Project 1 . Building a Scanner

1 Important information

- **Team:** You will work by yourself.
- **Dates:**
  1. Report/design due 11:59 pm 03/05/2021.
  2. Interview (10 minutes) on 03/06/2021 and 03/07/2021. Specific meeting time will be decided later.
  3. The project due 11:59 pm 03/26/2021.
- **Grade:** 8% of the final grade of this course
- **Deliverables:** One report and one program.
- **Grading:** design/report 50% and program50%
- **Submission:**
  1. Submit your design/report (either PDF or text files) to the Blackboard. name of your file: "Proj 1" + your first name + your last name.
  2. Final submission will be a zip file submitted to the Blackboard with name of the form: "Proj 1" + your first name + your last name.
     (a) A zip file consisting of
        i. The source code for the scanner which must be compilable to an executable file using command line instead of IDE. For C/C++: source code must be gcc/g++ compilable.
        ii. A readme file on how to compile the source code and how to run the executable.
        iii. Your revised report/design.
     (b) Name of the zip file: If your name is "Fred Jones" then name your zip file: "FJones-Proj1.zip".

2 The scanner

The scanner is for the tokens specified by the regular expression (including production rule for *comment*) given in page 54 of the text book.

You **must** use the DFA on page 57 as the basis for designing and implementing your scanner. Note that the DFA does not fully respect the production rule on *id* (i.e., the DFA will take `read` and `write` as *id* while they should be taken as a token of *read* and a token of *write* respectively). So, your scanner returns an ID token only if the id is not `read` or `write`. When the id is `read` or `write`, your scanner should return the token *read* and *write* respectively.

3 The task

Here is the task: *write a scanner using an imperative language: C/C++ (***in this option, your program must be compilable using gcc or g++.***), Java and Python.*

Your program **must have** the `scan` function. Its precondition is that the current pointer of the input file is NOT at the end of file (EOF). It scans from the current pointer of the input file and stops when either a valid token or invalid token can be decided. In the case of a valid token, `scan` returns the token type (i.e., the left hand side symbol of the corresponding production rule. E.g., for production rule `id -> letter letter^`, the token type of an input "abc" would be `id`); otherwise, it returns an error flag. Note here when deciding a token you must use the "longest possible token" rule. Your scanner should also deal with the `read` and `write` token correctly.

The scanner takes the name of a text file from the *command line*. It outputs to the console `error` if there is *any* non-valid token in the input file; the list of tokens otherwise.

Your implementation should be based on the automata in on Page 54 of the textbook.

Commandline Invocation: `scanner <input file name>`

If there is any non-token string in the input file, your program should exit and output to the console only:

```
error.
```

Otherwise, your program outputs only the list of tokens in the form of

```
(token1, token2, ...).
```

Example Input File: foo.txt with the following content:

```
read
/* foo
   bar */
*
five 5
```

Example Commandline: `scanner foo.txt`

Example Output: `(read, times, id, number).`

Note that your scanner does NOT output the token of *comment*. In fact, your function `scan` should completely ignore all comments in the input. In the example, the next token of `times` is `id`, instead of *comment*.

4 A More Challenging Task

One who needs more challenges may choose this task, instead of the previous task in Section 3.

In this task, the input of your program is a file containing the description of an automata, and a file F_2 containing a program.

If there is any non-token string in F_2, your program should exit and output to the console only:

```
error.
```

Otherwise, your program outputs only the list of tokens in the form of

```
(token1, token2, ...).
```

You must have a `scan` function whose input is an opened file, and output is the longest possible token type of the token starting from the current point of the input file. In this question, we talk about token type only. If you are interested in working out the token content for each token, the content should never contain white spaces separating the token from other tokens or comment. Remember, in this project, we don't take comment as a token. We simply ignore them when recognizing tokens in an input program. For example, input `/*123*/abc` has only one token whose content is `abc`.

The file containing the automata is a text file with 6 sets in order:

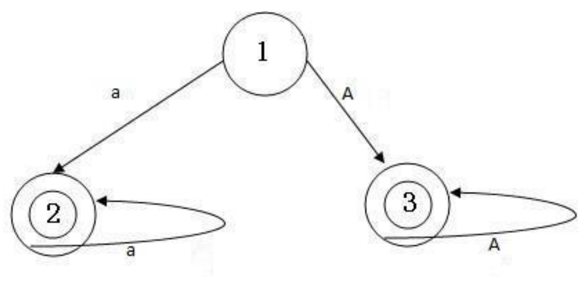
```
{c1,c2,...,ci,...}
{s1,s2,...,sj,...}
{i1}
{f1,f2...}
{(si,ck,sj),...}
{(f1,t1),(f2,t2),...}
```

Every set starts with {and ends with .} The first set lists all characters that a programmer can use EXCEPT the THREE characters '{', '}' and ','. The characters are separated by ','. The second contains all states of the automata which are separated by ','. Each state is a number with possibly more than two digits. The third set has a single element inside it which is the initial state of the automata. The fourth set includes all final states. The fifth is a set of triples each of which is of the format (*si*, *ck*, *sj*), where *ck* can be any member from the first set, that represents the transition that the character *ck* brings the automata from state *si* to state *sj*. The last set indicates the token type of each final state. Specifically, an element (*fj*, *tj*) of the set means that the token type of the final state *fj* is *tj* which is a sequence of letters.

The following is an example of a file that represents the automata in the picture below.

```
{a,A}
{1,2,3}
{1}
{2,3}
{(1,a,2),(2,a,2),(1,A,3),(3,A,3) }
{(3,captal),(2,low)}
```

In this automata, only characters *a* and *A* are allowed. There are 3 states: 1, 2, and 3. 1 is the initial state; both 2 and 3 are final states. State 2 and 3 represent token type *capital* and *low* respectively.



5 The report/design and implementation

Report. The report consists of four components.

1) Specify the important data structure(s) you will use in your scanner algorithm. The specification should be independent of any specific programming language such as JAVA. Here is an example of specifying a data structure.

To specify a data structure for the table in Fig 2.12 in the textbook, we need the following data structure with a name `transitionTable`. `transitionTable` is a two dimensional array. The first dimension is indexed from 1 to 18 (the number of states) and the second dimension is indexed from 1 to 14 with 1 representing white spaces (space, tab, newline), 2 representing the character `/`, ·,· 14 representing any letter. For any integer *i* and *j*, `transitionTable[`*i*`][`*j*`]` is a record with field names `action` and `newState`. `action` can take values of `move, recognize, stuck`. `transitionTable[`*i*`][`*j*`].action=move` means that the automata should move to next state and `transitionTable[`*i*`][`*j*`].newState` is the state to move to. `transitionTable[`*i*`][`*j*`].move = recognize` means that *i* is a final state and the automata can not move to any other state with the input character corresponding to the number *j*, i.e., we recognize a token! `transitionTable[`*i*`][`*j*`].move = stuck` means that the the automata can not get to any state from state *i*with a character corresponding to the number *j*.

2) The pseudo-code. For each function in your pseudo-code, you MUST have input, output and how output is related to input. As for pseudocode statements you should use to describe your algorithm, please refer to those in Fig 2.11 and the sample pseudo-code given on our website. The principle is not to use statements in a specific language such as JAVA, but your statements are still precise and understandable by a student who has taken CS1411, i.e., has basic knowledge about programming. Note that in your pseudo-code, you can use statement such

`If transitionTable[`*i*`][`*j*`]`.*action* `is move.`

Your pseudo-code should cover the major idea how you recognize a token, reflect the "longest possible" principle to find a token, how the tokens `read` and `write` are recognized etc.

3) The test cases to test the correctness of your program with explanation why you select them.

4) Acknowledgment of people and their contribution to your project.

Implementation. As for implementation (program), the minimal expectation is a finished implementation which means the source code is compilable to an executable and the executable behaves correctly at least most of the time.

6 Other Project Requirements

You must read 2.2.2 and 2.2.3 for this project. Your pseudo-code and implementation MUST be based on methods mentioned in those sections.

7 Grading

The grading will be based on the extent to which the requirements above are met and your understanding of the scanning algorithm. For example, one can expect almost no credit if his/her scanner is implemented without reflecting the process on how an automata is used to recognize a token (, which is covered in full detail in the class); if the principle of longest possible token is not followed, one can expect a significant deduction of grades etc. Your understanding of scanning algorithm and implementation is reflected from your report and the face to face interview.

Note, one might have done some projects which seem to be similar. Don't use the solution for those projects as a solution here simply because 1) it may not meet the requirements in this project and 2) we have learned new and better general ideas for this project in this class.

8 Acknowledgment

Edward Wertz, Xiaozhen Xue and Samira Talebi have contributed to the design and writing of the project spec- ification (section 3 and 4).