Lexical Analysis/Scanning

Input and Output

- The input is a stream of characters (ASCII codes) of the source program.
- The output is a stream of tokens or numarical codes corresponding to different syntactic categories. It also contains attributes (associated values) of tokens.
- Examples of tokens are keywords, identifiers, constants, operators, delimiters etc.

Input and Output

- This stage takes substantial time of the front-end as it does the actual I/O.
- A typical program line may contain a few tens of characters but its output, the number of tokens, may be less than ten.
- The standard character reading library of the implementation language may not be very efficient to read the input.

Input and Output

- In the past the size of memory was a constrain. Compilers used two-buffer technique to speed-up the I/O and also to handle the splitting of a token.
- In a modern machine getting a few megabytes of memory is not difficult. And the length of a program text is well within this bound.

Input and Output

- The whole program text can be read in a buffer using two or three system calls (`open()`, `fstat()` and `read()`).
- Open the file, get the file size, create a buffer and read the text in the buffer by one call.
- Getting the complete text in the memory has many advantages.

Input and Output

- Handling variable length tokens e.g. identifiers, numbers, strings is more difficult when characters are read one at a time.
- If the text is already available in a buffer, getting the size, allocating space and copying it is easier.
- Generation of error message, showing the context is also easier.

Note

- The scanner removes the comments, white spaces, evaluates the constants, keeps track of the line numbers etc.
- This stage reduces the complexity of the syntax analyzer.
- The syntax analyzer invokes the scanner whenever it requires a token.

Token

- A token is a code corresponding to a terminal symbol of the source language grammar.
- We often use different integer codes for different tokens.

Pattern

- A pattern is a description (formal or informal) of the set of objects corresponding to a terminal (token) symbol.
- Examples are the set of identifier, set of integer constants, keywords, operator symbols etc.

Lexeme and Attribute

- A lexeme is the actual string of characters that matches a pattern.
- An attribute of a token is a value that the scanner extracts from the corresponding lexeme. This is used for semantic action.
- Typical examples are value of constant, the string of characters of an identifiers etc.

Specification of Token

- The set of strings corresponding to a token (terminals) is often a regular language, and can be specified by a regular expression.
- So the collection of tokens of a programming language can be specified by a finite set of regular expressions or a big regular expression.

Scanner from the Specification

- A scanner or lexical analyzer of a language has an NFA or DFA in its core, corresponding to the set of regular expressions of its tokens.
- The automaton and the related actions of a scanner can be implemented directly as a program or can be synthesized from its specification by another program e.g flex.

Regular Expression

1. ε, $\emptyset$ and all $a \in \Sigma$ are regular expressions.
2. If r and s are regular expressions, then so are $(r|s)$, (rs), (r^*) and (r). Nothing else is a regular expression.

We can reduce the use of parenthesis by introducing precedence and associativity rules. Binary operators are left associative and the precedence rule is $* > \text{concat} > |$.

IEEE POSIX Regular Expressions

An enlarged set of operators (defined) for the regular expressions were introduced in different software e.g. awk, grep, flex etc.[a].

- `x` or `\x` is the character itself[b].
- `.` matches with any character except '`\n`'.
- `[xyz]` is any character `x, y, z`.

[a] Consult the manual pages of lex/flex and Wikipedia for the details of IEEE POSIX standard of regular expressions.

[b] '`\x`' is used when '`x`' is a meta-character of regular expression e.g. '\·'. A few exceptions are `\n, \t, \r` etc.

IEEE POSIX Regular Expression

- If $\texttt{r}_1$ and $\texttt{r}_2$ are regular expressions, there composition rules are same as before. $\texttt{r}_1\texttt{r}_2$ is the regular expression $\texttt{r}_1$ followed by $\texttt{r}_2$, and $\texttt{r}_1 \mid \texttt{r}_2$ is either $\texttt{r}_1$ or $\texttt{r}_2$.
- Basic repetition operators are `r?` is zero or one `r`, `r*` is zero or any finite number of `r`'s, and `r+` one or any finite number of `r`'s.
- `(r)` is used for grouping.

IEEE POSIX Regular Expression

There are other operators also.

- `[abg-pT-Y]` stands for any character `a`, `b`, `g`, $\cdots$, `p`, `T`, $\cdots$, `Y`.
- `[^G-Q]` not any one of `G`, `H`, $\cdots$, `P`, `Q`.
- `r{2,}` two or more `r`'s etc.

Language of a Regular Expression

The language of a regular expression is defined in a usual way on the inductive structure of the definition.
$L(\varepsilon) = \{\varepsilon\}$, $L(\emptyset) = \emptyset$, $L(a) = \{a\}$ for all $a \in \Sigma$,
$L(r|s) = L(r) \cup L(s)$, $L(rs) = L(r)L(s)$,
$L(r^*) = L(r)^*$, $L(r?) = L(r) \cup \{\varepsilon\}$,
$L(r^+) = L(r)^+$ etc.

An Identifier

The regular expression for an identifier may be
`[_a-zA-Z][_a-zA-Z0-9]*`
The first character is an English alphabet or an underscore. From the second character on a decimal digit can also be used.

Regular Name Definition

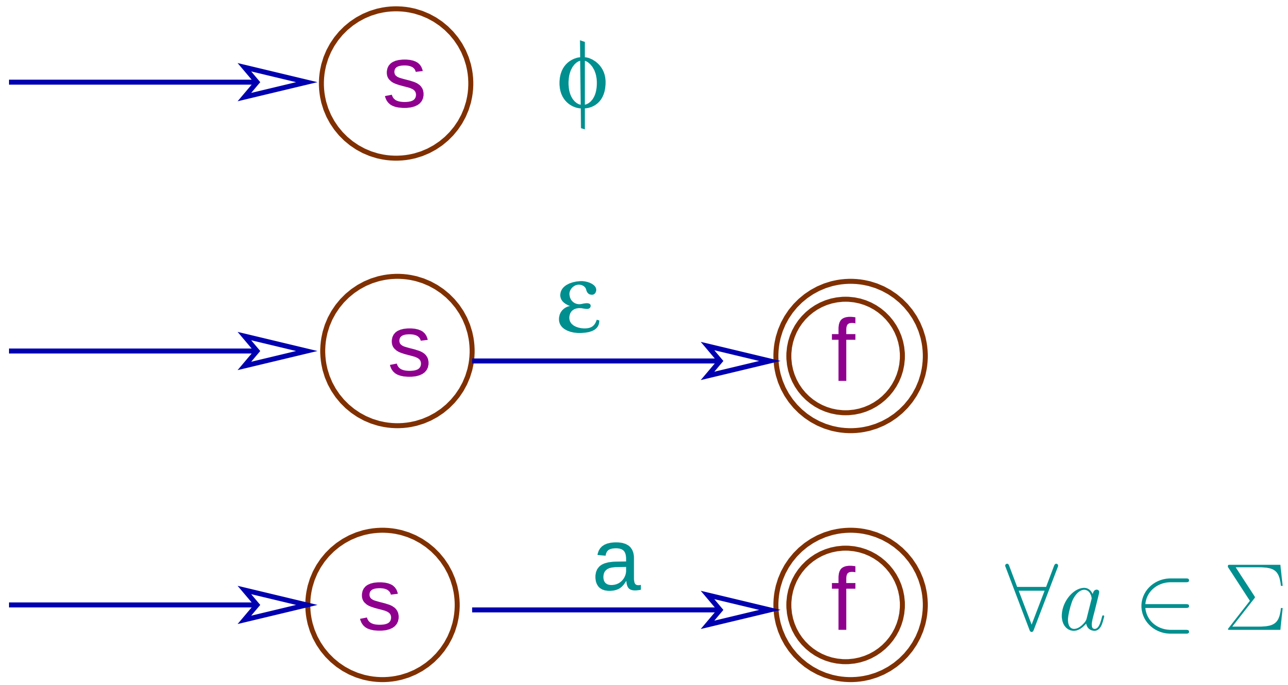
- Names can be given to sub-expressions of a regular expression for a better readability.
- A defined name can be use in subsequent expressions as a symbol that can be expanded.
- It is like a variables of a context-free grammar without recursion (EBNF).

Regular Name Definition: an Example

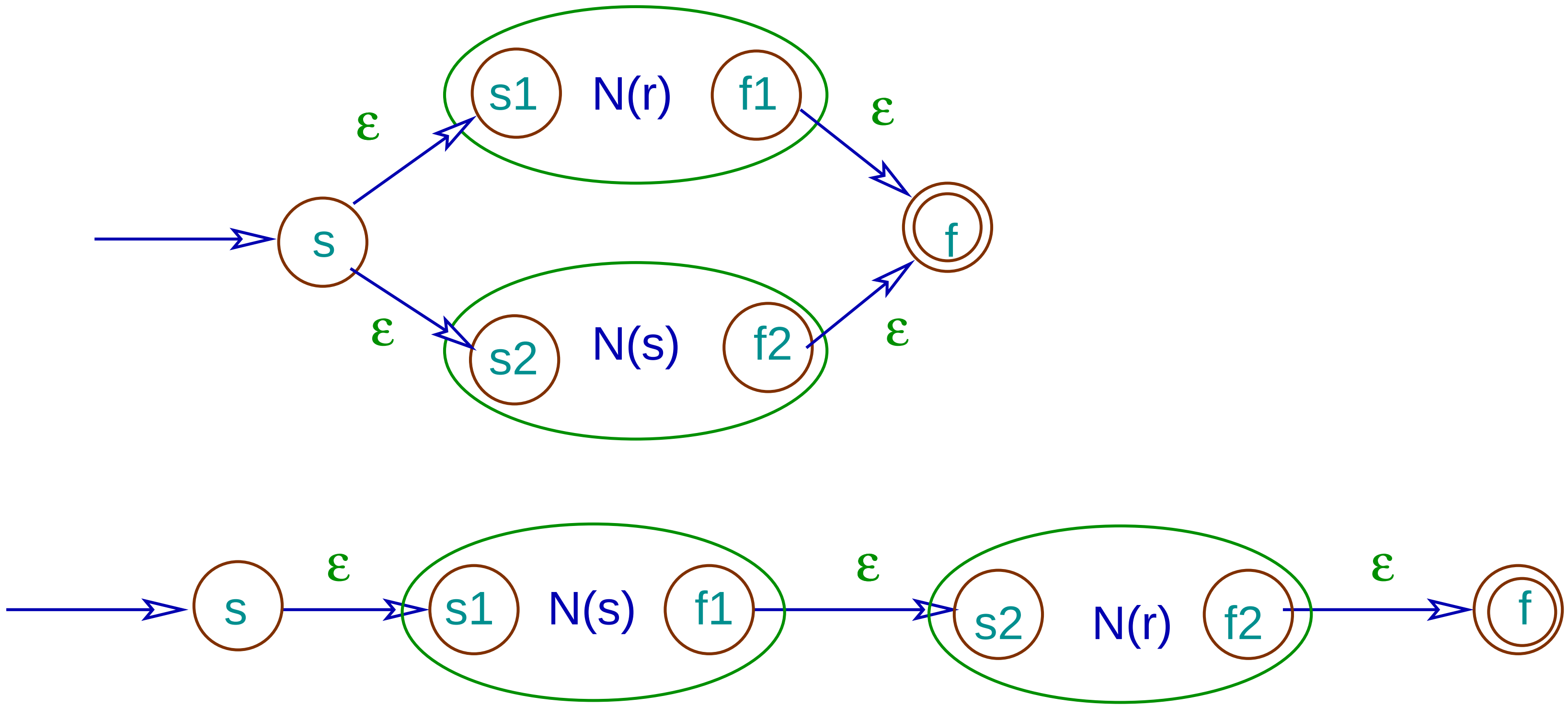
```
sign:     + | - | ε
digit:    [0-9]
digits:   {digit}*
nfrac:    \.{digits} | ε
frace:    \.{digit}{digits}
expo:     ((E | e){sign}{digit}{digit}?)  | ε
num:      {sign}(({digit}+ {nfrac} {expo}) |
          ({frace} {expo}))
```

RE to NFA: Thompson's Construction

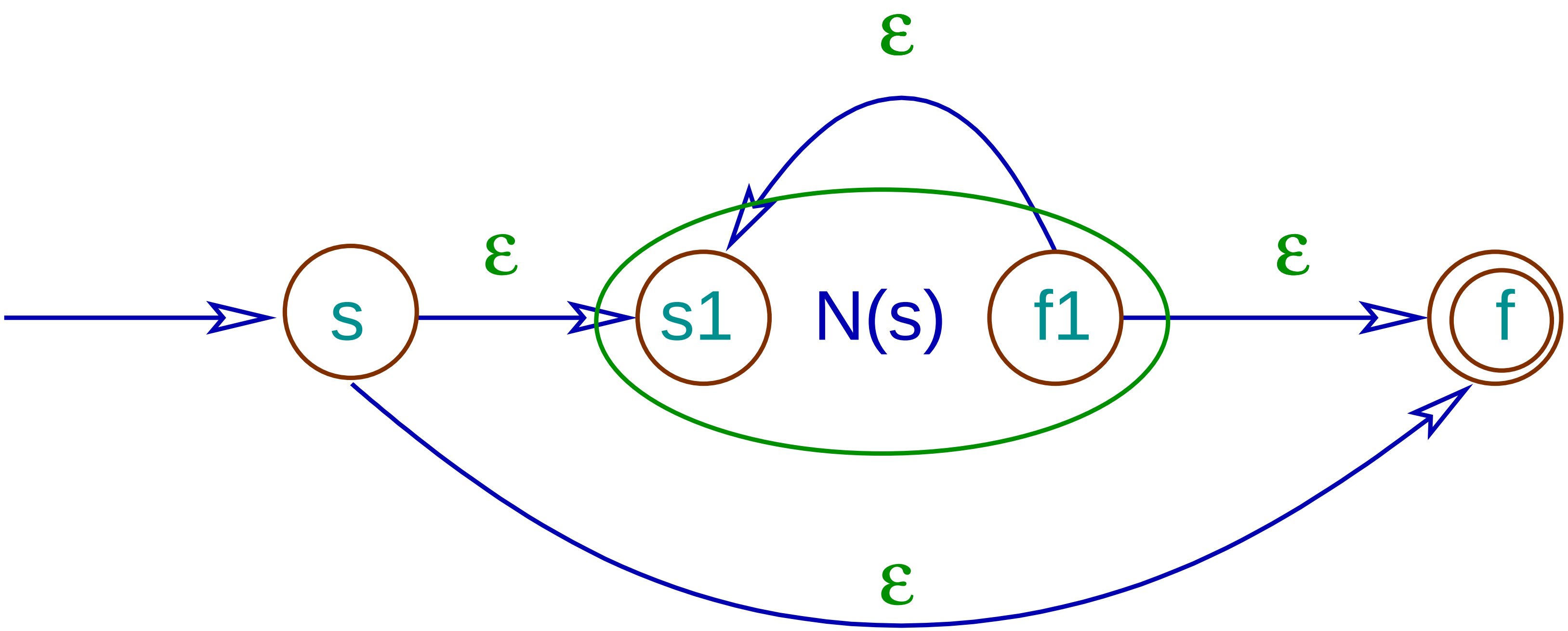
- For each $a \in \Sigma$ we can construct a 2-state NFA to recognize 'a'.
- We can combine these base NFAs using ε-transitions to build bigger NFAs.
- All these NAFs have one initial and one final state.


s
φ
s
ε
f
s
a
f
$\forall a \in \Sigma$

$(r|s)$ and (rs)



Kleene Closure: s^*



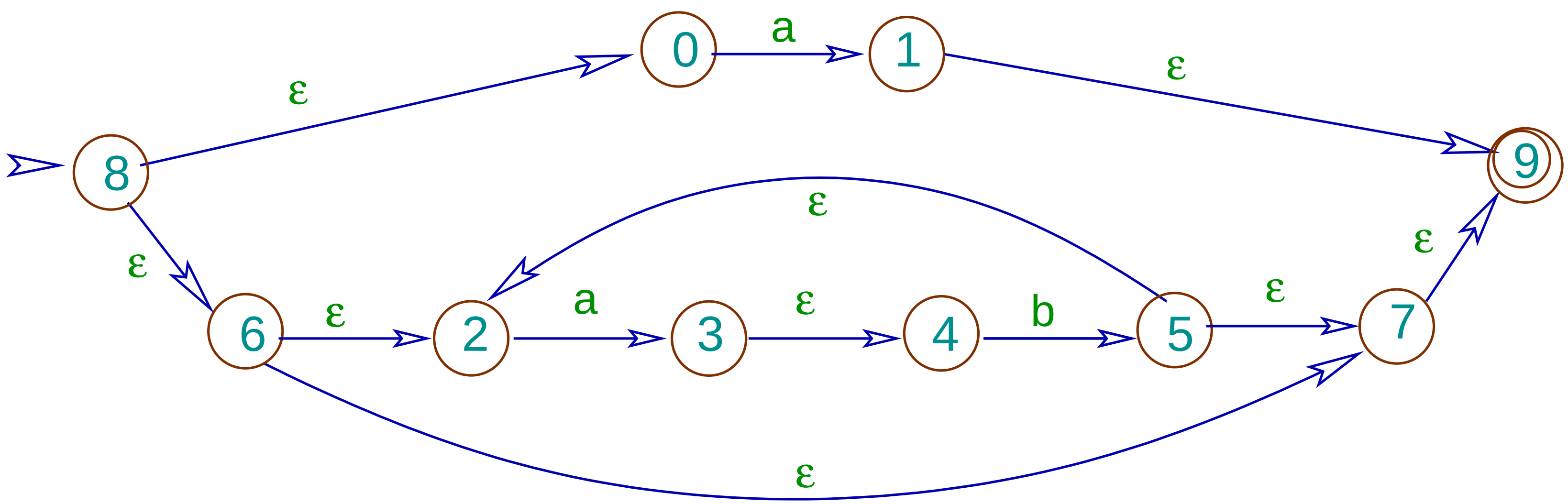
Properties of Thompson's Construction

- $|Q| \leq 2\text{length}(r)$, where Q is the number of states of the NFA and $\text{length}(r)$ is the number of alphabet and operator symbols in r.
- The constructed NFA has only one initial and one final state. There is no incoming edge to the initial state and no outgoing edge from the final state.

Properties of Thompson's Construction

- At most one incoming and one outgoing transition on a symbol of the alphabet. At most two incoming and two outgoing ε–transitions.

$a + (ab)*$ - An Example



Construction of DFA from NFA

Let the constructed ε-NFA be $(N, \Sigma, \delta_n, n_0, \{n_F\})$. By taking ε-closure of states and doing the subset construction we can get an equivalent DFA $(Q, \Sigma, \delta_d, q_0, Q_F)$.

Algorithm: Subset Construction

$q_0 = \varepsilon\text{-closure}(\{n_0\})$

$Q = L = \{q_0\}$

while$(L \neq \emptyset)$

    $q = \text{removeElm}(L)$

    for all $\sigma \in \Sigma$

        $t = \varepsilon\text{-closure}(\delta_n(q, \sigma))$

        $T[q][\sigma] = t$

        if $t \notin Q$

            $Q = Q \cup \{t\}$

            $L = L \cup \{t\}$

ε-closure(T)

for all $n \in T$ push(S, n) // S is stack

$\varepsilon T = T$

while(notEmpty(S))

    $n =$ pop(S)

    for all $n' \in \delta(n, \varepsilon)$

        if $n' \notin \varepsilon T$

            $\varepsilon T = \varepsilon T \cup \{n'\}$

            push(S, n')

Final State of the DFA

- The set of final states of the equivalent DFA is $Q_F = \{q \in Q : n_F \in q\}$.
- Different final states recognize different tokens. Also one final state may identify more than one tokens[a].

[a] A scanner may not be able to produce a token immediately from its final state, as there may be longer string matching with another token class. Often we need the maximal length match.

Time Complexity of Subset Construction

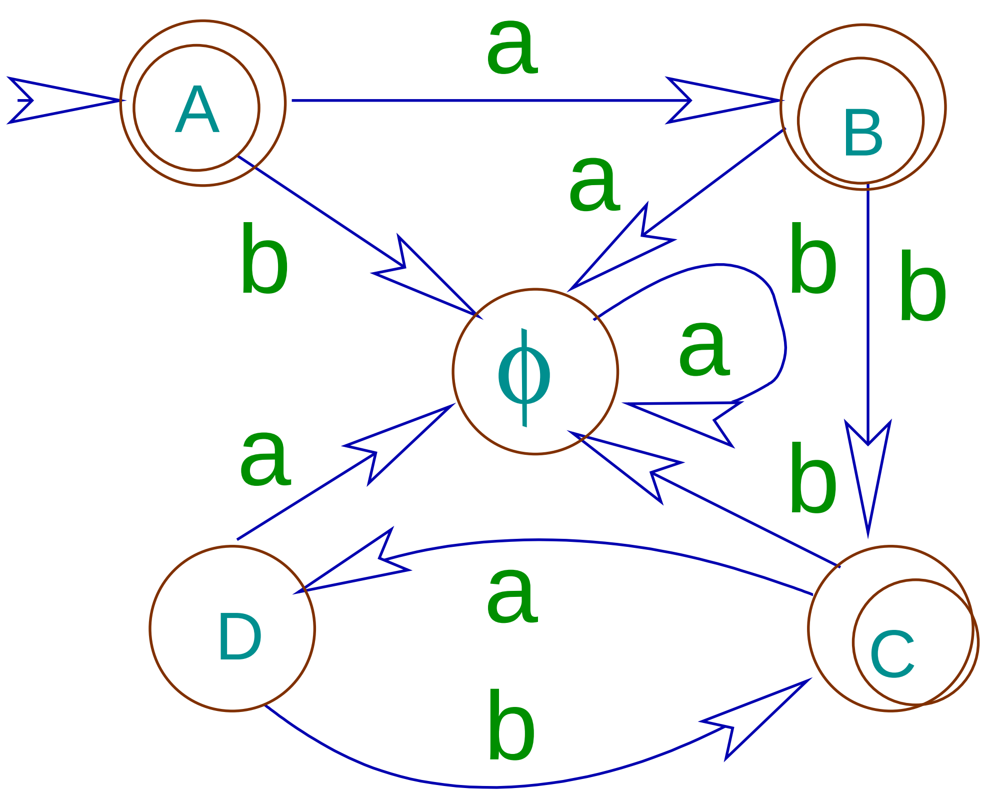
The size of Q is $O(2^{|N|})$ and so the time complexity is also $O(2^{|N|})$, where N is the set of states of the NFA. But this is one time construction.

$a + (ab)*$ - NFA to DFA

The state transition table of the DFA is

Current State	Next State	
	a	b
$A: \{0, 2, 6, 7, 8, 9\}$	$\{1, 3, 4, 9\}$	$\emptyset$
$B: \{1, 3, 4, 9\}$	$\emptyset$	$\{2, 5, 7, 9\}$
$C: \{2, 5, 7, 9\}$	$\{3, 4\}$	$\emptyset$
$D: \{3, 4\}$	$\emptyset$	$\{2, 5, 7, 9\}$
$\emptyset$	$\emptyset$	$\emptyset$

$a + (ab)*$ - NFA to DFA



Note

- We may drop the transitions to $\emptyset$ for designing a scanner. This makes the DFA incompletely specified.
- Absence of a transition from a final state identifies a token.
- But in a scanner absence of a transition from a non-final state may be due to crossing past a token.

DFA State Minimization

- The constructed DFA may have set of equivalent states[a] and can be minimized.
- The time complexity of a scanner with lesser number of states is not different from one with smaller number of states.
- Their code sizes may be different.

[a] Let $M = (Q, \Sigma, \delta, s, F)$ be a DFA. Two states $p, q \in Q$ are said to be equivalent if there is no $x \in \Sigma^*$ so that $\delta(p, x) \neq \delta(q, x)$.

DFA State Minimization

- Minimization starts with two non-equivalent partitions of Q: F and $Q \setminus F$.
- If p, q belongs to the same initial partition P of states, but there is some $\sigma \in \Sigma$ so that $\delta(p, \sigma) \in P_1$ and $\delta(q, \sigma) \in P_2$, where P_1 and P_2 are two distinct partitions, then p, q cannot remain in the same partition i.e. they are not equivalent.

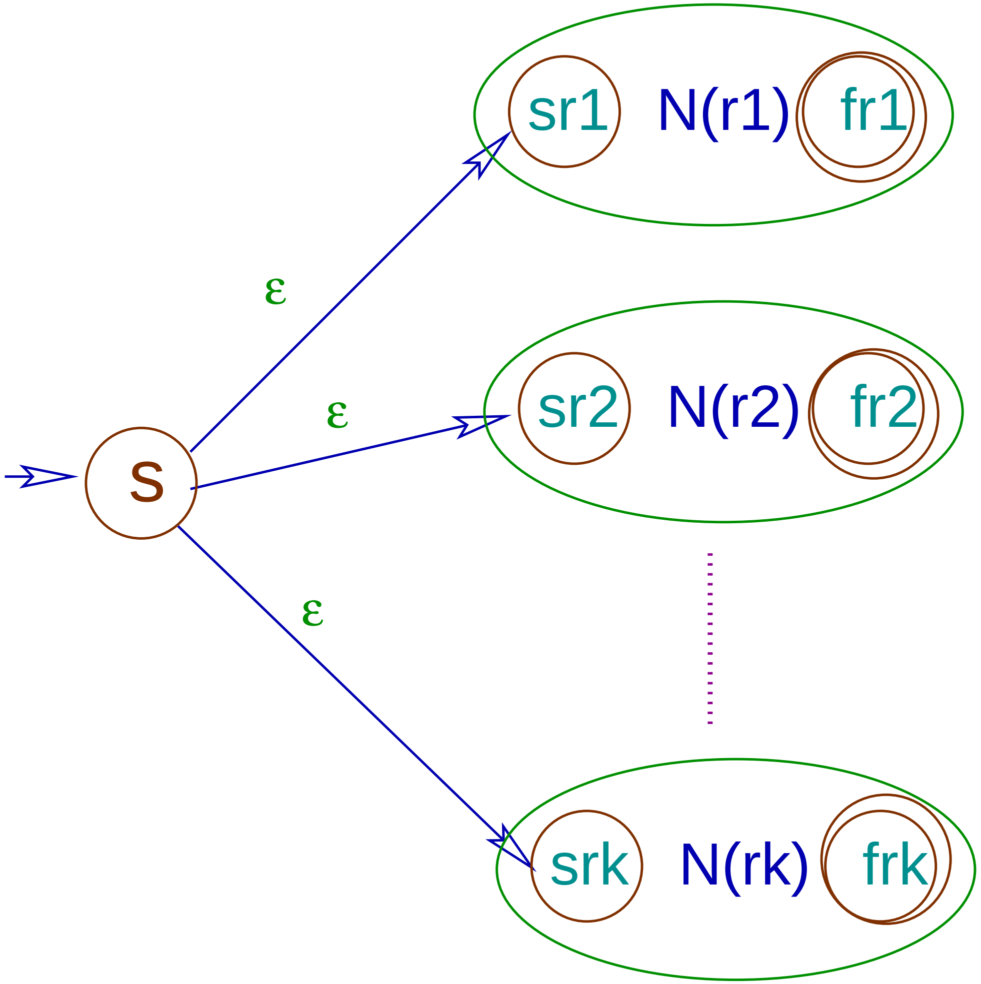
DFA to Scanner

- Given a regular expression r we can construct a recognizer of $L(r)$.
- For every token class or syntactic category of a language we have a regular expression.
- Let $\{r_1, r_2, \cdots, r_k\}$ be the total collection of regular expressions of a language. Then $r = r_1 | r_2 | \cdots | r_k$ represents objects of all syntactic categories.

DFA to Scanner

- Given the set of NFAs of $r_1, r_2, \cdots, r_k$ we construct the NFA for $r = r_1|r_2|\cdots|r_k$ by introducing a new start state and adding ε-transitions from this state to the initial states of the component NFAs.
- But we keep different final states as they are to identify different tokens.

Final Composite NFA



DFA to Scanner

The DFA corresponding to r can be constructed from the composite NFA. It can be implemented as a C program that will be used as a scanner of the language. But the following points are to be noted.

Note

- A lexically correct program is not a single word but a stream of words.
- The notion of acceptance of a token in a scanner is different from a simple DFA.

Note

- Word of one syntactic category may be a prefix of a word of another category e.g. `< << <<=`[a].
- Words of different categories are often not separated by delimiters e.g. `main(){`[b].

[a] The scanner should generate one token for $<<=$ and not three.

[b] The scanner generates four tokens, `id`, `(`, `)`, `{`

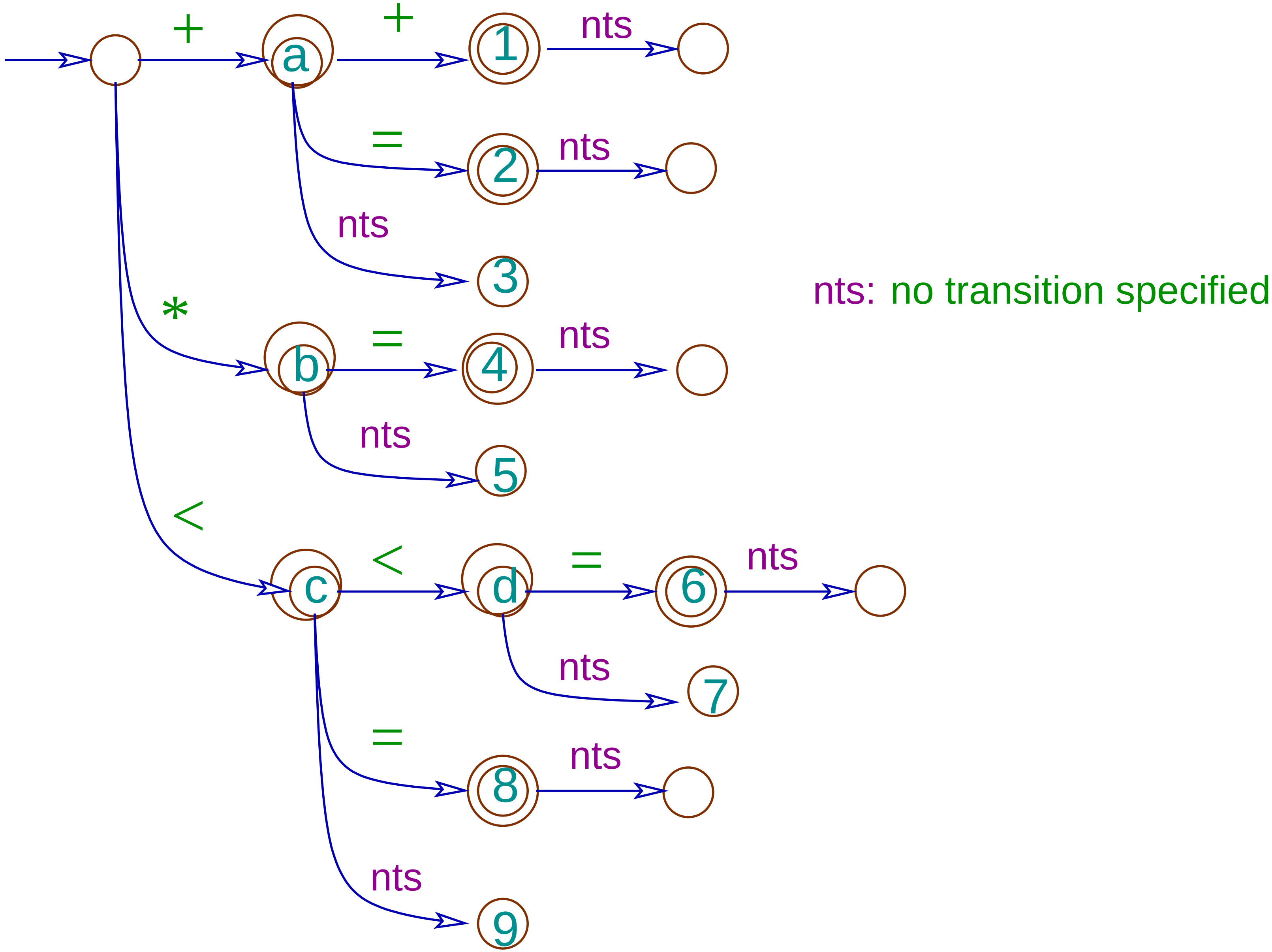
Note

We need to address the following questions.

- When does the scanner should report an acceptance?
- What does it do if the string (lexeme) matches with more than one regular expressions e.g. `int` which is a valid identifier and a keyword of C.

Example

Consider the following operators in C language:
`+ ++ += * *= < << <= <<=`
The state transition diagram of their DFA (incompletely specified) is as follows:


+
a
+
1
nts
=
2
nts
nts
3
*
b
=
4
nts
nts
5
<
c
<
d
=
6
nts
nts
7
=
8
nts
nts
9
nts: no transition specified

Note

- Both state a and 1 are final. The token for ++ can be generated at state 1 as it is not prefix to any other pattern.
- But it cannot be done at state a without a look-ahead. If the next symbol is other than $+$ or $=$, then the token for + can be generated.

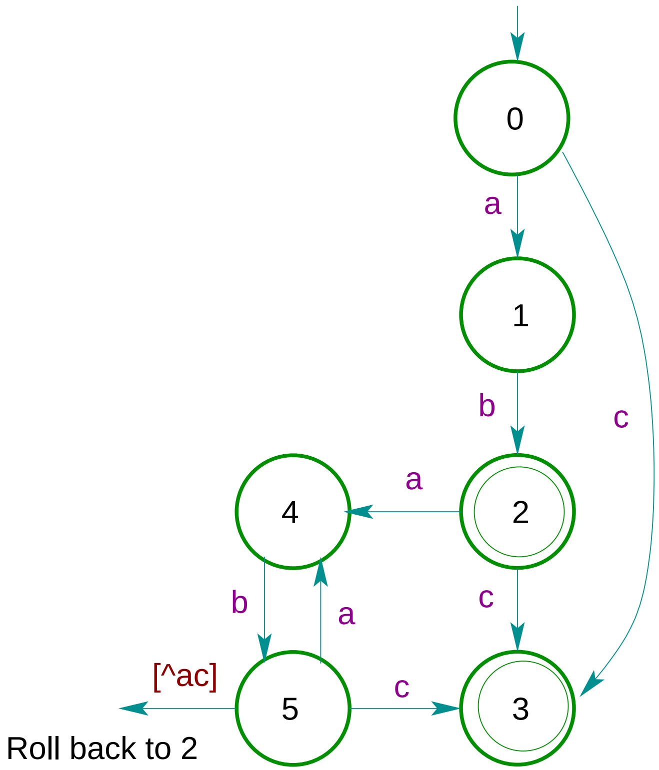
Note

- The amount of look-ahead may be more than one character.
- The look-ahead symbols are put back in the input stream (roll-back) before starting the matching for the next pattern (from the start state).

Note

- In a pathological situation the roll-back may be quadratic in complexity.
- Consider the regular expression $ab|(ab)^*c$. A maximal length tokenizer on input $ababababab c$EOF will read upto 'c' and will generate a single token.
- But what happens if the input is $ababababab$EOF?

The DFA



Sequence of states: 0 1 2 4 5 4 5 4 5 4 5 3 and
0 1 2 4 5 4 5 4 5 4 5

A Classic Example

- Here is a situations in Fortran where more than one look-ahead is necessary.

Fortran:
`DO 10 I = 1, 10` and `DO 10 I = 1.10`
The first one is a do-loop and the second one is an assignment `DO10I=1.10`. Fortran ignores blanks.
PL/I:
`IF ELSE THEN THEN = ELSE; ELSE ELSE = THEN`
`IF THEN` are not reserved as keyword.

Maximum Word Length Matching

- The scanner will go on reading input as long as there is a transition on it from the current state.
- Let there be no transitions from the current state q on the next input σ (the machine is incompletely specified).
- The state q may or may not be a final state.

q is Final

- If the final state q corresponds to only one regular expression r_i, the scanner returns the corresponding token[a].
- But if it matches with more than one regular expressions then the conflict is resolved by specifying priority to expressions e.g. a keyword over an identifier.

[a] It is necessary to identify the final state of the DFA with regular expressions. It is determined by the final states of the NFA present in the final state of the DFA.

q is not Final

- It is possible that while consuming symbols the scanner has crossed one or more final states. In a maximal length scanner, the token corresponding to the last final state is returned.
- So it is necessary to keep track of the sequence of states crossed before a final state is reached[a].

[a] A stack may be used for this purpose.

Another DFA Construction

Following is a construction of DFA from the collection of dotted items of regular expressions.

Regular Names and Dotted Items

Let $N = \alpha\beta$ be a regular expression.

- A dotted items or simply an item is a string of the form $\alpha \bullet \beta$.
- The notion of item is very useful when we try to match the regular expression with an input.

An Input and a Set of Items

- Let $x = uv \in \Sigma^*$ be the current input.
- Assume that we have already seen the part u of the input and yet to see v.
- An item $\alpha \bullet \beta$ may be valid for a situation like this.
- The regular expression α matches with the input 'u', and β is expected to match with a prefix of 'v'.

An Input and a Set of Items

- Given a set of regular expressions there will be a set of valid items for a particular situation. This set represents the state of a DFA of the regular expression.
- Consider three operator symbols of C language, `+ ++ +=`. We have three valid items after we have observed the first '+': $+\bullet$, $+\bullet+$ and $+\bullet=$.

Set of Items and State of DFA

- An item of the form $+\bullet$ is called a complete item.
- An item like $+\bullet+$ is called an incomplete or shift item.
- The state Q with $+\bullet$, $+\bullet+$, $+\bullet=$ has two incomplete and one complete items.

Transition of a Dot

- From this state Q there will be a transition to the state with item $+ + \bullet$ on input '$+$' and another transition to the state $+ = \bullet$ on input '$=$'.
- There is no other transition to any state of valid items on any other input[a]

[a]For all other input transitions reach ϕ state.

Transition of a Dot

In general

- If the item is $\alpha \bullet x\beta$, then on input symbol 'x', the transition[a] will be to $\alpha x \bullet \beta$, for $x \in \Sigma$.
- $\alpha \bullet \cdot\beta \rightarrow^{x} \alpha \cdot \bullet\beta$, for any $x \in \Sigma \setminus \{\texttt{\textbackslash n}\}$.
- $\alpha \bullet [xyz]\beta \rightarrow^{x,y,z} \alpha[xyz] \bullet \beta$, for any of $x, y, z \in \Sigma$.

[a] These are transitions of an NFA.

Transition of a Dot

- The item $\alpha \bullet (r_1|r_2)\beta$ is equivalent to two items $\alpha(\bullet r_1|r_2)\beta$ and $\alpha(r_1| \bullet r_2)\beta$. We expect to see a match for r_1 or a match for r_2.
- If there is a match for r_1, the new item is $\alpha(r_1 \bullet |r_2)\beta$. But if it is a match for r_2, the new item is $\alpha(r_1|r_2\bullet)\beta$. And both are equivalent to the item $\alpha(r_1|r_2) \bullet \beta$.

Transition of a Dot

- Item $\alpha \bullet (r)?\beta$ is equivalent to items $\alpha(\bullet r)?\beta$ and $\alpha(r)? \bullet \beta$.
- Either we expect to see a match for r or we expect to see a match for β - zero or one match for r.
- Item $\alpha(r\bullet)?\beta \equiv \alpha(r)? \bullet \beta$. Once we have seen an r, we expect a match for β.

Transition of a Dot

- Item $\alpha \bullet (r)^*\beta$ expects to see zero or any finite number of matches for the pattern r. So it is equivalent to $\{\alpha(r)^* \bullet \beta,\ \alpha(\bullet r)^*\beta\}$.
- Item $\alpha(r\bullet)^*\beta$ - after seeing an r, we again expect to see zero or any finite number of matches for the pattern r. So it is equivalent to $\{\alpha(r)^* \bullet \beta,\ \alpha(\bullet r)^*\beta\}$.

Transition of a Dot

Similarly,

- Item $\alpha \bullet (r)^{+}\beta \equiv \alpha(\bullet r)^{+}\beta$.
- Item $\alpha(r\bullet)^{+}\beta \equiv \{\alpha(r)^{+} \bullet \beta,\ \alpha(\bullet r)^{+}\beta\}$.

A Simple Example

- Consider two regular expressions, $r_1 = (ab)^*b$ and $r_2 = (a)^*b$ corresponding to two tokens.
- The combined regular expression is $r = r_1|r_2$.
- Our input should match any one of these patterns (or both). So the initial dotted item is $\bullet r$ equivalent to $\{\bullet r_1, \bullet r_2\}$. This is the start state q_0 of the DFA.

A Simple Example

- But then $\bullet r_1 = \bullet(ab)^*b \equiv \{(\bullet ab)^*b,\ (ab)^* \bullet b\}$ and $\bullet r_2 = \bullet(a)^*b = \{(\bullet a)^*b,\ (a)^* \bullet b\}$.
- So $q_0 = \{(\bullet ab)^*b,\ (ab)^* \bullet b,\ (\bullet a)^*b,\ (a)^* \bullet b\}$.
- In this way we construct the following state transition table.

A Simple Example

CS	Items	NS	
		a	b
q_0	$(\bullet ab)^*b$	q_1 :$(a \bullet b)^*b$	q_2 :$(ab)^*b\bullet$
	$(ab)^* \bullet b$	$(a)^* \bullet b$	$(a)^*b\bullet$
	$(\bullet a)^*b$	$(\bullet a)^*b$	
	$(a)^* \bullet b$		

In q_2 both items are complete.

A Simple Example

CS	Items	NS	
		a	b
q_1	$(a \bullet b)^*b$	$q_3 :(\bullet a)^*b$	$q_4 :(\bullet ab)^*b$
	$(a)^* \bullet b$	$(a)^* \bullet b$	$(ab)^* \bullet b$
	$(\bullet a)^*b$		$(a)^*b\bullet$

A Simple Example

CS	Items	NS	
		a	b
q_3	$(\bullet a)^* b$ $(a)^* \bullet b$	q_3	$q_5 : (a)^* b \bullet$

q_5 has one complete item.

A Simple Example

CS	Items	NS	
		a	b
q_4	$(\bullet ab)^* b$ $(ab)^* \bullet b$ $(a)^* b\bullet$	$q_6 : (a \bullet b)^* b$	$q_7 : (ab)^* b\bullet$

q_7 has a complete item.

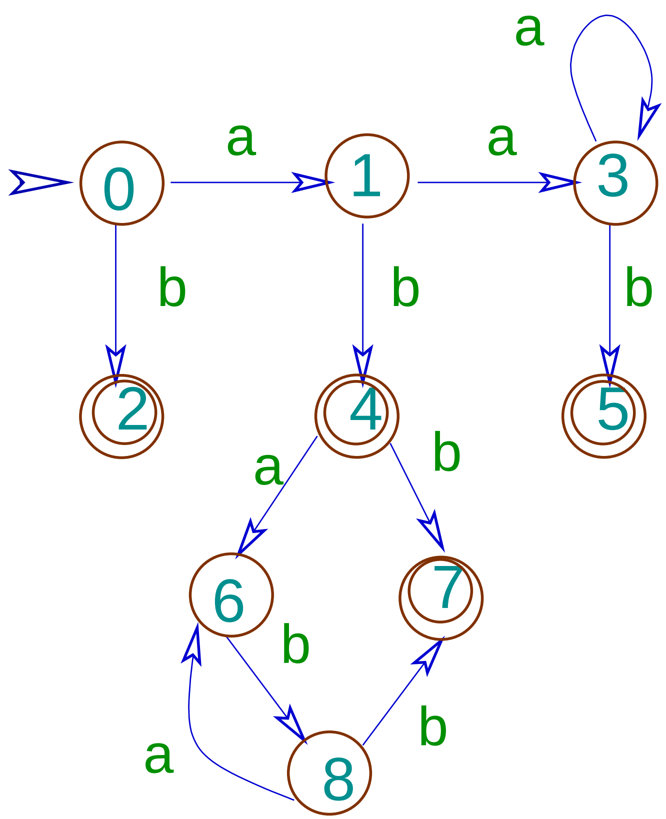
A Simple Example

CS	Items	NS	
		a	b
q_6	$(a \bullet b)^*b$		$q_8 : (\bullet ab)^*b$ $(ab)^* \bullet b$

A Simple Example

CS	Items	NS	
		a	b
q_8	$(\bullet ab)^*b$ $(ab)^* \bullet b$	q_6	q_7

A Simple Example: State Transition Diagram



A Simple Example: Note

- In q_2 there are two complete/reduce items. So two regular expressions match with the input (b). We need to decide which token to generate.
- In q_4 there are both reduce and shift items. We generate token if the input is other than a, b e.g. 'eof'.

A Simple Example: Note

- At q_5 token for a^*b is generated.
- At q_7 token for $(ab)^*b$ is generated.
- At q_6 if the input is a or 'eof', there will be a roll-back to state state q_4 and a token for a^*b is generated. But then it is an error as the last a has no token.

Components of a Scanner

1. The transition table of the DFA or NFA[a].
2. Set of actions corresponding to terminal[b] and final states.
3. Other essential functions.

---

[a] The table may be kept explicitly or implicitly (in the code).

[b] A state from where there is no transition on the current input.

Maximum Prefix on NFA

- Read input and keep track of the sequence of the set of states[a]. Stop when no more transition is possible (maximum prefix).
- Trace the sequence of the set of states backward and stop when a set of states with one or more final states is reached.

[a] In case of a DFA, there is only one element in the set. So it is a sequence of states.

Maximum Prefix on NFA

- Push back the look-ahead symbols in the input buffer and emit appropriate token along with its attribute value.
- The set of states may have more than one final states corresponding to different patterns. Action is taken corresponding to a pattern with highest priority.

From DFA to Code

Most often a DFA is used to implement a scanner. There are at least two possible implementations.

- Table driven,
- Direct coded,

We shall discuss about the table driven one.

Table Driven Scanner

There is a driver code and a set of tables. The driver code essentially has following components:

- Initialization,
- Main scanner loop,
- Roll-back loop,
- Token or error return.

Initialization

cs $\leftarrow$ q_0 # current state is the start state

lexeme $\leftarrow$ " " # null string

push$(S, \$)$ # push end of stack marker

Scanner Loop

while cs $\neq \phi$ # current state is not sink state
    if cs $\in Q_F$ then clear(S) # clear stack if cs is final
    push(S, cs) # push current state
    lexeme $\leftarrow$ lexeme + `(c = getchar())` # read next symbol
    sym $\leftarrow$ trans[c] # translate char to DFA symbol
    cs $\leftarrow \delta$(cs, sym) # current state is next state

Roll Back Loop

while cs $\notin Q_F$ and notEmpty(S)

current state is not a final state and stack is not empty

c = end(lexeme)

lexeme = lexeme - c

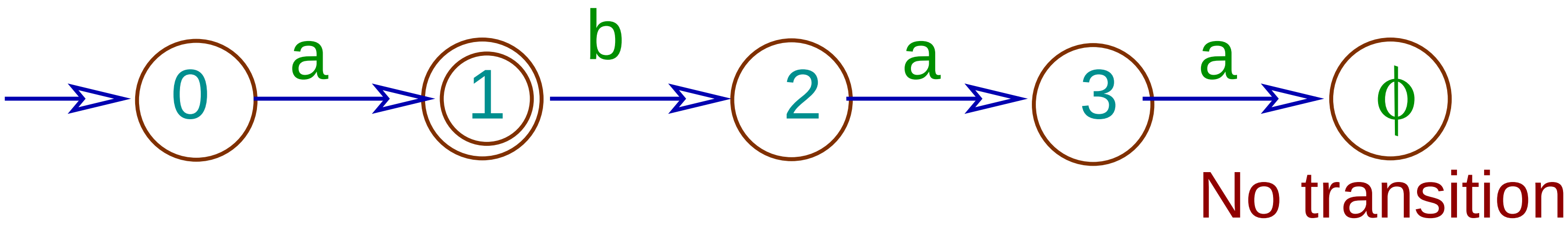
`unget(c)` # last symbol of lexeme to buffer

cs $\leftarrow$ pop(S) # pop new state from stack

Token or Error

if cs $\in Q_F$ return token[cs] # return token and attributes.
else Error # lexical error

An Example



Example

- After initialization: cs = 0, stack: empty [$], lexeme = null.
- After the scanner loop: cs = ϕ, stack: [$ 1 2 3], lexeme = "abaa".
- After the roll back loop: cs = 1, stack: empty [$], lexeme = "a"
- Token for state 1 is generated.

Tables

- `translate[]` converts a character to a DFA symbol (reduces the size of the alphabet).
- `delta[]` is the state transition table.
- `token[]` have token values corresponding to final states.

Quadratic Roll-Back

At times roll-back may be costly - consider the language $ab|(ab)^*c$ and the input `ababababab$`. There will be roll-back of $8 + 6 + 4 + 2 = 20$ characters.

Direct Coded Scanner

- Each state is implemented as a fragment of code.
- It eliminates memory reference for transition table access.

Code Corresponding to a State

- Code is labeled by the state name.
- Read a character and append it to lexeme.
- Update the roll-back stack.
- Go to next appropriate state - a valid transition, roll-back and token return state etc.

Reading Characters: Input Buffer

- We already have talked about it. The whole source file can be read in a single buffer.
- Another alternative is to map the file to the memory[a].

[a] Using `mmap()` in Linux. But the file should not be modified.

Another Construction of DFA from a Regular Expression

Another construction of deterministic finite automaton (DFA) from a given regular expression.

Important States: a Definition

- All initial states of the NFA are important.
- Any other state p of the NFA is also important if p has an out-transition on some $\sigma \in \Sigma$.
- Let the NFA be $(N, \Sigma, \delta_n, n_0, \{n_F\})$.

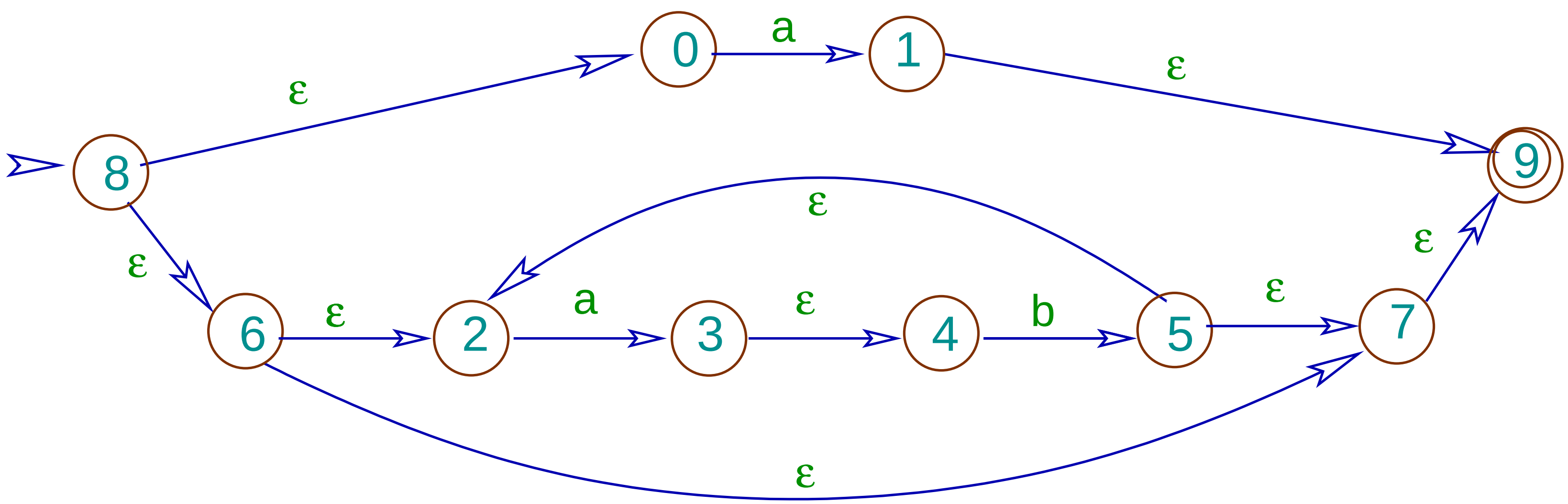
Important States

- During the construction of DFA $(Q, \Sigma, \delta_d, q_0, Q_F)$ from an NFA, we compute the next state of the DFA as ε-closure$(\delta_n(q, \sigma))$, where $q \subseteq N$ $(q \in Q)$ and $\sigma \in \Sigma$.
- In this computation $q \subseteq N$ contains only the important states of the NFA. And after transition on σ, their ε-closures is computed.

Important States

- Given a regular expression r, the important states, other than the initial state, corresponds to the positions of symbols in the regular expression.
- As an example in $a + (ab)^*$, there are four important states, the initial state and states corresponding to three positions of the symbols of Σ.

$a + (ab)*$ - An Example



Important states are $\{8, 0, 2, 4\}$.

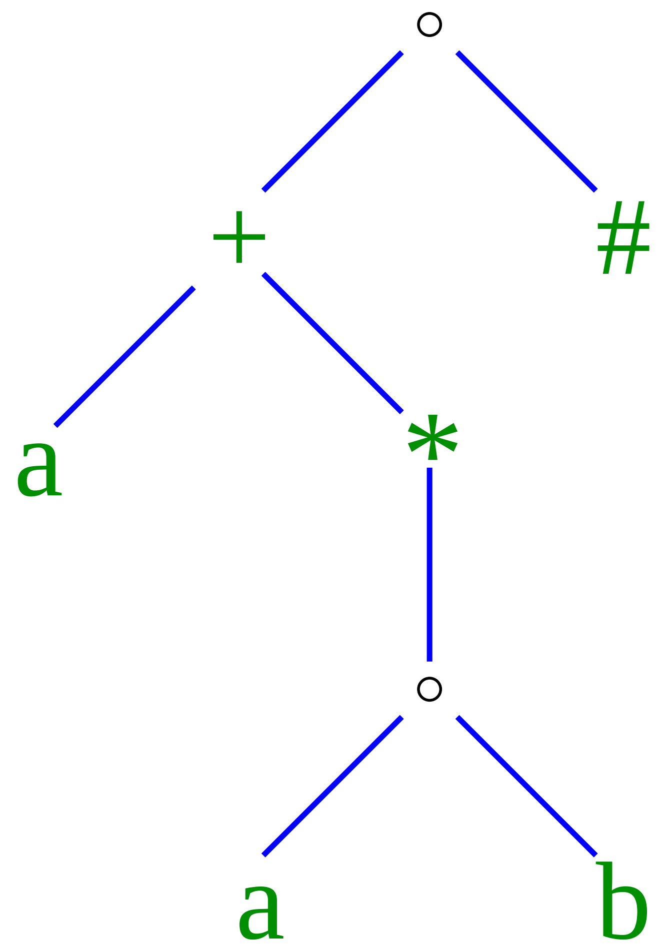
End Marker and Final State

- The final state with no transition is not important.
- We introduce a special end marker $\# \notin \Sigma$ to the regular expression, $r \rightarrow (r)\#$.
- This makes the final state(s) of the original NFA important.
- It also helps to detect the final state(s), a state having transition on $\#$.

Syntax Tree of a Regular Expression

A regular expression can be represented by a syntax tree where each leaf node corresponds to an operand $a \in \Sigma \cup \{\#, \varepsilon\}$. Each internal node corresponds to an operator symbol.

Syntax Tree of $(a + (ab)^*)\#$



Labeling the Leaf Nodes

- We associate a positive integer p with each leaf node of $a \in \Sigma \cup \{\#\}$ (not of ε). The positive integer p is called the position of the symbol of the leaf node.
- Following are a few definitions where n is a node and p is a position.

Definitions

- nullable(n): A node n is nullable if the language of its subexpression contains ϵ.
- firstpos(n): It is the set of positions in the subtree of n, from where the first symbol of any string of the language corresponding to the subexpression of n may come.

Definations

- lastpos(n): it is similar to the firstpos(n) except that these are the positions of the last symbols of strings.
- followpos(p): It is the set positions in the syntax tree from where a symbol may come after the symbol of the position p in a string of $L((r)\#)$.

Computation of nullable(n)

n is a

- leaf node with label ε: true.
- leaf node with label $a \in \Sigma$: false.
- internal node of the form $n_1 + n_2$: nullable(n_1) $\vee$ nullable(n_2).
- internal node of the form $n_1 \circ n_2$: nullable(n_1) $\wedge$ nullable(n_2).
- internal node of the form n_1^*: true.

Computation of firstpos(n)

n is a

- leaf node with label ε: $\emptyset$.
- leaf node with position p (label $a \in \Sigma \cup \{\#\}$): $\{p\}$.
- internal node of the form $n_1 + n_2$: firstpos(n_1) $\cup$ firstpos(n_2).
- internal node of the form $n_1 \circ n_2$: if nullable(n_1), then firstpos(n_1) $\cup$ firstpos(n_2), else firstpos(n_1).
- internal node of the form n_1^*: firstpos(n_1).

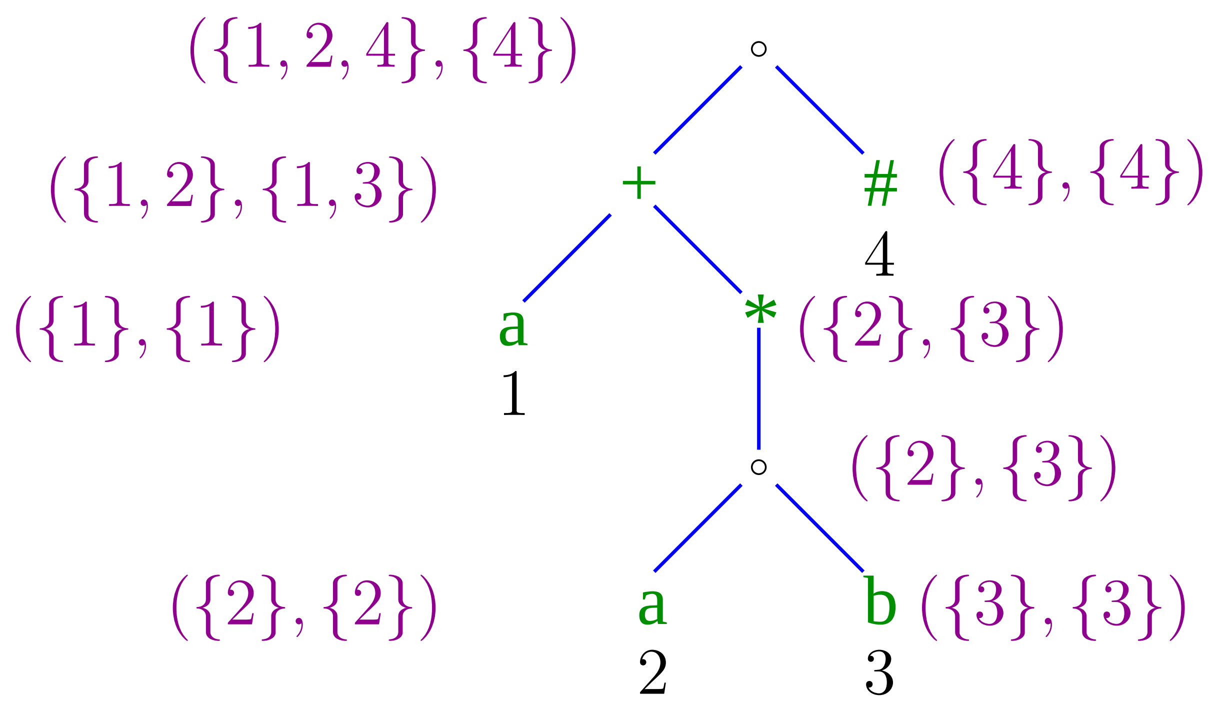
Computation of lastpos(n)

n is a

- leaf node with label ε: $\emptyset$.
- leaf node with position p (label $a \in \Sigma \cup \{\#\}$): $\{p\}$.
- internal node of the form $n_1 + n_2$: lastpos(n_1) $\cup$ lastpos(n_2).
- internal node of the form $n_1 \circ n_2$: if nullable(n_2), then lastpos(n_1) $\cup$ lastpos(n_2), else lastpos(n_2).
- internal node of the form n_1^*: lastpos(n_1).

Example

In our example there are two nullable nodes, the '$+$' and the '$*$' nodes. We decorate the syntax tree with firstpos() and lastpos() data.


({1, 2, 4}, {4})
○
+
#
({4}, {4})
4
({1, 2}, {1, 3})
({1}, {1})
a
1
*
({2}, {3})
○
({2}, {3})
({2}, {2})
a
2
b
3
({3}, {3})

Computation of followpos(p)

Given a regular expression r, a symbol of a particular position can be followed by a symbol of another position in a string of $L(r)$ in two different ways.

- If n is a concatenation node $n_1 \circ n_2$ of the syntax tree, then for each position p in lastpos(n_1), the followpos(p) contains each position q in the firstpos(n_2).

Computation of followpos(p)

- If n is a Kleene-star node of the syntax tree, then for each position p in lastpos(n), the followpos(p) contains each position q of firstpos(n).

Example

In our example,

- from the concatenation nodes we get that $3 \in \text{followpos}(2)$, $4 \in \text{followpos}(1)$ and $4 \in \text{followpos}(3)$.
- from the Kleene-star node we get $2 \in \text{followpos}(3)$.

Example

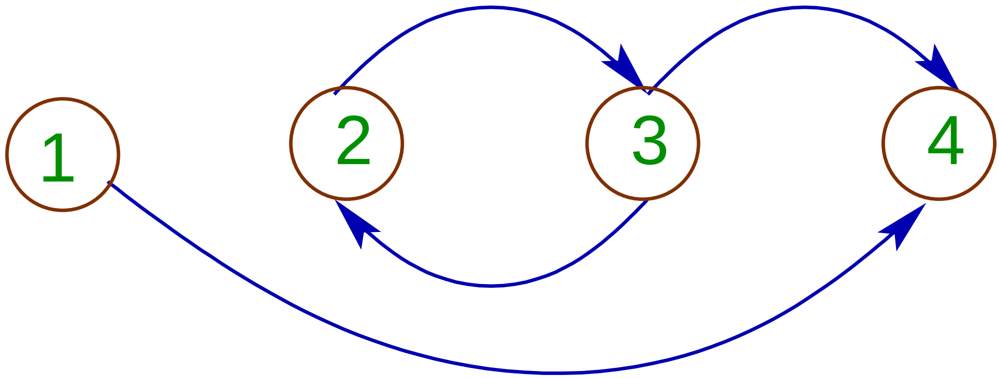
The following table summaries followpos() of different positions.

Position p	followpos(p)
1	$\{4\}$
2	$\{3\}$
3	$\{2, 4\}$
4	$\emptyset$

Directed Graph of followpos()

- Each position p is represented by a node.
- There is a directed edge from a position p to a position q, if $q \in \mathrm{followpos}(p)$.

Directed Graph of the Example

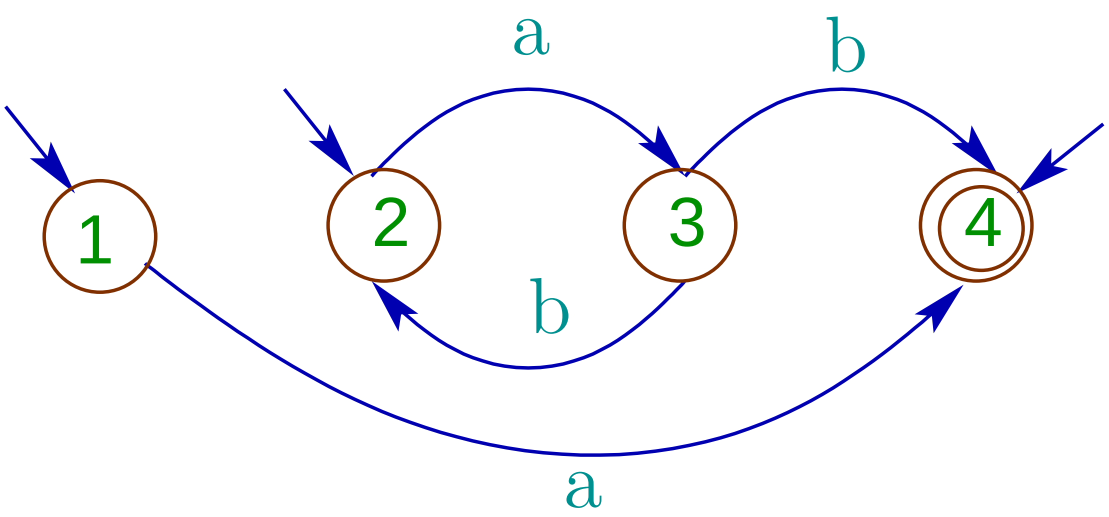


Directed Graph to NFA

This directed graph is actually an NFA without ε-transition.

- All positions in the firstpos(root) are initial states.
- A transition from $p \rightarrow q$ is labeled by the symbol of position p.
- The node corresponding to the position of $\#$ is the accepting state.

Directed Graph to NFA: the Example



DFA from Regular Expression - Direct Construction

Input: A regular expression r over Σ

Output: A DFA $M = (Q, \Sigma, s, F, \delta)$.

Algorithm:

1. Construct a syntax tree T corresponding to the augmented regular expression $(r)\#$, where $\# \notin \Sigma$.

DFA from Regular Expression - Directly

2. Compute nullable, firstpos, lastpos and followpos of the syntax tree T.

3. The construction of M is as follows: The set of states Q of M are the subsets of the positions of T. The start state $s = \text{firstpos}(\text{root}(T))$. The final states are all the subsets containing the position of $\#$.

Construction of δ

$\text{tag}[\text{firstpos}(\text{root}(T))] \leftarrow 0$

$Q \leftarrow \text{firstpos}(\text{root}(T))$

while $(\alpha \in Q$ and $\text{tag}[\alpha] = 0)$ **do**

  $\text{tag}[\alpha] \leftarrow 1$

  $\forall a \in \Sigma$ **do**

    $\forall$ positions $p \in \alpha$ of $a \in \Sigma$,

    collect followpos(p) in a set β

    if $(\beta \notin Q)$

      $\text{tag}[\beta] \leftarrow 0$

      $Q \leftarrow Q \cup \{\beta\}$

    $\delta(\alpha, a) \leftarrow \beta.$

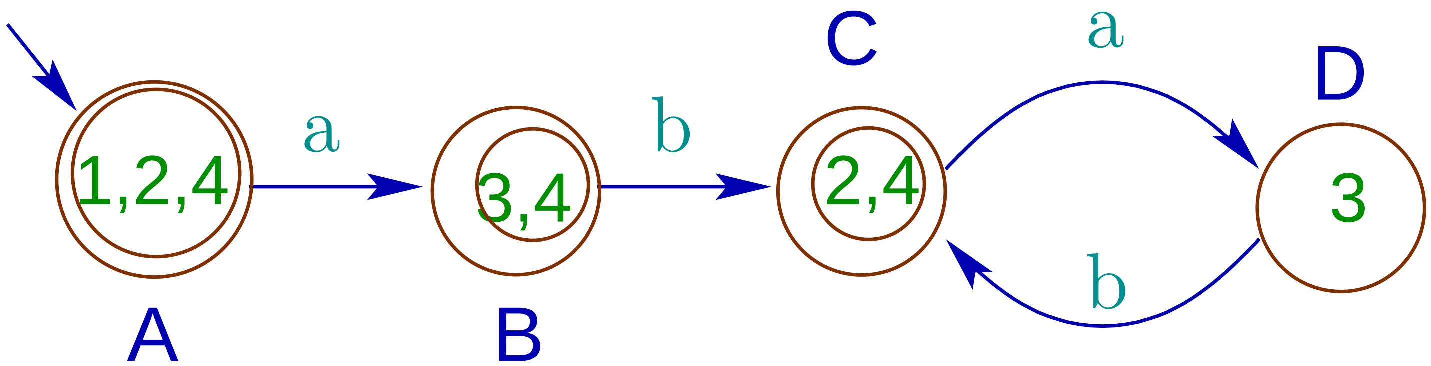
DFA of the Example

The state transition table:

Initial State	Final State	
	a	b
$A: \{1,2,4\}$	$\{3,4\}$	$\emptyset$
$B: \{3,4\}$	$\emptyset$	$\{2,4\}$
$C: \{2,4\}$	$\{3\}$	$\emptyset$
$D: \{3\}$	$\emptyset$	$\{2,4\}$

Start state: $A\{1,2,4\}$, Final states: $\{A\{1,2,4\}, B\{3,4\}, C\{2,4\}\}$.

DFA State Transition Diagram



Transition Table is Sparse

- Transitions from a state on most $a \in \Sigma$ are often to the sink state, $S_\emptyset$ (no acceptance).
- The number of valid items like $A : \alpha \bullet a\beta$, $a \in \Sigma$, are only a few in many states.
- The next state column of the transition table for most input $a \in \Sigma$ contains a small set of non-sink ($\neq S_\emptyset$) states.

Transition Table is Sparse

- The next state columns of two different input characterrs turns out to be either almost same or disjoint.

Transition Table Compression

- A sparse table can be compressed without compromising the speed and ease of access.
- Compression algorithms try to share rows of different states and put non-sink next state ($\neq S_\emptyset$) entries of one in locations of sink state ($S_\emptyset$) entries of the other.
- It also try to share identical rows of different states.

Transition Table Compression

- If two states share the same row where on input character a, one has transition to $S_\emptyset$ and the other to S_i, it is necessary to disambiguate the situation.
- Some algorithm maintains a bit map to indicates the presence of $S_\emptyset$ at an entry.
- If the bit is set for a state, the next state is $S_\emptyset$, otherwise it is S_i.

Table Compression: an Example

Let $\Sigma = \{a, b, c, d\}$, $Q = \{0, 1, 2, 3, 4\}$ and the transition table is as follows, where '-' stands for $S_\emptyset$.

CS	NS			
	a	b	c	d
0	2	–	–	–
1	–	3	–	0
2	–	3	4	–
3	2	–	–	–
4	3	4	–	1

The Bit Map for $S_\emptyset$

CS	Bit Map			
	a	b	c	d
0	0	1	1	1
1	1	0	1	0
2	1	0	0	1
3	0	1	1	1
4	0	0	1	0

Table Compression by Row Displacement

- Different rows of the original state transition table are merged to an one-dimensional transition vector, by sharing of locations and displacement of rows.
- Rows of states 0, 1, 2, 3 can be merged to a single row [2 3 4 0] as no input has conflicting transitions to the next states except the empty state ($S_\emptyset$).

Table Compression by Row Displacement

- The row corresponding to the state 4 can be partially merged by displacing it one position.

2	3	4	0	
	3	4	–	1

Table Compression by Row Displacement

- Displacements corresponding to different states are,

State	0	1	2	3	4
Displacement	0	0	0	0	1

- The compressed state transition array is (2 3 4 0 1).

State Transition in Compressed Table

The next state (q) of $\delta(p, \sigma)$ is computed as follows.

- If the bit-map of $[p, \sigma]$ is '1', $q = S_\emptyset$. $\delta(0, c) = S_\emptyset$, as '1' in the bit-map table.
- Otherwise, the state is found from the compressed table starting from the displacement of p. $\delta(4, d) = 1$ as '0' in bit-map and displacement is one.

Comparison of Space

- Let there be m states and n input symbols. If each transition table entry takes 4-bytes, then the space required is $4mn$ bytes in an uncompressed table.
- For the compressed version, there is an empty state bit-map table empty[m][n] which takes roughly $mn/32$ bytes of space (word size is 32-bits).

Comparison of Space

- The displacement vector takes 4m bytes of space and the compressed transition table vector takes 4k bytes, where k is its size.
- In the example, $m = 5, n = 4$ and $k = 5$. So the space used by the original table is 80 bytes. Space used after compression is $3 \times 5 \times 4 = 60$ bytes. We assume that each entry of the bit-map table is 1 byte.

Note

- For optimal compression it is necessary to find displacement of rows corresponding to different states so that the length of the transition vector is minimal.
- But that is an NP-complete problem[a]. So it is necessary to use heuristics to get a good solution (sub-optimal).

[a] Loosely speaking, as it is not a decision problem, but an optimization problem.

A Heuristic to Find Good Displacement

- Sort the rows according to the descending order of density (larger to smaller number of non-empty states).
- Rows are merged by first-fit.

Heuristic on Example

- Sorted rows: (3 4 - 1)(- 3 - 0)(- 3 4 -)(2 - - -)(2 - - -)
- But this doe not give minimal size transition vector.

Replacing Bit-Map by Marking

- For a large table the bit-map is replaced by markings in the entries of the state-transition vector.
- Marking can either be done using states or by the input characters.
- We shall not discuss the technique here.

Table Compression by Graph Colouring

- For a large table the set of states Q is partitioned in such a way that their next-state rows are compatible and can be combined[a].
- Given an empty-state bit-map table, compatible states can be combined to form a single row.

[a] Two states p, q are said to be compatible if for all $\sigma \in \Sigma$, either one of $\delta(p, \sigma)$ or $\delta(q, \sigma)$ is $S_\emptyset$, or they are same.

Table Compression by Graph Colouring

- The state partitioning can be done by constructing the interference graph of the states, and finding the minimum number of colours to colour the vertices.
- The states are nodes in the graph. There is an edge between the nodes of state p and q if the next-state vectors of them cannot be merged (not compatible).

Table Compression by Graph Colouring

- In our example there are five nodes $\{0, 1, 2, 3, 4\}$ and four edges $\{0, 4\}, \{1, 4\}, \{2, 4\}, \{3, 4\}$. The vertices can be coloured with two colours.
- States of same colour are in the same partition and can be merged.

Table Compression by Graph Colouring

- The next question is how to displace and merge the next state rows of the compatible states.
- If these rows are almost full (may be true for a large table), they can simply be concatenated.

References

[ASRJ] Compilers Principles, Techniques, and Tools, by A. V. Aho, Monica S. Lam, R. Sethi, & J. D. Ullman, 2nd ed., ISBN 978-81317-2101-8, Pearson Ed., 2008.

[DKHJK] Modern Compiler Design, by Dick Grune, Kees van Reeuwijk, Henri E. Bal, Ceriel J. H. Jacobs, Koen Langendoen, 2nd ed,, ISBN 978 1461 446989, Springer (2012).

[KL] Engineering a Compiler, by Keith D. Cooper & Linda Troczon, (2nd ed.), ISBN 978-93-80931-87-6, Morgan Kaufmann, Elsevier, 2012.