Project P1 Version B

Daniyal Javed cse31034

July 27, 201

1 Introduction

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

2 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.

3 Results of several test-runs of the application

s2.tcl:

```
red 13 % tclsh s2.tcl
Enter an EBNF rule to parse:
a = b | c
Next lexeme is a . Next token is IDENTIFIER
Next lexeme is = . Next token is EQUALS_OP
Next lexeme is b . Next token is IDENTIFIER
Next lexeme is | . Next token is ALTERNATE_OP
Next lexeme is c . Next token is IDENTIFIER
red 14 %
```

```
red 15 % tclsh s2.tcl
Enter an EBNF rule to parse:
a = x
Next lexeme is a . Next token is IDENTIFIER
Next lexeme is = . Next token is EQUALS_OP
Next lexeme is x . Next token is IDENTIFIER
red 16 %
```

```
red 21 % tclsh s2.tcl
Enter an EBNF rule to parse:
x = "a" | "aa"
Next lexeme is x . Next token is IDENTIFIER
Next lexeme is = . Next token is EQUALS_OP
Next lexeme is "a" . Next token is TERMINAL
Next lexeme is | . Next token is ALTERNATE_OP
Next lexeme is "aa" . Next token is TERMINAL
red 22 %
```

s3.tcl:

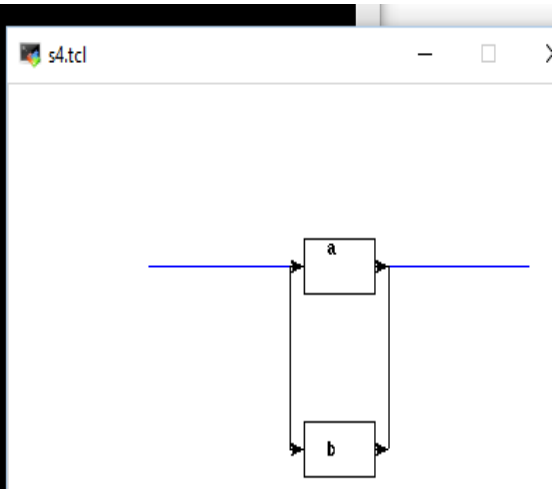
```
red 24 % tclsh s3.tcl
Enter an EBNF rule to parse:
a = b | c ;
Enter rule
Enter lhs
Next lexeme is a . Next token is IDENTIFIER
Exit lhs
Next lexeme is = . Next token is EQUALS_OP
Enter rhs
Enter term
Next lexeme is b . Next token is IDENTIFIER
Exit term
Next lexeme is | . Next token is ALTERNATE_OP
Enter rhs
Enter term
Next lexeme is c . Next token is IDENTIFIER
Exit term
Next lexeme is ; . Next token is END_OF_RULE
Exit rhs
Exit rhs
Exit rule
red 25 %
```

```
red 31 % tclsh s3.tcl
Enter an EBNF rule to parse:
a = d ;
Enter rule
Enter lhs
Next lexeme is a . Next token is IDENTIFIER
Exit lhs
Next lexeme is = . Next token is EQUALS_OP
Enter rhs
Enter term
Next lexeme is d . Next token is IDENTIFIER
Exit term
Next lexeme is ; . Next token is END_OF_RULE
Exit rhs
Exit rule
red 32 %
```

```
red 32 % tclsh s3.tcl
Enter an EBNF rule to parse:
a = "aa" | "bb" ;
Enter rule
Enter lhs
Next lexeme is a . Next token is IDENTIFIER
Exit lhs
Next lexeme is = . Next token is EQUALS_OP
Enter rhs
Enter term
Next lexeme is "aa" . Next token is TERMINAL
Exit term
Next lexeme is | . Next token is ALTERNATE_OP
Enter rhs
Enter term
Next lexeme is "bb" . Next token is TERMINAL
Exit term
Next lexeme is ; . Next token is END_OF_RULE
Exit rhs
Exit rhs
Exit rule
red 33 %
```

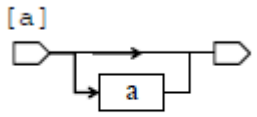
s4.tcl:

```
red 10 % wish s4.tcl
Enter an EBNF rule to parse:
c = a | b ;
Enter rule
Enter lhs
Next lexeme is c . Next token is IDENTIFIER
Exit lhs
Next lexeme is = . Next token is EQUALS_OP
Enter rhs
Enter term
Next lexeme is a . Next token is IDENTIFIER
Exit term
Next lexeme is | . Next token is ALTERNATE_OP
Enter rhs
Enter term
Next lexeme is b . Next token is IDENTIFIER
Exit term
Next lexeme is ; . Next token is END_OF_RULE
Exit rhs
Exit rhs
Exit rule
```



4 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:



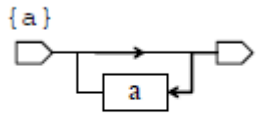
Optional parts

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.

5 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:



Repetitions (0 or more)

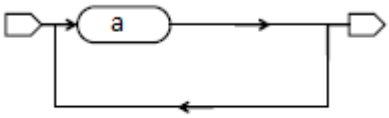
Advantage of using concatenation is that the item can repeat as many times as it wants or not repeat at all.

This is also a disadvantage as we cannot impose repetition on the item.

7. a,{a} (one or more repetitions of a) is a concatenation of a and {a}. Discuss an "improved" graphical representation of a,{a} that looks different from direct concatenation

We take an example that was discussed in class of letter{letter | digit} and we apply it for a{a}. This means that there can be 1 or more repetitions of a, graphically we represent it as:

To simplify it even more and to make it more optimal I came up with:



This is an improved graphical representation of a and {a}.