CS 316 E6TBA Queens College CUNY
Exercise Set #2

1. Consider the following EBNF grammar:
< id > → < letter > { < letter > | < digit > }
< extended id > → < id > { "_" < letters and digits > }
< letters and digits > → { < letter > | < digit > }$^+$
Construct a DFA that accepts the strings in < extended id >.

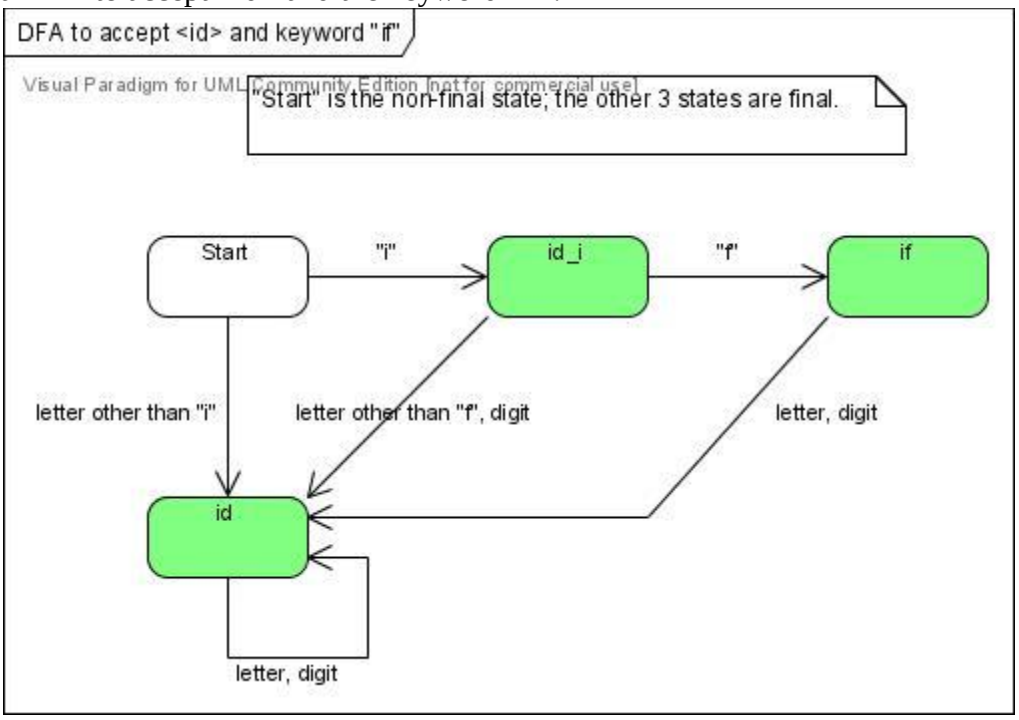
2. Given is the following BNF grammar:
< comp op > → "<" | "<=" | ">" | ">=" | "= =" | "!="
< arith op > → + | − | * | /
< paren > → "(" | ")"
< token > → < comp op > | < arith op > | < paren >
Construct a DFA that accepts the strings in < token >.

3. Given is the following EBNF grammar:
< float > → [+ | −] {< digit >} "." {< digit >}$^+$ [(E | e) [+ | −] {< digit >}$^+$]
Construct a DFA that accepts the strings in < float >. Note that according to this grammar, the integer part may be empty.

4. Building keyword recognition into DFAs is not conceptually difficult but considerably increases the number of states and transitions. The key fact about usual keywords is that an identifier may contain an initial segment or the whole of a keyword; e.g., "whil", "whil5", "whilea", etc. Based on this observation the following method can be used:

- Create a linear sequence of transitions for each keyword; any non-empty, proper initial segment of each keyword is accepted by a final state as an identifier.
- Whenever the input letter or digit breaks a keyword, the transition branches off to the <id> state.
- The final state for each keyword has the transition to the <id> state on the letters and digits.

The following is a DFA to accept <id> and the keyword "if":



Modify this DFA to accept additional keywords "else" and "while".

5. Consider the following EBNF grammar for <int> that defines a subset of the Java integer numerals.
<digit> → 0 | 1 | ... | 9
<nonZeroDigit> → 1 | ... | 9
<hexDigit> → <digit> | a | b | ... | f | A | B | ... | F
<octalDigit> → 0 | 1 | ... | 7
<int> → <decimal> | <hexadecimal> | <octal>
<decimal> → 0 | <nonZeroDigit> {<digit>}
<hexadecimal> → ("0x" | "0X") {<hexDigit>}$^{+}$
<octal> → 0 {<octalDigit>}$^{+}$
Construct a DFA that accepts the strings in < int >.
(Note: C++ integer numerals are almost identical; the difference is that "0" belongs to <octal>.)

6. Lexical analyzers are an example of pattern matching based on BNF grammars and finite automata, which has many other applications. Quite a number of modern script and programming languages provide built-in features or library functions to recognize patterns specified by EBNF grammars.

1. Suppose we want to find all file names containing an occurrence of "CS316" and ending with the extension ".txt". The "C" or "S" may be in lowercase. Give an EBNF grammar that defines this pattern.
2. Spam filters. Suppose we want to catch all email subject headers containing an occurrence of "pharmacy" where any of its characters may be in uppercase and any single ASCII character may be inserted between any of its characters. For example, "pHar%Ma?cy" and "p+hArMac!y". Give an EBNF Grammar that defines this pattern.

7. Consider the following grammar:
< sequence > → "(" < elements > ")"
< elements > → < element > { "," < element > }
< element > → < id > | < sequence >
< id > → < letter > { < letter > | < digit > }
Give a top-down parser in pseudo code for < sequence >, < elements >, and < element >. Presume that the function getToken( ) is given to extract the next token and assign it to the string variable *t*. The tokens are: <id>, "(", ")", and ",".

8. Consider the following grammar for arithmetic expressions, with the unary − operator in < primary >.
< E > → < term > { (+ | −) < term > }
< term > → < primary > { (* | /) < primary > }
< primary > → < id > | "(" < E > ")" | − < primary >
Presume that the function getToken( ) is given to extract the next token and assign it to the string variable *t*. The tokens are: <id>, "(", ")", +, −, *, and /.

1. Give pseudo code for a top-down parser for < E >, < term >, and < primary >; include pseudo code to emit suitable stack-based instructions *push*, *add*, *sub*, *mul*, *div*, and *neg*.
2. Give the sequence of *push*, *add*, *sub*, *mul*, *div*, and *neg* instructions that would be generated from each of the following expressions:
    a. X+Y+Z
    b. X−Y+Z−W
    c. X+Y*Z
    d. (X+Y)/(Z−W)
    e. X*−Y+Z/−W

9. Consider the following grammar:
< A > → (+|−) < B > < B >
< B > → < id > | < A >
Give pseudo code for a top-down parser for < A > and < B >. Presume that the function getToken( ) is given to extract the next token and assign it to the string variable *t*. The tokens are: <id>, +, and −.

10. In class we studied a top-down parser for the following grammar:
< S > → < assignment > | "{" < S List > "}"
< assignment > → < id > = < E > ;
< S List > → { < S > }$^{+}$
Let us extend the production rule for < S > with while-loops and conditionals:
< S > → < assignment > | "{" < S List > "}" | **while** "(" < B > ")" < S > | **if** "(" < B > ")" < S > [**else** < S >]
Give pseudo code for a top-down parser for the extended < S >, < assignment>, and < S List >. Presume that you are given the function B( ) to parse Boolean expressions < B >, E( ) to parse arithmetic expressions < E >, and getToken( ) to extract the next token and assign it to the string variable *t*. The tokens are: <id>, **if**, **else**, **while**, "{", "}", "(", ")", "=", ";". The top-down parser, if correctly constructed according to this grammar, should automatically build implicit parse trees in which each **else** matches the closest preceding unmatched **if**.