CS-4700/5700 Automata, Computability, and Formal Languages
PJ03.docx

PJ03 – JACK lexer

In this project, you will develop a DFA-based lexer for the JACK language. The language specification is contained in Chapter 10 of **The Elements of Computing Systems** by Noam Nisan and Shimon Schocken, ©2005, MIT Press. The entire chapter has been provided to you with the permission of the authors.

The lexical elements are defined in the first panel of Figure 10.5 on page 208. However, these categories are a too broad for the level we are focusing on in this course. We need to further divide the keyword and symbol token categories so that the language grammar (to be covered in the next set of projects) can use token categories as terminals without regard to the specific token value.

Furthermore, we will use source files that are purely ASCII encoded, so you may ignore references to Unicode.

Your program will scan the input file one character at a time, identifying tokens and outputting them to a file, one token per line, with the token type followed by a comma, a space, and then the token value which is the literal string from the JACK file that was accepted. The name of the file should be the same as the input .jack file except with a .tok extension.

Your program must adhere to the "max munch" principle, which is that the longest token that can be recognized is the token that WILL be recognized. For example, the string "letfred" will be recognized as a single identifier token and not as the keyword 'let' followed by the identifier token 'fred', even if the first one would ultimately result in an invalid program and the second one would result in a valid program.

Because this IS a course on finite automata, you are NOT allowed to use ANY form of regular expression processing tool or libraries nor to use any form of substring searches. You are to have clearly defined DFA machines that process the input strings one character at a time.

Jack files can have comments. These must be recognized but are NOT to be tokenized (exported to the output file). They are to be discarded. Comments are either EOL or BLK comments.

EOL: **CRLF** | **LF**

COMMENT_EOL: "**//**"' (any characters except EOL) EOL

COMMENT_BLK: "**/*** (any characters except "***/**") "***/**"

Whitespace (spaces, tabs, newlines) are ignored except that a token cannot span them (i.e., they can delimit tokens, but cannot be a part of a token)

CS-4700/5700 Automata, Computability, and Formal Languages
PJ03.docx

JACK Token Categories

- KW_CONST: **true** | **false** | **null** | **this**
- KW_TYPE: **int** | **char** | **boolean**
- KW_VARDEC: **static** | **field**
- KW_SUBDEC: **constructor** | **function** | **method**
- KW_VAR: **var**
- KW_VOID: **void**
- KW_CLASS: **class**
- KW_LET: **let**
- KW_IF: **if**
- KW_ELSE: **else**
- KW_WHILE: **while**
- KW_DO: **do**
- KW_RETURN: **return**
- SY_LPAREN: '**(**'
- SY_RPAREN: '**)**'
- SY_LBRACKET: '**[**'
- SY_RBRACKET: '**]**'
- SY_LBRACE: '**{**'
- SY_RBRACE: '**}**'
- SY_SEMI: '**;**'
- SY_PERIOD: '**.**'
- SY_COMMA: '**,**'
- SY_EQ: '**=**'
- SY_MINUS: '**-**'
- SY_NOT: '**~**'
- SY_OP: '**+**' | '*****' | '**/**' | '**&**' | '**|**' | '**<**' | '**>**'
- IDENT: A sequence of letters, digits, and underscores that does not start with a digit and is not a keyword.
- INTEGER: A decimal value in the range 0..32767 (inclusive).
- STRING: A sequence of ASCII characters (not to include a double-quote or newline) enclosed in double-quote characters.
- BADTOKEN: Anything that cannot be recognized as a valid token (as few characters as possible).

CS-4700/5700 Automata, Computability, and Formal Languages
PJ03.docx

SUBMISSION

You need to submit a single ZIP file. At the top level should be a single-spaced report (in PDF format) that is no more than five pages long (two to three is preferred). This report should focus on describing the technical approach taken in implementing your lexer. In particular, you should describe how you implemented the one-character-at-a-time processing constraint and the max-munch criteria. Included in this description should be a short section indicating the programming language and/or development environment that you used and any special configuration settings needed by someone that would like to reproduce your program as well as how to run your program.

There should also be a two folders at the top level. The first is named "results" and should contain your output .tok files.and log files for all of the machines. It does not need to have any other files, including the original .jack files. There should also be a folder named "code" that contains all of the source code for your program. Included should be all files needed by someone to reproduce your program assuming they have the appropriate development environment. This folder can contain subfolders as appropriate.

Keep in mind that the grader is very unlikely to even attempt to run your program – they likely will not know the language you used, let alone have the necessary tools. Thus it is important that your report file describe your approach and algorithm very clearly AND that your code be well organized and commented – the grader WILL review your code.

GRADING CRITERIA

Report/Code: 50%

- 70% Technical merit – algorithm is well explained.
- 10% Programming environment details well explained.
- 10% Code is well organized
- 10% Code is well documented

Token Lists: 50%

Each file will count equally toward this portion of the grade.

For EACH .jack file:

- 2% for each token category that has the correct overall count, regardless of order.
- -1% for each missing/extra token in the ordered token list (-50% max).

Note that since order counts, missing or extra tokens can quickly result in a very low score.

CS-4700/5700 Automata, Computability, and Formal Languages
PJ03.docx

A Word of Warning

It might be tempting to force your log files and output files to be what you know or believe they should be in the event that you do not get your code working completely. AVOID this temptation!

The grader will be looking at your algorithm for correctness and reviewing your code looking to see if it matches your algorithm. Any errors in the algorithm and code that should result in erroneous results BETTER result in those erroneous results. If the results in the files you submit for grading are demonstrably NOT the result of the algorithm/code that you submitted, you will receive a grade of -100% for the entire project (not just this mini-project, but all of them) and this score WILL count toward your final grade.

So DO NOT DO IT! It is FAR better to submit the files that result from running your code and take the hit or, for that matter, to simply not turn in the project at all, than to falsify the results.