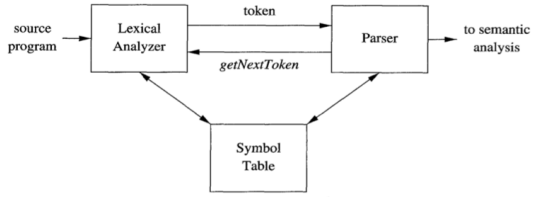
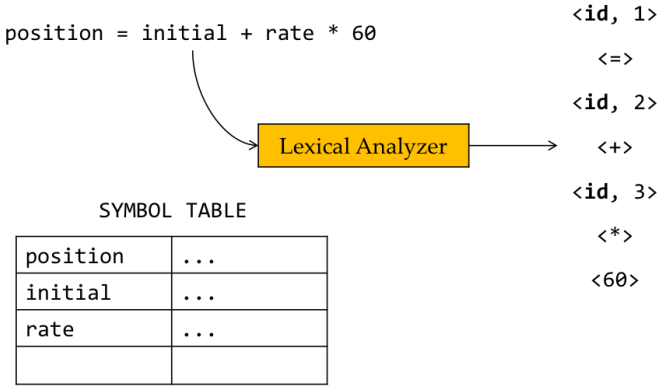
The Role of Lexical Analyzer

- Read the input characters of the source program, group them into lexemes, and produces a sequence of tokens
- Add lexemes into the symbol table when necessary

读入源程序的输入字符，生成token串.





Tokens, Patterns, and Lexemes

- A ***lexeme*** is a string of characters that is a lowest-level syntactic unit in programming languages
- A ***token*** is a syntactic category representing a class of lexemes. Formally, it is a pair <token name, attribute value>
  - Token name: an abstract symbol representing the kind of the token
  - Attribute value (optional) points to the symbol table
- Each token has a particular ***pattern***: a description of the form that the lexemes of the token may take

词素是最小的语法单元

token是一类词素

每个token都有特定的模式，以描述其类中的lexeme

Examples

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"`

Consider the C statement: `printf("Total = %d\n", score);`

Lexeme	printf	score	"Total = %d\n"	(	...
Token	id	id	literal	left_parenthesis	...

Attributes for Tokens

- When more than one lexeme match a pattern, the lexical analyzer must provide additional information, named *attribute values*, to the subsequent compiler phases
  - Token names influence parsing decisions
  - Attribute values influence semantic analysis, code generation etc.
- For example, an **id** token is often associated with: (1) its lexeme, (2) type, and (3) the location at which it is first found. Token attributes are stored in the symbol table.

```
A = B * 2  ⟶  <id, pointer to symbol-table entry for A>
               <assign_op>
               <id, pointer to symbol-table entry for B>
               <mult_op> <number, integer value 2>
```

token → parsing
attribute value → 语义分析，代码生成

Lexical Errors

- When none of the patterns for tokens match any prefix of the remaining input
- Example: `int 3a = a * 3;`

当没有匹配上任何token的模式时会报词法错误

Lexical errors and syntax errors in Java (learn by yourself):

- https://www.javatpoint.com/lexical-error
- https://www.javatpoint.com/syntax-error

Specification of Tokens

- **Regular expression (正则表达式, regexp for short)** is an important notation for specifying lexeme patterns
- Content of this part
  - Strings and Languages (串和语言)
  - Operations on Languages (语言上的运算)
  - Regular Expressions
  - Regular Definitions (正则定义)
  - Extensions of Regular Expressions

Strings and Languages

- **Alphabet (字母表)**: any finite set of symbols
  - Examples of symbols: letters, digits, and punctuations
  - Examples of alphabets: {1, 0}, ASCII, Unicode
- A **string (串)** over an alphabet is a finite sequence of symbols drawn from the alphabet
  - The length of a string s, denoted $|s|$, is the number of symbols in s (i.e., cardinality)
  - Empty string (空串): the string of length 0, ϵ

Terms : banana

- **Prefix (前缀) of string *s*:** any string obtained by removing 0 or more symbols from the end of *s* (ban, banana, ϵ)
- **Proper prefix (真前缀):** a prefix that is not ϵ and not *s* itself (ban)
- **Suffix (后缀):** any string obtained by removing 0 or more symbols from the beginning of *s* (nana, banana, ϵ).
- **Proper suffix (真后缀):** a suffix that is not ϵ and not equal to *s* itself (nana)
- **Substring (子串) of s:** any string obtained by removing any prefix and any suffix from *s* (banana, nan, ϵ)
- **Proper substring (真子串):** a substring that is not ϵ and not equal to *s* itself (nan)
- **Subsequence (子序列):** any string formed by removing 0 or more not necessarily consecutive symbols from *s* (bnn)

How many substrings & subsequences does banana have?

(Two substrings are different as long as they have different start/end index)

对串S，若|S| = n，则有

前(后)缀 n+1 个，真前(后)缀 n-1 个

子串 2n 个

子序列 $\sum_{i=0}^{n} C_n^i = 2^n$

子序列可以不连续移除

String Operations

- **Concatenation (连接):** the concatenation of two strings *x* and *y*, denoted *xy*, is the string formed by appending *y* to *x*
  - *x* = dog, *y* = house, *xy* = doghouse
- **Exponentiation (幂/指数运算):** $s^0 = \epsilon$, $s^1 = s$, $s^i = s^{i-1}s$
  - *x* = dog, $x^0 = \epsilon$, x^1 = dog, x^3 = dogdogdog

Language

- A **language** is any **countable set**[1] of strings over some fixed alphabet
  - The set containing only the empty string, that is $\{\epsilon\}$, is a language, denoted $\emptyset$
  - The set of all grammatically correct English sentences
  - The set of all syntactically well-formed C programs

[1] In mathematics, a countable set is a set with the same cardinality (number of elements) as some subset of the set of natural numbers. A countable set is either a finite set or a countably infinite set.

字符表固定、串可数.

Operations on Language

- 并，连接，Kleene闭包，正闭包

Operation	Definition and 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	$L^* = \cup_{i=0}^{\infty} L^i$
Positive closure of L	$L^+ = \cup_{i=1}^{\infty} L^i$

The exponentiation of L can be defined using concatenation. L^n means concatenating L n times.

https://en.wikipedia.org/wiki/Stephen_Cole_Kleene

Kleene闭包中含有空串而正闭包不一定有.

Examples

- L = {A, B, …, Z, a, b, …, z}
- D = {0, 1, …, 9}

$L \cup D$	{A, B, …, Z, a, b, …, z, 0, 1, …,9}
LD	the set of 520 strings of length two, each consisting of one letter followed by one digit
L^4	the set of all 4-letter strings
L^*	the set of all strings of letters, including ϵ
$L(L \cup D)^*$	?
D^+	?

Note: L, D might seem to be the alphabets of letters and digits. We define them to be languages: all strings happen to be of length one.

Regular Expressions : For Describing Languages/Patterns

Rules that define regexps over an alphabet Σ:

- **BASIS**: two rules form the basis:
  - ϵ is a regexp, $L(\epsilon) = \{\epsilon\}$
  - If a is a symbol in Σ, then a is a regexp, and $L(a) = \{a\}$
- **INDUCTION:** Suppose r and s are regexps denoting the languages L(r) and L(s)
  - (r)|(s) is a regexp denoting the language $L(r) \cup L(s)$
  - (r)(s) is a regexp denoting the language L(r)L(s)
  - $(r)^*$ is a regexp denoting $(L(r))^*$
  - (r) is a regexp denoting L(r). Additional parentheses do not change the language an expression denotes.

多余的括号不会影响表达式的语言。

- Following the rules, regexps often contain unnecessary pairs of parentheses. We may drop some if we adopt the conventions:
  - **Precedence:** closure * > concatenation > union |
  - **Associativity:** All three operators are left associative, meaning that operations are grouped from the left, e.g., a | b | c would be interpreted as (a | b) | c
- Example: (a) | ((b)*(c)) = a | b*c

Examples

- Examples: Let Σ = {a, b}
  - a|b denotes the language {a, b}
  - (a|b)(a|b) denotes {aa, ab, ba, bb}
  - a^* denotes {ϵ, a, aa, aaa, …}
  - $(a|b)^*$ denotes the set of all strings consisting of 0 or more *a*'s or *b*'s: {ϵ, a, b, aa, ab, ba, bb, aaa, …}
  - $a|a^*b$ denotes the string *a* and all strings consisting of 0 or more *a*'s and ending in *b*: {a, b, ab, aab, aaab, …}

Regular Language

- A **regular language** is a language that can be defined by a regexp
- If two regexps r and s denote the same language, they are *equivalent*, written as $r = s$
- Each algebraic law below asserts that expressions of two different forms are equivalent

两个正则表达式等价 ⇔ 两个正则表达式表示同一种语言.

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

Is $(\mathbf{a}|\mathbf{b})(\mathbf{a}|\mathbf{b}) = \mathbf{aa}|\mathbf{ab}|\mathbf{ba}|\mathbf{bb}$ true?

| can be viewed as + in arithmetics, concatenation can be viewed as ×, * can be viewed as the power operator.

| 可视作加法
|| 可视作乘法
* 可视作幂.

Regular Definitions

- For notational convenience, we can give names to certain regexps and use those names in subsequent expressions

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

$$d_1 \to r_1$$
$$d_2 \to r_2$$
$$\cdots$$
$$d_n \to r_n$$

where:

- Each d_i is a new symbol not in Σ and not the same as the other d's
- Each r_i is a regexp over the alphabet $\Sigma \cup \{d_1, d_2, \ldots, d_{i-1}\}$

Each new symbol denotes a regular language. The second rule means that you may reuse previously-defined symbols.

Examples

- Regular definition for C identifiers

*letter_* → `A` | `B` | ··· | `Z` | `a` | `b` | ··· | `z` | `_`
digit → `0` | `1` | ··· | `9`
id → *letter_* (*letter_* | *digit*)*

`_hello` valid?
`3times` valid?

- Regexp for C identifiers

```
(A|B|...|Z|a|b|...|z|_)((A|B|...|Z|a|b|...|z|_)|(0|1|...|9))*
```

Extensions of Regular Expressions

- **Basic operators:** union |, concatenation, and Kleene closure (proposed by Kleene in 1950s)
- A few **notational extensions**:
  - One of more instances: the unary, postfix operator +
    - $r^+ = rr^*$, $r^* = r^+ \mid \epsilon$
  - Zero or one instance: the unary postfix operator ?
    - $r? = r \mid \epsilon$
  - Character classes: shorthand for a logical sequence
    - $[a_1a_2\ldots a_n] = a_1 \mid a_2 \mid \ldots \mid a_n$
    - $[a\text{-}e] = a \mid b \mid c \mid d \mid e$
- The extensions are only for notational convenience, they do not change the descriptive power of regexps

Recognition of Tokens

- Lexical analyzer examines the input string and finds a prefix that matches one of the tokens
- The first thing when building a lexical analyzer is to define the patterns of tokens using regular definitions
- A special token: `ws` → `(blank | tab | newline)+`
  - When the lexical analyzer recognizes a whitespace token, it does not return it to the parser, but restart from the next character

Example: Patterns and Tokens

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	→	`<` | `>` | `<=` | `>=` | `=` | `<>`

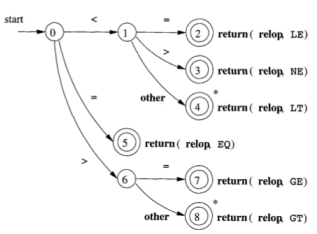
Patterns for tokens

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

Lexemes, tokens, and attribute values

Transition Diagrams

- An important step in constructing a lexical analyzer is to convert patterns into "transition diagrams"
- Transition diagrams have a collection of nodes, called *states* (状态) and *edges* (边) directed from one node to another

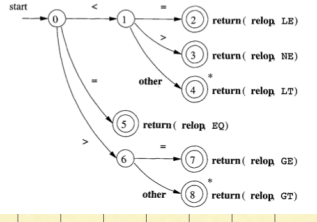


LEXEMES	TOKEN NAME	ATTRIBUTE VALUE
<	**relop**	LT
<=	**relop**	LE
=	**relop**	EQ
<>	**relop**	NE
>	**relop**	GT
>=	**relop**	GE

The transition diagram in the left recognizes **relop** tokens

States

- Represent conditions that could occur during the process of scanning (i.e., what characters we have seen)
- The *start state* (开始状态), or *initial state*, is indicated by an edge labeled "start", which enters from nowhere
- Certain states are said to be *accepting* (接受状态), or *final*, indicating that a lexeme has been found

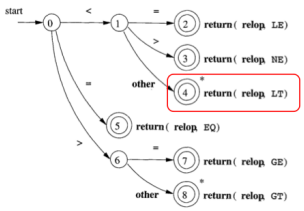


States 2-8 are accepting. They return a pair (token name, attribute value).

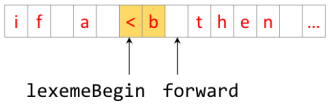
By convention, we indicate accepting states by double circles

The Retract Action

- At certain accepting states, the found lexeme may not contain all characters that we have seen from the start state (such states are annotated with *)
- When entering * states, it is necessary to retract (回退) the `forward` pointer, which points to the next char in the input string



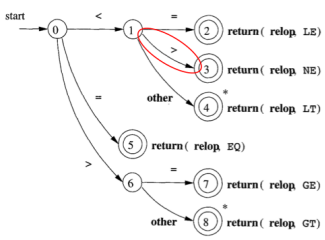
- The found lexeme: <
- The characters we've seen: <b

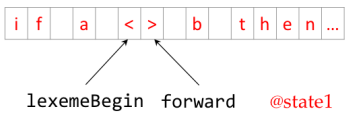


We should retract `forward` one step back

Edges

- *Edges* are directed from one state to another
- Each edge is labeled by a symbol or set of symbols





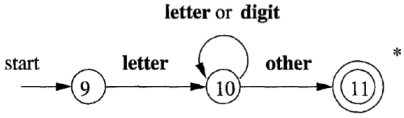
@state1

In the above case, we should follow the circled edge to enter state 3 and advance the `forward` pointer

Recognition of Reserved Words and Identifiers

保留字和标识符需要区分

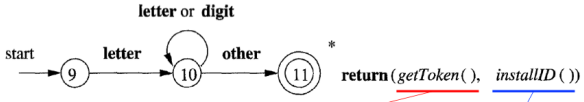
- In many languages, **reserved words** or **keywords** (e.g., `then`) also match the pattern of identifiers
- **Problem:** the transition diagram that searches for identifiers can also recognize reserved words



Is `then` an identifier?

Handling Reserved Words

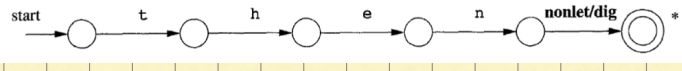
- **Strategy 1:** Preinstall the reserved words in the symbol table. Put a field in the symbol-table entries to indicate that these strings are not ordinary identifiers (预先存表方案)



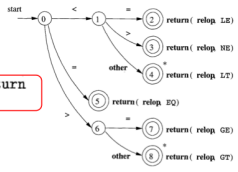
Examine the symbol table and return the token name (identifier or reserved word)

Place a recognized identifier/keyword in the symbol table if it is not already there and return a pointer to the entry

- **Strategy 2:** Create a separate transition diagram with **a high priority** for each keyword (多状态转移图方案)

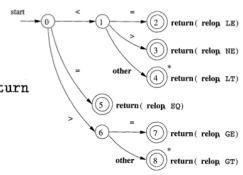


Building a Lexical Analyzer from transition Diagrams




```
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);
        }
    }
}
```

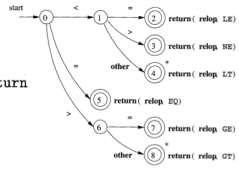
Sketch implementation of `relop` transition diagram




```
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);
        }
    }
}
```

Use a variable `state` to record the current state

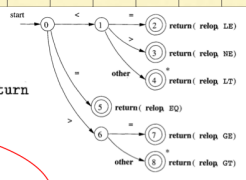
Sketch implementation of `relop` transition diagram




```
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);
        }
    }
}
```

A `switch` statement based on the value of `state` takes us to the processing code

Sketch implementation of `relop` transition diagram

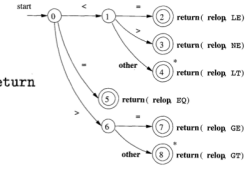



```
TOKEN getRelop()
{
    TOKEN retToken = new(RELOP);
    while(1) { /* repeat character processing until a return
                  or failure occurs */
        switch(state) {
            case 0: c = nextChar();
                    if ( c == '<' ) state = 1;
                    else if ( c == '='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

The code of a normal state:

1. Read the next character
2. Determine the next state
3. If step 2 fails, do error recovery

Sketch implementation of `relop` transition diagram




```
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);
        }
    }
}
```

The code of an accepting state:

1. Perform retraction if the state has *
2. Set token attribute values
3. Return the token to parser

Sketch implementation of `relop` transition diagram

Building the Entire Lexical Analyzer

- **Strategy 1:** Try the transition diagram for each type of token sequentially
  - `fail()` resets the pointer `forward` and tries the next diagram
- **Problem:** Not efficient
  - May need to try many irrelevant diagrams whose first edge does not match the first character in the input stream

- **Strategy 2:** Run transition diagrams in parallel
  - Need to resolve the case where one diagram finds a lexeme and others are still able to process input.
  - **Solution:** take the longest prefix of the input that matches any pattern
- **Problem:** Requires special hardware for parallel simulation, may degenerate into the sequential strategy on certain machines

- **Strategy 3:** Combining all transition diagrams into one
  - Allow the transition diagram to read input until there is no possible next state
  - Take the longest lexeme that matched any pattern

- This is a commonly-adopted strategy in real-world compiler implementation (efficient & requires no special hardware)

How? Be patient ☺, we will talk about this later.

The Lexical-Analyzer Generator Lex

- `Lex`, or a more recent tool `Flex`, allows one to specify a lexical analyzer by specifying regexps to describe patterns for tokens
- Often used with Yacc/Bison to create the frontend of compiler

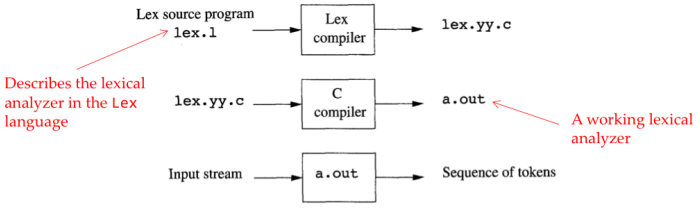


Figure 3.22: Creating a lexical analyzer with Lex

Structure of Lex Programs

- **A Lex program has three sections separated by %%**
  - Declaration (声明)
    - Variables, constants (e.g., token names)
    - Regular definitions
  - Translation rules (转换规则) in the form "Pattern {Action}"
    - Each pattern (模式) is a regexp (may use the regular definitions of the declaration section)
    - Actions (动作) are fragments of code, typically in C, which are executed when the pattern is matched
  - Auxiliary functions section (辅助函数)
    - Additional functions that can be used in the actions

Example

```
%{
    /* 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}+)?

%%
```

Anything in between %{ and }% is copied directly to `lex.yy.c`.
In the example, there is only a comment, not real C code to define manifest constants

Regular definitions that can be used in translation rules

Section separator

```
{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);}

%%
```

Literal strings*

Continue to recognize other tokens

Return token name to the parser

Place the lexeme found in the symbol table

A global variable that stores a pointer to the symbol table entry for the lexeme. Can be used by the parser or a later component of the compiler.

* The characters inside have no special meaning (even if it is a special one such as *).

- Everything in the auxiliary function section is copied directly to the file `lex.yy.c`
- Auxiliary functions may be used in actions in the translation rules

```
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 */
}
```

Variables defined and set automatically by the lexical analyzer Lex generates

Conflict Resolution

- When the generated lexical analyzer runs, it analyzes the input looking for prefixes that match any of its patterns.*
- **Rule 1:** If it finds multiple such prefixes, it takes the longest one
  - The analyzer will treat <= as a single lexeme, rather than < as one lexeme and = as the next
- **Rule 2:** If it finds a prefix matching different patterns, the pattern listed first in the `Lex` program is chosen.
  - If the keyword patterns are listed before identifier pattern, the lexical analyzer will not recognize keywords as identifiers

找匹配到最长的

找第一个匹配到的