Lexical Analysis

Lexical Analysis Basics

- Task
  - Replace sequence of characters by a sequence of tokens
  - Change alphabet from characters to tokens
- Byproducts
  - Remove comments
  - Convert cases for case-insensitive languages
  - Remove white spaces (space, tab, end-of-line)
- Implementation
  - Scanner
  - Finite state automaton/machine (FSA/FSM)
  - Two architectures
    - Produce explicit output file – program as a sequence of tokens
    - Be embedded in parser (*parser-directed translation*) producing one token at a time on demand

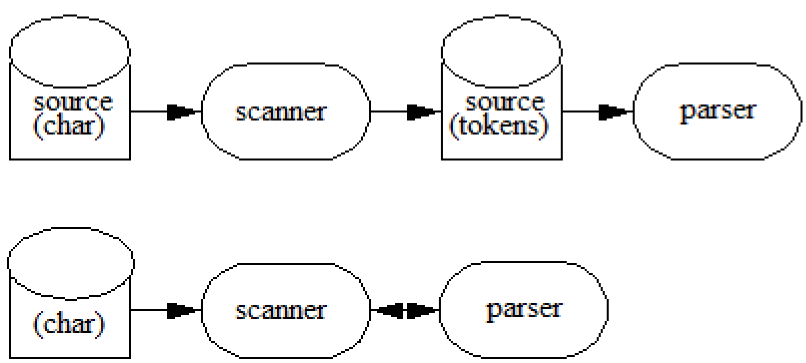

Figure 1. Two architectures for scanners: generator (above) and embedded (below). The second is more practical.

- Example: source code (characters) and the corresponding tokens below

```
if distance >= rate * (endTime - startTime) then distance := maxDist ;
```

ifTk idTk relopTk idTk *Tk (Tk idTk -Tk idTk)Tk thenTk idTk assignTk idTk ;Tk

- Some mapping/tokens decision
  - Keywords: one for all (keywordTk) or individual
  - Identifiers: one for all or individual
  - *Etc.*
  - Any mapping many-to-one must also include the instance, making token a pair
    - [keywordTk,"if"]
    - [idTk, "distance"]
  - In practice, line number is also added to make it triplet
- *Symbol table*
  - Symbol table is any list of symbols under consideration
  - In translation, we need the list of identifiers and they can be processed in a symbol table
  - Assemblers use symbol tables produced by lexical analyzer
  - In block-structured languages the list of identifiers is dynamic and thus cannot be built by translator (tokens have dynamic lifetime and scope)

Type 1: Regular Language and its Finite Automaton

- Finite state automaton (FSA/FSM) is equivalent to (one-to-one) a regular language:
- input alphabet $\sum$
- finite nonempty set of states Q(at least one)
- a starting state $q \in Q$
- a nonempty set of final states $F \subseteqq Q$
- state transition function ($Q \times \sum) \rightarrow Q$
- FSA accepts a string $\omega \epsilon \sum^*$ if the automaton, starting from the initial state arrives at a final state when the string ω is exhausted
  - Scanner will do it repeatedly until it produces EOFTk , each time starting from the beginning
- FSA can be represented as
  - a **labeled** and **directed** graph, good for design
  - transition table with a driver, good for implementation
- For error situations, one may use error states or alternatively one may assume lack of transition for $\sum$ element out of a state is an error situation

Lookahead

- In translation, *lookahead* refers to looking ahead into the program before making some decision
- For practical efficiency reasons, lookahead is often reduced to just 1
- In lexical analysis
  - characters are processed and thus lookahead refers to characters read ahead into the input
  - lookahead can be avoided by requirement to put spaces around all tokens
  - one lookahead is needed when spaces are not required
  - x+y vs. x+ y x23 vs. x 23

Example: scanner for identifiers

Design lexical analyzer (as graph) for the following language:

- identifiers that must start with a letter and continue with a sequence of letters, digits, or underscores.
  This can be represented by the following regular expression:
  where L denotes the language of all letters, D of all digits, and U the language with only the underscore.
- Alphabet

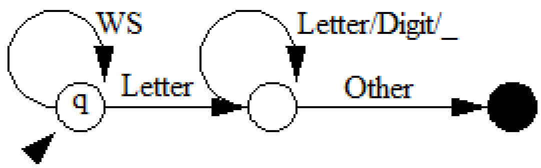

Figure 2. Scanner for identifiers only. “Other” refers to the remaining options out of the current node/state.

- Try
  - X23
  - Now assume alphabet also includes ‘+’, try
    - xyz + 2
    - xyz+2
- Lookahead is needed when no spaces required
  - Must look next character to know that it is too far
  - Cannot be consumed
    - Peek
    - Put back into input
    - Remember as first
- Now redesign if letter can be followed by
  - Exactly 2 characters
  - At most 2 characters

Nondeterminism

- *Nondeterminism* refers to not having enough information to make a proper choice decision when needed
- Nondeterminism can be handled in a number of ways
  - *Backtracking* implementation with stack support (recursive implementation)
  - Massively parallel computation
  - *Lookahead* (looking for needed information)
  - For efficiency, when possible we avoid nondeterminism if possible
- FSA can be
  - Deterministic DFSA/DFSM

        - in every state there is one or zero transitions for every element from ∑ (assuming 0 transitions is an error case)
    - Non-deterministic NDFSA/NDFSM
        - at least one state has ambiguity on at least one element from ∑, or
        - at least one state has an empty transition (no label in graph, meaning jump)
    - NDFSA implementations are not efficient
- For every NDFSA there exist DFSA representing the same language
    - NDFSA is thus convenience not extra power

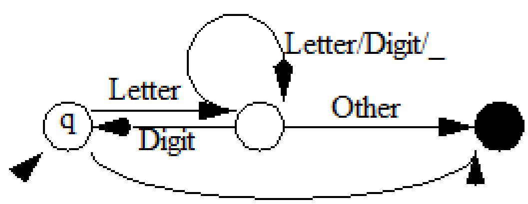

Figure 3. Nondeterministic FSA. The middle state has ambiguous transitions on Digit, and there is jump from the initial state to the final state.

More Facts

- For any regular language, there always exists a deterministic FSA to recognize the language
- For any regular language, there always exists a *minimal* DFA that recognizes the language
- One lookahead is needed unless WS are required separators in embedded scanner
    - WS={tab,space,eol}. EOF is usually a separate token.
- FSA do not have infinite memory
    - The only way to remember something is to have a state for that fact – being in that state means the fact is true
- All final states are equivalent
    - In scanner implementation, final states are often separated to specify the token that is recognized
- NDFSA must be converted to DFSA
- When FSA Is used for a scanner, it is usually relaxed in a number of ways
    - One lookahead is needed
    - Keywords are recognized as identifiers if using the same definition
    - Final states are separated to include token information
    - Line number and token instance must also be processed

Example: Design Scanner Based on FSA

- Suppose a language contains the following tokens
    - Keywords: if, then, begin, end
        - Keywords are *reserved*

- Operators: >, >=
- Integers: any sequence of decimal digits

- Suppose we merge keywords with identifiers since they follow the same definition
  - Postprocessing lookup on identifiers will separate the keywords
- Other design decisions
  - What many-to-one mappings will be used
    - Identifiers must be grouped
    - Integers must be grouped
    - Operators can be grouped or be separate tokens
  - Is any WS separating tokens - if not then one lookahead is needed
  - Embedded – yes
  - EOF – use special token

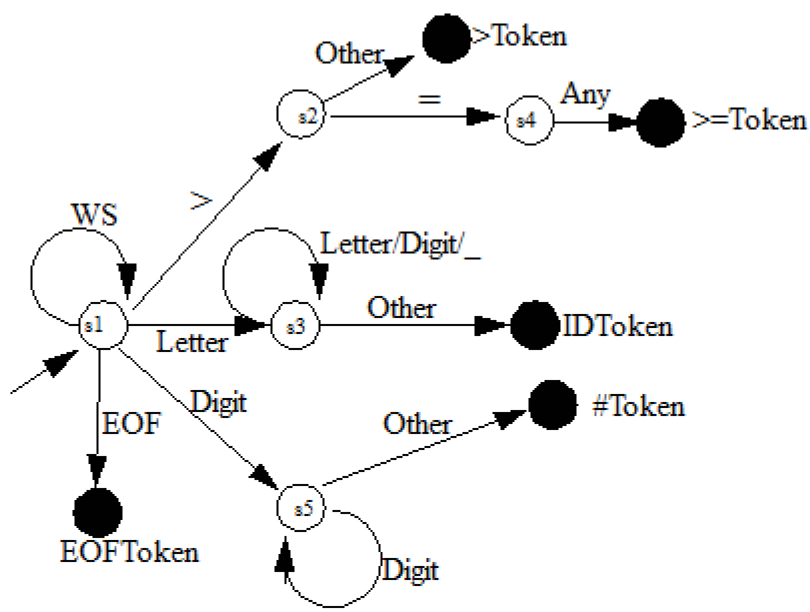

Figure 4. FSA implementing scanner for the given language. Keywords are not separated from identifiers. Different final states recognize different tokens.

- Scanner can be produced using
  - Lexical tool such as *lex*
  - Hard coded `ifs`
  - Recognized using multiple switch statements with language tokenizer especially assuming WS separators
  - Represented as graph, then graph as table with a driver

Index/ Label	>	=	Letter	Digit	WS	EOF	_
0/s1	s2	Error “no token starts with =”	s3	s5	s1	Final EOFToken	Error “no token starts with _”
1/s2	Final >Tk	s4	Final >Token	Final >Token	Final >Token	Final >Token	Final >Token
2/s3	Final IDTk	Final IDToken	s3	s3	Final IDToken	Final IDToken	s3
3/s4	Final >=Tk	Final >=Token	Final >=Token	Final >=Token	Final >=Token	Final >=Token	Final >=Token
4/s5	Final #Tk	Final #Token	Final #Token	s5	Final #Token	Final #Token	Final #Token

Figure 5. Table for the graph above. The actual table is just the white inside.

- Driver pseudocode

```
tokenType FADriver() // assume nextChar set, and used as column index
{     state_t state=INITIAL /* (0=s1 here) */
      nextState;
      tokeType token;
      string S=NULL;
      while (state!=FINAL)
      {     nextState=Table[state][nextChar];
            if (nextState==Error)
                  ERROR(); /* report and exit */
            if (nextState==FINAL)
                  if (token(state)==ID) // need reserved keyword lookup
                        if (S in Keywords)
                              return (KWtk,S) // or specific keyword
                        else
                              return (IDtk,S)
                  else
                        return (Table[state][Token],S)
            else /* not final */
                  state:=nextState;
                  append(S,nextChar);
                  nextChar=getchar();
}
```

More Table/Driver Implementation Questions

- Columns with exactly the same transitions can be combined with the use of a function mapping input character to a column number
- May use ASCII value of a character for a column number
- Row numbers may be row indexes, in which cane the table is a table of integers
    - Negative integers may represent error case
    - Other range can represent final states
- How to process comments
    - In table – complexity
    - Skip in preprocessor filer
- Preprocessor filer
    - Skip comments
    - Count line numbers
    - Map characters into column numbers
    - Error on invalid characters
- Error recovery
    - Usually none, termination