CST8152 - Compilers
Article #8
Finite Automata

DEFINITION

A **Finite Automaton (FA)** or **Finite State Machine (FSM)** is consists of the following components (parts):

1. A finite set of states, one of which is designated as the initial state, called the start state, and some states of which are designated as **final** states, called also **halting** states, **terminal** states or **accepting** states.
2. An alphabet Σ of possible input symbols (including ε), from which are formed the strings (words) of the language recognized by the FA.
3. A finite set of transition that determine for each state and for each symbol of the input alphabet which state to go next. In other words, the FA must have a transition function *move* or *next_state* that maps each state-symbol pair to a single state or set of states. If the function maps each state-symbol pair to a set containing only one state, the FA is defined as **Deterministic Finite Automaton (DFA)**. If the function maps each state-symbol pair to a set containing more than one state, the FA is defined as **Nondeterministic Finite Automaton (NFA).**
   NFA can be easily constructed from regular expressions and than converted to DFA. Each and every NFA can be converted to DFA.
   The set of transitions can be represented in different ways but the most common and the easiest for implementation in computers is the **transition table** representation. The transition table has one row for each state and one column for each input symbol. The entry for row ***i*** and column ***s*** is the state (for DFA) that can be reached by transition from state ***i***

on input symbol ***s***. The entry for row ***i*** and column ***s*** is the states (more than one state for NFA) that can be reached by transition from state ***i*** on input symbol ***s***. The transition table representation has the advantage that it can be build directly from the corresponding transition diagram and it provides very fast access; its disadvantage is that it takes a lot of space. A compromising solution is to compress the table to save space and make the access a bit slower.

Since the FA accept or recognize strings of a language they are also called **finite acceptors** or **language recognizers**.

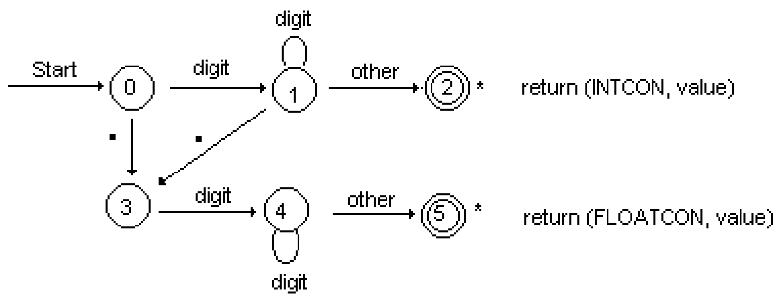
- A *recognizer* is able to take a string 'x' and determine whether the string belong to the language defined by the FA.
- A regular expression can be turned into a recognizer by making a transition diagram or transition table which represents a finite automata
- An automata can be considered either deterministic or non-deterministic
- Both types of automata can be used in developing a lexical analyzer
- Algorithms involving automata exist which can help improve the design of scanners

Deterministic Finite Automata (NFA)

- A DFA is a special case of an NFA, except for the following restrictions:
  - No empty ε-transitions exist within it
  - Every transition from a state is unique
- An NFA can always be converted into a DFA; however, the number of states will increase
- A DFA will often be faster than an NFA, because decisions regarding transitions don't have to be made

Implementing Transition Diagrams - Manually

- A transition diagram can be implemented manually, with states being represented by variables, and transitions being decided on by if statements.
- Example, consider a procedure which recognizes floating point and integer constants. Transitions are indicated by arrows with a input symbol label, states are indicated by circles with unique labels and final states are indicated by two concentric circles with unique labels. * indicates that the input must be retracted – the last symbol read must be returned back to the input stream of symbols.



- Cases would be developed which handle all states, and all transitions from one state to another

Example of Transition Diagram Implementation:

- The following (partial) code implements the previous transition diagram:

```
token next_token()
{
    while (1)
    {
        switch(state)
            case 0:
                c=nextchar();
                if (c == ' ')   // Skip blanks
                {
                    lexeme_start++;
                    state = 0;
                }
                else if (isdigit(c)) state = 1;
                else if (c == '.') state = 3;
                break;
            case 1:
                c = nextchar();
                if (isdigit(c)) state = 1;
                else if (digit == '.') state=3;
                else state= 2;
                break;
            case 2:
                retract(1); // Backtrack in input
                store_the_lexeme();
                return(INTCON);

            // Cases 3, 4, and 5 would
            // also be implemented here...
```

- Note that since functions can only return one value (and in this case, it is returning just the token), we are storing the actual lexeme somewhere else in memory by calling a function

Implementing Transition Tables - Arrays

- A two dimensional array can be used to implement a transition table.
- A table consists of rows representing states and columns representing character classes.
- Examples of character classes include:
  - digits
  - letters
  - . (or any punctuation)
- The classes for the previous example (recognizing integer or floating point constants), and an assigned value would be as follows:

Class	Value (column)
digit	0
.	1
other	2

- A table representing the transition diagram is as follows:

```
#define ES 6
#define IS -1

int Table[6][3] = {
/* State 0 */ { 1, 3, ES},
/* State 1 */ { 1, 3, 2},
/* State 2 */ { IS, IS, IS},
...
/* State ES */ { IS, IS, IS};
```

- Every row in our transition table represents one state, while every column represents a character class.
- Example: When in state 1, receiving a character of class 0 (ie. a digit) forces the machine to remain in state 1, a character in class 1 (a '.') forces a transition to state 3, and a character in any other class forces a transition to state 2.
- ES (Error State) and IS (Illegal State) are pre-defined constants indicating either an invalid transition (IS), or a final state (ES) reporting a semantic (spelling) error.
- Note that the rows and the columns can be swapped in the table

Implementation of Table-driven Transitions

Given the state table developed previously, the following functions can be used to find the transition from one state to another:

```
int next_ntate(int currentState, char ch)
{
     int column = get_column(ch)
     return Table[currentState][column];'
}

// Given a character, find what class it
// belongs to, to be used for table indexing

int get_column(char ch)
{
     if isdigit(ch) return(0);
     else if (ch = '.') return 1;
     else return 2;
}
```

The NFA implementation is a loop in which a character is taken from the input, the *next_state()* function is called and the type of the returned state is tested. If the state is not-accepting the loop continues. If the state is accepting the loop is terminated and the recognized string (lexeme) is processed by the corresponding accepting function. Usually an array of pointers to functions is used to call an accepting function using the accepting state number as an index.