CS-4700/5700 Automata, Computability, and Formal Languages
PJ04.docx

PJ04 – JACK lexer

In this project, you will develop a Recursive Decent parser 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 (referred to as ECS). 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, as in the previous Lexer project, 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. This makes the implementation of a recursive decent parser considerably easier.

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

Your program will use as its input the token list output from the previous Lexer project (i.e., the .tok files). Recall that these files contain 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.

IF the grammar is LL(k), you can write a parser that produces a reads the input string from left-to-right (the first L) and produces a leftmost derivation (the second L) by considering, at most, the next (k) token in the list. For this to happen, it must be possible for your parser to know which production to invoke next with only the knowledge of which variable is presently being processed, which production of that variable is being processed, where within that production the processing is occurring, and the identity of the next k tokens in the token list.

The grammar, as presented by the ECS authors, is technically an LL(2) grammar. However, it is a fairly straightforward manipulation to massage it into an LL(1) grammar, which is easier to implement.

Because the output of a parser is either a derivation or a parse tree, we want an output format that makes this easy for the reader (e.g., grader) to follow.

As a note on terminology, there are two camps with regards to the interpretation of the 'k' value in an LL(k) parser. Nearly everyone refers to the 'k' as the amount of "look ahead" needed by the parser. Where the ambiguity comes in is that if it is sufficient for the parser to only look at the next token, how many tokens is it "looking ahead". Some people, including the ECS authors, contend that you are not looking ahead at all, and hence call this an LL(0) parser. But the more widely accepted interpretation is that ‘k’ refers to the total number of tokens that must be available, including the next token. For these people, a parser that only needs to look at the next token is an LL(1) parser and an LL(0) parser is nonsensical since it would have to be able to parse the input without looking at any tokens.

We will stick with the more widely used convention, even though it is at odds with the ECS usage. You need to bear this in mind when reading the ECS material.

The output of your parser will be an XML file (using the same base failname but with an .xml extension) as specified in the ECS material in Section 10.2.4. Notice that the variables and token categories allowed in the XML file are a subset of those defined in the language grammar (ECS Figure 10.5), let alone the broadened set of token categories used in the Lexer in the prior project. This should not cause any significant problems if you keep in mind the hierarchical nature involved.

Also note that the requirements of the XML file specification conflict with some of the symbols used in the Jack language. The details, including how to deal with it, are discussed in ECS Chapter 10 on page 220.

<u>SUBMISSION</u>

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 parser. In particular, it should contain the entire definition of the grammar your parser is matched to, focusing on the modifications to the grammar that you made in order to make it LL(1).

Also included 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 .xml files. It does not need to have any other files, including the original .jack or .tok files, but it may contain them. 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

XML Files: 50%

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

For EACH .tok file:

- 5% for each XML category that has the correct overall count, regardless of order.
- -1% for each missing/extra variable/terminal in the tree (-50% max).

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

Since there are 15 non-terminals and 5 terminals that constitute the XML content, if your output is correct this portion of the grade will start at 100%. It can then lose up to 50% if the ordering of the content is incorrect.