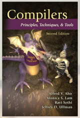
Compilers: Principles, Techniques, and Tools

Chapter 3 Lexical Analysis

Dongjie He
University of New South Wales
https://dongjiehe.github.io/teaching/compiler/

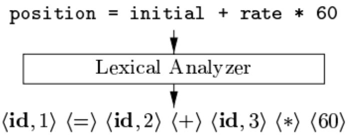
29 Jun 2023

THE UNIVERSITY OF NEW SOUTH WALES
SYDNEY · AUSTRALIA

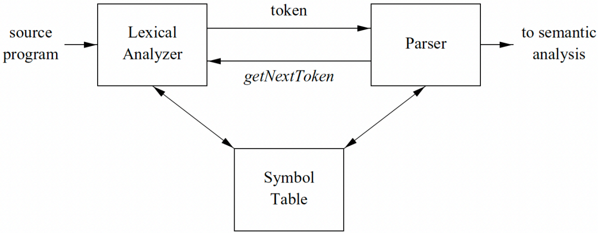


Lexical Analyzer

- A *lexical analyzer* groups multicharacter constructs as **tokens**
  - *scanning*: scan inputs, delete comments, compact whitespaces,...
  - *lexical analysis*: produce tokens from the output of scanner



- Interactions between the lexical analyzer and the parser



Lexical Analyzer

- Distinct Terms
  - *token*: ⟨**token name**, **optional attribute**⟩
  - *lexeme*: a token instance formed by a sequence of characters
  - *pattern*: the common form that the lexemes of a token may take
- Some common tokens in programming languages

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"`

- Attributes for Tokens
  - Information about the lexeme, e.g., lexeme, type, location, ...
  - a pointer to the symbole table entry

Review: Strings and Languages

- Tokens $\Leftarrow$ lexeme patterns $\Leftarrow$ regular expression
- *alphabet*: any finite set of symbols, e.g., {0, 1}, ASCII, Unicode
- *string*: a finite sequence of symbols in *alphabet*
  - synonyms: sentence, word
  - string length: $|s|$
  - *empty string*: ϵ
  - prefix/suffix/substring/subsequence
- *language*: any countable set of strings over some fixed alphabet
  - empty language: $\emptyset = \{\epsilon\}$
  - well-formed C programs, English sentences, ...
- Operations on Languages

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

Review: Regular Expression (RE)

- Inductive definition:
  - **BASIS 1**: ϵ is a RE, $L(\epsilon) = \{\epsilon\}$
  - **BASIS 2**: $a \in \Sigma$ is a RE, $L(\mathbf{a}) = \{a\}$
  - **inductive hypothesis**: r (s) is a RE denoting $L(r)$ $(L(s))$
  - **Induction 1**: $(r)|(s)$ is a RE denoting $L(r) \cup L(s)$
  - **Induction 2**: $(r)(s)$ is a RE denoting $L(r)L(s)$
  - **Induction 3**: $(r)^*$ is a RE denoting $(L(r))^*$
  - **Induction 4**: (r) is a RE denoting $L(r)$
- avoid unnecessary parentheses by adopting conventions:
  - unary operator *****, concatenation and $|$ are all **left associative**
  - **precedence**: unary operator ***** $>$ concatenation $>$ $|$
  - e.g., $(\mathbf{a})|((\mathbf{b})^*(\mathbf{c})) = \mathbf{a}|\mathbf{b}^*\mathbf{c}$
- An example: $\Sigma = \{a, b\}$
  - $\mathbf{a}|\mathbf{b}$: $\{a, b\}$
  - $\mathbf{a}^*$: $\{\epsilon, a, aa, aaa, \dots\}$
  - $(\mathbf{a}|\mathbf{b})^* = (\mathbf{a}^*\mathbf{b}^*)^*$: $\{\epsilon, a, b, aa, ab, ba, bb, aaa, \dots\}$
  - $\mathbf{a}|\mathbf{a}^*\mathbf{b}$: $\{a, b, ab, aab, aaab, \dots\}$

Review: Regular Expression (RE)

- r and s are ***equivalent***, $r = s$, if they denote the same language
  - e.g., $(\mathbf{a}|\mathbf{b}) = (\mathbf{b}|\mathbf{a})$
- Algebraic laws for RE

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

Review: Regular Definitions

- reason: notational convenience
- a sequence of definitions of the form: $d_1 \to r_1, \ldots, d_n \to r_n$
  - $d_i \notin \Sigma \cup \{d_1, \ldots, d_{i-1}\}$ is a fresh symbol
  - r_i is a RE over $\Sigma \cup \{d_1, d_2, \ldots, d_{i-1}\}$ (avoid recursive definitions)
- Example 1: C identifiers

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

- Example 2: Unsigned numbers
  - $digit \to 0 \mid 1 \mid \cdots \mid 9$
  - $digits \to digit\ digit^*$
  - $optFraction \to .\ digits \mid \epsilon$
  - $optExponent \to (\mathbf{E}\ (\ + \mid - \mid \epsilon\ )\ digits\ ) \mid \epsilon$
  - $number \to digits\ optFraction\ optExponent$

Review: Extensions of Regular Expressions (**Lex**)

Expression	Matches	Example
c	the one non-operator character c	`a`
$\backslash c$	character c literally	`\*`
"s"	string s literally	`"**"`
.	any character but newline	`a.*b`
^	beginning of a line	`^abc`
$	end of a line	`abc$`
$[s]$	any one of the characters in string s	`[abc]`
$[\hat{}s]$	any one character not in string s	`[^abc]`
$r*$	zero or more strings matching r	`a*`
$r+$	one or more strings matching r	`a+`
$r?$	zero or one r	`a?`
$r\{m,n\}$	between m and n occurrences of r	`a{1,5}`
$r_1 r_2$	an r_1 followed by an r_2	`ab`
$r_1 \mid r_2$	an r_1 or an r_2	`a|b`
(r)	same as r	`(a|b)`
r_1/r_2	r_1 when followed by r_2	`abc/123`

Review: Extensions of Regular Expressions (**others**)

- Filename expressions used by the shell command **sh**

Expression	Matches	Example
$'s'$	string s literally	`'\'`
$\backslash c$	character c literally	`\'`
$*$	any string	`*.o`
$?$	any character	`sort1.?`
$[s]$	any character in s	`sort1.[cso]`

- Shorthands: $[a_1 - a_2]$
  - $[a - z] = a \mid b \mid \cdots \mid z$
  - $[0 - 9] = 1 \mid 2 \mid \cdots \mid 9$
- Examples: identifiers and numbers
  - $id \rightarrow letter\_ \ (\ letter\_ \mid digit \ )^*$
  - $digit \rightarrow [0 - 9]$
  - $letter\_ \rightarrow [A - Za - z\_]$
  - $digits \rightarrow digit^+$
  - $number \rightarrow digits \ (. \ digits)? \ (\mathbf{E}[+-]? \ digits)?$

Lexical Analysis

- A brain route map

$$\longrightarrow \textit{source program} \longrightarrow \textit{lexemes} \longrightarrow \textit{tokens}$$

$$\longrightarrow \textit{regular expressions} \longrightarrow \textit{transition diagrams}$$

  - A *source program* consists of a sequence of *lexemes*
  - A *lexeme* is an instance any *token*
  - A *token* follows any *regular expression* pattern
  - identify the words of a *regular expression* by its *transition diagram*

Lexical Analysis: a running example

- Grammar

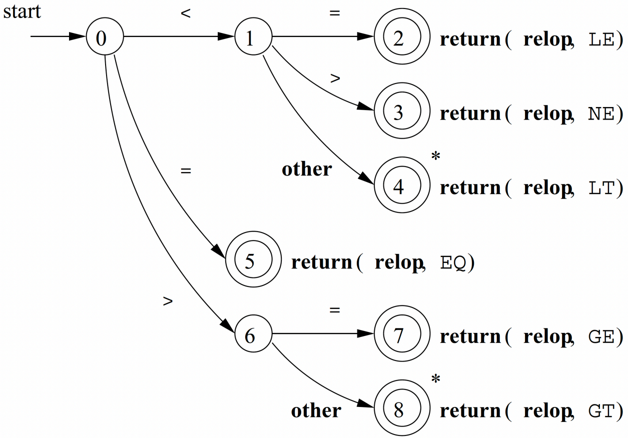
$$
\begin{array}{rcl}
stmt & \rightarrow & \textbf{if}\ expr\ \textbf{then}\ stmt \mid \epsilon \\
 & \mid & \textbf{if}\ expr\ \textbf{then}\ stmt\ \textbf{else}\ stmt \\
expr & \rightarrow & term\ \textbf{relop}\ term \mid term \\
term & \rightarrow & \textbf{id} \mid \textbf{number}
\end{array}
$$

- Tokens (Terminals): **if**, **then**, **else**, **relop**, **id**, and **number**
- Patterns: $ws \rightarrow (\ \textbf{blank} \mid \textbf{tab} \mid \textbf{newline})^+$

$digit \rightarrow [0-9]$	$digits \rightarrow digit^+$	$letter \rightarrow [A-Za-z]$
$number \rightarrow digits\ (.\ digits)?\ (\ \textbf{E}[+-]?\ digits\ )?$		
$id \rightarrow letter\ (\ letter \mid digit\ )^*$		
$if \rightarrow$ if	$then \rightarrow$ then	$else \rightarrow$ else
$relop \rightarrow < \mid > \mid <= \mid >= \mid = \mid <>$		

Lexical Analysis: a running example

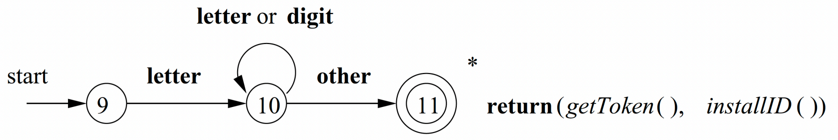
- Transition Diagrams: **relop**



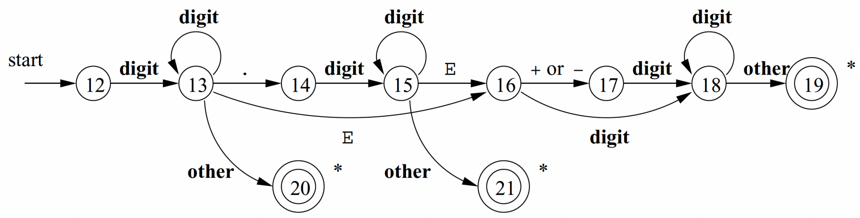
- *start* (initial) state, *accepting* (final) state, * retract character pointer

Lexical Analysis: a running example

- Transition diagram for identifiers and **keywords**

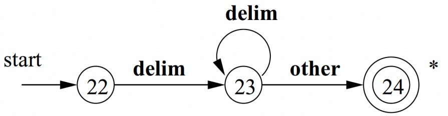


- Transition diagram for unsigned numbers

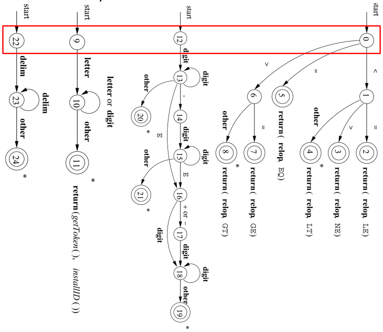


Lexical Analysis: a running example

- Transition diagram for whitespace

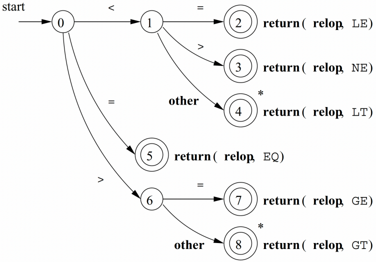


- Simulates the transition diagrams and identifies tokens
  - try one each time or try all in parallel
  - combine all into one, and simulate the one



Lexical Analysis: a running example

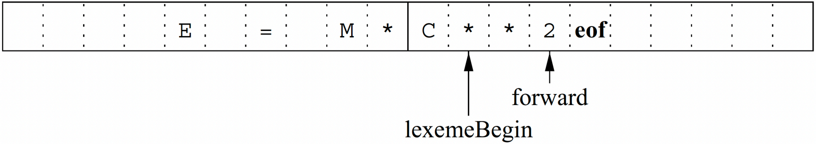
- Transition diagram Simulation
  - nextChar(), fail(), and retract()




```
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);
    }
}
```

Scanning

- How to implement nextChar()?
  - load one character each time?
  - efficient? retract()?
- Buffer Pairs



- load one buffer each time
- two buffers are alternately reloaded
- **lexemeBegin**: mark the beginning of the current lexeme
- **forward**: point to a position storing the next scanning character
- **eof**: sentinel character marking the end of a buffer or the entire input

Implementation of the running example

- An implementation of the Transition-Diagram-Based Lexical Analyzer
  `https://github.com/DongjieHe/cptt/tree/main/assigns/a3/TDBLexer`

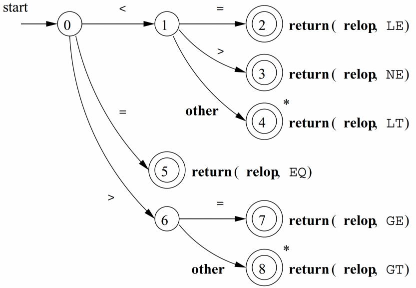
Play a Demo!

A problem remain unsolved

- How to transform regular expression into transition diagram?

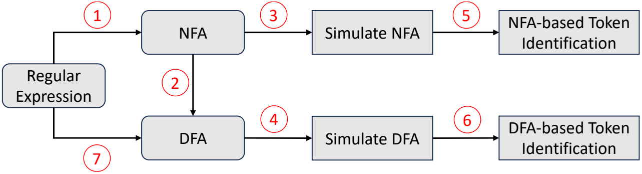
$$relop \to < \\ \mid > \\ \mid <= \\ \mid >= \\ \mid = \\ \mid <>$$

How? $\longrightarrow$



An overview of the solution

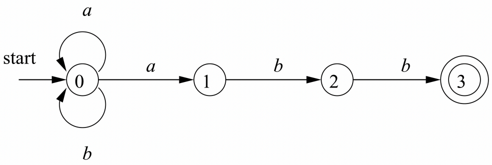
- How to transform regular expression into transition diagram?



- We first review Finite Automata: *recognizer*, say "yes" or "no"
  - *Nondeterministic Finite Automata* (NFA):
    - may have ϵ edges
    - no restrictions on edge labels
  - *Deterministic Finite Automata* (DFA):
    - no ϵ edge
    - no two edges out of any state share the same label

Review: Nondeterministic Finite Automata

- $NFA = \langle S, \Sigma \cup \{\epsilon\}, \delta, s_0, F \rangle$
  - S : finite states; s_0: start state; F: accepting states
  - Σ: input alphabet; δ: transition functions
- An example: $(\mathbf{a}|\mathbf{b})^*\mathbf{abb}$
  - transition graph

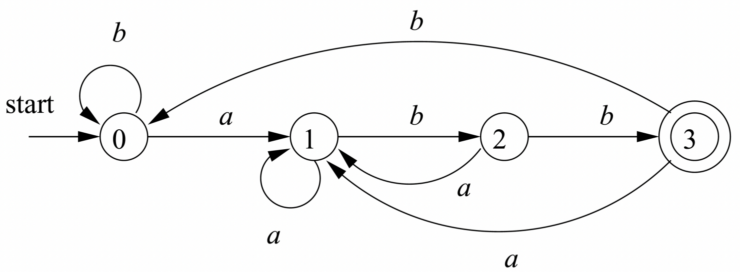


  - Transition Table

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

Review: Deterministic Finite Automata

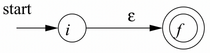
- $DFA = \langle S, \Sigma, \delta, s_0, F \rangle$ is a special NFA
  - no moves on ϵ
  - for each $s \in S$ and $a \in \Sigma$, only one edge labeled a out of s
- An example: $(\mathbf{a}|\mathbf{b})^*\mathbf{abb}$
  - transition graph for DFA



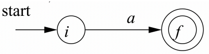
- $L(A)$: the language accepted by automaton A.

Step 1: Regular expression r to NFA $N(r)$

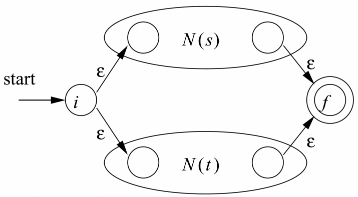
- McNaughton-Yamada-Thompson algorithm
- **Base**
  - NFA accepting ϵ



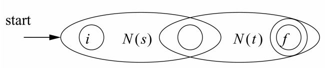
  - NFA accepting $a \in \Sigma$



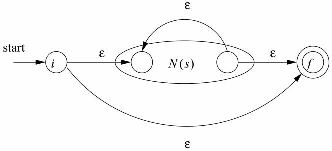
- **Induction**: $N(s)$ and $N(t)$ are NFA's for s and t
  - Union $r = s \mid t$



  - Concatenation $r = st$



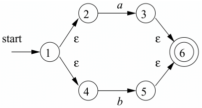
  - Closure $r = s^*$



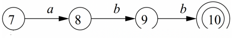
  - $r = (s)$, $N(s)$ and $N(r)$ are same

Step 1: Regular expression r to NFA $N(r)$

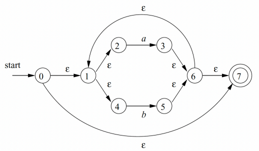
- An example: $(\mathbf{a}|\mathbf{b})^*\mathbf{abb}$
  - $(\mathbf{a}|\mathbf{b})$

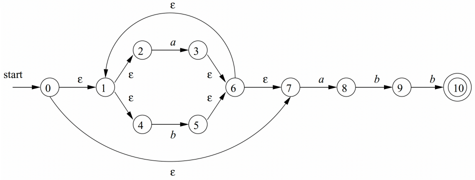


  - **abb**



  - $(\mathbf{a}|\mathbf{b})^*$





- Properties of the construced NFA $N(r)$
  - at most twice as many states as operators and operands in r
  - one start state with no incoming transition
  - one accepting state with no outgoing transition
  - one outgoing on $a \in \Sigma$ or two outgoing on ϵ for other states

Step 1: Regular expression r to NFA $N(r)$

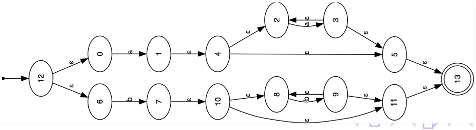
- A syntax-directed implementation in $O(|r|)$
- Grammar for Regular Expression

$$
\begin{aligned}
re &\rightarrow ur|ur \mid ur \\
ur &\rightarrow cr \cdot cr \mid cr \\
cr &\rightarrow sr^{*} \mid sr \\
sr &\rightarrow (\ re\ ) \mid op \\
op &\rightarrow \mathbf{a} \mid \mathbf{b} \mid \cdots
\end{aligned}
$$

- A link to the implementation

https://github.com/DongjieHe/cptt/tree/main/assigns/a3/RE2NFA

- An example: $\mathbf{aa}^{*}|\mathbf{bb}^{*}$



Step 2: NFA N to DFA D

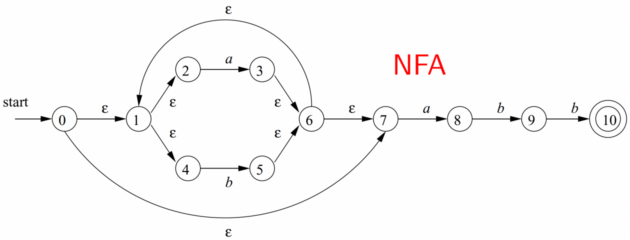
- Subset Construction Algorithm

```
initially, ε-closure(s_0) is the only state in Dstates, and it is unmarked;
while ( there is an unmarked state T in Dstates ) {
    mark T;
    for ( each input symbol a ) {
        U = ε-closure(move(T, a));
        if ( U is not in Dstates )
            add U as an unmarked state to Dstates;
        Dtran[T, a] = U;
    }
}
```

OPERATION	DESCRIPTION
$\epsilon\text{-}closure(s)$	Set of NFA states reachable from NFA state s on ϵ-transitions alone.
$\epsilon\text{-}closure(T)$	Set of NFA states reachable from some NFA state s in set T on ϵ-transitions alone; $= \cup_{s \text{ in } T}\ \epsilon\text{-}closure(s)$.
$move(T, a)$	Set of NFA states to which there is a transition on input symbol a from some state s in T.

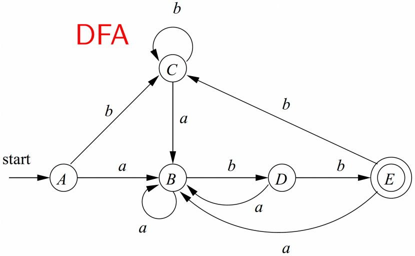
Step 2: NFA N to DFA D

- An Example: $(\mathbf{a}|\mathbf{b})^*\mathbf{abb}$



Transition Table

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



see an implementation: `https://github.com/DongjieHe/cptt/tree/main/assigns/a3/NFA2DFA`

Step 3: NFA Simulation

- Pseudo Code

```
1)  S = ϵ-closure(s0);
2)  c = nextChar();
3)  while ( c != eof ) {
4)          S = ϵ-closure(move(S, c));
5)          c = nextChar();
6)  }
7)  if ( S ∩ F != ∅ ) return "yes";
8)  else return "no";
```

- An Efficient Implementation
- run in $O(k \cdot (n+m))$, n states, m transitions, k input chars
- Link to An implementation:

https://github.com/DongjieHe/cptt/tree/main/assigns/a3/NFASimulator

- 1) : $oldStates = \epsilon\text{-}closure(s_0)$
- replace 4) with following code:

```
16) for ( s on oldStates ) {
17)         for ( t on move[s, c] )
18)                 if ( !alreadyOn[t] )
19)                         addState(t);
20)         pop s from oldStates;
21) }

22) for ( s on newStates ) {
23)         pop s from newStates;
24)         push s onto oldStates;
25)         alreadyOn[s] = FALSE;
26) }

9)  addState(s) {
10)         push s onto newStates;
11)         alreadyOn[s] = TRUE;
12)         for ( t on move[s, ϵ] )
13)                 if ( !alreadyOn[t] )
14)                         addState(t);
15) }
```

- replace S in 7) with $oldStates$

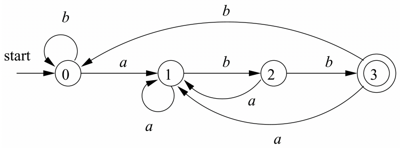
Step 4: DFA Simulation

- Pseudo Code

  $s = s_0$;
  $c = nextChar()$;
  while (c != **eof**) {
  $\quad s = move(s, c)$;
  $\quad c = nextChar()$;
  }
  if (s is in F) **return** "yes";
  else return "no";

- run in $O(k)$, k input chars

- An Example: $(\mathbf{a}|\mathbf{b})^*\mathbf{abb}$



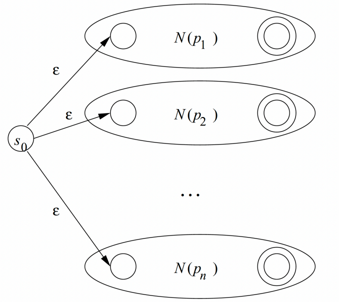
- test 1: "ababb"
- test 2: "abaabb"

Link to An implementation:

https://github.com/DongjieHe/cptt/tree/main/assigns/a3/DFASimulator

Step 5: NFA-based Lexical Analyzer

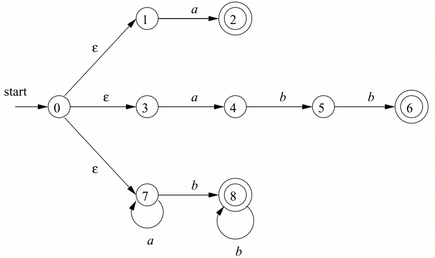
- Combine all patterns' NFA into one



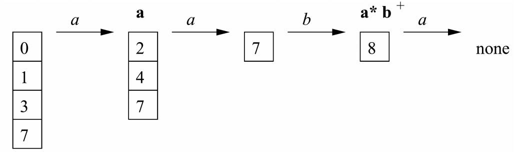
- move the pointer *forward* ahead from *lexemeBegin* until no next states
- look backwards until find a set including one or more accepting states
- pick up one associated with the earliest pattern p_i, perfom action A_i

Step 5: NFA-based Lexical Analyzer

- An example: p_1: **a**; p_2: **abb**; p_3: $\mathbf{a}^* b^+$
- combined NFA



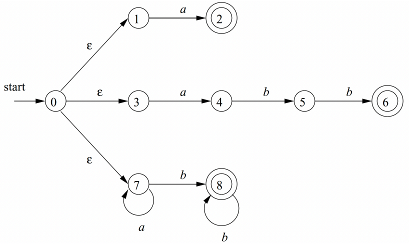
- Simulation: process input and compute the set of states



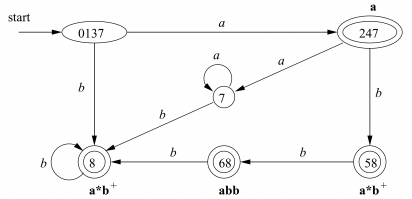
- aab, the longest prefix, is dentified to be an instance of p_3

Step 6: DFA-based Lexical Analyzer

- convert the combined NFA for all patterns into a DFA
- for each DFA state that has one or more accepting NFA states, choose the first pattern
  - An Example: p_1: **a**; p_2: **abb**; p_3: $\mathbf{a}^* b^+$



$\Longrightarrow$



- simulate DFA until no next state, look backwards until an accepting state, perform the associated action
  - An Example: input abba return abb as a lexeme

Step 7: directly from Regular Expression to DFA

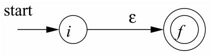
- *important* state: has a non-ϵ out-transition
- During the subset construction, S_1 and S_2 being *identified* if they
  - Have the same important states
  - *Either both have accepting states or neither does* $\Leftarrow$ Why need this?

The accepting state in NFA is not an important state

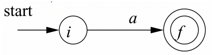
- Augmented regular expression $(r)\#$ $\Rightarrow$ NFA $\Rightarrow$ DFA
  - any state of DFA with a transition on $\#$ is an accepting state
  - DFA states could **only** be represented by *important* states
- Think about $(r)\# \Longrightarrow$ DFA ?
  - What are *important* states?
  - Initial state of DFA?
  - Given S_1 and $a \in \Sigma$, compute S_2 st. $Dtran[S_1, a] = S_2$

Review Step 1: Regular expression r to NFA $N(r)$

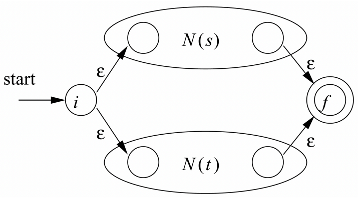
- McNaughton-Yamada-Thompson algorithm
- **Base**
  - NFA accepting ϵ



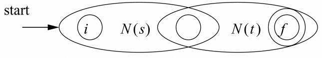
  - NFA accepting $a \in \Sigma$



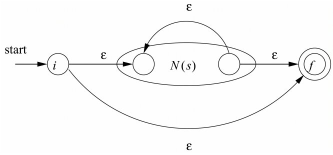
- **Induction**: $N(s)$ and $N(t)$ are NFA's for s and t
  - Union $r = s \mid t$



  - Concatenation $r = st$



  - Closure $r = s^*$



  - $r = (s)$, $N(s)$ and $N(r)$ are same

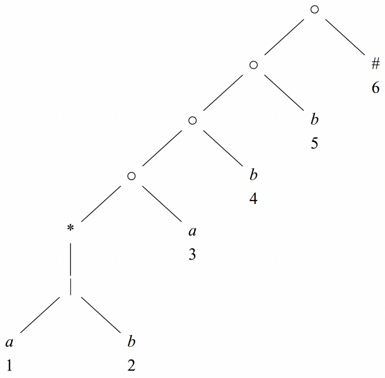
only initial states in **Base** for a particular symbol position are important

Important states from the syntax tree perspective

- Grammar for Regular Expression

$$
\begin{aligned}
re &\rightarrow ur | ur \mid ur \\
ur &\rightarrow cr \cdot cr \mid cr \\
cr &\rightarrow sr^{*} \mid sr \\
sr &\rightarrow (\ re\ ) \mid op \\
op &\rightarrow \mathbf{a} \mid \mathbf{b} \mid \cdots
\end{aligned}
$$

- Syntax tree for $(\mathbf{a}|\mathbf{b})^{*}\mathbf{abb}\#$
  - Leaves: operands, *position*
  - Interior nodes: $*$, $|$, $\cdot$



- Each node represents a subexpression

Step 7: directly from Regular Expression to DFA

- Algorithm from $(r)\#$ to DFA

```
initialize Dstates to contain only the unmarked state firstpos(n_0),
        where n_0 is the root of syntax tree T for (r)#;
while ( there is an unmarked state S in Dstates ) {
        mark S;
        for ( each input symbol a ) {
                let U be the union of followpos(p) for all p
                        in S that correspond to a;
                if ( U is not in Dstates )
                        add U as an unmarked state to Dstates;
                Dtran[S, a] = U;
        }
}
```

- *firstpos*(n): positions correspond to the first symbol of any $s \in L(n)$
  - *firstpos*(n_0) is the start state
- *followpos*(p): the positions follow the position p

Step 7: directly from Regular Expression to DFA

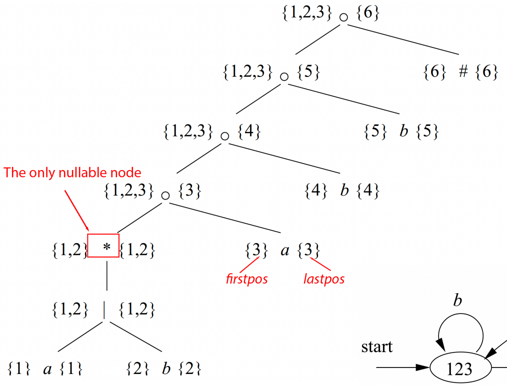
- How to compute *followpos* and *firstpos*?
  - *followpos* depends on *firstpos* and *lastpos*, which depend on *nullable*
- *nullable*(n): true iff $\epsilon \in L(n)$
- *lastpos*(n): positions correspond to the last symbol of any $s \in L(n)$
- Rules for computing *nullable*, *firstpos*, and *lastpos*

Node n	$nullable(n)$	$firstpos(n)$	$lastpos(n)$
A leaf with position i	**false**	$\{i\}$	$\{i\}$
An or-node $n = c_1 \mid c_2$	$nullable(c_1)$ **or** $nullable(c_2)$	$firstpos(c_1) \cup firstpos(c_2)$	$lastpos(c_1) \cup lastpos(c_2)$
A cat-node $n = c_1 \cdot c_2$	$nullable(c_1)$ **and** $nullable(c_2)$	**if**$(nullable(c_1))$ $firstpos(c_1) \cup firstpos(c_2)$ **else** $firstpos(c_1)$	**if** $(nullable(c_2))$ $lastpos(c_1) \cup lastpos(c_2)$ **else** $lastpos(c_2)$
A start-node $n = c_1^*$	**true**	$firstpos(c_1)$	$lastpos(c_1)$

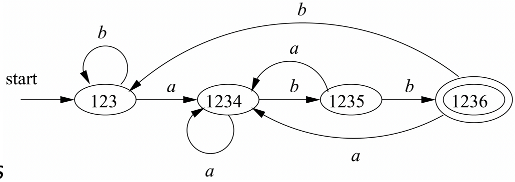
- Rules for computing *followpos*
  - $n = c_1 \cdot c_2$: $i \in lastpos(c_1) \implies firstpos(c_2) \subseteq followpos(i)$
  - $n = c_1^*$: $i \in lastpost(n) \implies firstpos(n) \subseteq followpos(i)$

Step 7: directly from Regular Expression to DFA

- An Example: $(\mathbf{a}|\mathbf{b})^*\mathbf{abb}\#$

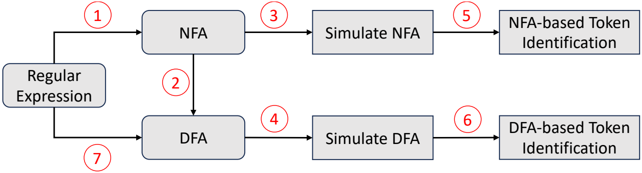


Position n	$followpos(n)$
1	$\{1,2,3\}$
2	$\{1,2,3\}$
3	$\{4\}$
4	$\{5\}$
5	$\{6\}$
6	$\emptyset$



- Rules for computing *followpos*
  - $n = c_1 \cdot c_2$: $i \in lastpos(c_1) \implies firstpos(c_2) \subseteq followpos(i)$
  - $n = c_1^*$: $i \in lastpost(n) \implies firstpos(n) \subseteq followpos(i)$

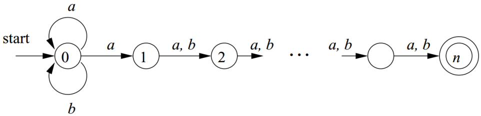
Complexity Analysis: NFA-based or DFA-based Simulation?



- Given the regular expression r and the input string x

	Complexity		Complexity
Step 1	$O(\|r\|)$	Step 3 and 5	$O(\|r\| \cdot \|x\|)$
Step 2 and 7	$O(\|r\|^2 \cdot s)$	Step 4 and 6	$O(\|x\|)$

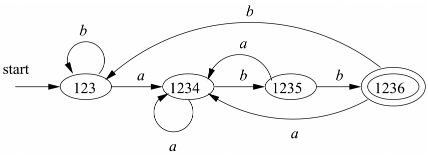
- Scale of DFA states $s = O(r)$ in typical case, $s = O(2^{|r|})$ in worst case
- An Example: $L_n = (\mathbf{a} \mid \mathbf{b})^*\mathbf{a}(\mathbf{a} \mid \mathbf{b})^{n-1}$

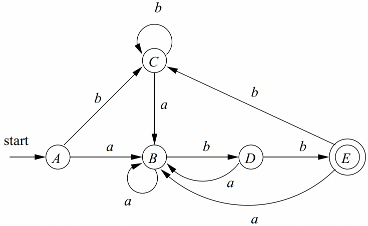


- Lexical Analyser chooses to simulate DFA while grep simulates NFA

Optimization 1: minimize the number of states of a DFA

- Many DFAs recognize the same language, e.g., $L((\mathbf{a} \mid \mathbf{b})^*\mathbf{abb})$





- DFA_1 and DFA_2 are ***the same up to state names***
  - if one can be transformed into the other by just renaming
- x ***distinguishes*** state s and state t
  - if exactly one reached from s and t by following x is an accepting state.
  - ϵ distinguishes any accepting state from anynonaccepting state.
- s is ***distinguishable*** from t if there is some string distinguishes them
- Idea: *partitioning* DFA states into groups that cannot be distinguished

Optimization 1: minimize the number of states of a DFA

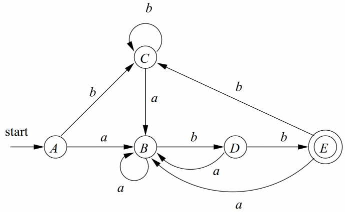
- Partitioning Algorithm: $D = \langle S, \Sigma, \delta, s_0, F\rangle$
- (1) $\Pi = [F, S - F]$
- (2) construct Π_{new}

```
initially, let Π_new = Π;
for ( each group G of Π ) {
        partition G into subgroups such that two states s and t
                are in the same subgroup if and only if for all
                input symbols a, states s and t have transitions on a
                to states in the same group of Π;
        /* at worst, a state will be in a subgroup by itself */
        replace G in Π_new by the set of all subgroups formed;
}
```

- (3) **if** $\Pi_{new} = \Pi$ **then** $\Pi_{final} = \Pi$, **goto** (4) **else** $\Pi = \Pi_{new}$, **goto** (2)
- (4) $D' = \langle S', \Sigma, \delta, s'_0, F'\rangle$, Π^i_{final} is the i-th group, $Rep(\Pi^i_{final})$
  - $s'_0 = Rep(\Pi^i_{final})$, where $s_0 \in \Pi^i_{final}$
  - $F' = \{Rep(\Pi^i_{final}) \mid \Pi^i_{final} \cap F \neq \emptyset\}$, $S' = \{Rep(\Pi^i_{final})\}$

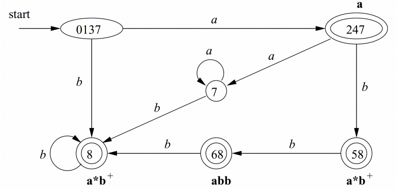
Optimization 1: minimize the number of states of a DFA

- An Example



  - $[\{A, B, C, D\}, \{E\}] \implies [\{A, B, C\}, \{D\}, \{E\}] \implies [\{A, C\}, \{B\}, \{D\}, \{E\}] = \Pi_{final}$

- State Minimization in Lexical Analyzers
  - Accepting states are initially partitioned into groups by tokens



[{0137, 7}, {8, 58}, {247}, {68}]

⇓

[{0137}, {7}, {8, 58}, {247}, {68}]

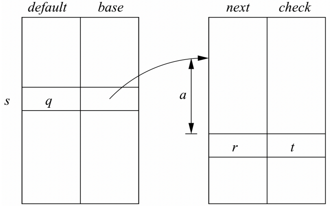
⇓

[{0137}, {7}, {8}, {58}, {247}, {68}]

Optimization 2: trade time for space in DFA Simulation

- A typical lexical analyzer uses < 1M memory/storage
  - two-dimensional table/array: $\langle$state id, input char$\rangle$
- Compilers appearing in very small devices
  - state $\mapsto [\langle$symbol, next state$\rangle, \cdots]$, less efficient but save space
  - A more subtle data structure, both time and memory efficient

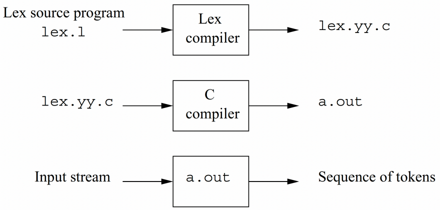
taking advantage of the similarities among states

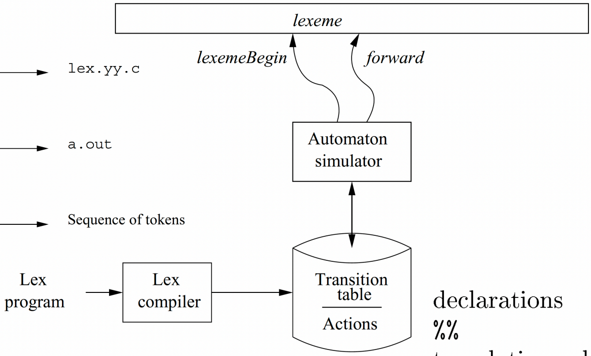


$$
\begin{array}{l}
\text{int } \textit{nextState}(s, a) \ \{ \\
\qquad \textbf{if} \ (\ \textit{check}[\textit{base}[s] + a] \ == \ s \ ) \ \textbf{return} \ \textit{next}[\textit{base}[s] + a]; \\
\qquad \textbf{else return} \ \textit{nextState}(\textit{default}[s], a); \\
\}
\end{array}
$$

Automation: Lex/Flex

- Workflow of Lex/Flex (`https://github.com/westes/flex`)






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

- Structure of Lex Programs
- A translation rule: Pattern { Action }
- declarations, actions, and auxiliary functions should in certain language, e.g., C/C++

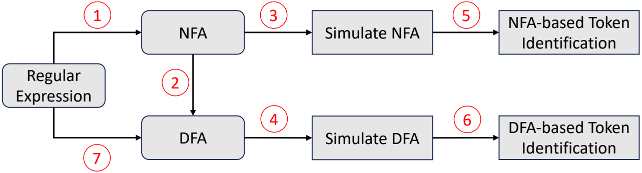
Automation: Lex/Flex

- An Example (see right figure)
  - prefer a longer prefix
  - prefer the pattern listed first
- An Implementation https://github.com/DongjieHe/cptt/tree/main/assigns/a3/flex

```
IF, THEN, ELSE, ID, NUMBER, RELOP */
#define LT 0
...
/* regular definitions */
delim [ \t\n]
ws {delim}+
letter [A-Za-z]
digit [0-9]
id {letter}({letter}|{digit})*
number {digit}+(\.{digit}+)?(E[+-]?{digit}+)?
%%
{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);}
%%
int installID() { ... }
int installNum() { ... }
```

Summary

- Review regular expression, DFA, NFA
- Implement a transition-diagram-based lexical analyzer
- Learn how to transform patterns into Automata



- Two optimization techniques
- Learn how to use Lex/Flex

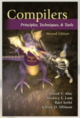
Compilers: Principles, Techniques, and Tools

Chapter 3 Lexical Analysis

Dongjie He
University of New South Wales
https://dongjiehe.github.io/teaching/compiler/

29 Jun 2023

THE UNIVERSITY OF NEW SOUTH WALES
SYDNEY · AUSTRALIA



Lab 3: Get Familiar with the Principle behind Lex

- Read the following implementations.
  - **IMP 1**: `https://github.com/DongjieHe/cptt/tree/main/assigns/a3/TDBLexer`
  - **IMP 2**: `https://github.com/DongjieHe/cptt/tree/main/assigns/a3/RE2NFA`
  - **IMP 3**: `https://github.com/DongjieHe/cptt/tree/main/assigns/a3/NFA2DFA`
  - **IMP 4**: `https://github.com/DongjieHe/cptt/tree/main/assigns/a3/NFASimulator`
  - **IMP 5**: `https://github.com/DongjieHe/cptt/tree/main/assigns/a3/DFASimulator`
  - **IMP 6**: `https://github.com/DongjieHe/cptt/tree/main/assigns/a3/flex`
- Modify **IMP 6** by
  - adding keyword **while**,
  - changing operators to be the **C** operators of that kind,
  - allowing underscore (_) as an additional letter
- Implement Step 7, i.e., transform regular expression to DFA (Hint, refer to **IMP 2**)
- Implement Optimization 1, i.e., minimize DFA (Optional)
  - refer to `https://dl.acm.org/doi/10.5555/891883`