)?
```

Translation Rules

```
%%

{ws}      {/* no action and no return */}
if        {return(IF);}
then      {return(THEN);}
else      {return(ELSE);}
{id}      {yylval = (int) installID(); return(ID);}
{number}  {yylval = (int) installNum(); return(NUMBER);}
"<"       {yylval = LT; return(RELOP);}
"<="      {yylval = LE; return(RELOP);}
"="       {yylval = EQ; return(RELOP);}
"<>"      {yylval = NE; return(RELOP);}
">"       {yylval = GT; return(RELOP);}
">="      {yylval = GE; return(RELOP);}
```

Example

- Figure is a Lex program that recognizes the tokens of the definitions and returns the token found.

Auxiliary Functions

```
%%

int installID() {/* function to install the lexeme, whose
                    first character is pointed to by yytext,
                    and whose length is yyleng, into the
                    symbol table and return a pointer
                    thereto */
}

int installNum() {/* similar to installID, but puts numer-
                     ical constants into a separate table */
}
```

LEXEMES	TOKEN NAME	ATTRIBUTE VALUE
Any *ws*	–	–
if	if	–
then	then	–
else	else	–
Any *id*	id	Pointer to table entry
Any *number*	number	Pointer to table entry
<	relop	LT
<=	relop	LE
=	relop	EQ
<>	relop	NE
>	relop	GT
>=	relop	GE

Declaration Section

```
%{
        /* definitions of manifest constants
        LT, LE, EQ, NE, GT, GE,
        IF, THEN, ELSE, ID, NUMBER, RELOP */
%}
```

- In the declarations section we see a pair of special brackets, %{and %}.
- Anything within these brackets is copied directly to the file lex.yy.c, and is not treated as a regular definition.
- It is common to place there the definitions of the manifest constants, using C #define statements to associate unique integer codes with each of the manifest constants.
- In our example, we have listed in a comment the names of the manifest constants, but have not shown them defined to be particular integer.

Declaration Section

```
/* regular definitions */
delim      [ \t\n]
ws         {delim}+
letter     [A-Za-z]
digit      [0-9]
id         {letter}({letter}|{digit})*
number     {digit}+(\.{digit}+)?(E[+-]?{digit}+)?

%%
```

- Also in the declarations section is a sequence of regular definitions.
- Notice that in the definition of id and number, parentheses are used as grouping metasymbols and do not stand for themselves. In contrast, E in the definition of number stands for itself.

Declaration Section

```
/* regular definitions */
delim      [ \t\n]
ws         {delim}+
letter     [A-Za-z]
digit      [0-9]
id         {letter}({letter}|{digit})*
number     {digit}+(\.{digit}+)?(E[+-]?{digit}+)?

%%
```

- If we wish to use one of the Lex metasymbols, such as any of the parentheses, +, *, or ?, to stand for themselves, we may precede them with a backslash.

Declaration Section

- Note that, although keywords like if match this pattern as well as an earlier pattern, Lex chooses whichever pattern is listed first in situations where the longest matching prefix matches two or more patterns.

- The action taken when id is matched is threefold.

1. Function installID() is called to place the lexeme found in the symbol table.
2. This function returns a pointer to the symbol table, which is placed in global variable yylval, where it can be used by the parser or a later component of the compiler.
3. The token name ID is returned to the parser.

Translation Rules

```
{ws}        {/* no action and no return */}
if          {return(IF);}
then        {return(THEN);}
else        {return(ELSE);}
{id}        {yylval = (int) installID(); return(ID);}
{number}    {yylval = (int) installNum(); return(NUMBER);}
"<"         {yylval = LT; return(RELOP);}
"<="        {yylval = LE; return(RELOP);}
"="         {yylval = EQ; return(RELOP);}
"<>"        {yylval = NE; return(RELOP);}
">"         {yylval = GT; return(RELOP);}
">="        {yylval = GE; return(RELOP);}
```

Translation Rules

- Note that installID() has available to it two variables that are set automatically by the lexical analyzer that Lex generates:

a) yytext is a pointer to the beginning of the lexeme, analogous to lexemeBegin.
b) yyleng is the length of the lexeme found.

- The action taken when a lexeme matching the pattern number is similar, using the auxiliary function installNum().

Auxiliary Functions

```
int installID() {/* function to install the lexeme, whose
                    first character is pointed to by yytext,
                    and whose length is yyleng, into the
                    symbol table and return a pointer
                    thereto */
}

int installNum() {/* similar to installID, but puts numer-
                     ical constants into a separate table */
}
```

- In the auxiliary-function section, we see two such functions, installID() and installNum().
- Like the portion of the declaration section that appears between %{...%} everything in the auxiliary section is copied directly to file lex.yy.c, but may be used in the actions.

Conflict Resolution in Lex

- We have alluded to the two rules that Lex uses to decide on the proper lexeme to select, when several prefixes of the input match one or more patterns:

1) Always prefer a longer prefix to a shorter prefix.
2) If the longest possible prefix matches two or more patterns, prefer the pattern listed first in the Lex program.

Conflict Resolution in Lex

- The first rule tells us to continue reading letters and digits to find the longest prefix of these characters to group as an identifier.
- It also tells us to treat <= as a single lexeme, rather than selecting < as one lexeme and = as the next lexeme.
- The second rule makes keywords reserved, if we list the keywords before id in the program.
- For instance, if then is determined to be the longest prefix of the input that matches any pattern, and the pattern then precedes id, then the token THEN is returned, rather than ID.

Thank You