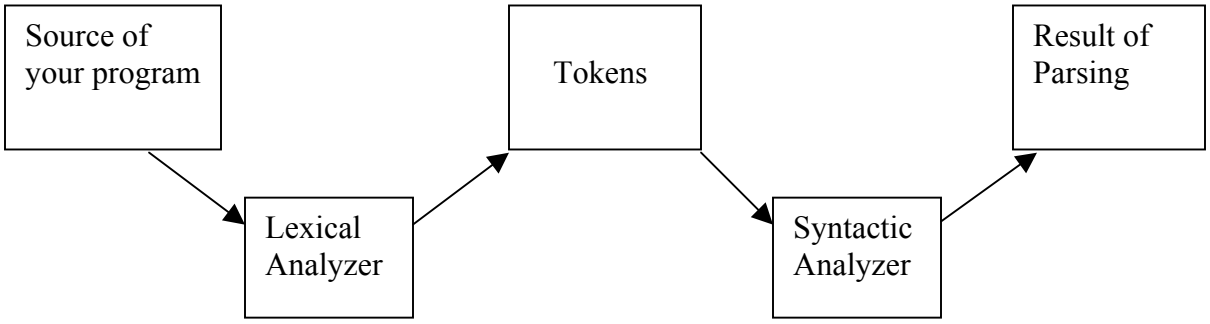
Chapter 4. Lexical and Syntax Analysis

- Building a lexical analyzer
- Syntax analyzer (Parsing problem)
  - Top-Down Parsing
  - Bottom-Up Parsing



Syntax analyzer (Parsing) is the process of analyzing an input sequence in order to determine its grammatical structure with respect to a given BNF grammar.

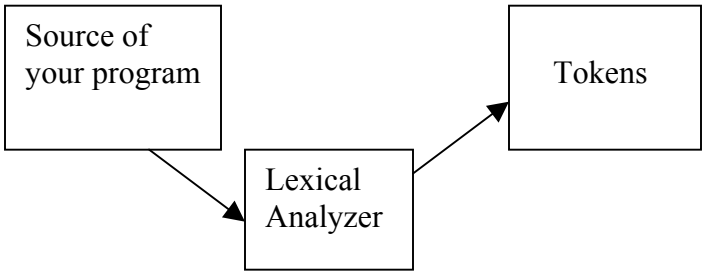
Reasons to Separate Lexical and Syntax Analysis

- *Simplicity* – Lexical analysis are less complex than syntax analysis, less complex approaches can be used for lexical analysis; separating them simplifies the parser.
- *Efficiency* – Lexical analysis requires a significant portion of total compilation time, separation allows optimization of the lexical analyzer and permits parallel development.

Building a lexical analyzer:

- Write a formal description of the tokens and use a software tool that automatically constructs lexical analyzers given such a description, for instance lex in Unix.
- Design a state diagram that describes the tokens and write a program that implements the state diagram

position := initial + rate * 60



Lexical analyzer is essentially pattern matching - each token is described by some pattern for its lexemes.

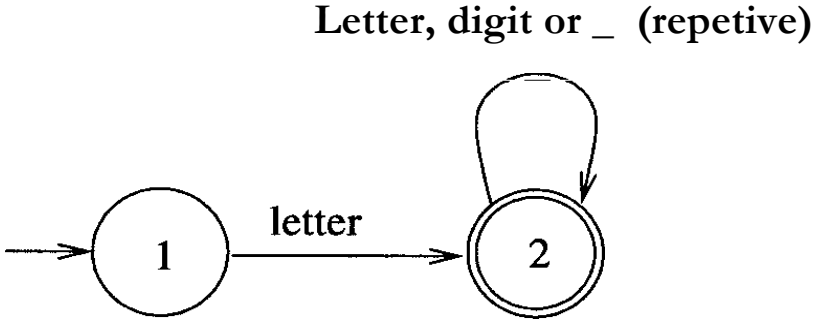
- The pattern for an identifier (token IDENT) is a letter followed by a sequence of letters, digits and underscores. Lexemes include: i3, foo, foo_7, ...

- The pattern for a floating point numeric literal (token INT_LIT) is an optional sign, followed by a sequence of digits, followed by a decimal point, followed by a sequence of digits. Lexemes include: -7.3, 3.44, ...

To generate a pattern matcher from regular expressions

- convert regular expressions to state transition diagrams
- generate pattern matching code from the diagrams

The state transition diagram for token **identifier:**



Each circle is a **state**
The arrows between the states are **transitions**.
Each transition is labeled with the character that causes the transition to occur.

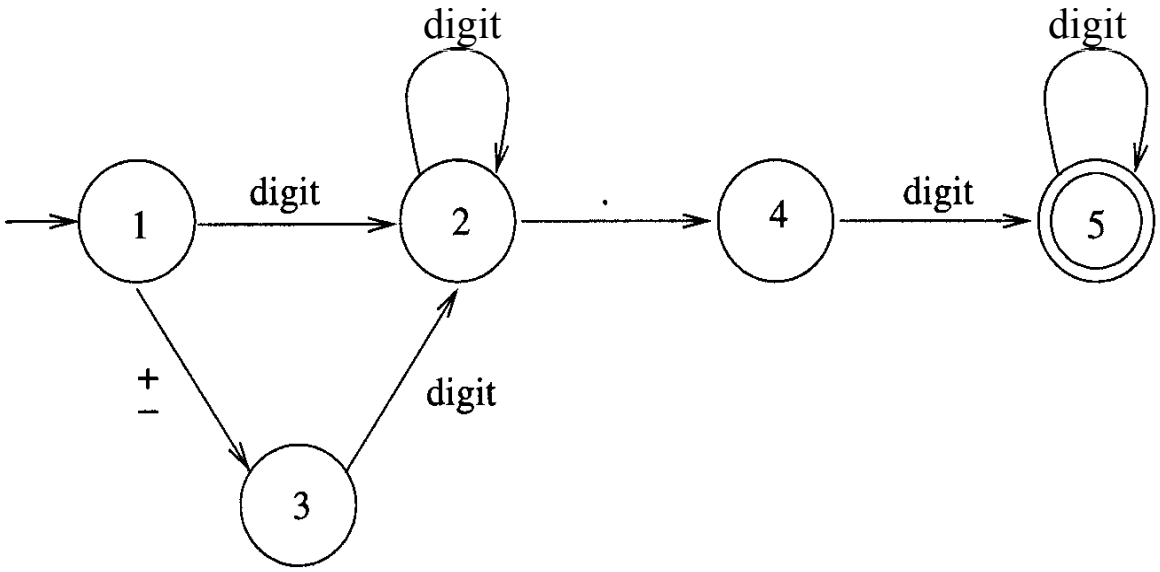
1 is the **start state** where pattern matching begins.

Any state consisting of 2 concentric circles is called an **accepting state**. When an accepting state is reached, the pattern has been matched.

Hence, state 1 moves to state 2 if the first input character is a letter. Any other input is an error.

The state transition diagram for a token Num

Num-> (+|-){digit}.{digit}



This form of state transition diagram is easily implemented using a big switch statement.

For example, the pattern matcher corresponding to the num above is:

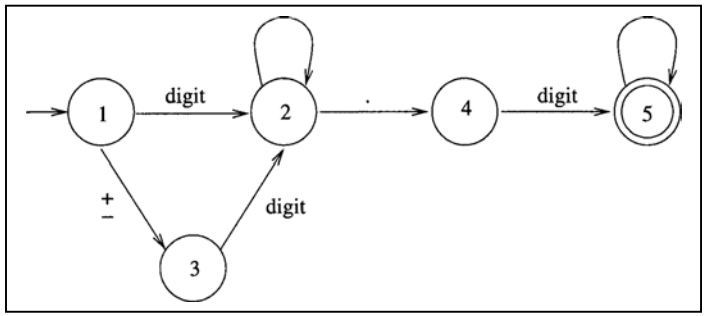
```
#include <ctype.h> // to use isdigit() #include <stdio.h>

int state = 1; // the start state int= 1;

char c; // the current input character
c = getchar(); // read next character of input

while (!done) {

switch (state) {
  case 1: if (isdigit( c )){
       state = 2;
       c = getchar();
       } else if (c == '+' || c == '-') {
       state = 3;
       c=getchar();
       } else error(); break;
  case 2: if (isdigit(c)) {
        // stay in same state
       c = getchar();
      }
```



```
       else if (c = '.') {
       state= 4;
       c=getchar(); } else error(); break;
 case 3: if(isdigit(c)) { state = 2;
        c = getchar(); } else error();
       break;
 case 4: if(isdigit(c)) { state = 5;
        c = getchar(); } else error();
       break;
 case 5: if (isdigit(c)) {
// stay in same state
       c = getchar();
    } else done = 1; // matched the longest string of input
              //characters that match the pattern
       break;
       }//switch
}//loop
```