CS 406: Lexical Analysis

Stefan D. Bruda

Winter 2016

The Lexical Analyzer

- Main role: split the input character stream into tokens
  - Usually even interacts with the symbol table, inserting identifiers in it (especially useful for languages that do not require declarations)
  - This simplifies the design and portability of the parser
- A token is a data structure that contains:
  - The token name = abstract symbol representing a kind of lexical unit
  - A possibly empty set of attributes
- A pattern is a description of the form recognized in the input as a particular token
- A lexeme is a sequence of characters in the source program that matches a particular pattern of a token and so represents an instance of that token
- Most programming languages feature the following tokens
  - One token for each keyword
  - One token for each operator or each class of operators (e.g., relational operators)
  - One token for all identifiers
  - One or more tokens for literals (numerical, string, etc.)
  - One token for each punctuation symbol (parentheses, commata, etc.)

Example of Tokens and Attributes

`printf("Score = %d\n", score);`

Lexeme	Token	Attribute
`printf`	**id**	pointer to symbol table entry
`(`	**open_paren**	
`"Score = %d\n"`	**string**	
`,`	**comma**	
`score`	**id**	pointer to symbol table entry
`)`	**cls_paren**	
`;`	**semicolon**	

`E = M * C ** 2`

Lexeme	Token	Attribute
`E`	**id**	pointer to symbol table entry
`=`	**assign**	
`M`	**id**	pointer to symbol table entry
`*`	**mul**	
`C`	**id**	pointer to symbol table entry
`**`	**exp**	
`2`	**int_num**	numerical value 2

Input Buffering

- Buffering is often used to speed up the process of recognizing lexemes
  - Also facilitates the process of looking ahead beyond the current lexeme
- Typical buffer arrangement:
  - Two buffers of size N = the size of a disk sector (usually 4096 bytes)
  - One buffer is loaded while the other is being processed
  - One system call fills in a whole buffer
  - Two pointers per buffer: lexemeBegin (the beginning of the current lexeme) and forward (moves forward until a pattern is found, but can also move backward)
- Problem: each time we advance the forward pointer we need to tests: one for the current character, the other for the end of the buffer
  - Solution: place a special sentinel character (e.g., `EOF`) at the end of the buffer
  - A single test will then suffice

Specification of Tokens

- Token patterns are simple enough so that they can be specified using regular expressions
- Alphabet Σ: a finite set of symbols (e.g. binary digits, ASCII)
- Strings (not sets!) over an alphabet; empty string: ε
  - Useful operation: concatenation ($\cdot$ or juxtaposition)
  - ε is the identity for concatenation ($\varepsilon w = w\varepsilon = w$)
- Language: a countable set of strings
  - Abuse of notation: For $a \in \Sigma$ we write a instead of $\{a\}$
  - Useful elementary operations: union ($\cup$, $+$, $|$) and concatenation ($\cdot$ or juxtaposition): $L_1 L_2 = L_1 \cdot L_2 = \{w_1 w_2 : w_1 \in L_1 \wedge w_2 \in L_2\}$
  - Exponentiation: $L^n = \{w_1 w_2 \cdots w_n : \forall 1 \leq i \leq n : w_i \in L\}$ (so that $L^0 = \{\varepsilon\}$)
  - Kleene closure: $L^* = \bigcup_{n \geq 0} L^n$
  - Positive closure: $L^+ = \bigcup_{n > 0} L^n$
- An expression containing only symbols from Σ, ε, $\emptyset$, union, concatenation, and Kleene closure is called a regular expression
  - A language described by a regular expression is a regular language

Syntactic Sugar for Regular Expressions

Notation	Regular expression	
r^+	rr^*	one or more instances (positive closure)
$r?$	$r\|\varepsilon$ or $r + \varepsilon$ or $r \cup \varepsilon$	zero or one instance
$[a_1 a_2 \cdots a_n]$	$a_1 \| a_2 \| \cdots \| a_n$	character class
$[a_1 - a_n]$	$a_1 \| a_2 \| \cdots \| a_n$	provided that a_1, a_2, $\ldots a_n$ are in sequence
$[\hat{\ } a_1 a_2 \cdots a_n]$		anything except a_1, a_2, $\ldots a_n$
$[\hat{\ } a_1 - a_n]$		

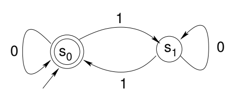
- The tokens in a programming language are usually given as regular definitions = collection of named regular languages

Examples of Regular Definitions

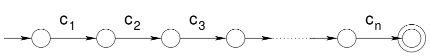
$$
\begin{aligned}
\text{letter\_} &= [A - Za - z\_] \\
\text{digit} &= [0 - 9] \\
\text{id} &= \text{letter\_ (letter\_} \mid \text{digit)}^* \\
\text{digits} &= \text{digit}^+ \\
\text{fraction} &= \text{. digits} \\
\text{exp} &= E\ [+-]?\ \text{digits} \\
\text{number} &= \text{digits fraction? exp?} \\
\text{if} &= i\ f \\
\text{then} &= t\ h\ e\ n \\
\text{else} &= e\ l\ s\ e \\
\text{rel\_op} &= < \mid > \mid <= \mid >= \mid == \mid !=
\end{aligned}
$$

State Transition Diagrams

- Also called deterministic finite automata (DFA)
- Finite directed graph
- Edges (transitions) labeled with symbols from an alphabet
- Nodes (states) labeled only for convenience
- One initial state
- Several accepting states



- A string $c_1 c_2 c_3 \ldots c_n$ is accepted by a state transition diagram if there exists a path from the starting state to an accepting state such that the sequence of labels along the path is c_1, c_2, $\ldots$, c_n



  - Same state might be visited more than once
  - Intermediate states might be final
- The set of exactly all the strings accepted by a state transition diagram is the language accepted (or recognized) by the state transition diagram

Software Realization

- Big practical advantages of DFA: very easy to implement:
  - Interface to define a vocabulary and a function to obtain the input tokens

```
typename vocab;        /* alphabet + end-of-string */
const vocab EOS;       /* end-of-string pseudo-token */
vocab gettoken(void); /* returns next token */
```

  - Variable (state) changed by a simple `switch` statement as we go along

```
int main (void) {
    typedef enum {S0, S1, ... } state;
    state s = S0;    vocab t = gettoken();
    while ( t != EOS ) {
        switch (s) {
            case S0: if (t == ...) s = ...; break;
                     if (t == ...) s = ...; break;
                     ...
            case S1: ...
            ...
        } /* switch */
        t = gettoken();   } /* while */
    /* accept iff the current state s is final */
}
```

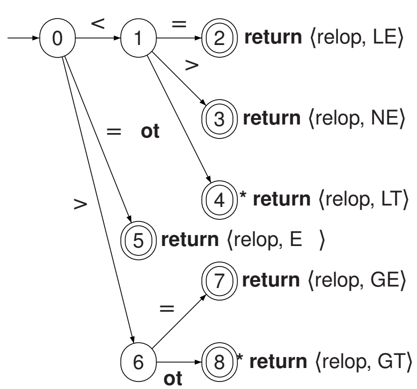

Software Realization: Example

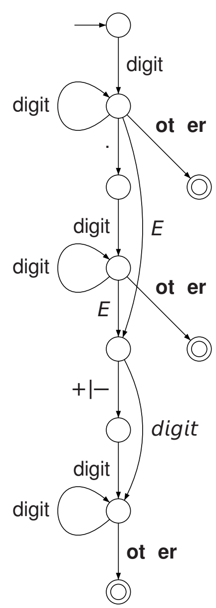
```
typedef enum {ZERO, ONE, EOS} vocab;

vocab gettoken(void) {
    int c = getc(stdin);
    if (c == '0')  return ZERO;
    if (c == '1')  return ONE;
    if (c == '\n') return EOS;
    perror("illegal character");    }

int main (void) {
    typedef enum {S0, S1 } state;
    state s = S0;    vocab t = gettoken();
    while ( t != EOS ) {
        switch (s) {
            case S0: if (t == ONE)  s = S1; break;
                  /* if (t == ZERO) s = S0; break */
            case S1: if (t == ONE)  s = S0; break;
                  /* if (t == ZERO) s = S1; break */   } /* switch */
        t = gettoken(); } /* while */
    if (s != S0) printf("String not accepted.\n");     }
```


Examples of State Transition Diagrams





When returning from *-ed states must retract last character

Lex, the Lexical Analyzer Generator

- The Lex language is a programming language particularly suited for working with regular expressions
  - Actions can also be specified as fragments of C/C++ code
- The Lex compiler compiles the Lex language (e.g., `scanner.l`) into C/C++ code (`lex.yy.c`)
  - The resulting code is then compiled to produce the actual lexical analyzer
  - The use of this lexical analyzer is through repeatedly calling the function `yylex()` which will return a new token at each invocation
  - The attribute value (if any) is placed in the global variable `yylval`
  - Additional global variable: `yytext` (the lexeme)

- Structure of a Lex program:
  Declarations
  %%
  translation rules
  %%
  auxiliary functions

- Declarations include variables, constants, regular definitions
- Transition rules have the form

  Pattern { Action }

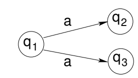
  where the pattern is a regular expression and the action is arbitrary C/C++ code

Lex Behaviour

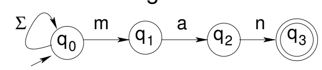
- LEX compile the given regular expressions into one big state transition diagram, which is then repeatedly run on the input
- LEX conflict resolution rules:
  - Always prefer a longer to a shorter lexeme
  - If the longer lexeme matches more than one pattern then prefer the pattern that comes first in the LEX program
- LEX always reads one character ahead, but then retracts the lookahead character upon returning the token
  - Only the lexeme itself in therefore consumed

Nondeterministic State Transition Diagrams

- Deterministic = for any pair (state, input symbol) there can be at most one outgoing transition
- A nondeterministic diagram allows for the following situation:



- The acceptance condition remains unchanged:
  - A string $c_1c_2c_3 \ldots c_n$ is accepted by a state transition diagram if there exists some path from the starting state to an accepting state such that the sequence of labels along the path is $c_1, c_2, \ldots, c_n$
- Why nondeterminism?
  - Simplifies the construction of the diagram



  - A nondeterministic diagram can be much smaller than the smallest possible deterministic state diagram that recognizes the same language
- Also known as nondeterministic finite automata (NFA)

Software Realization

- As for the deterministic version, except that we have to keep track of a set of states at any given time

```
typedef enum { Q0, Q1, Q2, Q3 } state;

int main (void) {
    vocab t = gettoken(); StateSet A;  A.include(Q0);
    while (t != EOS) {
        StateSet NewA;
        for (state s in A) {
            switch (s) {
              case Q0: NewA.include(Q0);
                       if (t == 'm') NewA.include(Q1); break;
              case Q1: if (t == 'a') NewA.include(Q2); break;
              case Q2: if (t == 'n') NewA.include(Q3); break;
              case Q3: break;
            }
        }
        A = NewA;  t = gettoken();
    }
    /* accept iff (Q3 in A) */
}
```


Software Realization (cont'd)

- This kind of implementation is fine for "throw-away" automata
  - Text editor search function searches for a pattern in the text
  - The next search is likely to be different so a brand new automaton needs to be created
- Some times the automaton is created once and then used multiple times
  - The lexical structure of a programming language is well established
  - Lexical analysis in a compiler is accomplished by an automaton that never changes
  - In such a case it is more efficient to precalculate the set of states
    - Exactly as in the previous program
    - Except that we no longer have an input to guide us, so we calculate the sets `NewA` for all possible inputs
    - We obtain a DFA that is equivalent to the given NFA (i.e., recognizes the same language)

ε-TRANSITIONS

- Useful at times to have "spontaneous" transitions = transitions that change the state without any input being read = ε-transitions
  - Only available for nondeterministic state transition diagrams!
- Example of usefulness: Construct the state transition diagram for the language

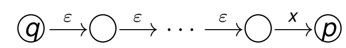
$$\{0,1\}^*01\{0,1\}^* + \{w \in \{0,1\}^* : w \text{ has an even number of 1's}\}$$

- Even better ε-transitions can be eliminated afterward

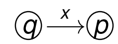
ELIMINATING ε-TRANSITIONS

For every diagram M with ε-transitions a new diagram without ε-transitions can be constructed as follows:

1. Make a copy M' of M where the ε-transitions have been removed. Remove states that have only ε-transitions coming in except for the starting state
2. Add transitions to M' as follows: whenever M has a chain of ε-transitions followed by a "real" transition on x:



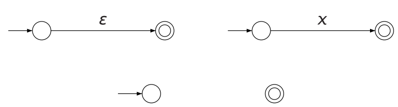
   add to M' a transition from state q to state p labeled by x:



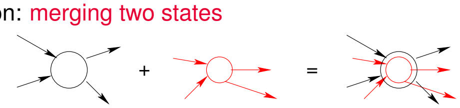
   - Note that q and p may be any states
   - In particular this step is also used in the case where $q = p$
3. If M has a chain of ε-transitions from a state r to an accepting state, then r is made to be an accepting state of M'.

FROM REGULAR EXPRESSIONS TO FA

- Construct a finite automaton for every elementary regular expression (ε, $x \in \Sigma$, $\emptyset$):



- Then starting from component finite automata we show how we can construct finite automata for each possible operator appearing in regular expressions ($+$, $\cdot$, *)
  - Useful operation: merging two states

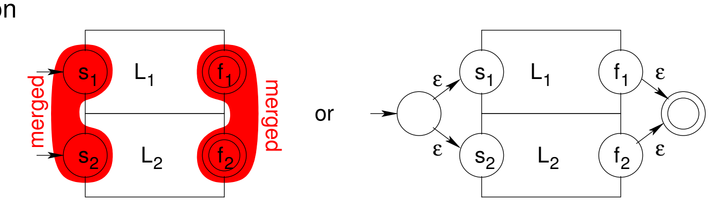


  - Properties to be maintained:
    - One accepting state
    - Initial state different from the accepting state
    - No transitions out of the accepting state

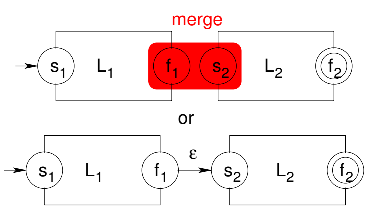
FROM REGULAR EXPRESSIONS TO FA (CONT'D)

- We start from the following two automata:

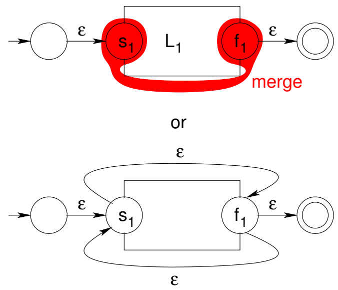
- Union



- Concatenation



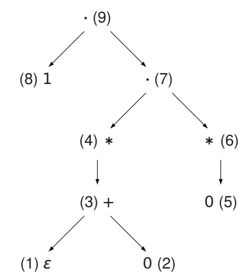
- Closure



- All regular expressions can be converted step by step to the equivalent finite automaton by using these constructions
  - Construct a tree that represents the operations in the regular expression
    - Leafs are labeled with elementary regular expressions
    - Internal nodes are labeled with operation, and their children are the operands
  - Traverse the tree from leaves to root using the previous constructions

Example: $1(\varepsilon + 0)^*0^*$

1. FA for ε
2. FA for 0
3. FA for $\varepsilon + 0$
4. FA for $(\varepsilon + 0)^*$
5. FA for 0
6. FA for 0^*
7. FA for $(\varepsilon + 0)^*0^*$
8. FA for 1
9. FA for $1(\varepsilon + 0)^*0^*$



- The finite automaton thus obtained can either be converted into a deterministic finite automaton or realized as is