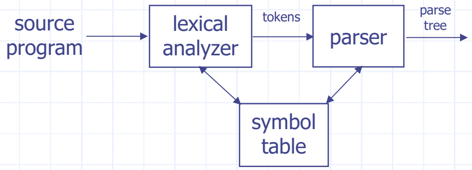
編譯器設計

Chapter 3: Lexical Analysis

Lexical Analysis

- The role of the lexical analyzer
  - To read the input characters
  - To produce a sequence of tokens



- Reasons to separate lexical analysis
  - Simpler design is the most import consideration
  - Compiler efficiency is improved
  - Compiler portability is enhanced

Tokens, Patterns, and Lexemes

- Tokens
  - The terminal symbols in the grammar
- Patterns
  - Rule describing the set of strings that correspond to the same token
- Lexeme
  - A sequence of characters matched by a pattern

Token	Sample Lexemes	Informal Description of Pattern
const	const	const
if	if	if
relop	<, <=, =, <>, >, >=	< or <= or = or <> or > or >=
id	pi, count, D2	letter followed by letters and digits
num	0, 3.14, 6.02E23	any number constant

Operations on Languages

- Union of *L* and *M*, written as $L \cup M$

$$L \cup M = \{s \mid s \in L \vee s \in M\}$$

- Concatenation of *L* and *M*, written as *LM*

$$LM = \{st \mid s \in L \wedge t \in M\}$$

- Kleene closure of *L*, written as L^*
  - denotes "zero or more concatenations of *L*

$$L^* = \bigcup_{i=0}^{\infty} L^i$$

- Positive closure of *L*, written as L^+

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

Operations on Languages

- Example

  Let L ={A, B,…, Z, a, b,…, z} and D ={0, 1, 2,…, 9}
  - $L \cup D$
    - the set of letters and digits
  - LD
    - the set of strings consists of a letter followed by a digit
  - L^4
    - the set of all four-letter strings
  - L^*
    - the set of all strings of letters, including ϵ
  - $L(L \cup D)^*$
    - the set of all strings of letters and digits beginning with a letter

Regular Expressions

- Each regular expression r denotes a language $L(r)$
- Rules to define the regular expressions over Σ
  - ϵ is regular expression that denotes $\{\epsilon\}$
  - If a is a symbol in Σ, then a is a regular expression that denote {a}
  - Suppose r and s are regular expressions, then
    - $(r)|(s)$ is a regular expression denoting $L(r) \cup L(s)$
    - $(r)(s)$ is a regular expression denoting $L(r)L(s)$
    - $(r)^*$ is a regular expression denoting $(L(r))^*$
    - $(r)^+$ is a regular expression denoting $(L(r))^+$
  - If two regular expressions r and s denote the same language, we say r and s are equivalent and write $r = s$

Regular Expressions

- Example: let Σ= {a, b}
  - The regular expression $a \mid b$ denotes {a, b}
  - (a|b)(a|b) denotes {aa, ab, ba, bb}
    - i.e. the set of all 2-letter strings of a's and b's
    - Another form is $aa \mid ab \mid ba \mid bb$
  - a^* denotes the set of all strings of a's
    - i.e. {ϵ, a, aa, aaa, …}
  - $(a \mid b)^*$ denotes the set of all strings of a's and b's
    - Another form is $(a^*b^*)^*$
  - $a \mid a^*b$ denotes the set containing the string a and all strings consisting of zero or more a's and followed by b's

Precedence

- Unnecessary parentheses can be avoided in regular expressions if
  - The unary operator * has the highest precedence and is left associative
  - Concatenation has the second highest precedence and is left associative
  - | has the lowest precedence and is left associate
  - Example
    - $(a)|((b)^*(c)) \equiv a \mid b^*c$

Algebraic Properties of Regular Expressions

- Some algebraic laws that hold for regular expressions *r*, *s*, and *t*

Axiom	Description
r|s = s|r	| is commutative
r|(s|t) = (r|s)|t	| is associative
(rs)t = r(st)	concatenation is associative
r(s|t) = rs|rt (s|t)r = sr|tr	concatenation distributes over |
ϵr = r rϵ = r	ϵ is the identity element of concatenation
r* = (r|ϵ)*	relationship between * and ϵ
r** = r*	* is idempotent

Regular Definitions

- For notational convenience, we may wish to give names to regular expressions
  - To define regular expressions using these names as if they were symbols
  - If Σ is an alphabet, then a regular definition is a sequence of definitions of the form

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

    where each r_i is a regular expression over the symbols in $\Sigma \cup \{r_1, r_2, \ldots, r_{i-1}\}$

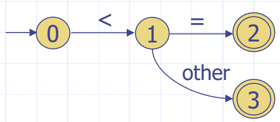
Regular Definitions

- Notational Shorthands
  - +: one or more instance
  - ?: zero or one instance
  - [abc]: character classes, i.e. ≡ a|b|c
- Example

  digit → 0 | 1 | … | 9
  letter → [A-Za-z]
  id → *letter* (*letter* | *digit*)*
  digits → digit+
  optional_fraction → (. *digits*)?
  optional_exp → (E (+ | -)? *digits*)?
  num → *digits optional_fraction optional_exp*

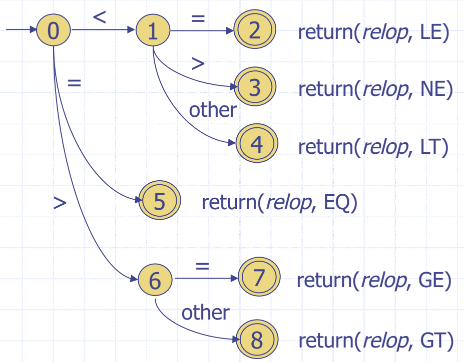
Recognition of Tokens

- State transition diagrams
  - depicts the actions that take place when a lexical analyzer is called by the parser to get the next token
  - States
    - Positions in a transition diagram
      - drawn as circles
    - States are connected by arrows, call *edges*
    - Start state
      - the initial state of the transition diagram



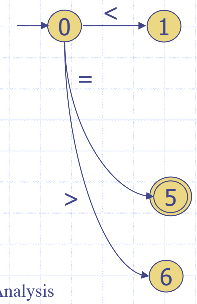
Recognition of Tokens

- Example: *relop* $\rightarrow$ < | <= | = | <> | > | >=



Implementing a Transition Diagram

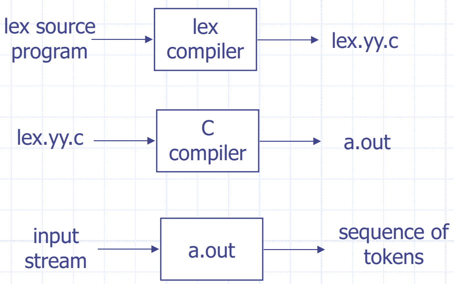
- A transition diagram can be converted into a program
  - Each state gets a segment of code
    - If there are edges leaving a state, then its code reads a character and selects an edge to follow
      - If there is an edge labeled by the character read, then control is transferred
      - If there is no such edge, then fail




```
token nexttoken( ) {
  switch(state) {
  case 0:            /* state 0 */
    c = nextchar();
    if (c == '<') state = 1;
    else if (c == '=') state = 5;
    else if (c == '>') state = 6;
    else state = fail;
    break;
  ...
}
```


Implementing a Transition Diagram

- Several tools have built for constructing lexical analyzer from regular expressions
  - e.g. lex



Finite Automata

- We compile a regular expression into a lexical analyzer by constructing a generalized transition diagram called a *finite automaton*
  - $M = \{S, \Sigma, \delta, s_0, F\}$
    - S : a finite, nonempty set of *states*
    - Σ : an *alphabet*
    - δ : mapping $S \times \Sigma \rightarrow S$
    - s_0: start state
    - F : the set of accepting (or final) states
  - A finite automaton is called a deterministic finite automaton (DFA) if
    - No state has an ϵ-transition, and
    - For each state s and input symbol a, there is at most one edge labeled a leaving s

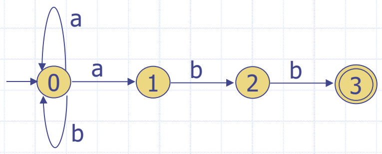
Simulating a DFA

- Input
  - An input string x terminated by an eof
  - A DFA $D = \{S, \Sigma, \delta, s_0, F\}$
- Output
  - "yes" if D accepts x; "no" otherwise
- Method

```
s = s0
c = nextchar();
while c ≠ eof do
  s = moveto(s, c);
  c = nextchar();
end
if s is in F then return "yes"
            else return "no"
```


Nondeterministic Finite Automata (NFA)

- $M = \{S, \Sigma, \delta, s_0, F\}$
  - S: a finite, nonempty set of *states*
  - Σ : an *alphabet*
  - δ : mapping $S \times \Sigma \rightarrow 2^S$
  - s_0: start state
  - F : the set of accepting (or final) states



$\delta(0, a) = \{0, 1\}$
$\delta(0, b) = \{0\}$
$\delta(1, b) = \{2\}$
$\delta(2, b) = \{3\}$

Simulating an NFA

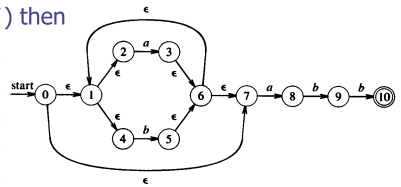
- Input
  - An input string x terminated by an eof
  - A NFA N
- Output
  - "yes" if N accepts x; "no" otherwise
- Method

```
S = ε-closure(s0)
c = nextchar;
while c ≠ eof do
  S = ε-closure(moveto(S, c));
  c = nextchar();
end
if S ∩F ≠ψ then return "yes"
             else return "no"
```


Operations on NFA States

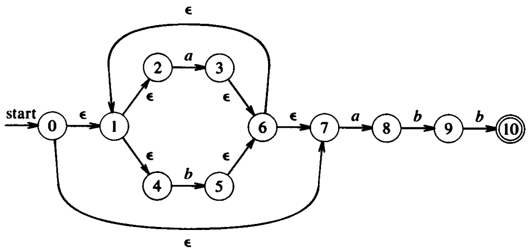
- ϵ-*closure*(s)
  - Set of NFA states reachable from state s on ϵ-transitions only
- ϵ-*closure*(T)
  - Set of NFA states reachable from some state s in T on ϵ-transitions only

```
push all states in T onto stack
initialize ε-closure(T) to T
while stack is not empty do
  pop t, the top element, off the stack
  for each state u with an edge from t to u labeled ε do
    if u is not in ε-closure(T) then
      add u to ε-closure(T)
      push u onto stack
    end if
  end
end
```



Operations on NFA States

- *moveto*(T, a)
  - Set of NFA states to which there is a transition on input symbol a from some NFA state s in T
- Example: NFA for (a|b)*abb
  - ϵ-*closure*(0) = {0, 1, 2, 4, 7} ≡ A
  - *moveto*(A, a) = ϵ-*closure*({3, 8}) = {1, 2, 3, 4, 6, 7, 8} ≡ B



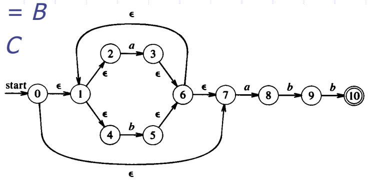
Constructing a DFA from an NFA

- Algorithm (Subset construction)
  - Input. An NFA N
  - Output. A DFA D accepting the same language
  - Method. Constructs a transition table *Dtran* for D

initially, ϵ-*closure*(s_0) is the only state in *Dstates* and unmarked
while there is an unmarked state T in *Dstates* do
  mark T
  for each input symbol a do
    U = ϵ-*closure*(*moveto*(*T*, a))
    if *U* is not in *Dstates* then
      add *U* to as an unmarked state to *Dstates*
    Dtran[*T*, a] = *U*
  end
end

Constructing a DFA from an NFA

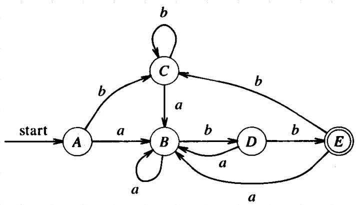
- Example: NFA for (a|b)*abb
  - ϵ-*closure*(0) = {0, 1, 2, 4, 7} ≡ *A*
  - *moveto*(*A*, a) = ϵ-*closure*({3, 8}) = {1, 2, 3, 4, 6, 7, 8} ≡ *B*
  - *moveto*(*A*, b) = ϵ-*closure*({5}) = {1, 2, 4, 5, 6, 7} ≡ *C*
  - *moveto*(*B*, a) = ϵ-*closure*({3, 8}) = *B*
  - *moveto*(*B*, b) = ϵ-*closure*({5, 9}) = {1, 2, 4, 5, 6, 7, 9} ≡ *D*
  - *moveto*(*C*, a) = ϵ-*closure*({3, 8}) = *B*
  - *moveto*(*C*, b) = ϵ-*closure*({5}) = *C*
  - *moveto*(*D*, a) = ϵ-*closure*({3, 8}) = *B*
  - *moveto*(*D*, b) = ϵ-*closure*({5, 10}) = {1, 2, 4, 5, 6, 7, 10} ≡ *E*
  - *moveto*(*E*, a) = ϵ-*closure*({3, 8}) = *B*
  - *moveto*(*E*, b) = ϵ-*closure*({5}) = *C*



Constructing a DFA from an NFA

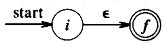
- Example: NFA for (a|b)*abb

State	Input Symbol	
	a	b
A	B	C
B	B	D
C	B	C
D	B	E
E	B	C

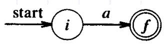


Constructing NFA from Regular Expression

- Algorithm (Thompson's construction)
  - Input. A regular expression *r* over Σ
  - Output. An NFA *N* accepting *L*(*r*)
  - Method.
    - For ϵ, construct the NFA



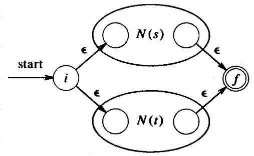
      - This NFA recognizes {ϵ}
    - For a in Σ, construct the NFA



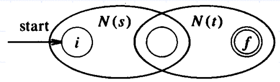
      - This NFA recognizes {a}

Constructing NFA from Regular Expression

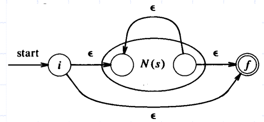
- Suppose *N*(*s*) and *N*(*t*) are NFA's for regular expressions *s* and *t*
  - For the regular expression *s* | *t*
    - This NFA recognizes L(s) ∪ L(t)



  - For the regular expression *st*
    - This NFA recognizes L(s)L(t)

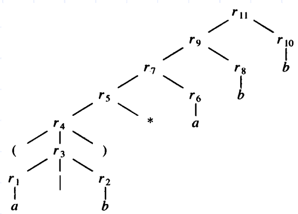


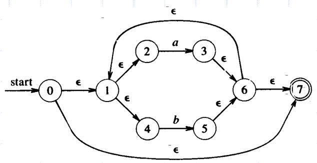
  - For the regular expression *s**
    - This NFA recognizes L(s)*

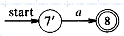


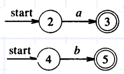
Constructing NFA from Regular Expression

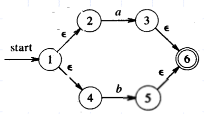
- Example: *r* = (a|b)*abb
  - Parse *r* into its constituent subexpressions

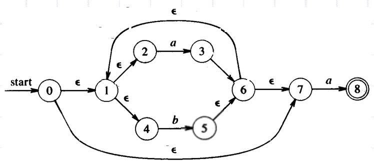












Constructing NFA from Regular Expression

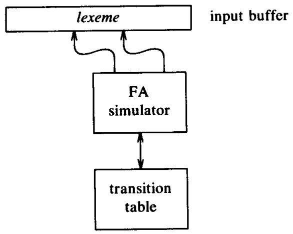
- The construction produces an NFA *N*(*r*) with the following properties
  - *N*(*r*) has at most twice as many as states as the number of symbols and operators in *r*
    - Each step creates at most two new states
  - *N*(*r*) has exactly one start state and one accepting state
    - The accepting state has no outgoing transitions
  - Each state of *N*(*r*) has either one outgoing transition on a symbol in Σ or at most two outgoing ϵ-transitions

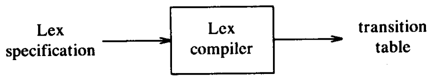
Design of a Lexical Analyzer Generator

- Assuming a specification of a lexical analyzer

  p_1 { $action_1$ }
  p_2 { $action_2$ }
  …
  p_n { $action_n$ }

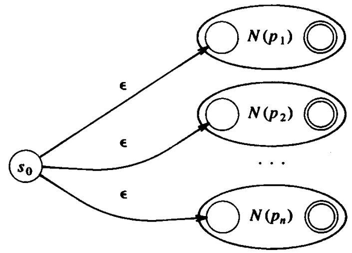
- A finite automaton is a natural model





Design of a Lexical Analyzer Generator

- Pattern matching based on NFA's
  - To construct the transition table of a NFA N for the pattern $p_1 | p_2 | \dots | p_n$



Minimizing the Number of States

- An important theoretical result
  - Every regular set is recognized by a minimum-state DFA that is unique up to state names
- String *w distinguishes* state *s* from state *t* if
  - By starting with the DFA *M* in state *s* and feeding it input *w*, we end up in an accepting state, but
  - By starting in state *t* and feeding it input *w*, we end up in a nonaccepting state, or vice versa
  - e.g. ϵ distinguishes any accepting state from any nonaccepting state
- DFA states can be minimized by finding all groups of states that can be distinguished by some input string
  - Each group of states that cannot be distinguished by some input staring is then merged into single state

Minimizing the Number of States

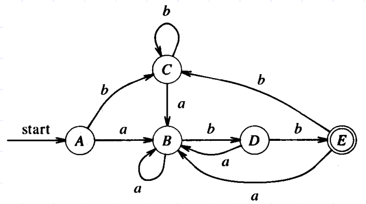
- Algorithm
  - Construct an initial partition $\Pi = \{F, S - F\}$
  - Construct a new partition:

    for each group $G \in \Pi$ do
      partition G into subgroups such that
        two states s and $t \in G$ are in the same subgroup iff
        for all input symbol a,
        s and t have transitions on a to states in the same group in Π
      replace G in Π_{new} by the set of subgroups formed
    end

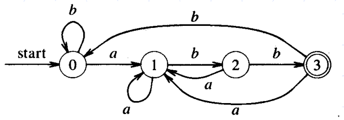
  - If $\Pi_{new} \neq \Pi$, then let $\Pi = \Pi_{new}$. Repeat.
  - Choose one state in each group as the representative
    - Update the transitions

Minimizing the Number of States

- Example
  - Initially, Π = {(ABCD), (E)}
    - (ABCD) ⇒ (ABC)(D)
  - Π = {(ABC), (D), (E)}
    - (ABC) ⇒ (AC)(B)
  - Π = {(AC), (B), (D), (E)}



State	Input Symbol	
	a	b
A (0)	B	A
B (1)	B	D
D (2)	B	E
E (3)	B	A



Building Regular Grammar from NFA

- Input. An NFA *N*
- Output. A regular grammar *r*. *N* accepts *L*(*r*)
- Method
  - For each state *i* of N, create a nonterminal symbol A_i
  - If state *i* goes to state *j* on symbol a, introduce the production $A_i \rightarrow aA_j$
  - If state *i* goes to state *j* on symbol ∊, introduce the production $A_i \rightarrow A_j$
  - If state *i* is a final state, introduce the production $A_i \rightarrow \epsilon$
  - If state *i* is the start state, make A_i be the start symbol

Building Regular Grammar from NFA

- Example
  - Nonterminal symbol:

    A_0, A_1, A_2, A_3
  - Productions

    $A_0 \rightarrow aA_0$

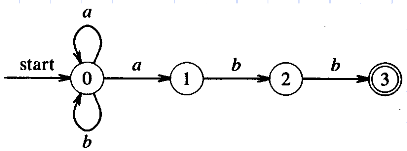
    $A_0 \rightarrow bA_0$

    $A_0 \rightarrow aA_1$

    $A_1 \rightarrow bA_2$

    $A_2 \rightarrow bA_3$

    $A_3 \rightarrow \epsilon$
  - Start symbol: A_0



Building NFA from Regular Grammar

- Input. A regular grammar $G = (V_N, V_T, P, S)$
- Output. An NFA $N = (K, V_T, \delta, s_0, F)$ that accepts $L(G)$
- Method
  - $K = V_N \cup \{A\}$
  - If $S \rightarrow \epsilon \in P$, $F = \{S, A\}$

    Otherwise, $F = \{A\}$
  - If $B \rightarrow a \in P$ then

    $A \in \delta(B, a)$
  - If $B \rightarrow aC \in P$ then

    $C \in \delta(B, a)$
  - For every $a \in V_T$

    $\delta(A, a) = \emptyset$