Chapter2. Lexical Analysis

2019年12月3日 11:38

NOTE TAKING AREA

The role of lexical analyzer

Read input of source program, group into lexemes and produces tokens.
Add lexemes into symbol table.
Strip out comments and whitespace.
Token: a pair $< token\ name, attribute\ value >$.

- Token names: influence parsing decisions.
- Attribute values: influence semantic analysis, code generation.

Pattern: description of the form that the lexemes of a token may take.
Lexeme: a sequence of characters matches the pattern for a token (an instance of the token).

Specification of tokens (regular expression)

Strings and languages:
Alphabet: any finite set of symbols.
String over an alphabet: a finite sequence of symbols drawn from the alphabet.

- Empty string: the string of length 0, ϵ.
- Prefix of string, proper prefix, suffix, proper suffix: proper doesn't empty or equal to the string.
- Substring, proper substring, subsequence: subsequences can be non-consecutive.
- String-related operations: concatenation and exponentiation.

Language: any countable set of strings over some fixed alphabet.

A countable set is either a **finite** set or a **countably infinite** set.
Operation on languages: union, concatenation, Kleene closure, positive closure.

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	$L^+ = \cup_{i=1}^{\infty} L^i$

Regular expressions: rules that define regular expressions over an alphabet $\sum$.

- Basis: $L(\epsilon) = \{\epsilon\}, L(a) = \{a\}\ for\ symbol\ a\ in\ \sum$.
- Induction: $(r)|(s), (r)(s), (r)*, (r)$.
    - Precedence: closure > concatenation > union.
    - Associativity: left associativity.
- Regular language: a language that can be defined by a regular expression.

  Algebraic laws assert that expressions of equivalent:

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

- Regular definition for notational convenience.

  $d_1 \rightarrow r_1$
  $d_2 \rightarrow r_2$
  $\dots$
  $d_n \rightarrow r_n$

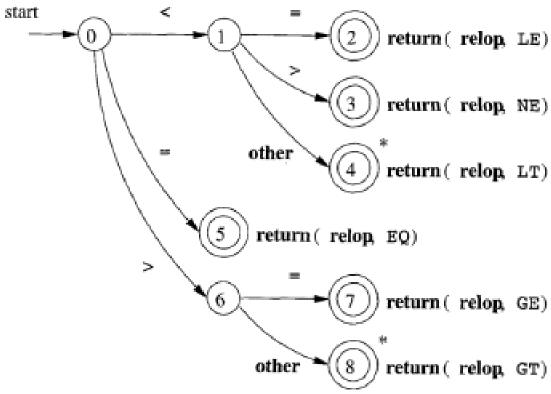
- The extension of regular expressions: one or more instances*, zero or one instance?, character classes[].

Recognition of tokens (transition diagrams)

Lexical analyzer examines the longest prefix of matched string.

Transition diagram

Consists of nodes (states) and edges (labeled by a symbol or set of symbols) from one node to another.



- Start state (initial state): enters from nowhere;
- Accepting (final) state: a lexeme has been found (denoted by double circle);
- Retract: retract forward pointer to previous character.

Handling reserved words

1. Preinstall the reserved words in the symbol table.
2. Create a separate transition diagram with a high priority for each keyword.

Strategies to build the entire lexical analyzer

1. Try the transition diagram for each token sequentially;
2. Run transition diagrams in parallel;
3. Combining all transition diagrams into one (preferred).

The lexical-analyzer generator

Lex / Flex is tool to specify a lexical analyzer by specifying regular expressions to describe patterns for tokens.

Structure of Lex program: declaration, translation rules (pattern, actions), auxiliary functions section.

Global variable $yylval$: pointer to the symbol table entry for the lexeme.

Conflict resolution: looking for prefixes that match any of its patterns.

- Rule 1: Take the **longest** one of multiple prefixes.
- Rule 2: For prefix matching different patterns, take the pattern listed **first**.

Finite automata

Nondeterministic finite automata (NFA): allowing multiple target states, and empty string ϵ is a possible lable.

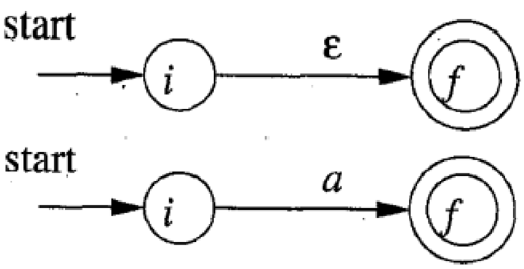
- A finite set of states S, a set of input symbols $\sum$ (input alphabet), a transition function, a start state s_0, and a set of accepting states F.

- NFA accepts input string if and only if there is some path from the start state to one of the accepting states.

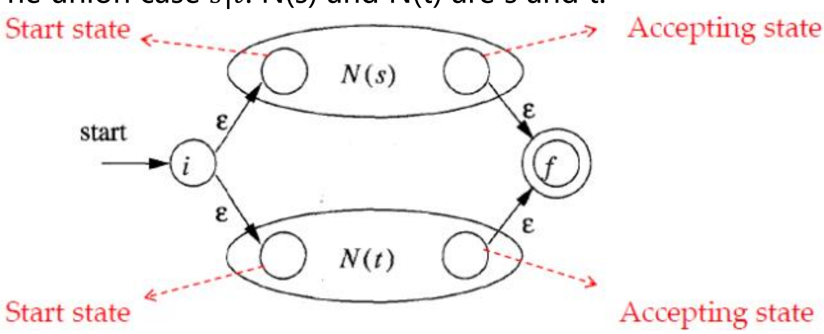
<u>Deterministic finite automata (DFA): one edge for one symbol and one state.</u>
- A special case of an NFA, more efficient.
- Each NFA can be converted to DFA.

NFA and DFA recognize the same languages, called regular languages.
Convert regular expression to NFA, then convert NFA to DFA.

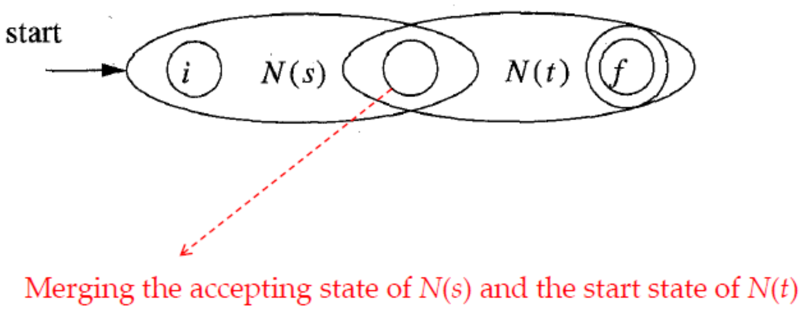
Conversion of an NFA to a DFA
The <u>subset construction</u> algorithm:
    Each state of the constructed DFA corresponds to a set of NFA states, simulate DFA after reading input as the same state as DFA, simulate in parallel all possible moves.
The number of DFA states is possible **exponential** in the number of NFA states.
    In practice, NFA and DFA have **approximately the same** number of states.
<u>Operations used in algorithm: $\epsilon - closure([s|T]), move(T, a)$.</u>
    $Dtran[T, a]$ and Dstates are also constructed.
<u>Thompson's construction algorithm: recursively by splitting an expression into its constituent subexpressions.</u>
- Two basic rules: subexpressions with no operators.



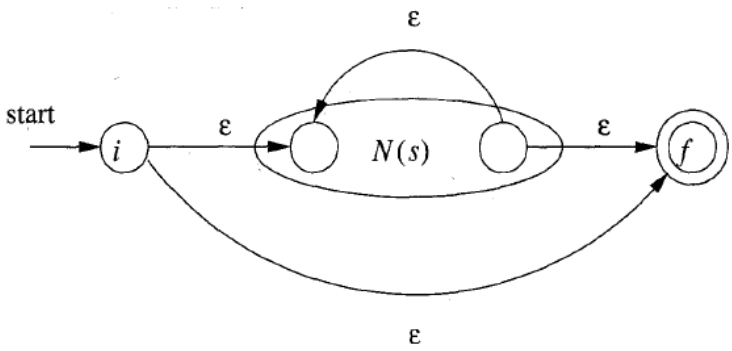
- Three inductive rules: subexpression of a given expression.
    - The union case $s|t$: N(s) and N(t) are s and t.



    - The concatenation case st:



    - The Kleene star case $s*$:



An accepting state of the DFA corresponds to a subset of the NFA states.

If multiple accepting states occur, means conflicts arise (take first one).

Minimize the number of states of a DFA: **grouping sets of equivalent states**.
Distinguishing states: string x distinguishes s and t if **exactly one** of the states reached from s and t by following the path with label x is an accepting state.
DFA state-minimization algorithm: **partition**, **distinguish**, loop.

State minimization in lexical analyzers: initial partition F and S-F differently.

CUE COLUMN

Token, pattern, lexeme example

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"`

Regular expression example

Example: Let $\Sigma = \{a, b\}$

- **a | b** denotes the language $\{a, b\}$
- **(a | b)(a | b)** denotes $\{aa, ab, ba, bb\}$
- **a*** denotes $\{\epsilon, a, aa, aaa, \ldots\}$
- **(a | b)*** denotes the set of all strings consisting of 0 or more a's or b's: $\{\epsilon, a, b, aa, ab, ba, bb, aaa, \ldots\}$
- **a | a*b** denotes the string a and all strings consisting of 0 or more a's and ending in b: $\{a, b, ab, aab, aaab, \ldots\}$

Regular definition example:

- Regular definition for C identifiers

$$
\begin{aligned}
letter\_ &\rightarrow \texttt{A} \mid \texttt{B} \mid \cdots \mid \texttt{Z} \mid \texttt{a} \mid \texttt{b} \mid \cdots \mid \texttt{z} \mid \_ \\
digit &\rightarrow \texttt{0} \mid \texttt{1} \mid \cdots \mid \texttt{9} \\
id &\rightarrow letter\_ \; (\; letter\_ \mid digit \;)^*
\end{aligned}
$$

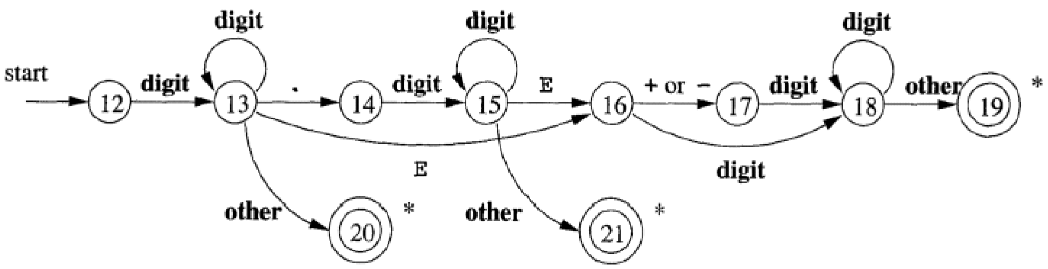
`_hello` valid?

`3times` valid?

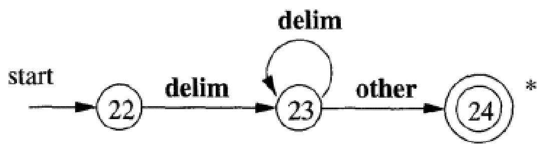
- Regular expression for C identifiers

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

Example of transition diagrams

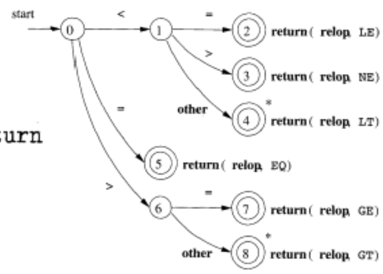


A transition diagram for unsigned numbers



A transition diagram for whitespace

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 of implementation of `relop` transition diagram

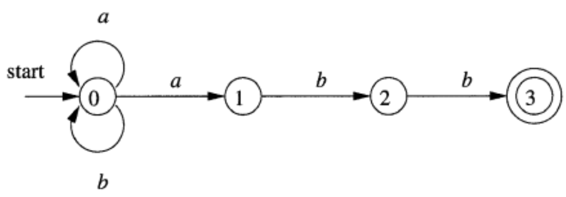
- Use variable **state** to record the current state;
- A **switch** statement based on the value of state takes us to code for each of the possible states;
- The code of a normal state:
  a. Read the next character;

   b. Determine the next state;
   c. If step 2 fails, do error recovery.
- The code of an accepting state:
   a. Perform retraction if the state has *;
   b. Set token attribute values;
   c. Return the token to parser.

NFA example

- S = {0, 1, 2, 3}
- Start state: 0
- Accepting states: {3}
- Transition function
   - (0, a) → {0, 1}   (0, b) → {0}
   - (1, b) → {2}   (2, b) → {3}

Graph representation of the NFA:

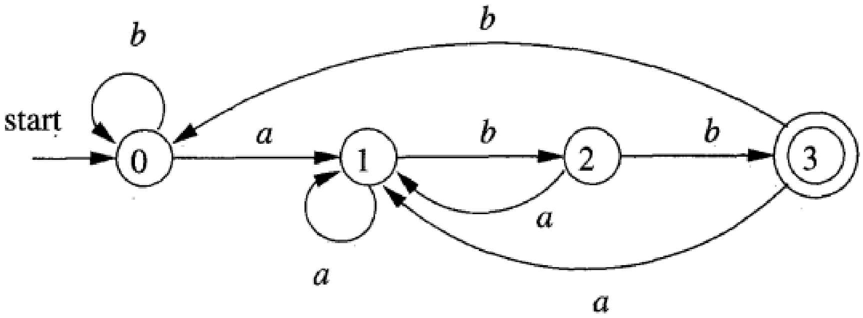


This graph is equivalent to the transition table below:

STATE	a	b	ϵ
0	$\{0,1\}$	$\{0\}$	$\emptyset$
1	$\emptyset$	$\{2\}$	$\emptyset$
2	$\emptyset$	$\{3\}$	$\emptyset$
3	$\emptyset$	$\emptyset$	$\emptyset$

Rows correspond to states, columns correspond to the input symbols, the table entries are transition functions, and ∅ means no such transition function.
This example describes the language: $L((a|b)*abb)$.

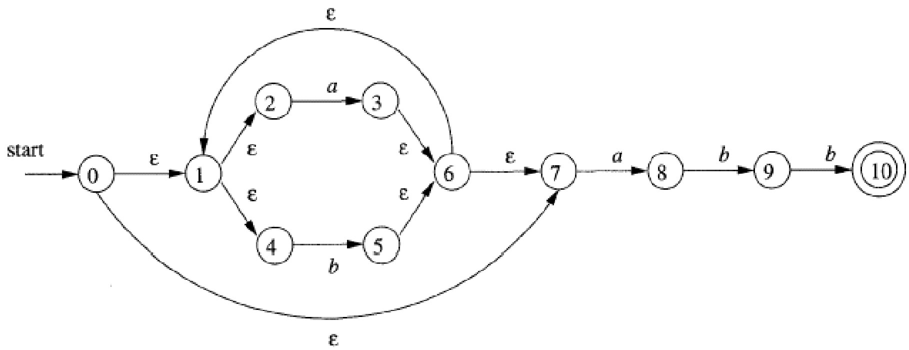
DFA example



The DFA enters the input string $ababb$ and returns "yes".

NFA convert to DFA example

- A: ϵ-$closure(0) = \{0, 1, 2, 4, 7\}$
- B: Dtran[A, a] = ϵ-$closure(\{3, 8\}) = \{1, 2, 3, 4, 6, 7, 8\}$
- C: Dtran[A, b] = ϵ-$closure(\{5\}) = \{1, 2, 4, 5, 6, 7\}$
- D: Dtran[B, b] = ϵ-$closure(\{5, 9\}) = \{1, 2, 4, 5, 6, 7, 9\}$
- ...

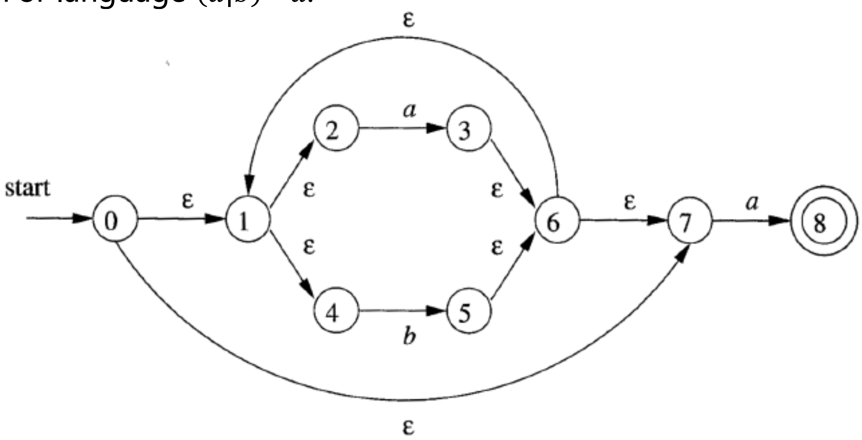


Transition table of the DFA: start state A and accepting states {E}.

NFA STATE	DFA STATE	a	b
$\{0,1,2,4,7\}$	A	B	C
$\{1,2,3,4,6,7,8\}$	B	B	D
$\{1,2,4,5,6,7\}$	C	B	C
$\{1,2,4,5,6,7,9\}$	D	B	E
$\{1,2,4,5,6,7,10\}$	E	B	C

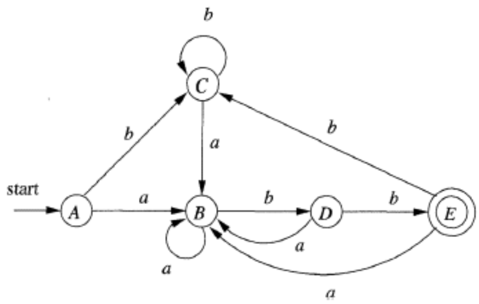
Convert NFA to DFA example

For language $(a|b) * a$:



DFA states minimization algorithm example

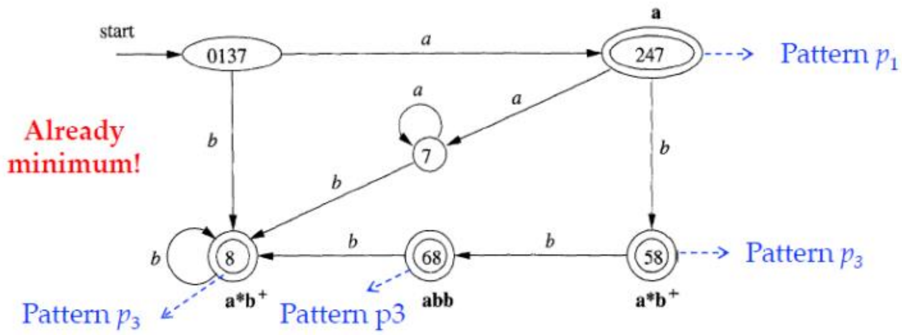
- Initial partition: {A, B, C, D}, {E}
- Handling group {A, B, C, D}: b splits it to two subgroups {A, B, C} and {D}
- Handling group {A, B, C}: b splits it to two subgroups {A, C} and {B}
- Picking A, B, D, E as representatives to construct the minimum-state DFA



STATE	a	b
A	B	A
B	B	D
D	B	E
E	B	A

In lexical analyzer, the example is:

- Initial partition: {0137, 7}, {247}, {68}, {8, 58}, {∅}
  - We add a dead state ∅: we suppose has transitions to itself on inputs a and b. It is also the target of missing transitions on a from states 8, 58, and 68.



Be aware that the initial partition is different.

SUMMARIES

1. The role of lexical analyzer: lexeme, token, pattern.
2. Specification of tokens (regular expression): alphabet, string, language.
3. Recognition of tokens (transition diagram).
4. The lexical-analyzer generator.
5. Finite automata: NFA, DFA, conversion, minimization of states.