CS 363 – Homework #1 – Recursive-descent Parsing
Due Monday September 25, 2017

The purpose of this assignment is to practice scanning and parsing for a realistic programming language. You will write a program that takes as input a program written in a made-up language called Sumatra, and then translates it into Java. If you combine your parser with an ordinary Java compiler – we will have the ability to compile and run any program written in Sumatra.

The program will run interactively. Begin by asking the user for the basename of the Sumatra source file. Assume that the name of the input file will end with ".sum" and your output file should have the same basename with the ".java" extension. Your program will read the source file, scan (tokenize) it, and then parse it. Your program will emit Java code during the parse. Finally, write a message to standard output announcing that the parsing is finished.

Here is a hint on the scanning. In my implementation, I defined a Token class. This is because I found it convenient to conceptualize a token as having two attributes: a token-type and a lexeme. The token's type tells us what kind of token it is, and the lexeme is the string text of the token. For example, if the Sumatra program declares a variable called "n", the type of token would be an identifier, and the string "n" is the lexeme of the token.

The grammar of Sumatra is given on the next page, and you can adapt it directly into recursive-descent parsing functions. Most of these functions will also perform output of the corresponding Java code.

The natives of Sumatra do not believe in object-oriented programming. In fact, they don't like modularity either. So, your Java output will have just a single "main" function. Since the program will likely do I/O, your Java output program should declare a BufferedReader object called in. (Alternatively, you could use the Scanner class.) You may assume that the Sumatra program will not use "in" as an identifier. We are not performing any semantic checking, so you may assume that the Sumatra program has all its variables declared appropriately.

The Java code that your program produces should be indented appropriately and have vertically aligned curly braces as shown in the example output.

Finally, once you get the basic functionality to work, there is one more feature to add to your translator: Syntax error messages. A syntax error would be an unexpected or missing token. If your program finds an error, print an appropriate error message and immediately halt. The message should also include the line number where the error occurred. Don't work on this feature of your translator until you are sure that it already translates correctly. It is better to have a correct translator that doesn't emit syntax error messages, than a translator that can emit error messages but outputs incorrect code.

Here is the grammar for the Sumatra language. Some notation conventions are:

- The nonterminals are enclosed in angle brackets like this: <program>.
- Short tokens like '=' are enclosed in single quotes.
- Curly braces { } indicate that the enclosed terminals and/or nonterminals are optional. For example, an if-statement may contain an optional else portion.

```
<program>  →  <program_header> <declarations> <stmts> <program_footer>

<program_header>  →  program <identifier> ';'

<program_footer>  →  end program ';'

<declarations>  →  declare <varlist> ';' |

<varlist>  →  <identifier> { '[' <unsigned_number> ']' } { ',' <varlist> }

<stmts>  →  <stmt> | <stmt> <stmts>

<stmt>  →  <asg_stmt> | <read_stmt> | <print_stmt> | <if_stmt> |
           <while_stmt> | <do_stmt>

<asg_stmt>  →  <variable> ':=' <expr> ';'

<read_stmt>  →  read <variable> ';'

<print_stmt>  →  print <expr> ';'

<if_stmt>  →  if <cond> then <stmts> { else <stmts> } end if ';'

<while_stmt>  →  while <cond> do <stmts> end while ';'

<do_stmt>  →  do <stmts> until <cond> ';'

<cond>  →  <expr> <relop> <expr>

<relop>  →  '=' | '/=' | '<' | '>' | '<=' | '>='

<expr>  →  <variable> | { '-' } <unsigned_number> |
           '(' <expr> <arithop> <expr> ')'

<arithop>  →  '+' | '-' | '*' | '/'

<variable>  →  <identifier> { '[' <expr> ']' }
```

Here are some regular expressions to help you identify tokens:

```
<identifier>  →  <letter> ( <letter> | <digit> )*

<unsigned_number>  →  <digit>+

<letter>  →  'a' .. 'z' | 'A' .. 'Z'

<digit>  →  '0' .. '9'
```

Comments in Sumatra are enclosed in curly braces.

Here is an example program written in the Sumatra language. (bubblesort.sum)

```
program bubblesort;
declare A[10], i, j, n, tmp;

{ read in list of numbers until a -1 is encountered }

i := 0;
read n;
if n /= -1 then
  do
    A[i] := n;
    read n;
    i := (i + 1);
  until n = -1;
end if;
n := i;

{ sort the numbers in ascending order }

i := 0;
while i < n do
  j := (i + 1);
  while j < n do
    if A[i] > A[j] then
      tmp := A[i];
      A[i] := A[j];
      A[j] := tmp;
    end if;
    j := (j + 1);
  end while;
  i := (i + 1);
end while;

{ print the sorted list of numbers }

i := 0;
while i < n do
  print A[i];
  i := (i + 1);
end while;
end program;
```

Here is the same program, automatically translated into Java. (bubblesort.java)

```
import java.io.BufferedReader;
public class bubblesort
{
  public static void main(String [] args) throws IOException
  {
    BufferedReader in = new BufferedReader(new InputStreamReader(System.in));
    int [] A = new int [10];
    int i;
    int j;
    int n;
    int tmp;
    i = 0;
    n = Integer.parseInt(in.readLine().trim());
    if (n != -1)
    {
      do
      {
        A[i] = n;
        n = Integer.parseInt(in.readLine().trim());
        i = (i + 1);
      } while (! (n == -1));
    }
    n = i;
    i = 0;
    while (i < n)
    {
      j = (i + 1);
      while (j < n)
      {
        if (A[i] > A[j])
        {
          tmp = A[i];
          A[i] = A[j];
          A[j] = tmp;
        }
        j = (j + 1);
      }
      i = (i + 1);
    }
    i = 0;
    while (i < n)
    {
      System.out.println(A[i]);
      i = (i + 1);
    }
  }
}
```