COMP2022 Assignment 2

October 2018

1 The Grammar G [10%]

Consider the following grammar G, which represents a fragment of a simple programming language:

$$L \to LE|E$$

$$E \to (C)|(F)|V|T$$

$$C \to ifEE|ifEEE$$

$$F \to +L| - L| * L|printL$$

$$V \to a|b|c|d$$

$$T \to 0|1|2|3$$

1.1 List the variables of G

L, E, C, F, V, T are the variables of G

1.2 List the terminals of G

a, b, c, d, 0, 1, 2, 3, (,), if, +, -, *, print are the terminals of G.

1.3 What is the start variable of G?

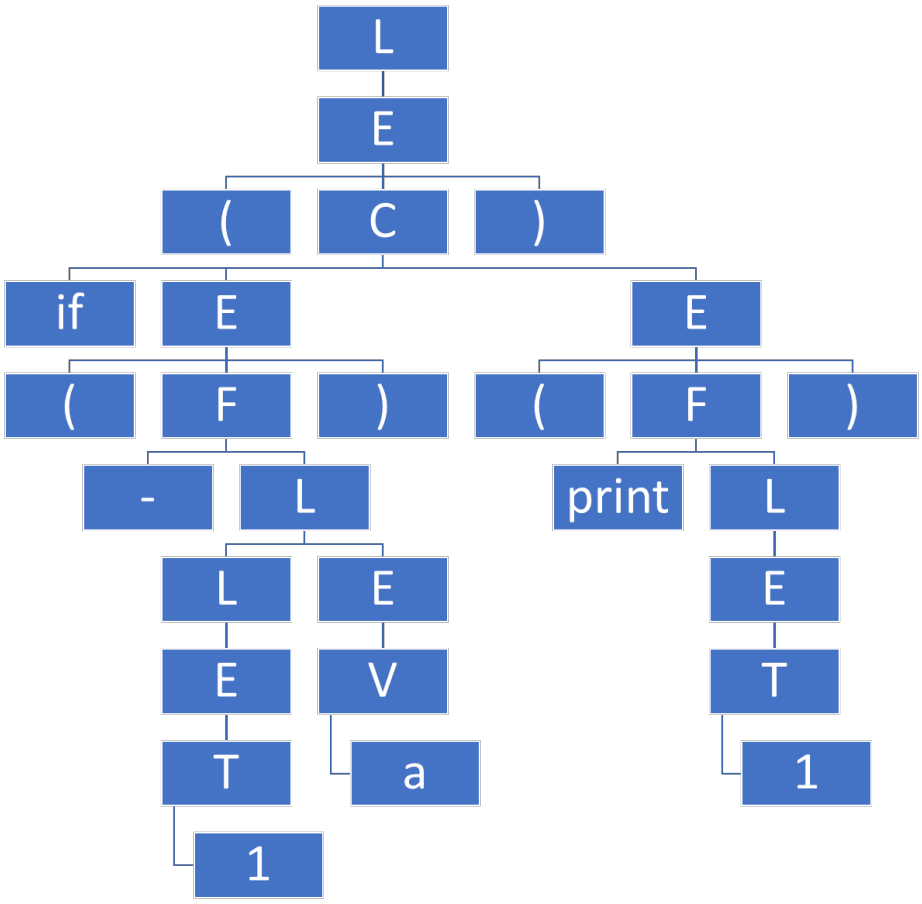
L is the start variable of G.

1.4 Give a leftmost derivation of the string (if(-1a)(print1))

L
E
(C)
(ifEE)
(if(F)E)
(if(-L)E)
(if(-LE)E)
(if(-EE)E)
(if(-TE)E)
(if(-1E)E)
(if(-1V)E)

(if(-1a)E)
(if(-1a)(F))
(if(-1a)(printL))
(if(-1a)(printE))
(if(-1a)(printT))
(if(-1a)(print1))

1.5 Draw a parse tree for (if(-1a)(print1))



2 Prove G is not LL(1) [10%]

Prove that G is not an LL(1) grammar.

The definition of an LL(1) grammar is a context free grammar whose parsing table has no multiple entries, and the 1 stands for using 1 input symbol to look ahead at each step of the parsing action decision.

The requirements for a grammar to be LL(1) are:

- The grammar must be left recursive
- The grammar needs to be left factoring
- The first sets of the production rules for a variable are not disjoint

With this knowledge, we can prove that the grammar G is not LL(1) due to the E, C and F variables. All of these have symbols that do not lead forward. These symbols are **(, if, +, -, *** and **print**. When these are parsed in alone, the grammar becomes stuck since there is no way to move forward. In order to follow the grammar, we must look multiple steps ahead, therefore making this not an LL(1) grammar, and thus proving the question.

An additional proof that is related to the previous point is that two of the production rules of the E variable can produce the same result. $E \rightarrow (C)$ & $E \rightarrow (F)$ both will return (which leads to their being two entries on the parse table. This also proves that the grammar G is not LL(1).

3 Find an equivalent LL(1) grammar G' [10%]

Find an equivalent grammar G' which is LL(1), by using the grammar transformation techniques shown in lectures, or otherwise. Describe the process and show your working.

My equivalent grammar G' which is LL(1) is as follow:

$$L \rightarrow EN$$
$$N \rightarrow EN|\epsilon$$
$$E \rightarrow (G)|V|T$$
$$G \rightarrow C|F$$
$$C \rightarrow ifEEI$$
$$I \rightarrow E|\epsilon$$
$$F \rightarrow +L| - L| * L|printL$$
$$V \rightarrow a|b|c|d$$
$$T \rightarrow 0|1|2|3$$

This grammar has removed the left recursive nature of the original Grammar G L variable, and has erased the repeated FIRST of multiple variables such as E through left factoring.

An example of this working can be seen with the same (if(-1a)(print1)):

L
EN
(G)N
(C)N
(ifEEI)N
(if(G)EI)N
(if(F)EI)N
(if(-L)EI)N
(if(-EN)EI)N
(if(-TN)EI)N
(if(-1N)EI)N
(if(-1EN)EI)N
(if(-1VN)EI)N
(if(-1aN)EI)N
(if(-1a)EI)N

(if(-1a)(G)I)N
(if(-1a)(F)I)N
(if(-1a)(printL)I)N
(if(-1a)(printEN)I)N
(if(-1a)(printTN)I)N
(if(-1a)(print1N)I)N
(if(-1a)(print1)I)N
(if(-1a)(print1))N
(if(-1a)(print1))

4 LL(1) parse table [15%]

Complete the LL(1) parse table for G'. Describe the process and show your working, including:

1. FIRST sets for all the production rules of G'
2. FOLLOW sets for variables of G' only if they are needed

	a	b	c	d	0	1	2	3	(	)	if	+	-	*	print	$
L	EN	EN	EN	EN	EN	EN	EN	EN	EN							
N	EN	EN	EN	EN	EN	EN	EN	EN	EN	ϵ						ϵ
E	V	V	V	V	T	T	T	T	(G)						(G)	
G											C	F	F	F	F	
C											ifEEI					
I	E	E	E	E	E	E	E	E		ϵ						
F												+L	-L	*L	printL	
V	a	b	c	d												
T					0	1	2	3								

My first attempt at construction a FIRST and FOLLOW set can be found on the page below.

Production Rule	First Set	Follow Set
$L \rightarrow EN$	FIRST(EN) = {(,a,b,c,d,0,1,2,3,ϵ}	FOLLOW(L) = {$}
$N \rightarrow EN$	FIRST(EN) = {(,a,b,c,d,0,1,2,3,ϵ}	FOLLOW(N) = {$}
$N \rightarrow \epsilon$	FIRST(ϵ) = {ϵ}	FOLLOW(N) = {$}
$E \rightarrow (G)$	FIRST((G)) = {(}	
$E \rightarrow V$	FIRST(V) = {a,b,c,d}	
$E \rightarrow T$	FIRST(T) = {0,1,2,3}	
$G \rightarrow C$	FIRST(C) = {if}	
$G \rightarrow F$	FIRST(F) = {+,-,*,print}	
$C \rightarrow ifEEI$	FIRST(ifEEI) = {if}	
$I \rightarrow E$	FIRST(E) = {(,a,b,c,d,0,1,2,3}	
$I \rightarrow \epsilon$	FIRST(ϵ) = {ϵ}	
$F \rightarrow +L$	FIRST(+L) = {+}	
$F \rightarrow -L$	FIRST(-L) = {-}	
$F \rightarrow *L$	FIRST(*L) = {*}	
$F \rightarrow printL$	FIRST(printL) = {print}	
$V \rightarrow a$	FIRST(a) = {a}	
$V \rightarrow b$	FIRST(b) = {b}	
$V \rightarrow c$	FIRST(c) = {c}	
$V \rightarrow d$	FIRST(d) = {d}	
$T \rightarrow 0$	FIRST(0) = {0}	
$T \rightarrow 1$	FIRST(1) = {1}	
$T \rightarrow 2$	FIRST(2) = {2}	
$T \rightarrow 3$	FIRST(3) = {3}	

This however, I feel was not a perfection set, and produced errors later on. Upon making the FIRST and FOLLOW set I went to try and create the set in my code. The visualisation of how I encoded it can be seen below:

```
static final String[][] lookup = new String[][]{
  //10x17 Matrix. This is the hardcoded version of the grammar G' parse/lookup table
  {"",  "a",   "b",   "c",   "d",   "0",   "1",   "2",   "3",  "(",   ")",   "+",   "-",   "*",  "print", "if", "$"},
  {"L", "EN",  "EN",  "EN",  "EN",  "EN",  "EN",  "EN",  "EN",  "EN", "BAD", "BAD", "BAD", "BAD","BAD",   "BAD","BAD"},
  {"N", "EN",  "EN",  "EN",  "EN",  "EN",  "EN",  "EN",  "EN",  "EN",  "",   "BAD", "BAD", "BAD","BAD",   "BAD", ""},
  {"E", "V",   "V",   "V",   "V",   "T",   "T",   "T",   "T",   "(G)", "BAD", "BAD", "BAD", "BAD", "BAD", "(G)", "BAD"},
  {"G", "BAD", "BAD", "BAD", "BAD", "BAD", "BAD", "BAD", "BAD", "BAD", "BAD", "F",   "F",   "F",  "F", "C", "BAD"},
  {"C", "BAD", "BAD", "BAD", "BAD", "BAD", "BAD", "BAD", "BAD", "BAD", "BAD", "BAD", "BAD", "BAD", "BAD", "ifEEI", "BAD"},
  {"I", "E", "E", "E", "E", "E", "E", "E", "E", "E", "BAD", "", "BAD", "BAD", "BAD", "BAD", "BAD", "BAD", "BAD"},
  {"F", "BAD", "BAD", "BAD", "BAD", "BAD", "BAD", "BAD", "BAD", "BAD", "BAD", "+L",  "-L",  "*L",  "printL","BAD", "BAD"},
  {"V", "a",   "b",   "c",   "d",   "BAD", "BAD", "BAD", "BAD", "BAD", "BAD", "BAD", "BAD", "BAD", "BAD", "BAD", "BAD"},
  {"T", "BAD", "BAD", "BAD", "BAD", "0",   "1",   "2",   "3", "BAD", "BAD", "BAD", "BAD", "BAD", "BAD", "BAD", "BAD"}
};
```

Then, through trial and error I made minor tweaks till it gave me the correct grammar for multiple inputs.

5 Implementation [25%]

Implement a program which parses strings using an LL(1) table driven parser, using the table you determined for G' in the previous exercise. You may use Python, Java, C, C++, or Lisp. If you'd like to use a different language then please check with us first.

- Input: The first command line argument is the filename of a file containing the string of characters to test.

- Output:
    1. Print a trace of the execution, showing the steps followed by the program as it performs the left-most derivation. This should look similar to parsing the string through a PDA. An example of this is given in the appendices.
    2. After parsing the whole input file, print ACCEPTED or REJECTED, depending on whether or not the string could be derived by the grammar.
    3. If there is a symbol in the input string which is not a terminal from the grammar, the program should output `ERROR_INVALID_SYMBOL` (This could be during or before trying to parse the input.)
- All whitespace in the input file should be ignored (line breaks, spaces, etc.). The output will be easier to read if you remove the whitespace *before* starting the parse.
- Examples of the program output syntax are provided in the appendices.

To test the outputs of this code, I ran the example zips over it to ensure that the Parse Table accepted and rejected the proper inputs. The final test I did was running an extended string which is as follows.

$$if(print(*2c)1(if(-(+20)1(ifa(if(print(+bc)a(if(+(+bd)1(if2(ifaab)b)(+0))a2)(+0))11)b)(+0))1c)$$

$$(+3(print21(if(print(*c1)1(if(print(+2)1(ifa(if(printa1(if(+(+ba)1(ifb(ifccb)b)(+0))a2)(+02))11)b)$$
$$(+0))aa)(+3(print022)))ab))))11)$$

This is an acceptable string as defined by the grammar, and it returned an ACCEPTED result. The logfile for the output is significantly too long for the report and will not be shared, but examples of other testcases being ran can be found below.

The final test that I added was just adding nonterminal characters to the input string. I ran a test called bad.txt which consisted of the string "xyz" to ensure that invalid symbol occured when this was entered. This also passed the test.

The Accepted Strings

```
nick@DESKTOP-6OT9TCM:/mnt/c/Users/Ninjawaffle/Documents/University/COMP Subjects/COMP2022/Assignment 2$ java Parser acceptExample
(if(-1a)(print1))$ || L$
(if(-1a)(print1))$ || EN$
(if(-1a)(print1))$ || (G)N$
if(-1a)(print1))$ || G)N$
if(-1a)(print1))$ || C)N$
if(-1a)(print1))$ || ifEEI)N$
(-1a)(print1))$ || EEI)N$
(-1a)(print1))$ || (G)EI)N$
-1a)(print1))$ || G)EI)N$
-1a)(print1))$ || F)EI)N$
-1a)(print1))$ || -L)EI)N$
1a)(print1))$ || L)EI)N$
1a)(print1))$ || EN)EI)N$
1a)(print1))$ || TN)EI)N$
1a)(print1))$ || 1N)EI)N$
a)(print1))$ || N)EI)N$
a)(print1))$ || EN)EI)N$
a)(print1))$ || VN)EI)N$
a)(print1))$ || aN)EI)N$
)(print1))$ || N)EI)N$
)(print1))$ || )EI)N$
(print1))$ || EI)N$
(print1))$ || (G)I)N$
print1))$ || G)I)N$
print1))$ || F)I)N$
print1))$ || printL)I)N$
1))$ || L)I)N$
1))$ || EN)I)N$
1))$ || TN)I)N$
1))$ || 1N)I)N$
))$ || N)I)N$
))$ || )I)N$
)$ || I)N$
)$ || )N$
$ || N$
$ || $
ACCEPTED
```

```
nick@DESKTOP-6OT9TCM:/mnt/c/Users/Ninjawaffle/Documents/University/COMP Subjects/COMP2022/Assignment 2$ java Parser acceptIfIf
(if1(ifab))$ || L$
(if1(ifab))$ || EN$
(if1(ifab))$ || (G)N$
if1(ifab))$ || G)N$
if1(ifab))$ || C)N$
if1(ifab))$ || ifEEI)N$
1(ifab))$ || EEI)N$
1(ifab))$ || TEI)N$
1(ifab))$ || 1EI)N$
(ifab))$ || EI)N$
(ifab))$ || (G)I)N$
ifab))$ || G)I)N$
ifab))$ || C)I)N$
ifab))$ || ifEEI)I)N$
ab))$ || EEI)I)N$
ab))$ || VEI)I)N$
ab))$ || aEI)I)N$
b))$ || EI)I)N$
b))$ || VI)I)N$
b))$ || bI)I)N$
))$ || I)I)N$
))$ || )I)N$
)$ || I)N$
)$ || )N$
$ || N$
$ || $
ACCEPTED
```

```
nick@DESKTOP-6OT9TCM:/mnt/c/Users/Ninjawaffle/Documents/University/COMP Subjects/COMP2022/Assignment 2$ java Parser acceptPrintList
(printabc)$ || L$
(printabc)$ || EN$
(printabc)$ || (G)N$
printabc)$ || G)N$
printabc)$ || F)N$
printabc)$ || printL)N$
abc)$ || L)N$
abc)$ || EN)N$
abc)$ || VN)N$
abc)$ || aN)N$
bc)$ || N)N$
bc)$ || EN)N$
bc)$ || VN)N$
bc)$ || bN)N$
c)$ || N)N$
c)$ || EN)N$
c)$ || VN)N$
c)$ || cN)N$
)$ || N)N$
)$ || )N$
$ || N$
$ || $
ACCEPTED
```

The Rejected Strings

```
nick@DESKTOP-6OT9TCM:/mnt/c/Users/Ninjawaffle/Documents/University/COMP Subjects/COMP2022/Assignment 2$ java Parser rejectParenthesesNumberAlone
(1)$ || L$
(1)$ || EN$
(1)$ || (G)N$
1)$ || G)N$
REJECTED
```

```
nick@DESKTOP-6OT9TCM:/mnt/c/Users/Ninjawaffle/Documents/University/COMP Subjects/COMP2022/Assignment 2$ java Parser bad.txt
ERROR_INVALID_SYMBOL
REJECTED
```

```
nick@DESKTOP-6OT9TCM:/mnt/c/Users/Ninjawaffle/Documents/University/COMP Subjects/COMP2022/Assignment 2$ java Parser rejectHangingOpenParens
(if(-1a)(print1)$ || L$
(if(-1a)(print1)$ || EN$
(if(-1a)(print1)$ || (G)N$
if(-1a)(print1)$ || G)N$
if(-1a)(print1)$ || C)N$
if(-1a)(print1)$ || ifEEI)N$
(-1a)(print1)$ || EEI)N$
(-1a)(print1)$ || (G)EI)N$
-1a)(print1)$ || G)EI)N$
-1a)(print1)$ || F)EI)N$
-1a)(print1)$ || -L)EI)N$
1a)(print1)$ || L)EI)N$
1a)(print1)$ || EN)EI)N$
1a)(print1)$ || TN)EI)N$
1a)(print1)$ || 1N)EI)N$
a)(print1)$ || N)EI)N$
a)(print1)$ || EN)EI)N$
a)(print1)$ || VN)EI)N$
a)(print1)$ || aN)EI)N$
)(print1)$ || N)EI)N$
)(print1)$ || )EI)N$
(print1)$ || EI)N$
(print1)$ || (G)I)N$
print1)$ || G)I)N$
print1)$ || F)I)N$
print1)$ || printL)I)N$
1)$ || L)I)N$
1)$ || EN)I)N$
1)$ || TN)I)N$
1)$ || 1N)I)N$
)$ || N)I)N$
)$ || )I)N$
$ || I)N$
REJECTED
```

6 Extension [30%]

You may pick one of the following extensions to improve your parser. Any one of the following ideas would be sufficient (do not implement more than one). They are listed roughly in order of increasing difficulty/effort, but are worth the same marks:

- Use the FIRST and FOLLOW sets to implement an error recovery feature. This should give the user suggestions on possible corrections which could change strings which could not be derived in G' For this extension you need to:
  - Implementation: If a second command line argument error is given, then instead of rejecting a string that is not in the language, it should make suggestions about how to correct it. The user chooses one of the options, then the program should continue the parse. Some examples are provided in the appendices. This should be included in the code you submit to PASTA.
  - Report: Explain how your error recovery feature uses the FIRST and FOLLOW sets to work, and show some useful examples.

The Extension was not fully completed due to being unable to figure out how to follow the variables to get the correct terminals. Instead, they tell the variables and terminals which should of gone there, rather than just the end possible terminals. No other features were added.