Structure of Programming Languages – Lecture 3

CSCI 6636 – 4536

Spring 2017
1 Finite Languages
   - Deterministic Finite State Machines
   - Lexical Analysis
   - Regular Expressions

2 Creating a Lexer
   - Lexer Generators
   - Flex

3 Homework
Part 1

1. Finite Languages

Deterministic Finite State Machines
Lexical Analysis
Finite Languages

Deterministic Finite Automata (DFA)

A deterministic finite automaton is an abstract machine for classifying strings. It reads an input string and either accepts or rejects it. It has:

- A finite set of states, including one called $S_0$, the initial state.
- A subset of the states, $A$, called accepting states.
- An input alphabet.
- A state transition rule.

  - The machine starts in state $S_0$.
  - At each step, it reads an input character, $c$.
  - Then it moves to another state, depending on $c$ and the current state.
  - After the last symbol has been read and processed, the string is accepted if the current state is in the set $A$.

In some generalizations, each state transition has an associated action.
Automaton 1

Alphabet: a, b
States: S0, D, P, R
Starting state: S0
Accepting states: {S0, R}
What Language is Defined by this DFA?

A *language* is a set of strings. Automaton 1 accepts this language:

- The null string
- ab
- abab
- ababab
- abbbbbbbbb
- abbbbbbbbbbabbbbab

It rejects:

- Anything that begins with b.
- Anything that ends in a.
- Anything with two consecutive a’s.

Since the alphabet for this machine is a, b, we don’t concern ourselves with any other letters or symbols.
Automaton 2

Alphabet: a, b, d
States: S0, D, P, R
Starting state: S0
Accepting states: {R}
What Language is Defined by this DFA?

Automaton 2 accepts this language:
- bb
- dd
- bad
- bab
- dad
- daaaab
- baaaaaaaaaaaaaad

It rejects:
- Anything that begins with a or ends with a.
- Anything that ends in a.
- Anything with more than two of b/d.
A Shorthand Notation

- We use DFA is a mathematical abstraction with an alphabet, states, and a transition rule.
- A DFM is a generalization of the DFA with outputs.
- We use DFM’s to do the lexical analysis for programming languages.
- We represent the DFM with a graphical language of circles and transition links.
- I have introduced graphs that are fully explicit:
  - Every state is shown in the graph.
  - Every state has a transition rule for every symbol in the alphabet.
- Some professionals use graphs with implicit links to a Dead state. Any “missing” transition goes, by default, to the Dead state, which may not be shown, but is listed among the states of the DFA.
Short form of Automaton 2

Alphabet: a, b, d
States: S0, D, P, R
Starting state: S0
Accepting states: {R}
Lexical analysis is the first stage of translation.

- The input is the source code of a program or a program unit.
- The Lexer reads the characters in the source code one at a time and identifies the primitive language units (words and punctuation) that make up the program.
- Illegal characters (if any) are identified.
- Comments are identified and discarded.
- The remaining units are called lexemes.
- A lexeme, together with its category-code is called a token.
- A lexer outputs a stream of tokens, ready for preprocessing or for parsing.
Lexing a number in C

- A number must start with a digit, which may be followed by other digits.
- It may have a decimal point at the beginning or end of the digits, or in the middle.
- If a decimal point is present, an exponent may follow the last digit or the decimal point.
- An exponent consists of the letter E or e, followed by a + or − sign, followed by one or more digits, and ending in a precision indicator (D, d, F, f, L, or l).

See: LexC.pdf
Lexing FORTH

- A rest-of-line comment starts with a backslash in column and ends with the next newline.
- An inline comment starts with \(< \text{space} >\) and ends with the first \(\) .
- A string literal starts with ." \(< \text{space} >\) and ends with the first ".
- A number is a sequence of digits delimited by whitespace.
- Any other collection of visible characters form a word.

See: LexFORTH.pdf
Regular Expressions

A regular expression is a string that describes a whole set of strings in some finite language, according to certain rules.

- An American logician named Stephen Cole Kleene studied regular languages in the 1950’s and introduced one notation for writing regular expressions.
- We use regular expressions to specify the lexical rules for computer languages.

**Theorem (Kleene)**

A language is definable by a regular expression if and only if it is recognized by a finite-state automaton.
Kleene Regular Expressions

To write a Kleene regular expression, you need:

- An alphabet. Each symbol in the alphabet is a regular expression.
- A set of operators, in order of precedence: \( \ast \cdot + \)
  - \( x^\ast \) means 0 or more copies of \( x \).
  - \( x \cdot y \) means \( x \) followed by \( y \).
  - \( x + y \) means the union of \( x \) and \( y \): either \( x \) or \( y \) or both.
- Grouping symbols to override the precedence: ( )

Variations on this idea are widely used in CS applications (grep, flex, Perl, …).
Examples: Finite Languages Revisited

- Expression for Automaton 1: \((a \cdot b \cdot b^*)^*\)
- Expression for Automaton 2: \((b + d) \cdot a^* \cdot (b + d)\)

We use regular expressions to define the lexical structure of languages. These are translated into efficient computer programs (lexers), based on finite state machines, that recognize the language forms.
Part 2

2. Lexical Analysis

Lexer Generators

Flex: Fast Lexer Generator
A person implementing a compiler must produce a lexer for the language. This can be done in two ways.

(See: Wikipedia–Lexical Analysis)

- Write a custom lexer. This can be more efficient and can be the only solution if the language is complex or its lexemes are recursively defined.

- Write regular expressions that define the lexical structure. Then use these definitions as input to a lexer-generator, such as *lex* or *flex*, which turns them into a lexer for the language, implemented as a finite-state machine.
Flex: Fast Lexer Generator

Flex is the GNU lexer generator.

- It reads user-specified input files for a lexer description in the form of pairs of regular expressions and corresponding action-rules coded in C.
- The result of running flex is a C program, which must be compiled to create a lexer.
- The resulting lexer analyzes its input (source code) for occurrences of text matching the original regular expressions. Whenever it finds a match, it executes the corresponding action-rule.
Flex Regular Expressions-1

This is the extended regular expression syntax recognized by *flex*:

- **x** match a particular character
- **xy** an x followed by a y (this is Kleene’s · operator)
- **.** match any single character (not the same as Kleene’s ·)
- **x|y** either an x or a y (this is Kleene’s + operator).
- **x** zero or more x’s
- **x+** one or more x’s (not the same as Kleene’s +).
- **x?** zero or one x (an optional x)
- **d{2}** exactly 2 d’s
- **d{2,}** 2 or more d’s
- **d{2,5}** from 2 to 5 d’s
- **[^A]** match any character except A. (A negated character.)
This is the extended regular expression syntax recognized by *flex*:

- \[xyz\]  
  a character class: x or y or z

- \[xyz\]*  
  zero or more chars from the set x, y, z, in any order or combination.

- \[0-9\]  
  a character class with a range of numbers (describes a decimal literal)

- \[a-zA-Z\]  
  a character class with two ranges

- \[^A-Z\n\]  
  a negated set of characters that includes a range

- \[a..z\]{-}\[lmno\]  
  the set a...z with lmno removed

Plus several increasingly complex ways to name patterns and make patterns out of other patterns.
Creating a Lexer

Flex

Examples: Flex Regular Expressions

\[(abb\ast\ast)\ast\]
\[(ab\ast\ast)\ast\]
\[[bd]a\ast[bd]\]
\[ab?c\]

Automaton 1, Kleene syntax: \((a \cdot b \cdot b\ast\ast)\ast\)
Automaton 1, using the extended Flex syntax
Automaton 2, Kleene syntax: \((b+d) \cdot a\ast \cdot (b+d)\)
A string that starts with and a and ends with a c
and has an optional b in the middle.

\[0\[xX\][a\text{-}fA\text{-}F0\text{-}9]\ast\]
\[\[^\"\]\]
\[a\text{-}z\]{\sim}\[eo\]

A hex literal, in C.
Any character except a double quote.
The set of characters a...z, excluding e and o.
Flex for FORTH

To analyze a FORTH program, the following rules should be applied in order. If one fails, try the next. If the first four rules fail, the last one will always succeed.

Let WS stand for a sequence of 1 or more whitespace characters.
Input symbols are given in red ink, lexer symbols are black.

```plaintext
Comment \ WS [ ^\n ]* \n
Comment ( WS [ ^) ]* ) WS

StringLiteral .” WS [ ^” ]* ” WS

Integer –? [0–9 ^ WS]* WS

Word [^WS]* WS
```

A recognizer for this language needs five accepting states. If you are in an accepting state when a newline or WS terminates a rule, the entire string that has been matched should be output as a token, with the category from the left column.
Hw 3: Finite-State Machines and Regular Expressions

1. Define a DFA that will process input strings of letters from a to e. Accept all strings that contain only vowels (‘a’ and ‘e’) . Reject all other inputs.

2. Write a Flex regular expression that defines strings of letters that contain only the letters ‘a’ and ‘e’ , in any combination, any order, and any number.

3. Define a DFA that will process input strings of letters from a to e. Accept all strings of exactly 3-letters that start with a vowel. (Examples: add, ade, edc, eee) Reject all other inputs (Examples: dad, eat).

4. Write a Flex regular expression that defines strings of 3-letters that contain only the letters ‘a’ through ‘e’ and start with a vowel.
5. Define a DFA that accepts legal C identifiers. The rules are:
   1. An identifier can be composed of letters (both uppercase and lowercase letters), digits and underscore ‘_’ only.
   2. The first letter of identifier cannot be a digit.
   3. There is no limit on the length of an identifier.

6. Write FORTH function that displays your name. (Keep it simple.)

Read chapter 4.1 through 4.3 of the text.