Lexical Analyzers

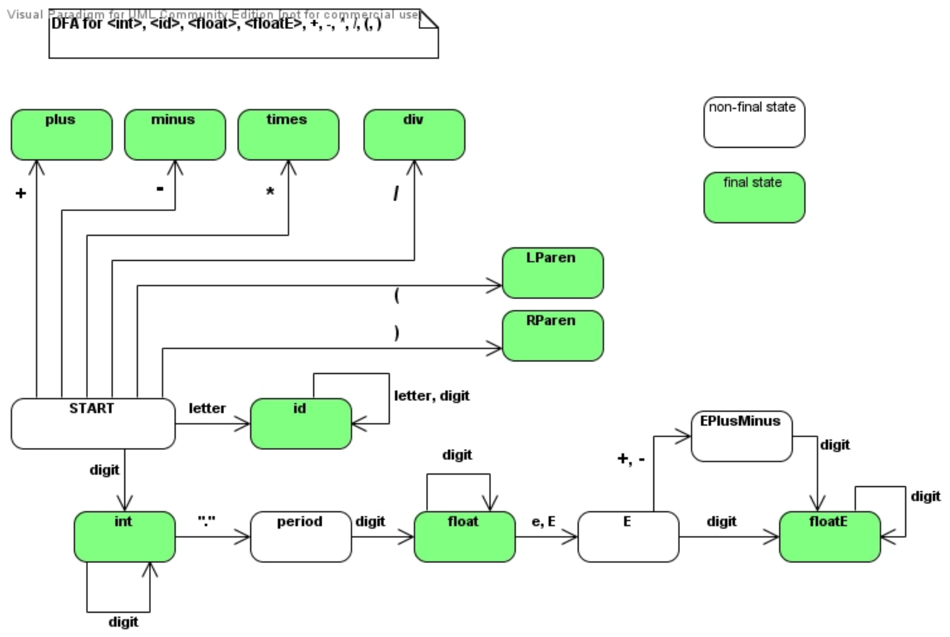
The standard lexical analyzer construction method is based on the fact that tokens are definable by a limited form of EBNF grammars that can be accepted by deterministic finite automata (DFAs). This limited form of EBNF is equivalent to regular expressions. As described in that note, tokens are definable without use of recursive production rules. Given a token set specified by an EBNF, we construct a DFA to accept it and feed its states and transition function to a DFA driver program. This way, construction of lexical analyzers is systematically reduced to construction of DFAs.

A DFA consists of:

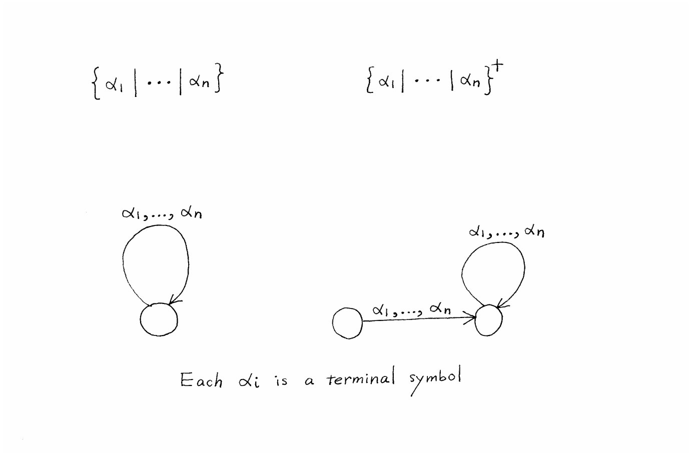
- a finite set I of input symbols,
- a finite set S of states,
- a special state $s \in S$, called the *start state*,
- a subset $F \subseteq S$, called the *final states*,
- a state-transition function δ: $S \times I \to S$. The function δ may be partial, i.e., may be undefined on some (state, input symbol) pairs.

DFAs are conveniently specified by state-transition diagrams. For each $\delta(s, c) = s'$, we draw a directed arrow from state s to state s' labeled with input symbol c. In application to lexical analyzers, the input symbols are single characters in source program texts. The following is an example DFA to accept the tokens <int>, <id>, <float>, <floatE>, +, −, *, /, (,).

<plus> → +
<minus> → −
<times> → *
<div> → /
<LParen> → "("
<RParen> → ")"
<int> → { <digit> }$^{+}$
<id> → <letter> { <letter> | <digit> }
<float> → { <digit> }$^{+}$ "." { <digit> }$^{+}$
<floatE> → <float> (E|e) [+|−] { <digit> }$^{+}$



The iterative forms { α_1 | ... | α_n } and { α_1 | ... | α_n }$^+$, where each α_i is a terminal, are implemented respectively by the following state-transition patterns:



$\{\alpha_1 | \cdots | \alpha_n\}$
$\{\alpha_1 | \cdots | \alpha_n\}^+$
$\alpha_1, \ldots, \alpha_n$
$\alpha_1, \ldots, \alpha_n$
$\alpha_1, \ldots, \alpha_n$
Each α_i is a terminal symbol

Here is an example of more complex notation for floating-point numbers: Java floating-point numbers.

In coding DFAs as programs, it is useful to cleanly separate the driver, which remains invariant for all DFAs, from states and state-transition functions, which will change from DFA to DFA and will be superimposed on the driver. (The driver is an "interpreter" for DFAs while states and state-transition functions are "programs" of DFAs.) The following is an example algorithm for a DFA driver to extract the next token on the input stream.

```
//   t holds the string extracted so far
//   c is the current character on the input stream

int driver()
{
    t = empty string;
    state = START;
    if ( c is whitespace char )
        advance c to the next non-whitespace char;
    if ( EndOfStream ) return -1;

    while ( not EndOfStream )
    {
        nextState = δ(state, c);
        if ( nextState == UNDEFINED ) // DFA will halt.
        {
            if ( state is a final state )
                return 1;  // valid token extracted
            else // c is an unexpected char
            {
                t = t+c;
                c = the next char on the input stream;
                return 0; // invalid token found
            }
        }
        else // DFA will go on.
        {
            state = nextState;
            t = t+c;
            c = the next char on the input stream;
        }
    }

    // EndOfStream is reached while a token is being extracted.

    if ( state is a final state )
        return 1; // valid token extracted
    else
        return 0; // invalid token found
}
```

The task of DFAs when used for lexical analyzers is to extract the tokens in the source program one by one, rather than to decide if a given string in its entirety is a valid token. Sequences of tokens may appear without any whitespaces in between, like "12.34e−5−3.15E+3+98.3". The DFA drivers for lexical analyzers, including the above one, implement the *longest token rule*: always extract the longest possible tokens by going as far as possible in state transitions until no further transition is possible. If the DFA then ends in a final state, a valid token has been extracted; if the DFA ends in a non-final state, an invalid token has been found. The driver should not halt at an arbitrary final state. This is because a valid token may contain a shorter valid token as its initial segment. For example, any non-empty initial segments of <int> and <id> are valid <int> and <id> respectively, and any <floatE> with an exponentiation part contains a valid <float> without the exponentiation part. Of all these the longest ones are the intended tokens.

The parser calls driver( ) whenever it needs the next token in the source program for construction of a parse tree.

State-transition functions can be implemented by nested "2-dimensional conditionals" or by a 2-dimensional array – see example Java programs.

There are lexical analyzer generators that automate generation of lexical analyzer programs from EBNF grammars defining tokens. The LEX software is one of the most popular. These systems take EBNF token grammars as input, convert them to non-deterministic finite automata accepting the tokens, then convert them to equivalent DFAs, and output programs implementing the DFAs.