Project P1 Version B

Daniyal Javed cse31034

July 27, 201

* Introduction

This report is written on the version B of the project for EECS3301 and it is programmed in the programming language TCL/Tk.

* Comments on your experience with this assignment (explain if anything went wrong with the design and implementation)

This project was a very big challenge to complete in Tcl/tk. This was my first time writing something substantial in a scripting language. One of the more serious shortcomings I found of Tcl/Tk was the lack of a rich set of data structures. The only true data types in Tcl are *strings* and *associative arrays*. The *string* type is also used to provide support for integers and floating point numbers and for lists as well. However, there are several possible workarounds to these inherent problems.

The first part was to illustrate lexical-analyzer construction with a state diagram. The state diagram includes states and

transitions for each and every token pattern. However, this approach, according to the book results in a very large and complex diagram, because every node in the state diagram would need a transition for every character in the character set of the language being analyzed. Thus the book simplifies it further. We need a lexical analyzer that recognizes only arithmetic expressions, including variable names and integer literals as operands. All characters of the English language have 52 different characters which include both upper and lower class. A lexical analyzer is interested only in determining that it is a name and is not concerned with which specific

name it happens to be. Therefore, we define a character class named LETTER for all 52 letters and use a single transition on the first letter of any name.

Another opportunity for simplifying the transition diagram is with the

integer literal tokens. 10 characters could begin an integer lexeme thus we named it as DIGIT.

The second part was implementing the state diagram made above into Tcl code. First we process the input from the user. We then parse this input and create a function to lookup operators and parentheses. We separated functions to parse LETTERS, DIGITS and OPERATORS and return them at the EOF.

For part three we have to come up with a Recursive-Descent Parser for the grammar. A recursive-descent parser consists of a collection of subprograms, many of which are recursive, and it produces a parse tree in top-down order.

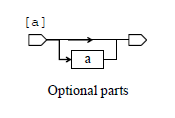
For the last part we had to represent the input from the RDP in the form of graphics. This was the hardest part of the assignment for me and the group members as this is very complex to problem to implement in a brand new language with such little time.

* Results of several test-runs of the application

// tests screen shots

* Graphical representation of optional components, e.g. [a].

An optional item is enclosed between [ and ] (square brackets); the item can either be included or discarded. It is represented by this graphic:

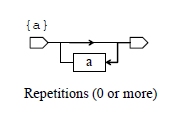


From the start the process could end or it could go to ‘a’ meaning that it is optional.

In a sequence, we must go through each item. In an option we must go through either the bottom rung including the item, or the top rung discarding it.

* Graphical representation of repetitive components, e.g. {a}. Outline the pros and cons of graphical representations with and without inversion of concatenation order.

The repeatable item is enclosed between { and } (curly–braces); the item can be repeated zero times. It is represented graphically:



//The first shows a repetition of two alternatives, where any number of intermixed As and Bs are legal: AAA, BB, or BABBA. A different choice can be made for each repetition. The second shows two alternatives where a repetition of As or a repetition of Bs are legal: AAA or BB, but not AB: once the choice is made, only one of these characters can be rep