

# Notes on Lexical Analysis

By Dan Beatty

Reference: “Compilers: Principles, Techniques and Tools”  
by Alfred V Aho, Ravi Seth, and Jeffrey D. Ullman

# Lexical Analysis

- “A simple way to build a lexical analyzer is to construct a diagram that illustrates the structure of the tokens of the source language, and then to hand translate the diagram into a program for finding tokens.”
- “The underlying problem is the specification and design of the programs that execute actions triggered by patterns in strings.”
- Automated lexical analysis
  - Jarvis’ lexical analysis generator for finding imperfections in printed circuit boards.
  - Advantage “is that it can utilize the best known pattern matching algorithms and thereby create efficient lexical analyzers for people who are not experts in pattern matching techniques.”

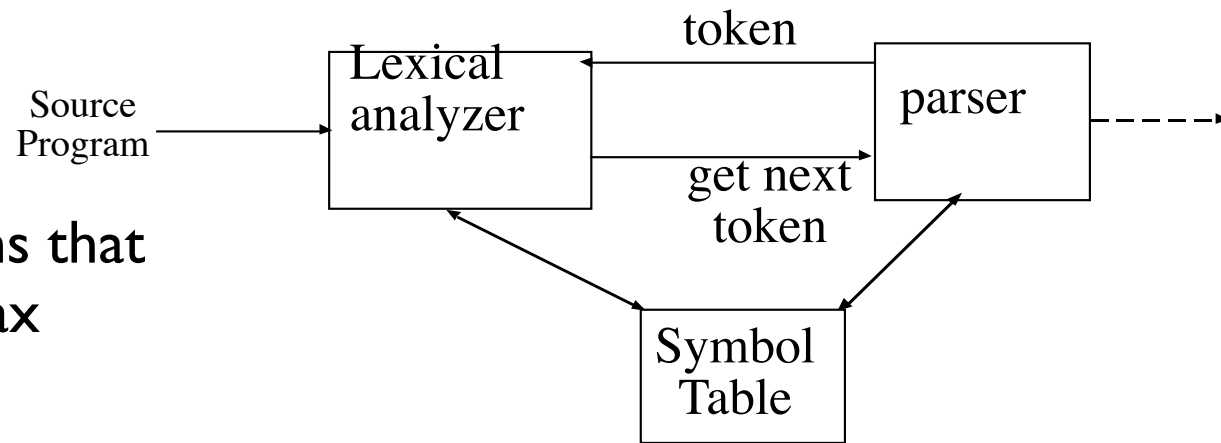
# Reasons for Lexical Analysis

- Simpler Design: Separation of lexical analysis from syntax analysis
- Improved compiler efficiency with specialized buffers
- Enhanced portability

# Role of the first phase of the Compiler

- Tasks:

- Read input characters
- Output a series of tokens that the parser uses for syntax analysis
- Strips comments and white space(s)
- Keeps track of compiler errors with line number references.
- Preprocessor functions



# Tokens, Patterns, and Lexemes

- “A lexeme is a sequence of characters in the source program that is matched by the pattern for a token”
- A pattern associated with a token is a set of strings described by a rule describing the set of lexemes that can represent a particular token in source programs.
- Tokens are treated symbols in the grammar for the source language. Examples common in programming languages are “keywords, operators, identifiers, constants, literal string, and punctuation symbols.” Tokens are often referred to with integer references.
- “Lexemes matched by the pattern for the token represent strings of characters in the source program that can be treated together as a lexical unit.”

# Lexical Errors

- Few errors are discernible at the lexical level alone.
- Lexical analyzers have a very localized view of a source program
- Errors discovered by Lexical Analyzer have the following resolutions:
  - Panic mode
  - delete an extraneous character
  - inserting a missing character
  - replacing an incorrect character by a correct character
  - transposing two adjacent characters.
  - Trial and error prefix adjustment
  - Minimum correction computation

# Input Buffering

- How efficient is buffering the input for a lexical analyzer and what are the methods?

# Regular Expressions to an NFA



# A Review of Finite Automata

“ A *recognizer* for a language is a program that takes as input a string  $x$  and answers ‘yes’ if  $x$  is a sentence of the language and ‘no’ otherwise.” Non-deterministic finite automata has more than one transition out of a state on the same symbol. A deterministic finite automata has only one state transition per symbol. Machine accepts a string if and only if it enters a final state (element of  $F$ ) and there are no other items in its input.

## References

- [1] Jeffrey D. Ulman, Ravi Sethi, Alfred V. Aho *Compilers: Principles, Techniques and Tools* copyright 1986 by Bell Telephone Laboratories, Incorporated page 113

# Non-deterministic Finite Automata

A non-deterministic finite automata (NFA) is a mathematical model that consists of

1. a set of states  $S$
2. a set of input symbol  $\Sigma$  ( the input symbol alphabet)
3. a transition function move that maps state-symbol pairs to sets of states
4. a state  $s_0$  that is distinguished as the start (or initial) state
5. a set of states  $F$  distinguished as accepting (or final) states

“An NFA accepts an input string  $x$  if and only if there is some path in the transition graph from the start state to some accepting state, such that the edge labels along this path spell out  $x$ . ”

## References

- [1] Jeffrey D. Ulman, Ravi Sethi, Alfred V. Aho *Compilers: Principles, Techniques and Tools* copyright 1986 by Bell Telephone Laboratories, Incorporated page 114-115

NFA is typically represent pictorially by a transition graph. A transition graph is a graph where the states are the nodes and labeled edges represent the transition functions. NFA may include edges represented by  $\epsilon$ .

A NFA may also be represented by a transition table. A transition table consists of:

- a row for each state
- a column for each symbol
- An entry in each cell representing the states that can be reached by the row's state and column's input symbol

## References

- [1] Jeffrey D. Ulman, Ravi Sethi, Alfred V. Aho *Compilers: Principles, Techniques and Tools* copywrite 1986 by Bell Telephone Laboratories, Incorporated page 114

# Deterministic Finite Automata

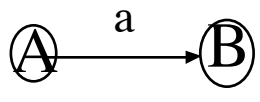
A deterministic finite automaton (DFA) is a special case of NFA in which

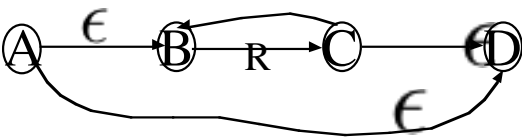
1. no state has an  $\epsilon$ -transition
2. for each state  $s$  and input symbol  $a$ , there is at most one edge labeled  $a$  leaving  $s$

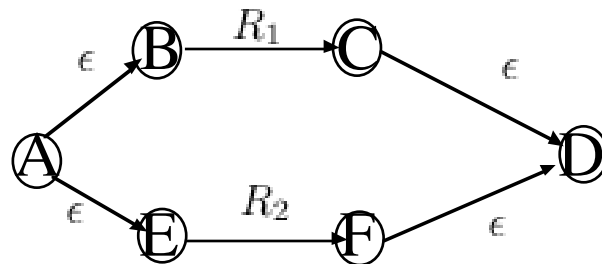
## References

- [1] Jeffrey D. Ulman, Ravi Sethi, Alfred V. Aho *Compilers: Principles, Techniques and Tools* copyright 1986 by Bell Telephone Laboratories, Incorporated page 115-116

# Regular Expressions $\rightarrow$ NFA

1. Symbol  $a$  : 

2.  $R^*$  : 

3.  $R_1 | R_2$  : 

4.  $R_1 \dot{R}_2$  : 

5.  $R^+$  :  $R \dot{R}^*$

**Example: Minimal DFA**  $NFA \rightarrow DFA$   $\epsilon$  closure / composite states

“The DFA uses its state to keep track of all possible states the NFA can be in after reading each input symbol.”

**Algorithm** Subset construction: Constructing a DFA from an NFA

*Input* An NFA  $N$

*Output:* A DFA  $D$  accepting the same language

*Goal:* “Each DFA state is a set of NFA states and we construct  $D$ -tran so that  $D$  will simulate in parallel all possible moves  $N$  can make on a given input string.”

*Method.* Construct a transition table  $D$ -tran for  $D$ . Apply  $\epsilon$  closure and move methods,

Table 1: default

$\epsilon$ -closure( $s$ )	Set of NFA states reachable from NFA state $s$ on $\epsilon$ -transitions alone
$\epsilon$ -closure( $T$ )	Set of NFA states reachable from some NFA state $s \in T$ on $\epsilon$ -transitions alone
$move(T, a)$	Set of NFA states to which there is a transition on input symbol $a$ from some NFA state $s \in T$

# Notes on subset construction

- $\epsilon$ -closure ( $s_0$ ) shows a pseudo-state composed of the states  $s \in T$  reachable from  $s_0$ . For symbol  $a$ , these states include  $\epsilon closure(move(s_0, a))$
- Each state of  $D$  corresponds to a set of NFA states that  $N$  could be after reading a sequence of input symbols including all possible  $\epsilon$ -transitions before or after the symbols are read.
- Starting state of  $D$  is  $\epsilon - closure(s_0)$ .
- An accepting state in  $D$  is defined as the state is a set of NFA states containing at least one accepting state of  $N$ .
- A simple algorithm to compute  $\epsilon$  - closure ( $T$ ) uses a stack to hold states whose edges have not been checked for  $\epsilon$  labeled transitions.

# Design of a Lexical Analyzer Generator

- How to construct a lexical analyzer program
- Lexical analyzer specification:
  - $p_1 = \{action_1\}$
  - $p_2 = \{action_2\}$
  - ...
  - $p_n = \{action_n\}$

Such that “each pattern  $p_i$  is a regular expression and each  $action_i$  is a program fragment that is to be executed whenever a lexeme matched by  $p_i$ ” [1]

## References

- [1] Jeffrey D. Ulman, Ravi Sethi, Alfred V. Aho *Compilers: Principles, Techniques and Tools* copyright 1986 by Bell Telephone Laboratories, Incorporated



# Problem Statement

“Our problem is to construct a recognizer that looks for lexemes in the input buffer.”

Three cases exist for this statement:

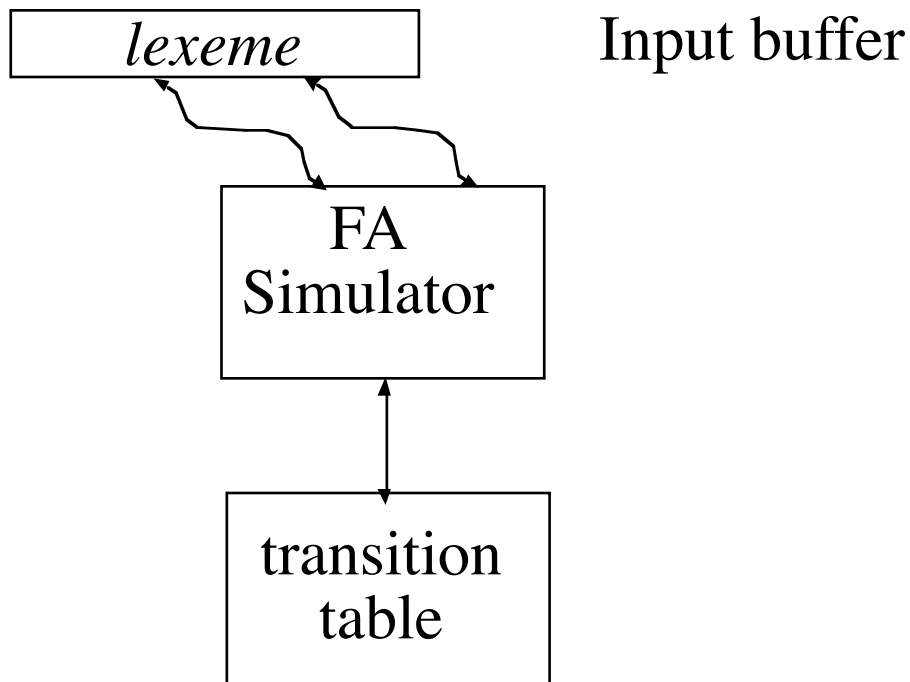
1. No pattern matches (report the error)
2. One pattern matches (chose the matching lexeme)
3. More than one pattern matches
  - Chose the longest lexeme matched
  - Chose the first listed matching pattern lexeme.

A finite automation is a natural model to build a lexical analyzer around.

## References

- [1] Jeffrey D. Ulman, Ravi Sethi, Alfred V. Aho *Compilers: Principles, Techniques and Tools* copyright 1986 by Bell Telephone Laboratories, Incorporated

# A finite automation example



## Hints on the lexical analyzer: A tokenizer

1. Read each line of input into a buffer. Assume 80 characters per line. This is to setup a string for each line. March through the string for lexical analysis.
2. Print each string when it is read to a listing file (text file). The syntax analyzer is going to call the lexical analyzer for the next token. Error handling is part of this scheme. The errors can be written for the line printed out with the string. (has all error messages).
3. Symbol table - Preload symbol table with all reserved words and symbols. In our example, the first 36 entries will be reserved words and symbols. Starting at the 37th element, all of the identifiers and constants and non-reserved words. The easiest data structure should be chosen for the symbol table. Reserve the hard work for the compiler itself. There should be a few items in this record. Also fix the identifier type to token type 37 and constants to token type 38. The row index and token type should be returned with the record.
  - Name String
  - location
  - Token Type
  - Index of token
  - constants
  - integer
4. Lexical analyzer returns token type row in symbol table.
5. Constants and Id 's you first search (these constants must be fixed types). The purpose is to see if the string already exists in the symbol table.
6. A match of a symbol, simply returns its location in the symbol table and record in the symbol table.
7. Operators, and special symbols are delimiters. How the delimiter is identified is immaterial. What matters is that when one of these delimiters is encountered, the token acquisition stops. Special cases exist for multiple character special symbols, such as `:=`. Stopping on space implies that a delimiter on beginning the reading of next token - throw out the spaces. End of line is a delimiter.
8. Token type and symbol table location should be made integer types, and global variables, contrary to SE's idea. The symbol table may be useful as a global variable.
9. One can include the length of identifiers. Another can be identifier types should be alphabetical characters only.

At code generation time is where the location item is used. The object code will stick the data at the bottom of object code memory. In that data section, each symbol will live in the data section. The location in the symbol identifies the location in the data section for that symbol. This location applies to constants as well.

Ways of implementing lexical analyzer for NFA, NDFA and DFA.

1. Pattern matching based on NFA's
2. DFA for Lexical Analysis
3. "Tokenizers"

# Pattern Matching NFA Includes

- A transition table containing a non-deterministic automaton  $N$  for the composite pattern of  $p_1|p_2|\dots|p_n$ 
  - Create an NFA for each pattern  $p_i$
  - Add a new start state  $s_0$
  - Link  $s_0$  to the start state of each  $N(p_i)$
- The combined NFA must recognize the longest prefix of the input that is matched by a pattern.
- Method
  1. Add an accepting state to the current set of states
  2. Record the current input position and the pattern  $p_i$  corresponding to the accepting state
  3. Continue making transitions until termination is reached, and mark as accepting state positions.
  4. On termination, move the forward pointer to the last match that occurred.
- If no pattern matches, then an error has occurred.

## References

- [1] Jeffrey D. Ulman, Ravi Sethi, Alfred V. Aho *Compilers: Principles, Techniques and Tools* copyright 1986 by Bell Telephone Laboratories, Incorporated page 130-131

## Pattern Matching DFA Includes

1. Convert the NFA version to a DFA form. Note that there may be several accepting states.
2. Follow the NFA scheme of state transitions until a no-next state has been reached for the current input symbol.
3. Return the last input position at which the DFA entered an accepting state

## References

- [1] Jeffrey D. Ulman, Ravi Sethi, Alfred V. Aho *Compilers: Principles, Techniques and Tools* copyright 1986 by Bell Telephone Laboratories, Incorporated page 132-133

# Example: Lexical Analysis Through Semantics

Note that whole point of the lexical analyzer in this case is to identify tokens. The calling program reads in a buffer of characters (lines of 80 characters or less). This string is passed to the lexical analyzer. As a debugging tool, each string will be printed when processed. Also comments as to success or failure shall also be printed.

It has been alluded that the syntax analyzer will actually be doing the calling, trying to acquire tokens. One feature that appears to be necessary is a common symbol table. Whether this symbol table is global, is simply referenced by both is something to be tested. There is a suggestion that makes sense. Pre-load the symbol table with all reserved words and symbols. All reserved words have a token type identical to its index in the symbol table. The suggested record structure is as follows:

- Name string
- Location
- Token Type
- Index of Token
- Constants
- Integer

# Special Suggestions

1. Operators, special symbols, and spaces are delimiters. How the delimiter is determined is immaterial. What matters is that when one of these delimiters is encountered, the token acquisition stops.
2. Generate Selection Sets for the grammar;
3. You are to write a phased implementation of a compiler to translate a program written in the language described above into an object language to be defined at a later date. Your compiler will consist of a lexical analyzer (or scanner), syntax analyzer, semantic analyzer, and code generator.

## Reserved Words/Symbols:

1. program
2. array
3. integer
4. read
5. write
6. rdln
7. wrln
8. when
9. until
10. from
11. from
12. ..
13. and
14. or
15. mod

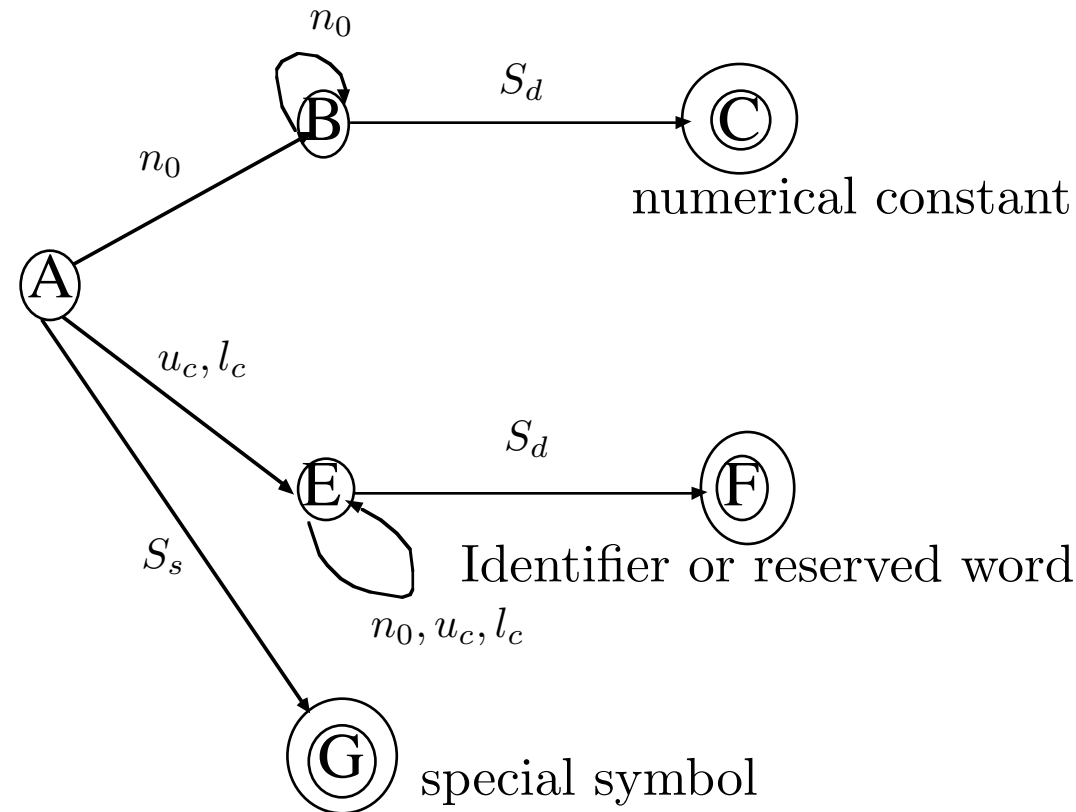
## delimiter symbols

- eof
- {
- }
- :
- [
- ]
- .
- <
- >
- =
- <=
- >=
- <>
- +
- -
- \*
- /
- :



**Tokens of the Language** Tokens of the language can be either a space ( $S_0$ ), special symbol ( $S_s$ ), identifier, numerical constant, or a reserved word. Identifiers and reserved words can be considered to almost the same except, the reserved words are preloaded into the symbol table.

- $S_d = S_s \cup S_0$
- $u_c = \{A, \dots, Z\}$
- $l_c = \{a, \dots, z\}$
- $n_0 = \{0, \dots, 9\}$



# A cute token analyzer

- input: String  $s$  integer position
- output Token identifier
- Support variables: character  $c$ , integer tokenid, integer index

