LEXICAL ANALYSIS

Based on Chapter 3 of Aho, Lam, Sethi, Ullman:

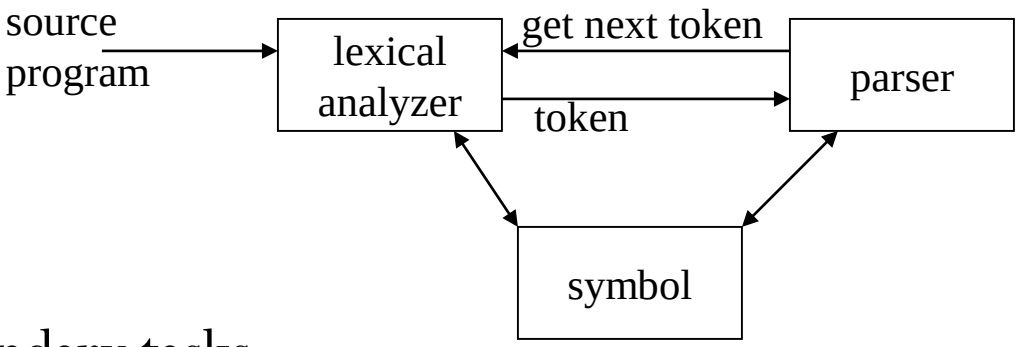
Compilers: Principles, Techniques, & Tools

2nd Ed, Addison Wesley, 2007

Table of Contents

The Role of the Lexical Analyzer

- The lexical analyzer
  - to read the input characters,
  - to group them into lexemes,
  - to produce as output a sequence of tokens
- Interactions of the lexical analyzer with the parser



- Secondary tasks
  - strip out comments and white space from source
  - keep track of the line number for error message

Implementing the Lexical Analyzer

- Hand-written
  - to construct a diagram that illustrates the structure of the tokens of the source language
  - to *hand-translate* the diagram into a program

- Pattern-directed
  - to use *pattern-action* language
  - to describe patterns in a form of *regular expressions*
  - e.g. lex

Tokens, Lexemes, Patterns

- Tokens
  - lowest-level of syntactic units
  - units to communicate with a parser
  - classifying lexemes in categories (represented in symbols used in a grammar)
  - examples:
    - keywords (**if**, **then**, **while**, etc.)
    - operators (**plus**, **minus**, …), punctuation symbols (**comma**, **lparen**,…)
    - identifiers (**id**), numbers (**num**), literal strings (**literal**)
- Lexemes
  - an actual sequence of input characters for a token
  - example: "if", "<", "position", "60.0", etc.
- Patterns
  - a rule describing a token
  - usually in a form of regular expressions

Tokens, Lexemes, Patterns

- Example

Token	Sample Lexemes	Pattern(informal)
const	const	const
if	if	if
lt	<	<
id	pi, count, D2	letter followed by letters and digits
num	3.14, 0, 6.2E23	any numeric constant
literal	“core dumped”	any characters between “ and “

Tokens

- A token is formed in a pair <*token name*, *attribute value*>
  - the *token name* represents a kind of lexical unit
  - the *token attribute* value differentiates its token from other tokens

    e.g. <num, 2> and <num, 5>

    e.g. <id, 1> and <id, 2> ; its attribute is a pointer to symbol table
  - some tokens do not need attributes

    e.g. <const, > , < assign_op, >, etc
- Example

  position = initial + rate * 60

  <id, *pointer to symbol table entry for* position>
  <assign_op, >
  <id, *pointer to symbol table entry for* initial>
  <add_op, >
  <id, *pointer to symbol table entry for* rate>
  <mult_op, >
  <num, integer value 60>

Lexical Errors

- Lexical Error
  - none of the patterns for tokens matches a prefix of the remaining input

    e.g. in C language: 1e+% “abc<EOF> 0x0g ‘aa’
  - lexical error vs syntax error

    e.g. fi (a == f(x)) …
- Error Recovery
  - Panic mode: delete successive characters from the remaining input until lexical analyzer can find a well-formed token
  - Word-level
    - delete an extraneous character
    - insert a missing character
    - replace an incorrect character by a correct character
    - transpose two adjacent character

Specification of Tokens

- An *Alphabet* denotes any finite set of symbols

  e.g. {0, 1}, ASCII, …
- A *string* over an alphabet is a finite sequence of symbols in the alphabet (including *empty* string **ε)**

  e.g. **banana** is a string over the alphabet ASCII
  - the length of a string : $|s|$
  - the concatenation of two strings : $s_1 \cdot s_2$ (also denoted as $s_1 s_2$)
  - the exponentiation of strings : $s^1, s^2, \ldots, s^i$
- A *language* (L) or *formal language* denotes any set of strings over some fixed alphabet

  e.g.
  - the set of all sequences over {0, 1}
  - the set of all strings of letters and digits beginning with a letter
  - empty set {}
  - the set of all well-formed C programs (a sequence of *tokens*)

Specification of Tokens

- Operations on Languages
  - Union of L and M ($L \cup M$) : { s | s in L or s in M }
  - Concatenation of L and M (LM) : {st | s in L and t in M }
  - Kleene Closure (L^*) : zero or more concatenations of L

    $L^* = L^0 \cup L^1 \cup L^2 \cup L^3 \cup \ldots$
  - (Positive) Closure (L^+) : one or more concatenations of L

    $L^+ = L^1 \cup L^2 \cup L^3 \cup \ldots = L^* - \{\varepsilon\}$

e.g.

L is the set of letters

D is the set of digits

$L \cup D$ is the set of letters and digits

LD is the set of 520 strings, each consisting of one letter followed by one digit

L^4 is the set of all 4-letter strings

$L(L \cup D)^*$ is the set of all strings of letters and digits beginning with a letter

L^* is the set of all strings of letters, including ε

Regular Expressions

- Regular expressions represent patterns of strings of characters
  - e.g. 1(1+0)*
- A regular expression *r* denotes a language(regular set) *L*(*r*)
- Definition: regular expressions over alphabet Σ
  - ε is a regular expression that denotes $\{\varepsilon\}$
  - if $a \in \Sigma$, *a* is a regular expression denoting $\{a\}$
  - suppose *r* and *s* are regular expressions denoting L(*r*) and L(*s*)
    - (*r*) | (*s*) is 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*)
  - operator precedence * > concatenation > |

Regular Expressions

- Example 3.4 Let $\Sigma = \{a, b\}$
  - a | b denotes the set {a, b}
  - (a|b)(a|b) denotes the set {aa, ab, ba, bb}
  - a* denotes {ε, a, aa, aaa, … }
  - (a|b)* denotes the set of all strings containing zero or more instance of an a or b
  - a*b denotes the set {b, ab, aab, aaab, … }
- Algebraic properties

  r|s = s|r

  r|(s|t) = (r|s)|t

  (rs)t = r(st)

  r(s|t) = rs | rt : concatenation distributes over |

  (s|t)r = sr | tr

  εr = r rε= r : ε is the identity for concatenation

  r** = r* : * is idempotent

Regular Definitions

- A regular definition is a sequence of definitions of the form

  $d_1 \rightarrow r_1$

  $d_2 \rightarrow r_2$

  …

  $d_n \rightarrow r_n$

  d_i : unique name not in Σ

  r_i : regular expression over $\Sigma \cup \{d_1, d_2, .. d_{i-1}\}$

- Example: identifiers

  *letter_* -> A | B | . | Z | a | b | … | z | _

  digit -> 0 | 1 | … | 9

  id -> *letter_*(*letter_* | *digit*)*

- Unsigned number

  digits -> *digit digit**

  optional_fraction -> . *digits* | ε

  optional_exponent -> (E (+ | - | ε)*digits*) | ε

  number -> *digits optional_fraction optional_exponent*

Notational Shorthands

- One or more instances
  - (r)+ = r r*

  e.g. digits -> digit+
- Zero or one instance
  - r? = r| ε

  e.g. optional_fraction -> (. digits)?

  optional_exponent -> (E (+|-)? digits)?
- A range of characters
  - [abc] = a|b|c
  - [a-z] = a|b|…|z

  e.g. id -> [A-Za-z][A-Za-z0-9]*

Recognition of Tokens

- Consider the following grammar

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

- Regular definitions that generates or recognize the tokens

  if -> if
  then -> then
  else -> else
  relop -> < | <= | = | >= | > | <>
  id -> **letter**(**letter** | **digit**)*
  number -> **digit**+ (. **digit**+)?(E(+|-)?)**digit**+)?

Recognition of Tokens

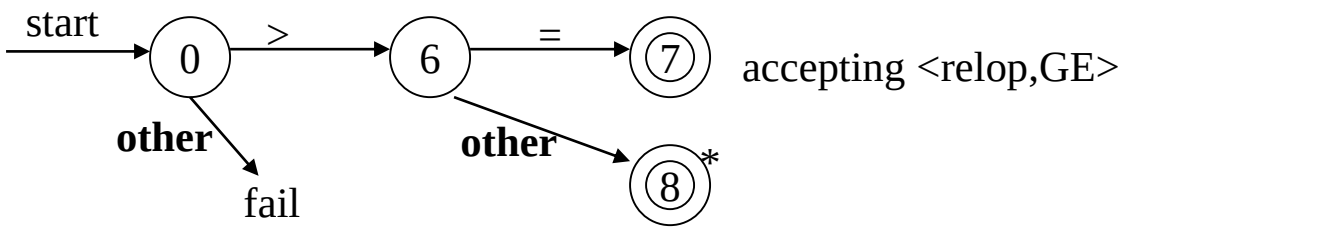
- Regular expressions and tokens

regular exp	token name	attribute value
ws	-	-
if	if	-
then	then	-
else	else	-
id	id	pointer to table entry
number	num	pointer to table entry
<	relop	LT (constant symbol)
<=	relop	LE
>	relop	GT
>=	relop	GE
=	relop	EQ
<>	relop	NE

* ws -> (blank | tab | newline)+

Transition Diagrams

- Transition diagram
  - states
    - *start* state
    - *accept* state (*final* state)
  - edges (or transitions) (*state* → *state* including a label)
  - actions
    - goto (assume the diagram to be deterministic)
    - accept
    - retract
    - fail
  - example : > and >=



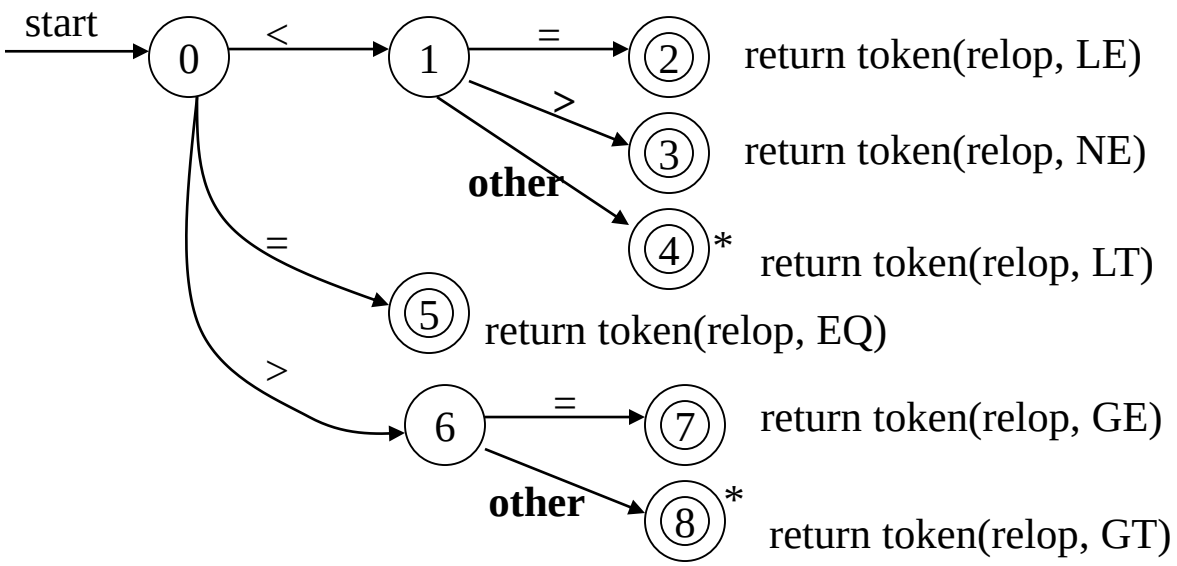
  - ***The principle of longest string***

Transition Diagrams

- There may be several transition diagrams, each specifying a group of tokens
  - if *failure* occurs while we are following one transition diagram, then we *retract* to where it was in the start state, and activate the next diagram
  - if failure occurs in all transition diagrams, the lexical error has been detected and an *error-recovery* routine is invoked

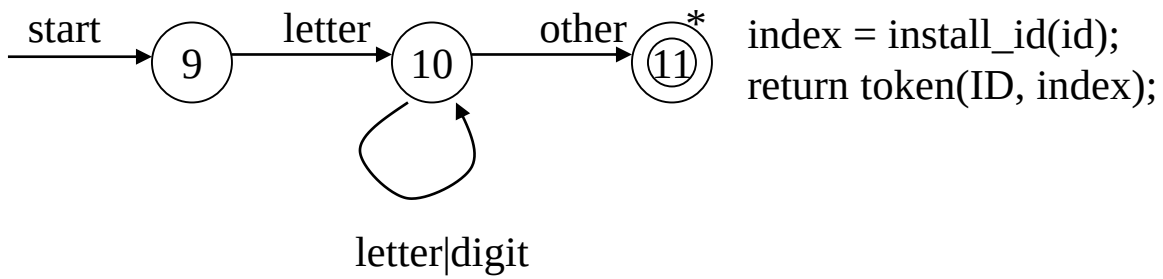
Transition Diagrams

- Transition diagram for the token **relop**



Transition Diagrams

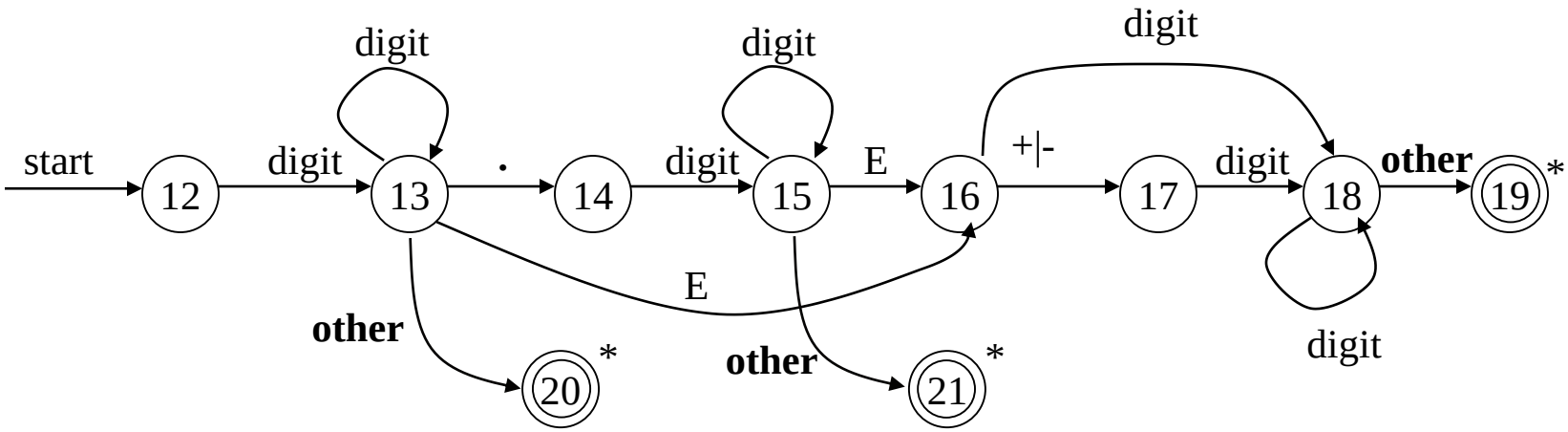
- Recognition of identifiers and keywords
  - keywords are exceptional cases of identifiers
  - rather than encoding the exceptions into a transition diagram, a trick to treat keywords as special identifiers is useful
  - when accepting state is reached, some code is executed to determine if the lexeme is a keyword or an identifier
  - simple technique : install keywords in the symbol table before any identifier is installed



Transition Diagrams

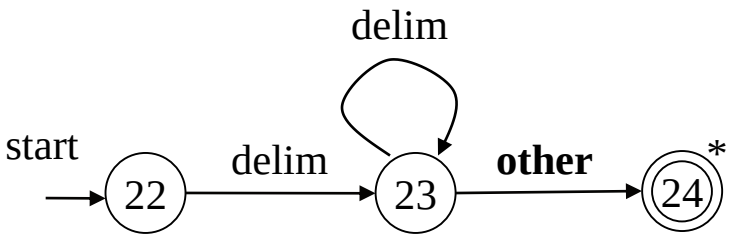
- Recognition of unsigned numbers

  num -> digit+ (.digit+)?(E(+|-)? digit+)?



Transition Diagrams

- Transition diagram for whitespace

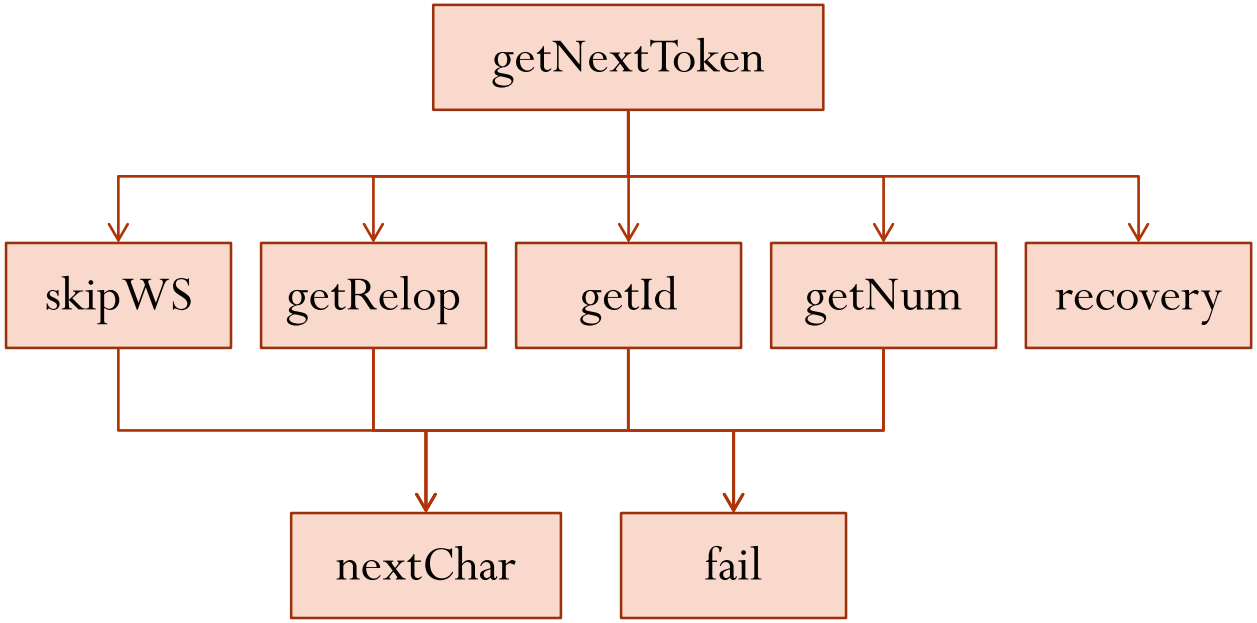


Implementing Transition Diagrams

- Systematic converting state transition diagrams into a program
  - each state is converted to some code
  - Non-accepting state
    - for next input character, if there exists leaving edge with a label for the character then go to next state on the edge
    - otherwise, fail()
  - Accepting state
    - retract() if any
    - return the token
  - Fail()
    - go to the code for the next transition diagram
    - if no more transition diagram, do the error recovery

Architecture of a Lexical Analyzer

- A transition diagram can be simulated by a function



Architecture of a Lexical Analyzer

- Main routine for recognizing one token

```
TOKEN getNextToken()
{
    state = 0;
    while(1) {
        skipWS();
        switch(state) {
            case 0:  token= getRelop();
                     if (token == failToken) { state = 9; break; }
                     else return token;
            case 9:  token = getId();
                     if (token == failToken) { state = 12; break; }
                     else return token;
            case 12: token = getNum();
                     if (token == failToken) { recovery(); state = 0; break; }
                     else return token;
            default: error();
        }
    }
}
```

Architecture of a Lexical Analyzer

- The function recognizing the token **relop**

```
TOKEN getRelop()
{
   TOKEN retToken = {REOP, _};

   while(1) {
      switch(state) {
         case 0: c = nextChar();
                  if(c == '<')   state = 1;  // goto state 1
                  else if (c == '=' ) state =5;
                  else if (c == '>' ) state = 6;
                  else  {fail(); return failToken; }
                  break;
         case 1: c = nextChar();
                 if(c == '=') state = 2;
                 else if (c=='>') state = 3;
                 else state  = 4;
                 break;
         case 2: retToken.attribute = LE;  return(retToken);
         case 3: retToken.attribute = NE; return(retToken);
         case 4: retract();
                   retToken.attribute = LT; return(retToken);
         …
         case 8: retract();
                  retToken.attribute = GT; return(retToken);

      } // end of switch
   } // end of while
} // end of function
```

Architecture of a Lexical Analyzer

- nextChar()
  - New token begins with *lexeme_start* pointer
  - Move *forward* pointer one character
- fail() function
  - reset the *forward* pointer with *lexeme_start* pointer
- recovery()
  - delete one character – panic mode