CS 3520 - Programming Language Structures

Program: 4
Points: 50
Due Date: **Sunday**, **December 6, 11:59 pm**

You are going to implement a parser for the DOT language. This assignment will provide you with experience in parser development, specifically with the development of a recursive descent parser.

The parser takes as input a stream of tokens and determines whether those tokens follow the grammatical structure of DOT. That is, the parser should determine whether a DOT file is well-formed; if not, the parser should produce an error. The relevant source files for this task include the Ruby files created for the Lexer in program3 (token.rb and dot_lexer.rb), dot_parser.rb, and prog4.rb.

The parser you are going to build should be based on the following EBNF grammar that defines DOT. Please refer to the recursive descent algorithm we discussed in class to get an idea on how you might implement a parser for the EBNF grammar. Make sure you correctly handle the EBNF meta symbols "{}, [], and ()". A character or multiple characters wrapped within single quotes indicates a terminal symbol. Note that you do not have to implement a complete parser for DOT. However, your parser should be able to handle all test cases and sample inputs provided in appendix. All test cases are correct DOT code and should be successfully recognized by your parser.

```
graph -> ('digraph' | 'DIGRAPH') [id] '{' stmt_list '}'

stmt_list -> {stmt [';']}

stmt -> edge_stmt | id '=' id | subgraph

edge_stmt -> (id | subgraph) edge edgeRHS ['['attr_list']']

attr_list -> id ['=' id] {',' id ['=' id]}

edgeRHS -> (id | subgraph) {edge (id | subgraph)}

edge -> '->' | '--'

subgraph -> ('subgraph' | 'SUBGRAPH') [id] '{' {stmt_list} '}'

id -> ID | STRING | INT
```

In the file dot_parser.rb, you need to define an associated function for each non-terminal to match language constructs defined by the rules of the non-terminal. You may want to maintain a list of all tokens (including EOF) generated by the lexer to make lookahead easy. You may also need to maintain a read pointer to indicate the next lookahead token in the input token list.

Notes:

1. As shown in Test Case 1, a digraph has to have a name after the keyword "`digraph`" or "`DIGRAPH`". We call the body of a digraph as a cluster that may contain a set of properties, edges, and subgraphs. For example, "`rankdir=LR;`" is a property; "`0 -> "1" [label = "A"];`" is a complete edge statement where an arrow connects two nodes followed by an optional list of properties. The property list is delimited by a pair of brackets and separated by commas. A subgraph has its own cluster.
2. A node in DOT can be either an ID, an INT, or a string. Two operands in a property (e.g., `label = "A"`) can be either an ID, an INT, or a string.
3. To make your implementation simple, for DOT code with multiple syntax errors, you only need to print out the first error detected and stop parsing. To achieve this, you simply print an error message when a mismatch is found. Please refer to **Sample Input and Output 2** for more information on how to handle syntax errors.

4. This program MUST be done individually. Overly similar programs will result in 0 for all involved.
5. Make comments to your code whenever necessary.
6. Fully test your program before submission.
7. You will lose points if your final submission is after the due time:
   -5 if your final submission is by 11:59 PM, December 6
   -10 if your final submission is by 11:59 PM, December 7
   -15 if your final submission is by 11:59 PM, December 8
   Completion credit after 11:59 PM, October 20
8. For all programs in this class, you must turn in a running solution in order to pass the course. The solutions do not have to work correctly for all test cases but must solve a significant portion of the problem.
9. Match the output exactly and follow the Ground Rules below.

Ground rules for this program:

1. Variable and method names must be in lowercase with underscores between words in a name
2. Use spaces between operators
3. Comment at the top of each file saying what the class does and your name as the Author
4. One- or two-line comment for each method
5. Methods must be less than 40 lines long
6. Indent **two** spaces in all the normal places (if, while, for, etc.)
7. Line length MUST be 78 characters or less

0.5 points off for each violation. Maximum 5 points off for formatting errors.

Appendix

Test Case 1:

```
digraph NFA {
  rankdir=LR;
  0 -> "1" [label = "A"];
  0 -> "2" [label = "B", color = red];
}
```

Test Case 2:

```
DIGRAPH NFA {
  0 -> "1" [label = "A"];
  0 -> "2" [label = "B"];
}
```

Test Case 3:

```
digraph trees {
  subgraph t {
    0 -> "1" [label = "A"];
    0 -> "2" [label = "B"];
  }
  subgraph u {
    Animal -> Cat [label = "feline"];
    Animal -> Dog [label = "canine"];
  }
}
```

Test Case 4:

```
digraph trees {
  subgraph t {
    0 -> "1" [label = "A"];
  }
  SUBGRAPH u {
    Animal -> Cat [label = "feline", shape="record"];
    Animal -> Dog [label = "canine"];
    Animal -> Bird [label = "tweety", shape="record"];
  }
}
```

Test Case 5:

```
digraph G {
    main -> compare -> execute;
    main -> init;
    main -> cleanup;
    execute -> makeString;
    execute -> printf;
    init -> makeString;
    main -> printf;
    execute -> compare;
}
```

Test Case 6:

```
digraph t { }
```

Sample Input and Output 1:

```
digraph trees {
  rankdir=LR;
  subgraph t {
    0 -> "1" [label = "A"];
    0 -> "2" [label = "B"];
  }
  SUBGRAPH u {
    Animal -> Cat [label = "feline", shape="record"];
    Animal -> Dog1 [label = "canine"];
  }
}
Start recognizing a digraph
Start recognizing a cluster
Start recognizing a property
Finish recognizing a property
Start recognizing a subgraph
Start recognizing a cluster
Start recognizing an edge statement
Start recognizing a property
Finish recognizing a property
Finish recognizing an edge statement
Start recognizing an edge statement
Start recognizing a property
```

```
Finish recognizing a property
Finish recognizing an edge statement
Finish recognizing a cluster
Finish recognizing a subgraph
Start recognizing a subgraph
Start recognizing a cluster
Start recognizing an edge statement
Start recognizing a property
Finish recognizing a property
Start recognizing a property
Finish recognizing a property
Finish recognizing an edge statement
Start recognizing an edge statement
Start recognizing a property
Finish recognizing a property
Finish recognizing an edge statement
Finish recognizing a cluster
Finish recognizing a subgraph
Finish recognizing a cluster
Finish recognizing a digraph
```

Sample Input and Output 2:

```
digraph G {{ ->
    main -> compare -> execute;
    main -> init;
    main -> cleanup;
    execute -> makeString;
    execute -> printf;
    init -> makeString;
    main -> printf;
    execute -> compare;
}
Start recognizing a digraph
Start recognizing a cluster
Error: expecting property, edge or subgraph, but found: {
```