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 symbols corresponding to different syntactic categories. The output also contains attributes of tokens. Examples of tokens are different keywords, identifiers, constants, operators, delimiters etc.

Note

The scanner removes the comments, white spaces, evaluates the constants, keeps track of the line numbers etc.

This stage performs the main I/O and reduces the complexity of the syntax analyzer.

The syntax analyzer invokes the scanner whenever it requires a token.

Token

A token is an identifier (name/code) corresponding to a syntactic category of the language grammar. In other word it is the terminal alphabet of the grammar. Often we use an integer code for this.

Pattern

A pattern is a description (formal or informal) of the set of objects corresponding to a terminal (token) symbol of the grammar. Examples are the set of *identifier* in C language, set of integer constants etc.

Lexeme and Attribute

A lexeme is an actual string of characters that matches with a pattern and generates a token.

An attribute of a token is a value that the scanner extracts from the corresponding lexeme and supplies to the syntax analyzer.

Specification of Token

The set of strings corresponding to a token (terminals) of a programming language is often a regular set and is specified by a regular expression.

Scanner from the Specification

The collection of tokens of a programming language can be specified by a set of regular expressions. A scanner or lexical analyzer for the language uses a DFA (recognizer of regular languages) in its core. Different final states of the DFA identifies different tokens. Synthesis of this DFA from the set of regular expressions can be automated.

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 Expression

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

- `\x` is the character itself (a few exceptions are `\n`, `\t`, `\r` etc.).
- `.` is any character other than '`\n`'.
- `[xyz]` is `x` | `y` | `z`.

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

IEEE POSIX Regular Expression

- `[abg-pT-Y]` any character `a`, `b`,`g`, $\cdots$, `p`, `T`, $\cdots$, `Y`.
- `[^G-Q]` not any one of `G`, `H`, $\cdots$, `P`, `Q`.
- `r+` one or more `r`'s.
- `r?` one or zero `r`'s.
- `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 $a \in \Sigma$,
$L(r|s) = L(r) \cup L(s)$, $L(rs) = L(r)L(s)$,
$L(r^*) = L(r)^*$, $L(r?) = L(r) \cup \{\varepsilon\}$ etc.

C Identifier

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

Regular definition

We can give name to a regular expression for the convenience of use. The name of a regular expression can be used within the following regular expressions.

Example of a Regular Definition

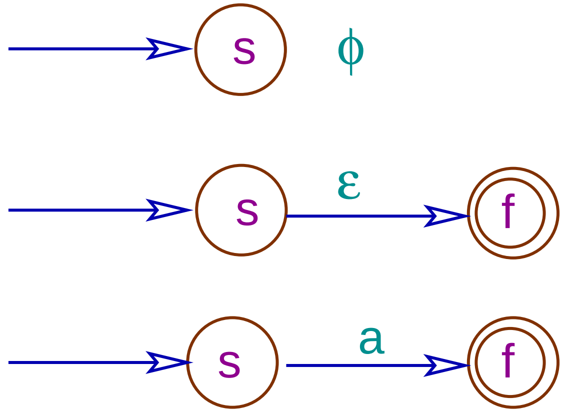
```
sign:    + | - | ε
digit:   [0-9]
digits:  {digit}*
frac:    \.{digits} | ε
frace:   \.{digit}{digits}
expo:    ((E | e){sign}{digit}{digit}?)  | ε
num:     {sign}(({digit}+ {frac} {expo}) |
         ({frace} {expo}))
```

RE to NFA: Thompson's Construction

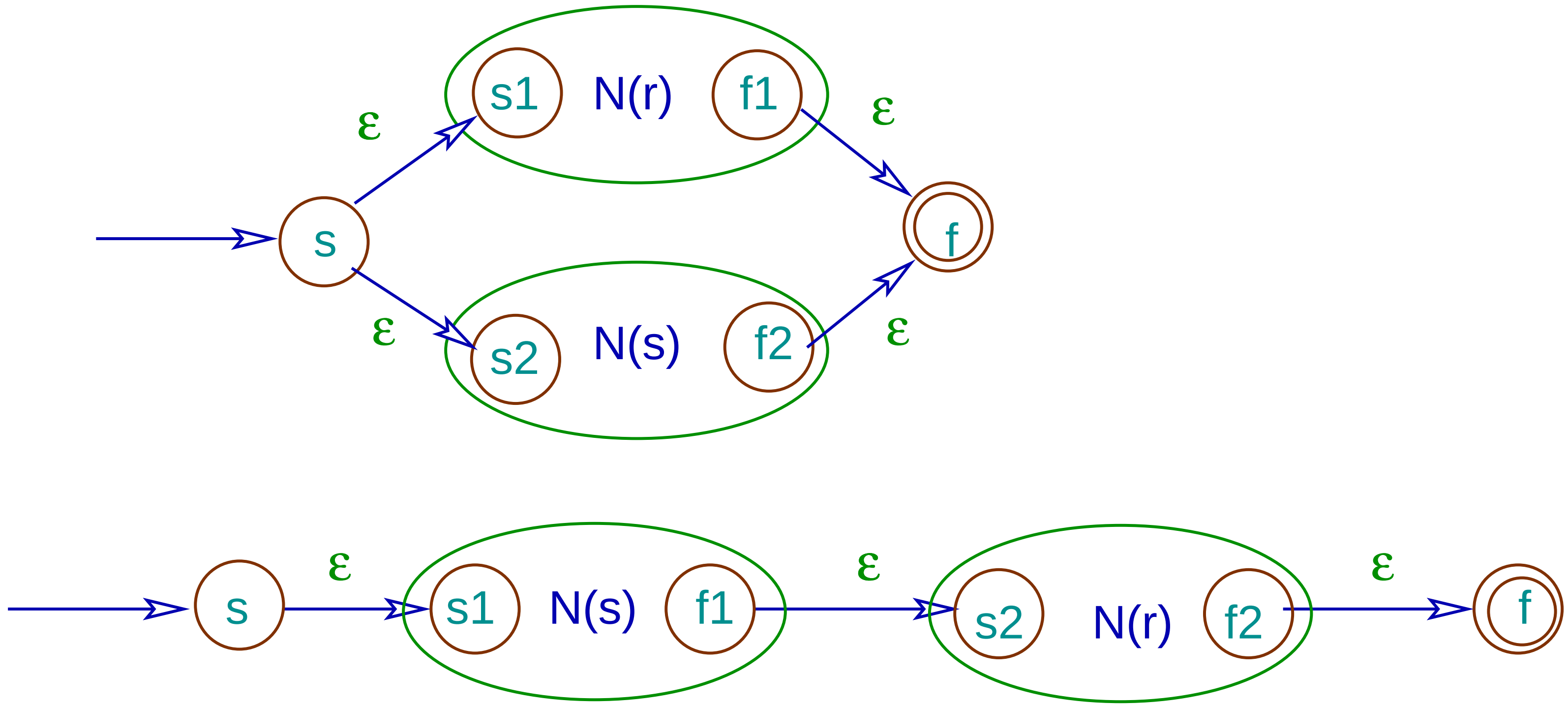
We can mechanically construct a *non-deterministic finite automaton (NFA)* with only one *initial* and only one *final state* from a given *regular expression*. The total number of states of the NFA is linear in the number of symbols of the regular expression[a]

[a] The construction is on the inductive structure of the definition of the regular expression.

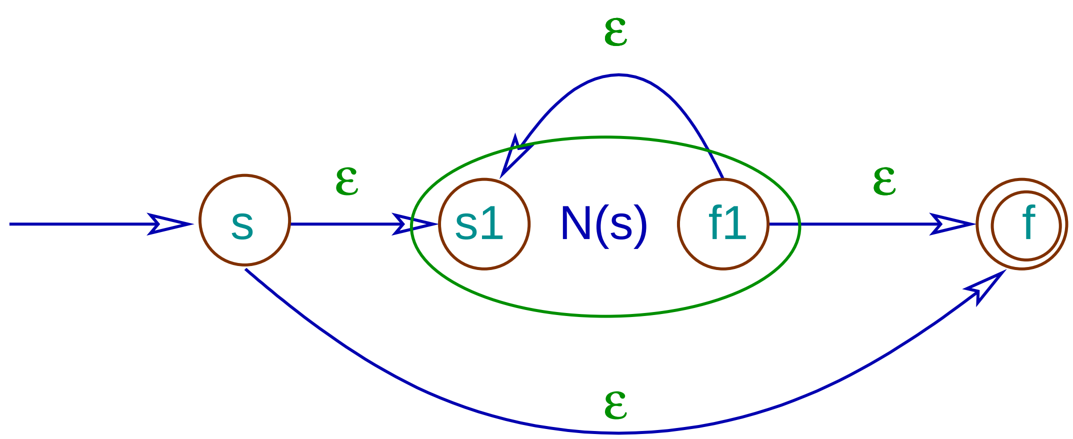
Base Cases



Union and Concatenation



Kleene Closure



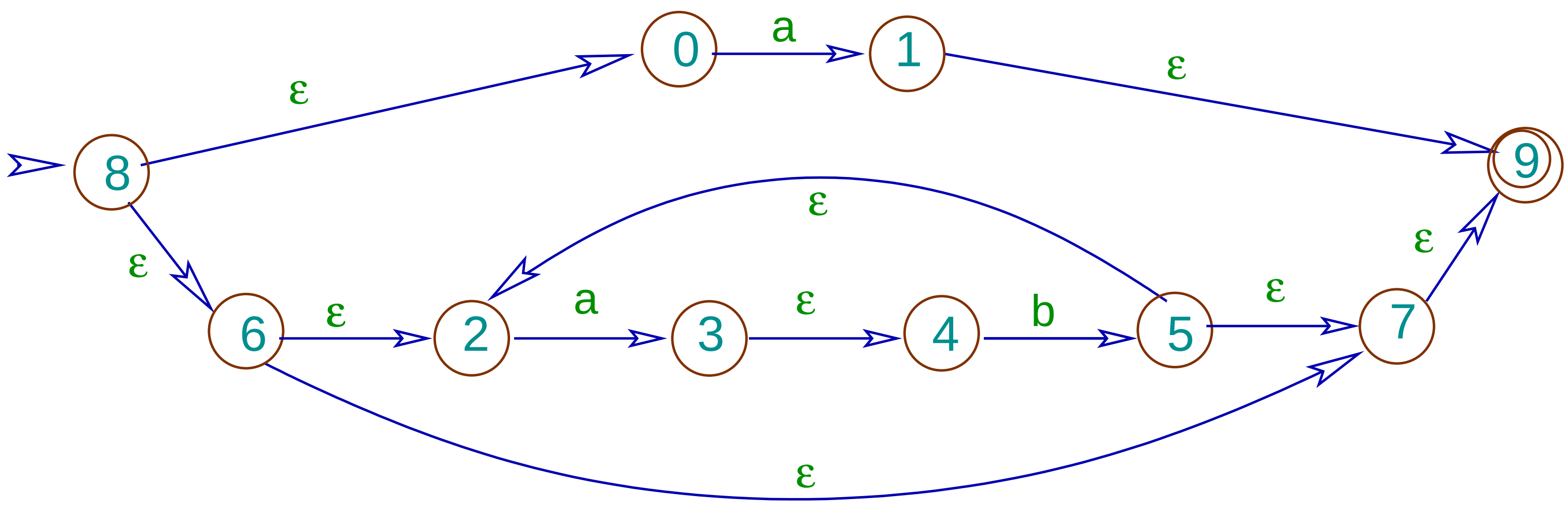
Properties of Thompson's Construction

- $|Q| \leq 2\text{length}(r)$, where Q is the number of states of the NFA and $length(r)$ is the number of alphabet and operator symbols in r.
- Only one initial and one final state. 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 $\epsilon-$transitions.

$a + (ab)*$ - An Example



Context-Free Grammar of RE

The set of regular expression can be specified by a context-free grammar.

$$
\begin{aligned}
E &\rightarrow \emptyset \mid \varepsilon \mid \sigma,\ \forall\sigma \in \Sigma \\
&\rightarrow E.E \mid E + E \mid E* \mid (E)
\end{aligned}
$$

We have put a '.' for concatenation to make it an *operator grammar* and have replaced '$|$' by '+' for clarity[a].

---

[a] This *ambiguous grammar* can be used with proper *precedence* and *associativity* rules.

Syntax Directed Thompson's Construction

Rules of Thompson's construction can be associated with the production rules of the grammar. We assume the following data structures.

Syntax Directed Thompson's Construction

- Global state counter `S` initialized to 0, and the state transition table: `T[][]`.
- With every occurrence of the non-terminal E we associate two attributes `E.ini` and `E.fin` to store the initial and the final states of the NFA, corresponding to the regular expression generated by this occurrence of E.

Some of the Rules: Basis

$E \to \varepsilon$: {T[S][ε] = S+1; E.ini = S;
S = S+1; E.fin = S; S = S+1;}

$E \to a$: {T[S][a] = S+1; E.ini = S;
S = S+1; E.fin = S; S = S+1;}

The second rule depends on the symbol of the alphabet

Concatenation Rules

$E \rightarrow E1.E2$: {`E.ini = S; S = S+1;`

`E.fin = S+1; S = S+1;`

`T[E.ini][`ε`]=E1.ini;`

`T[E1.fin][`ε`]=E2.ini;`

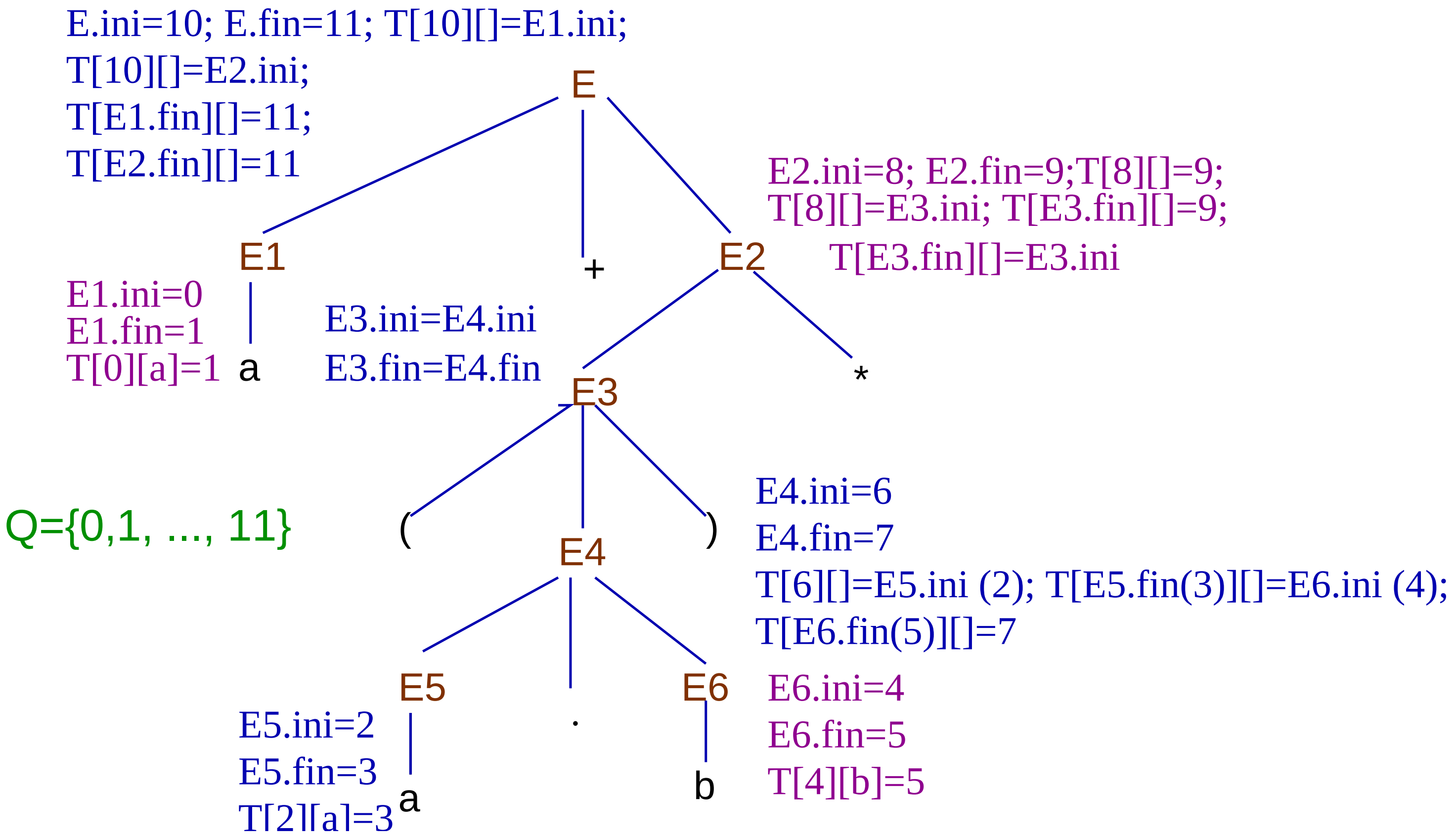
`T[E2.fin][`ε`]=E.fin;`}

Similarly other rules can be derived.

The Final NFA

The states of the final NFA are $\{0, 1, \cdots, S-1\}$. The *initial* state is in `E.ini` and the *final* state is in `E.fin`. The state transitions are in `T[][]`.

$a + (a.b)*$ - An Example



NFA to DFA

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

Algorithm

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

$\textbf{while}(L \neq \emptyset)$

$\quad q = removeElm(L)$

$\quad \textbf{for all } \sigma \in \Sigma$

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

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

$\quad\quad \textbf{if } t \notin Q$

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

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

ε-closure(T)

for all $q \in T$ $push(St, q)$

$\varepsilon T = T$

while$(isEmpty(St) == false)$

  $t = pop(St)$

  for all $u \in \delta(t, \varepsilon)$

    if $u \notin \varepsilon T$

      $\varepsilon T = \varepsilon T \cup \{u\}$

      $push(St, u)$

Note

The time complexity of the ε-closure algorithm for each state is $O(|M|) = O(|N| + |\delta|)$.

Final State of the DFA

The set of final states of the equivalent DFA is $Q_F = \{q \in Q : n_F \in q\}$. It is to be noted that different final states will recognize different tokens. It is also possible that one final state identifies more than one tokens.

Time Complexity of Subset Construction

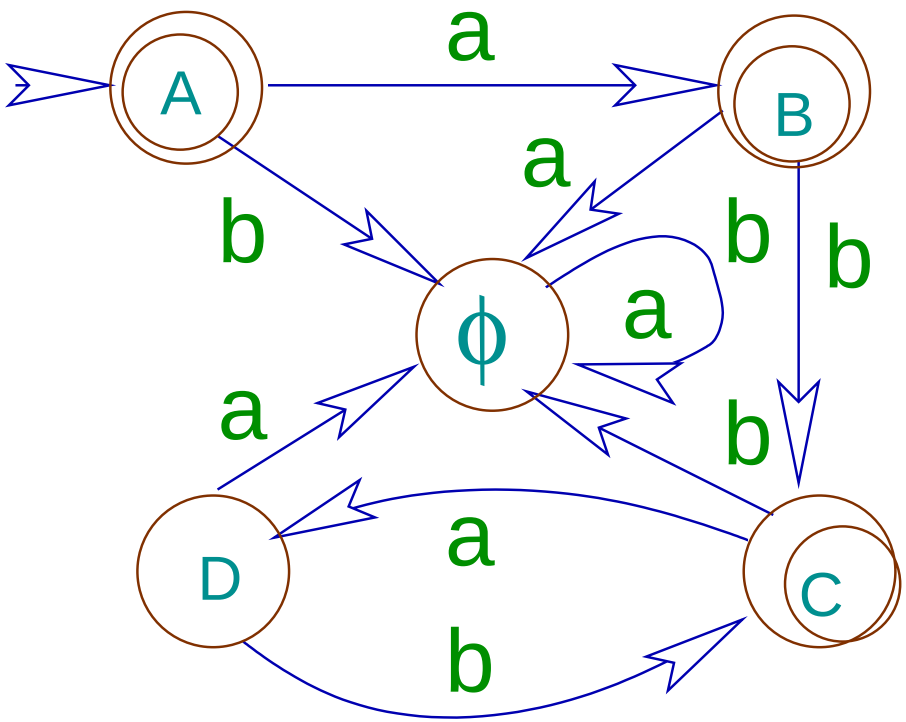
The size of Q is $O(2^{|N|})$ and so the time complexity is also $O(2^{|N|})$, where N is the set of states of the NFA. But this is one time construction.

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

The state transition table of the DFA is

Initial State	*Final 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

It may be of advantage to drop the transitions to $\emptyset$ for designing a scanner. This makes the DFA incompletely specified. Absence of a transition from a final state may identify a token.

DFA State Minimization

The constructed DFA may have set of equivalent states[a] and can be minimized. It is to be noted that the time complexity of a scanner of a DFA with a larger number of states is not different from the scanner of a DFA having a smaller number of states. Their code sizes are different and that may give rise to some difference in their speeds.

[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

We start with two non-equivalent partitions of Q: F and $Q \setminus F$.

If p, q belongs to the same initial partition P 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.

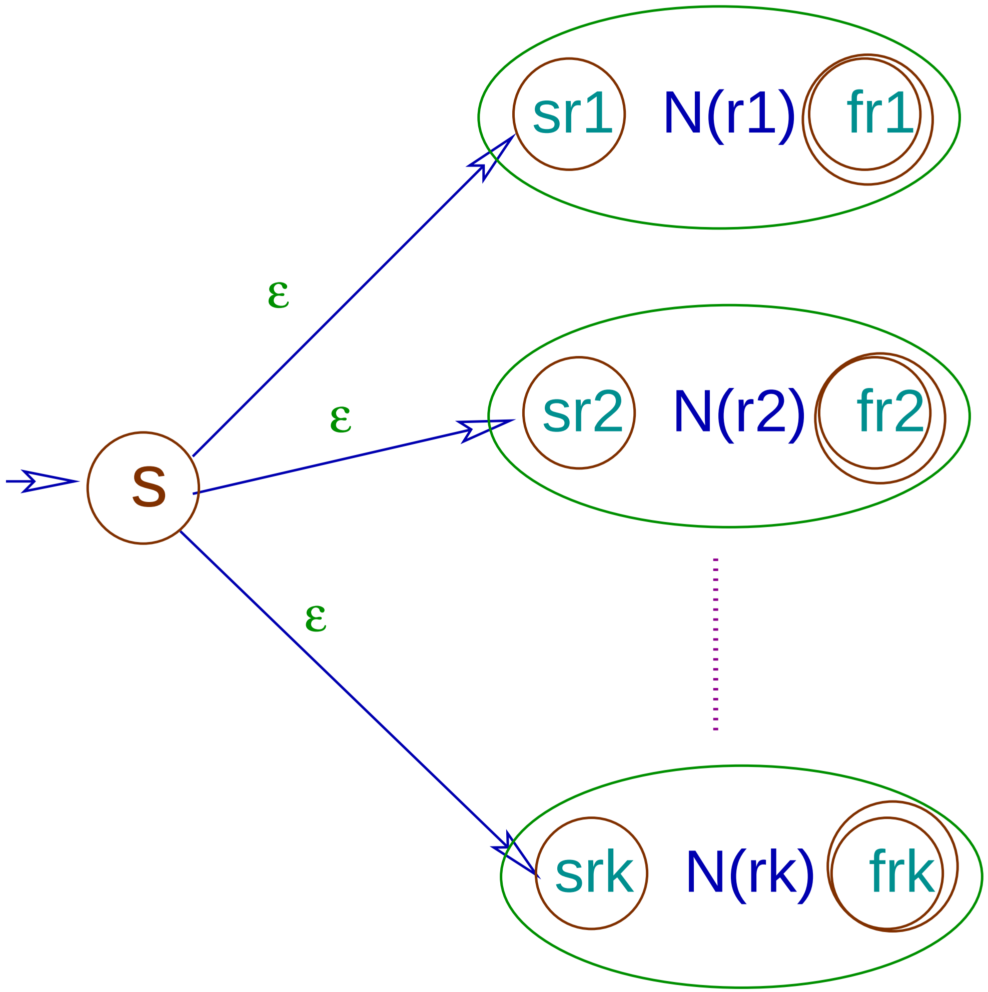
DFA to Scanner

Given a regular expression r we can construct the 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 all *regular expressions* of a language. The regular expression $r = r_1|r_2|\cdots|r_k$ represents objects of all syntactic categories.

DFA to Scanner

Give the 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 token classes.

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

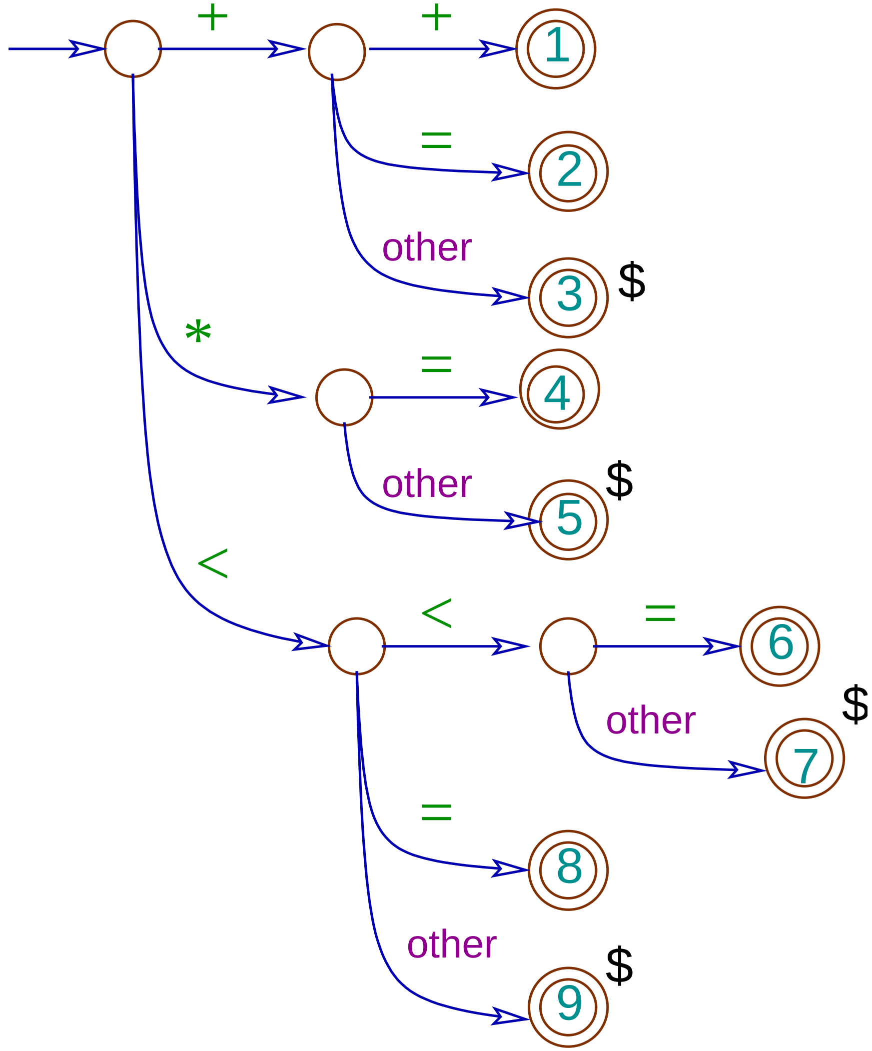
A program is not a single word but a stream of words and the notion of acceptance of a scanner should be different from a simple DFA. The following questions are of importance:

- when does the scanner report an acceptance?
- what does it do if the word (lexeme) matches with more than one regular expressions?

Example

Consider the following subset of C language operators: `+ ++ += * *= < << <= <<=`

State Transition Diagram



Note

At the final state 1 we know that we have "++". But we cannot decide whether it is *pre* or *post* increment operator. Though scanner can take that decision, but it is better to delay it for the parser.

Note

At the final state 3 we know that we have '+'. But we do not know whether it is *binary* or *unary*. Again that decision is defered. Moreover the last consumed symbol is not part of the *lexeme*. It is a look-ahead symbol. We mark such a final state with the number of look-ahead symbols to un-read before going back to the start state. Here we have done that by one $.

Note

- There are situations where there may be more than one look-ahead.

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`.
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. Let there be no transitions for the current state q on the input σ (the machine is incompletely specified). The state q may or may not be final.

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 it is necessary to resolve the conflict. This is often done by specifying priority of expressions e.g. keyword over an identifier.

[a] It is necessary to identify the final state with the regular expression r_i.

q is not Final

It is possible that while consuming symbols the scanner has crossed one or more final states. The decision may be to report the last final state. But then it is necessary to keep track of the final states and the position of the input.

Components of a Scanner

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

Maximum Prefix on NFA

1. Read input and keep track of the sequence of the set of states. Stop when no more transition is possible (*maximum prefix*).
2. Trace back the *last set of states* with a final state.
3. Push back the look-ahead symbols in the buffer and emit appropriate *token* along with attribute value(s).

Note

It is possible that the last set of states has more than one final states corresponding to different *patterns*. Take action corresponding to a pattern with *highest priority*[a].

[a] A pattern specified earlier may have higher priority.

From DFA to Code

Three possible implementations of DFA -

- table driven,
- direct coded,
- hand coded.

Table Driven Scanner

There is a driver code and a set of tables. The driver code essentially has three parts:

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

Initialisation

```
currect state <-- start state
lexeme <-- Nil
push(stack,$)
```

Main Scanner Loop

```
while currect state not = error state
      lexeme <-- lexme + (c = getchar())
      if current state is an accept state
         clear(stack)
      push(stack, current state)
      sym <-- translate[c]
      next state <-- delta(current state, sym)
```

Roll Back Loop

```
while not a final state or stack is not empty
      state <-- pop(stack)
      unget() last symbol of lexeme
```

Token or Error

```
if final state
   return token[state]
   else Error
```

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.

Note

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 labelled 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

A scanner or lexical analyser reads the input character by character. The process will be very inefficient if it sends request to the OS for every character read.

Input Buffer

- OS reads a block of data, supplies the requesting process the required amount, and stored the remaining portion in a buffer called buffer cache. In subsequent calls, the actual IO does not take place as long as the data is available in the buffer.
- Requesting OS for single character is also costly due to context-switching overhead. So the scanner uses its own buffer.

Input Buffer

- A buffer at its end may contain an initial portion of a lexeme. It creates problem in refilling the buffer. So a 2-buffer scheme is used. The buffers are filled alternatively.
- A sentinel-character is placed at the end-of-buffer to avoid two comparisons - character and end-of-buffer.
- We may run out of buffer space for a long

character string or a comment.

Brozozowski's Minimal DFA from NFA

Let N be be an NFA. The reverse of N, the NFA N^R, constructed by introducing (i) a new state as its initial state and making ε-transitions from it to all the final states of N, (ii) making the initial state of N as the final state, and (iii) reversing all transitions of N. We call $N^R = reverse(N)$.

Brozozowski's Minimal DFA from NFA

reachable(N) is the subset of states and transitions of N that are reachable from the start state.
subset(N) gives the DFA M equivalent to N.

Brozozowski's Minimal DFA from NFA

The minimal DFA M equivalent to the given NFA N is
$reachable(subset(reverse(reachable(subset(reverse(N))))))$