Assignment #1

MacMillan, Kyle

October 10, 2018

Contents

Title

Table of Contents i

List of Figures ii

List of Tables ii

1 Problem 1 1
- 1.1 Compilers and Interpreters . . . 1
- 1.2 Statically and Dynamically Typed Languages . . . 1
- 1.3 JIT Compiler and Interpreters . . . 1
- 1.4 Top-Down and Bottom-Up Parsing . . . 1

2 Problem 2 2
- 2.1 Minimum-State DFA . . . 2
- 2.2 Input String: abbcabcba . . . 2
- 2.3 Input String: abbcababa . . . 2
- 2.4 Regular Expression for L . . . 2
- 2.5 Regular Expression generate: abbcabcba . . . 3

3 Problem 3 4
- 3.1 LL1 Grammar . . . 4
- 3.2 Generation Explanation . . . 4
- 3.3 Proof of LL(1) . . . 4
- 3.4 Predictive Parse Table . . . 4
- 3.5 Predictive Parser: abbbaa . . . 6

4 Problem 4 7
- 4.1 Lex/Yac Interpreter . . . 7
- 4.2 Calculator Output . . . 7

5 Problem 5 10
- 5.1 Turing Machine Double Click . . . 10

List of Figures

1 Minimum-State DFA 2
2 Evaluating abbcabcba 3
3 Evaluating abbcababa 3
4 LL1 Grammar “ab” 5
5 LL1 Grammar “abbbaa” 6
6 Lex Code 7
7 Yacc Code 8
8 Lex/Yacc parse + 1 - 2 3 9
9 Lex/Yacc parse + 1 - 2 9

List of Tables

1 Predictive Parse Table 4

1 Briefly explain the difference between

1.1 A compiler and an interpreter

Compilers are different from interpreters or they would not be separate things. In general terms a compiler takes something and turns it into something else. An interpreter does the same thing but not as efficiently. The compiler we use to compile C++ will read through our code and reduce it to assembly instructions for the machine we are on. While doing this it will evaluate some parts of the code and place things in an "optimal" ordering. An interpreter does not look at the code until execution, meaning it has no room to perform optimizations. Another difference is that interpreters are able to take code on the fly, whereas a compiled language must have everything at compile time.

1.2 A statically typed language and a dynamically typed language

Statically typed languages strictly enforce *type*, whereas dynamically typed languages determine which *type* to use. There are two primary benefits to statically typed languages:

- type-safe – You must declare the type and it will be enforced
- speed – Knowing the type ahead of time means it doesn't have to figure it out on the fly

A dynamically typed language is able to determine the type on the fly which reduces development time but can also be confusing. In Python, for example, a function with a variable named "flow". A user may not initially know if flow is a string such as *'fast', 'normal'* or *'slow'*, or a float such as a flow rate. This confusion can make maintenance more difficult. It can also hamper someone trying to learn the codebase. There are pros and cons of both that need to be taken into consideration when choosing a language.

1.3 A just-in-time compiler and an interpreter

A just-in-time (JIT) compiler compiles as you send it commands, meaning at runtime. A JIT, such as what Java uses, will take the code and compile it to bytecode. An interpreter executes instructions. A JIT will have marginally better performance compared to an interpreter.

1.4 Top-down parsing and bottom-up parsing

Top-down (TD) and bottom-up (BU) parsing refers to the methods of constructing a parse tree. As can be assumed, they each begin at their respective "ends" of a tree. TD parsers can be attacked with a predictive *LL(k)* method where *LL* means it descends the left nodes and has a look-ahead of *k*. BU parsing involves constructing a parse tree from a given input by starting at the bottom, ascending towards the root. BU parsing can use a shift-reduce method where the input string is reduced by shifting out the left-most element of the input to be evaluated. A tool for both Top-Down and Bottom-Up parsers is the **FIRST** and **FOLLOW** method.

2 Let L be the language $\{abxba \mid x$ is any string of a's, b's, and c's not containing the substring $ba\}$

2.1 Construct a minimum-state deterministic finite automaton for L

The wording on this is a little ambiguous and I'm going to take it to mean that the string denoted by x can be an empty string because technically the empty string does not contain ba. It could also be interpreted that the string x must contain at least one a, b, and c. This assignment is hard/long enough as is and I don't believe that was the intent.

Any string of a's, b's, and c's not containing the substring ba could be rewritten as:

$$y = (a^*b^*c^*)^* \tag{1}$$

$$x \subset y \mid \{ba\} \cap x = \varepsilon \tag{2}$$

In order for ba to not intersect x we have to ensure the minimum-state DFA cannot go from $b \rightarrow a$. The visualization is dentoted in Figure 1. Blue denotes the x section and orange are empty sets. Of special note is that where b terminates it cannot pass ε.

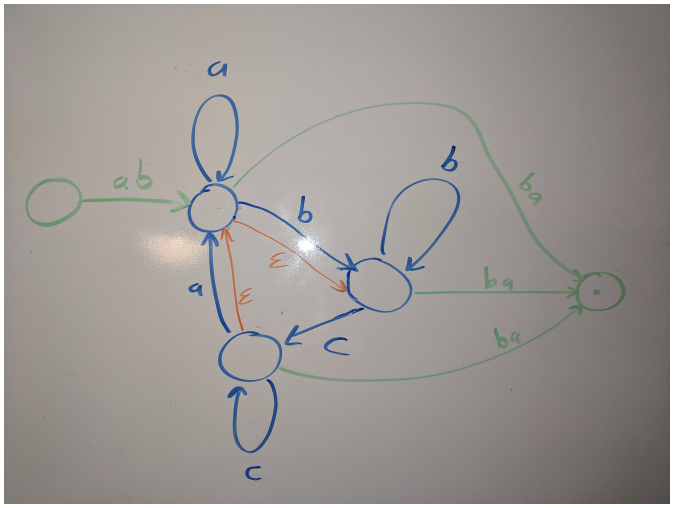

Figure 1: Minimum-State DFA

2.2 Show how your DFA processes the input string abbcabcba

ab is the standard entry point, which leaves us with $bcabcba$. While sitting at the a node it will advance to b, then to c, and back to a, leaving us with $bcba$. From a it will advance to b, then to c, exiting the "x" portion of our DFA, following ba to q_{final}. This process is outlined in Figure 2.

2.3 Show how your DFA processes the input string abbcababa

ab is the standard entry point, which leaves us with $bcababa$. While sitting at the a node it will advance to b, then to c, and back to a, leaving us with $baba$. From a it will advance to b, there is no path from b to a, so it will hop to q_{final} and still have characters left to evaluate, creating a critical failure. The process is outlined in Figure 3.

2.4 Construct a regular expression for L

$$ab(a|b^*c|c)^*b^*ba \tag{3}$$

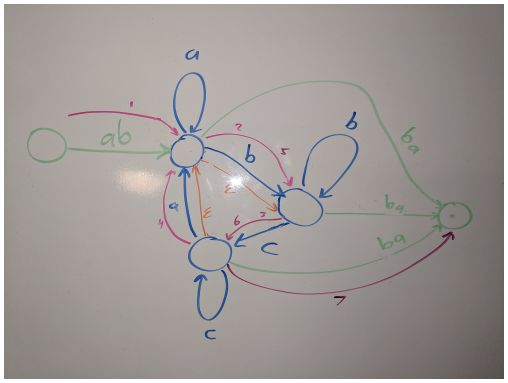

Figure 2: Evaluating abbcabcba

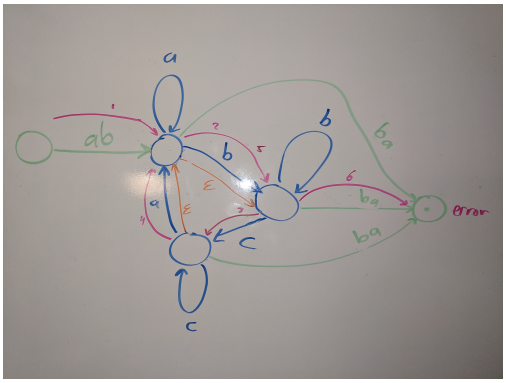

Figure 3: Evaluating abbcababa

2.5 Show how your regular expression generates the input string abbcabcba

ab	$\rightarrow$	ab	ab
bc	$\rightarrow$	b^*c	abbc
a	$\rightarrow$	a^*	abbca
bc	$\rightarrow$	b^*c	abbcabc
ba	$\rightarrow$	ab	abbcabcba

3 Let L be the set of all strings of a's and b's with the same number of a's as b's

3.1 Construct an LL(1) grammar that generates L

$$
\begin{aligned}
\text{V} = \quad & \{S, A, B\} \\
T = \quad & \{a, b\} \\
P = \quad & \{ \\
& \qquad S \to SASBS \mid S \to SBSAS \mid S \to \varepsilon \\
& \qquad A \to a \\
& \qquad B \to b \\
& \} \\
S = \quad & \{S\}
\end{aligned}
$$

Where each of those components is a part of the grammar defined by:

$$G = \langle V, T, P, S \rangle \tag{4}$$

3.2 Explain how your grammar generates L

Let's assume the simplest case: ab. Figure 4 shows the steps taken to generate input string ab with the given grammar. We start with S since it is our *start position*. Given a look ahead of 1 we can assume the production rule $S \to SASBS$ instead of $S \to SBSAS$, given the prediction rules in Table 1. From there we walk each step, taking the appropriate prediction. An important note is that the implementation of this grammar needs to look at the full stack.

For example in Figure 4 we can see on the second line:

$$SASBS \quad ab \quad S \to \varepsilon$$

Observe that ε is chosen. This is because $LL(1)$ sees an a on the input, and when evaluating it's stack it sees an A immediately after the left-most S, that means it does not need to pull in another $SASBS$. Compare that to Figure 5's line:

$$SAS \quad baa \quad S \to SBSAS$$

The input variable seen by $LL(1)$ is b, and the variable following S is A, therefore it follows the prediction table. A more cleanly written grammar wouldn't need the added complexity but eventually you have to give up and take a solution that works.

3.3 Prove that your grammar is LL(1)

By Table 1 we can see there are no cells containing more than one production rule, therefore we assume the grammar is LL(1).

3.4 Construct a predictive parsing table for your grammar

Table 1: Predictive Parse Table

	a	b	\$
S	$S \to SASBS$	$S \to SBSAS$	ε
A	$A \to a$		
B		$B \to b$	

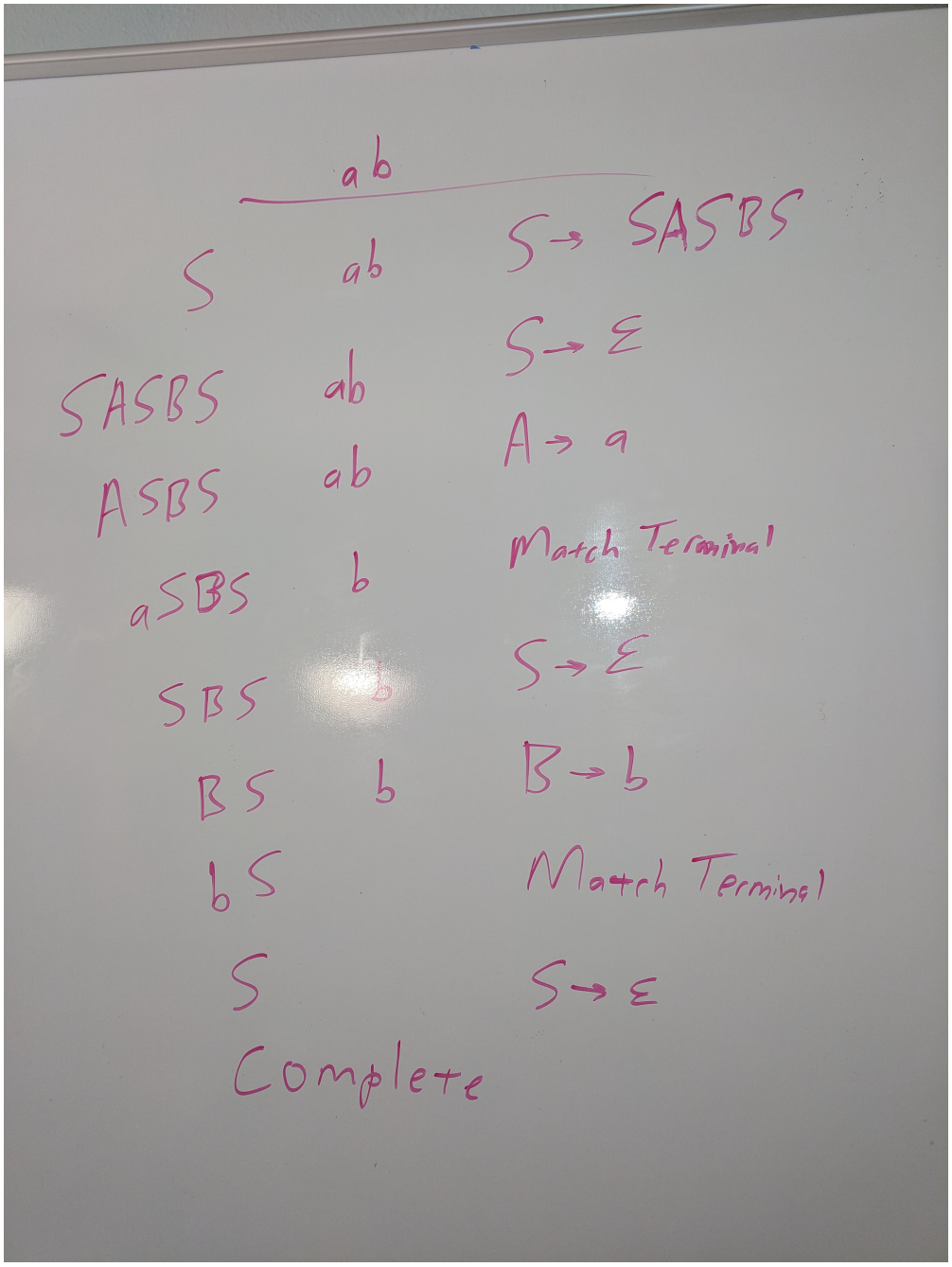

Figure 4: Simple LL1 grammar where the input string is “ab”

3.5 Show how your predictive parser processes the input string abbbaa

Figure 5 details the walkthrough of input string *abbbaa*.

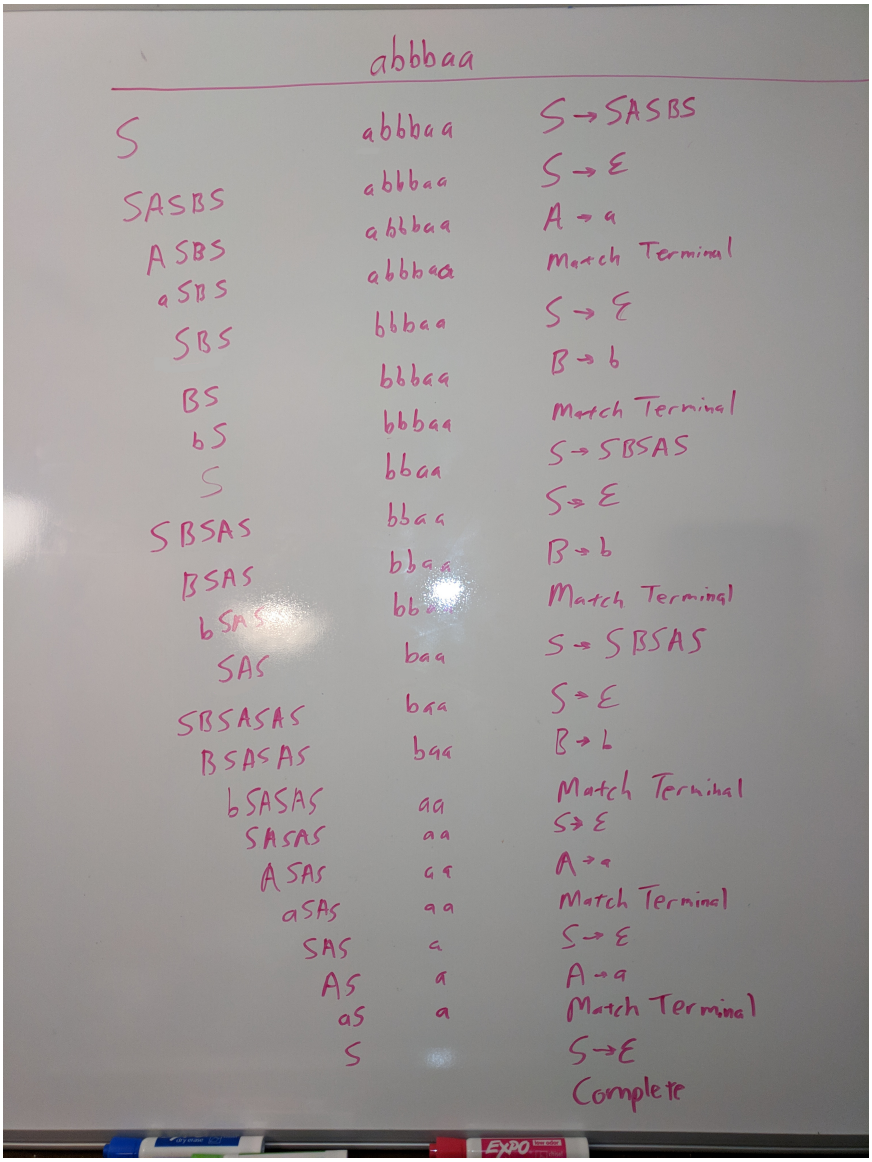

Figure 5: LL1 grammar where the input string is “abbbaa”

4 Interactive desk calculator for prefix expressions

4.1 Using Lex and Yacc or their equivalents, implement an interpreter that evaluates newline-terminated input lines of prefix arithmetic expressions. The expressions may contain integers and operators for addition, subtraction, multiplication, division, and negation. The answers are to be integers. Show the Lex-Yacc code for your calculator

No matter what I tried I could not figure out how to get an "operator" parse rule to work so the code is not what I wanted, at all. Figure 7 shows the yacc code, and Figure 6 shows the Lex code.

The section starting on Figure 7 line 43 should have been something to the effect of *operator NUMBER operator*. The production rule after that should have been something to the effect of *operator NUMBER operator NUMBER*. Even that isn't right though. I needed a stack and I could not figure it out in Lex/Yacc and searches online yielded nothing.

```
1  %{
2  #include "ocalc.tab.h"
3  extern int yylval;
4  %}
5
6  %%
7
8
9  [0-9]+  { yylval = atoi (yytext);
10     printf ("scanned the number %d\n", yylval);
11     return NUMBER; }
12 "+" return PLUS;
13 "-" return MINUS;
14 "*" return TIMES;
15 "/" return DIVIDE;
16 [ \t]   { printf ("skipped whitespace\n"); }
17 \n  { printf ("reached end of line\n");
18     return 0;
19     }
20
21
22 .   { printf ("found other data \"%s\"\n", yytext);
23     return yytext[0];
24     /* so yacc can see things like '+', '-', and '=' */
25     }
26
27 %%
```

Figure 6: Lex code for prefix calculator

4.2 Show the output generated by your calculator for

1. `+ 1 - 2 3` See Figure 8
2. `+ 1 - 2` See Figure 9

```
/*  Tried to use http://www-h.eng.cam.ac.uk/help/tpl/languages/flexbison/ to help but could
    not figure out how to make the operators grouped into an operator production.
*/

%{
#include <math.h>
#include <stdio.h>
#include <stdlib.h>
%}

%token NAME NUMBER
%token OPERATOR
%token PLUS MINUS TIMES DIVIDE

%left PLUS MINUS
%left TIMES DIVIDE
%left NEG
%%
statement:  NAME '=' expression { printf("pretending to assign %s the value %d\n", $1, $3); }
    /* Note that in the current version, the lexer never actually
returns NAME, so this rule will never happen. */
    ;
    |   expression      { printf("= %d\n", $1); }
    ;
expression:
    NUMBER          { $$ = $1;
                      printf ("Recognized a number.\n");
                      }
    /* BASIC OPERATIONS */
    |   TIMES expression NUMBER { $$ = $1 * $3 ;
                      printf ("Recognized '*' expression.\n");
                      }
    |
        DIVIDE expression NUMBER   { $$ = $1 / $3;
                      printf ("Recognized '/' expression.\n");
                      }
    |   PLUS expression NUMBER   { $$ = $2 + $3;
                      printf ("Recognized '+' expression.\n");
                      }
    |   MINUS expression NUMBER   { $$ = $2 - $3;
                      printf ("Recognized '-' expression.\n");
                      }
    /* CONVOLUTED NONSENSE */
    |   PLUS  NUMBER MINUS expression   { $$ = $2;
                      printf ("Recognized a nested operation.\n");
                      }
    |   PLUS  NUMBER MINUS NUMBER expression   { $$ = $2;
                      printf ("Recognized a nested operation.\n");
                      }
    |   MINUS expression %prec NEG { $$=-$2; }
    ;

%%
int main (void) {
    return yyparse();
    }

/* Added because panther doesn't have liby.a installed. */
int yyerror (char *msg) {
    return fprintf (stderr, "YACC: %s\n", msg);
    }
```

Figure 7: Yacc code for prefix calculator

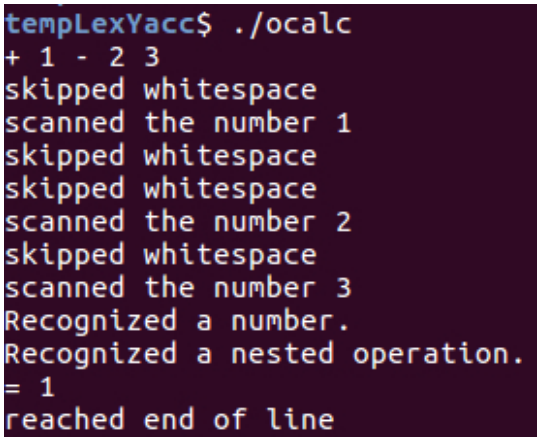

Figure 8: Lex/Yacc parsing of string: + 1 - 2 3

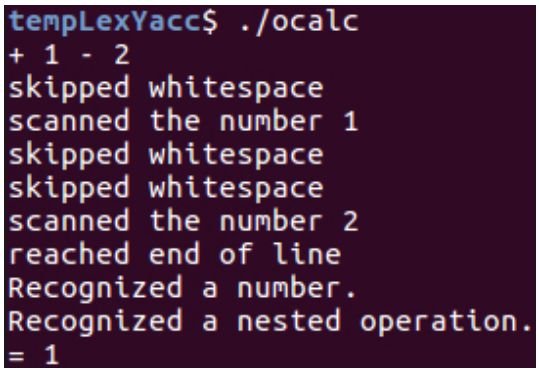

Figure 9: Lex/Yacc parsing of string: + 1 - 2

5 A Turing machine is an abstract computer that can do any computation that a real computer can do. Because the Turing machine is an abstract machine, it does not have the physical properties that a real machine has. For example, its tapes are infinite (or at least unbounded for purposes of any particular calculation) in length. Another important characteristics is that it lacks a clock that is tied to time in the real world.

5.1 Explain how a Turing machine can detect the fact that two mouse clicks occur close together in real-world time even though it has no clock.

From what I found online the minimum definition for a Turing Machine is:

$$M = \langle Q, \Gamma, b, \Sigma, \delta, q_0, F \rangle$$

Anything following that formula is a Turing Machine, but I found nothing stating a Turing Machine can't be added to. If L and R shift operations are given a time component t, which defines an 8th variable for the tuple: Δt, we can assume a Turing Machine would be capable of detecting a double click event if the tape were programmed with symbols representing the click events.

$$M = \langle Q, \Gamma, b, \Sigma, \delta, q_0, F, \Delta t \rangle$$

Another possibility is to construct an imitation Turing Machine. If we remove the abstraction and build something of similar functionality we would be constrained by computation time. Using this constraint to our advantage we would be able to use an integer symbol that counts the number of times it's able to add one to the symbol between key presses, resetting the symbol after checking for a double click. If the symbol value is in a predefined range it would be considered to be in the state of a double-click.