COMP2922 ASSIGNMENT 2

Min Joo Kim
460 495 204

Table of Contents

1. The Grammar G

$L \rightarrow LE \mid E$

$E \rightarrow (C) \mid (F) \mid V \mid T$

$C \rightarrow$ if EE | if EEE

$F \rightarrow$ +L | -L | *L | print L

$V \rightarrow a \mid b \mid c \mid d$

$T \rightarrow 0 \mid 1 \mid 2 \mid 3$

2. Proving G is not LL(1)

The grammar is LL(1) if and only if for every variable of G, we can decide which rule to follow just by observing the next symbol. But for some of the terminals of G, this criteria does not meet – for example the rules of E, (C) and (F) both start with the opening bracket '(' and therefore it is impossible to decide which rule to follow just by the next symbol, and therefore G cannot be LL(1).

3. Find an Equivalent LL(1) Grammar G'

$L \rightarrow EA$ (eliminate left recursion)

$A \rightarrow L \mid \varepsilon$

$E \rightarrow V \mid T \mid (B)$ (left factorisation)

$B \rightarrow C \mid F$

$C \rightarrow$ if EED (left factorisation)

$D \rightarrow E \mid \varepsilon$

$F \rightarrow$ +L | -L | *L | print L

$V \rightarrow a \mid b \mid c \mid d$

$T \rightarrow 0 \mid 1 \mid 2 \mid 3$

4. LL(1) Parse Table

Process

	FIRST	FOLLOW
L → EA	{a, b, c, d, 0, 1, 2, 3, (}	{$,)}
A → L | ε	{ a, b, c, d, 0, 1, 2, 3, (, ε }	{$,)}
E → V | T | (B)	{a, b, c, d, 0, 1, 2, 3, (}	
B → C | F	{if, +, -, *, print}	{)}
C → if EED	{if}	{)}
D → E | ε	{a, b, c, d, 0, 1, 2, 3, (, ε }	{)}
F → +L | -L | *L | print L	{+, -, *, print}	{)}
V → a | b | c | d	{a, b, c, d}	
T → 0 | 1 | 2 | 3	{0, 1, 2, 3}	

Parse Table

	a	**b**	**c**	**d**	**0**	**1**	**2**	**3**	**+**	**-**	*****	**print**	**if**	**(**	**)**	**$**
L	EA	EA	EA	EA	EA	EA	EA	EA						EA		
A	L	L	L	L	L	L	L	L							ε	ε
E	V	V	V	V	T	T	T	T						(B)		
B									F	F	F	F	C			
C													ifEED			
D	E	E	E	E	E	E	E	E						E	ε	
F									+L	-L	*L	printL				
V	a	b	c	d												
T					0	1	2	3								

5.1 PARSING STRINGS WITH AN LL(1) TABLE DRIVEN PARSER

Instructions to run the code

Put parser.py and input files in the same repository. Then simply run with “python parser.py input_file” on the command line. Input_file should be replaced with the file the user is going to test with.

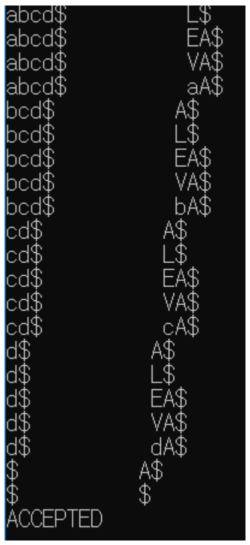
Steps

1. Read the string from the input file.
2. Clean up the white spaces from the string.
3. Create two lists, one should store the unscanned list and another one should store the scanned output. They should store the strings in reverse order so that their functionality should be same as the stack data structure. For example, if the input string is “abcd”, then in the list it should be stored as $ in index 0, d in index 1, c in index 2 and so on.
4. Parse table is stored as a pivot data frame.
5. In a loop,
   a) Base case: if both of the lists only have $ left, print ACCEPTED and break.
   b) If the symbols at the top of the lists are equal, remove that symbol from the both.
   c) Otherwise, let’s say the top symbol of unscanned list is x and the top symbol of stack is y. Use the parse table data frame, check if data_frame[x][y] has a valid rule. If true, replace y with the rule retrieved from the data frame, else print REJECTED and terminate.

Example Outputs - ACCEPTED

Input: abcd

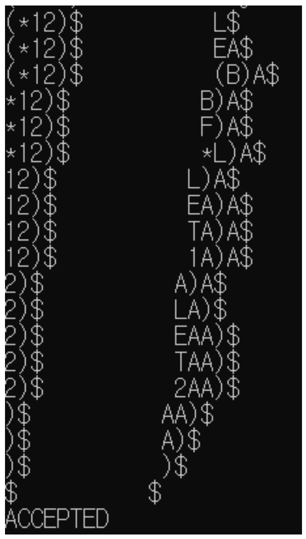
Output: ACCEPTED



```
abcd$          L$
abcd$          EA$
abcd$          VA$
abcd$          aA$
bcd$          A$
bcd$          L$
bcd$          EA$
bcd$          VA$
bcd$          bA$
cd$          A$
cd$          L$
cd$          EA$
cd$          VA$
cd$          cA$
d$          A$
d$          L$
d$          EA$
d$          VA$
d$          dA$
$          A$
$          $
ACCEPTED
```

Input: (*12)

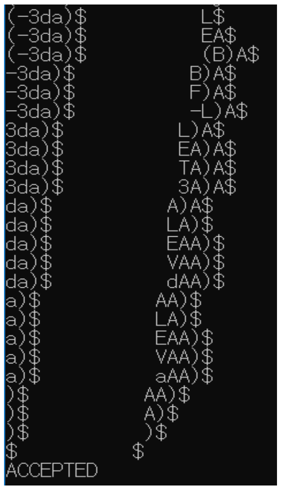
Output: ACCEPTED



```
(*12)$          L$
(*12)$          EA$
(*12)$           (B)A$
*12)$          B)A$
*12)$          F)A$
*12)$           *L)A$
12)$          L)A$
12)$          EA)A$
12)$          TA)A$
12)$           1A)A$
2)$          A)A$
2)$          LA)$
2)$          EAA)$
2)$          TAA)$
2)$          2AA)$
)$          AA)$
)$          A)$
)$           )$
$          $
ACCEPTED
```

Input: (-
3d a   )

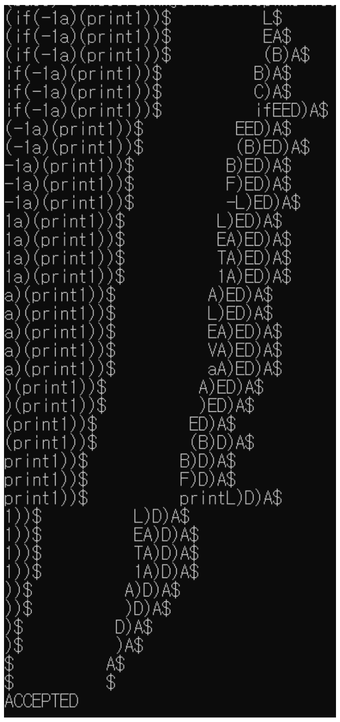
Output: ACCEPTED, whitespaces removed first



```
(-3da)$          L$
(-3da)$          EA$
(-3da)$           (B)A$
-3da)$          B)A$
-3da)$          F)A$
-3da)$          -L)A$
3da)$          L)A$
3da)$          EA)A$
3da)$          TA)A$
3da)$          3A)A$
da)$          A)A$
da)$          LA)$
da)$          EAA)$
da)$          VAA)$
da)$          dAA)$
a)$          AA)$
a)$          LA)$
a)$          EAA)$
a)$          VAA)$
a)$          aAA)$
)$          AA)$
)$          A)$
)$           )$
$          $
ACCEPTED
```

Input: (if(-1a)(print1))

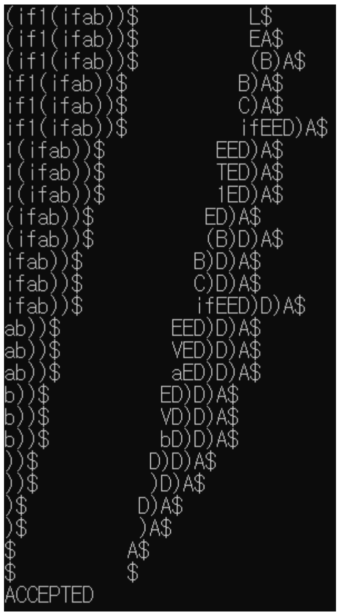
Output: ACCEPTED



```
(if(-1a)(print1))$         L$
(if(-1a)(print1))$         EA$
(if(-1a)(print1))$          (B)A$
if(-1a)(print1))$          B)A$
if(-1a)(print1))$          C)A$
if(-1a)(print1))$           ifEED)A$
(-1a)(print1))$          EED)A$
(-1a)(print1))$           (B)ED)A$
-1a)(print1))$          B)ED)A$
-1a)(print1))$          F)ED)A$
-1a)(print1))$           -L)ED)A$
1a)(print1))$          L)ED)A$
1a)(print1))$          EA)ED)A$
1a)(print1))$          TA)ED)A$
1a)(print1))$           1A)ED)A$
a)(print1))$          A)ED)A$
a)(print1))$          L)ED)A$
a)(print1))$          EA)ED)A$
a)(print1))$          VA)ED)A$
a)(print1))$          aA)ED)A$
)(print1))$          A)ED)A$
)(print1))$           )ED)A$
(print1))$          ED)A$
(print1))$           (B)D)A$
print1))$          B)D)A$
print1))$          F)D)A$
print1))$          printL)D)A$
1))$          L)D)A$
1))$          EA)D)A$
1))$          TA)D)A$
1))$           1A)D)A$
))$          A)D)A$
))$           )D)A$
)$          D)A$
)$           )A$
$          A$
$          $
ACCEPTED
```

Input: (if 1 (if a b))

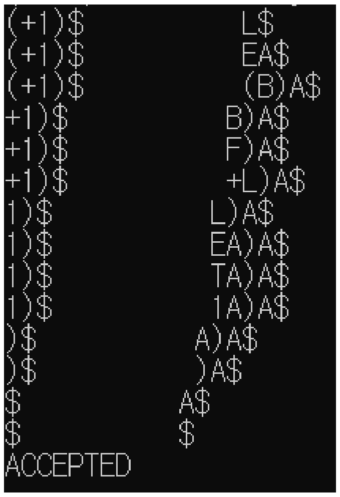
Output: ACCEPTED



```
(if1(ifab))$          L$
(if1(ifab))$          EA$
(if1(ifab))$           (B)A$
if1(ifab))$          B)A$
if1(ifab))$          C)A$
if1(ifab))$           ifEED)A$
1(ifab))$          EED)A$
1(ifab))$          TED)A$
1(ifab))$           1ED)A$
(ifab))$          ED)A$
(ifab))$           (B)D)A$
ifab))$          B)D)A$
ifab))$          C)D)A$
ifab))$           ifEED)D)A$
ab))$          EED)D)A$
ab))$          VED)D)A$
ab))$           aED)D)A$
b))$          ED)D)A$
b))$          VD)D)A$
b))$          bD)D)A$
))$          D)D)A$
))$           )D)A$
)$          D)A$
)$           )A$
$          A$
$          $
ACCEPTED
```

Input: (+ 1)

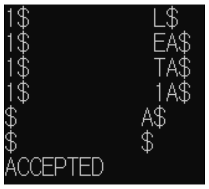
Output: ACCEPTED



```
(+1)$          L$
(+1)$          EA$
(+1)$           (B)A$
+1)$          B)A$
+1)$          F)A$
+1)$           +L)A$
1)$          L)A$
1)$          EA)A$
1)$          TA)A$
1)$           1A)A$
)$          A)A$
)$           )A$
$          A$
$          $
ACCEPTED
```

Input: 1

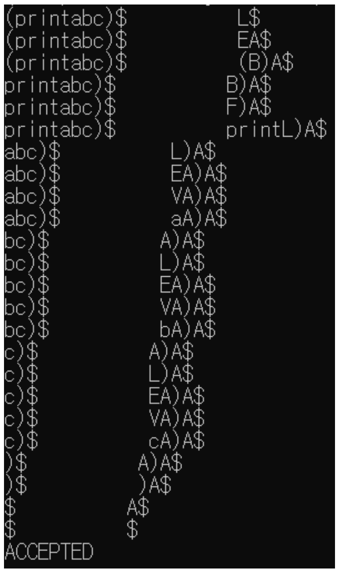
Output: ACCEPTED



```
1$          L$
1$          EA$
1$          TA$
1$           1A$
$          A$
$          $
ACCEPTED
```

Input: (print a b c)

Output: ACCEPTED

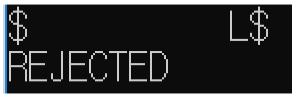


```
(printabc)$          L$
(printabc)$          EA$
(printabc)$           (B)A$
printabc)$          B)A$
printabc)$          F)A$
printabc)$          printL)A$
abc)$          L)A$
abc)$          EA)A$
abc)$          VA)A$
abc)$          aA)A$
bc)$          A)A$
bc)$          L)A$
bc)$          EA)A$
bc)$          VA)A$
bc)$          bA)A$
c)$          A)A$
c)$          L)A$
c)$          EA)A$
c)$          VA)A$
c)$          cA)A$
)$          A)A$
)$           )A$
$          A$
$          $
ACCEPTED
```

Example Outputs - REJECTED

Input: (empty)

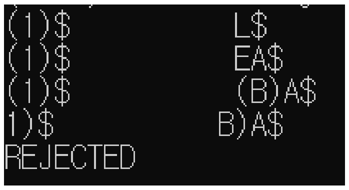
Output: REJECTED, because there is no rule for empty terminal and variable L in the parse table.



```
$                L$
REJECTED
```

Input: (1)

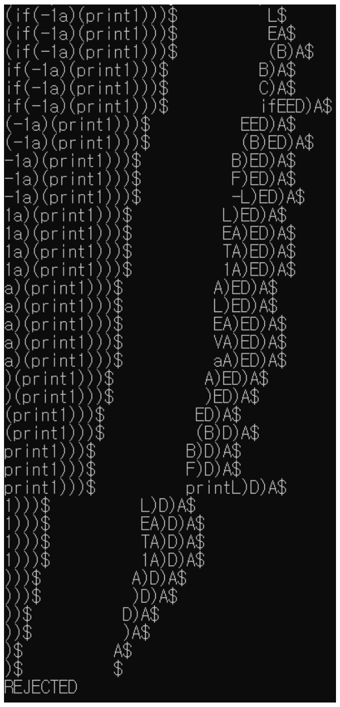
Output: REJECTED, because FIRST(B) = {if, +, -, *, print} but 1 was given.



```
(1)$            L$
(1)$            EA$
(1)$             (B)A$
1)$             B)A$
REJECTED
```

Input: (if (- 1 a) (print 1)))

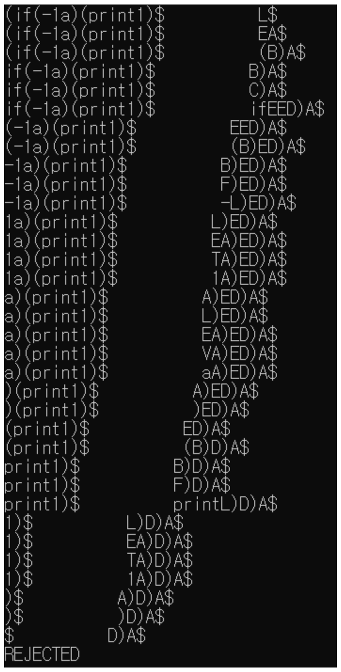
Output: REJECTED, because the closing parenthese is still left on the unscanned list but no more variable left in the stack.



```
(if(-1a)(print1)))$              L$
(if(-1a)(print1)))$              EA$
(if(-1a)(print1)))$               (B)A$
if(-1a)(print1)))$              B)A$
if(-1a)(print1)))$              C)A$
if(-1a)(print1)))$               ifEED)A$
(-1a)(print1)))$              EED)A$
(-1a)(print1)))$               (B)ED)A$
-1a)(print1)))$              B)ED)A$
-1a)(print1)))$              F)ED)A$
-1a)(print1)))$              -L)ED)A$
1a)(print1)))$              L)ED)A$
1a)(print1)))$              EA)ED)A$
1a)(print1)))$              TA)ED)A$
1a)(print1)))$               1A)ED)A$
a)(print1)))$              A)ED)A$
a)(print1)))$              L)ED)A$
a)(print1)))$              EA)ED)A$
a)(print1)))$              VA)ED)A$
a)(print1)))$              aA)ED)A$
)(print1)))$              A)ED)A$
)(print1)))$               )ED)A$
(print1)))$              ED)A$
(print1)))$               (B)D)A$
print1)))$              B)D)A$
print1)))$              F)D)A$
print1)))$              printL)D)A$
1)))$              L)D)A$
1)))$              EA)D)A$
1)))$              TA)D)A$
1)))$               1A)D)A$
))))$              A)D)A$
))))$               )D)A$
)))$              D)A$
)))$               )A$
))$              A$
))$              $
REJECTED
```

Input: (if (- 1 a) (print 1)

Output: REJECTED, in this case one more opening parenthese, unscanned list is empty but stack is not.



```
(if(-1a)(print1)$              L$
(if(-1a)(print1)$              EA$
(if(-1a)(print1)$               (B)A$
if(-1a)(print1)$              B)A$
if(-1a)(print1)$              C)A$
if(-1a)(print1)$               ifEED)A$
(-1a)(print1)$              EED)A$
(-1a)(print1)$               (B)ED)A$
-1a)(print1)$              B)ED)A$
-1a)(print1)$              F)ED)A$
-1a)(print1)$              -L)ED)A$
1a)(print1)$              L)ED)A$
1a)(print1)$              EA)ED)A$
1a)(print1)$              TA)ED)A$
1a)(print1)$               1A)ED)A$
a)(print1)$              A)ED)A$
a)(print1)$              L)ED)A$
a)(print1)$              EA)ED)A$
a)(print1)$              VA)ED)A$
a)(print1)$              aA)ED)A$
)(print1)$              A)ED)A$
)(print1)$               )ED)A$
(print1)$              ED)A$
(print1)$               (B)D)A$
print1)$              B)D)A$
print1)$              F)D)A$
print1)$              printL)D)A$
1)$              L)D)A$
1)$              EA)D)A$
1)$              TA)D)A$
1)$               1A)D)A$
)$              A)D)A$
)$               )D)A$
$              D)A$
REJECTED
```

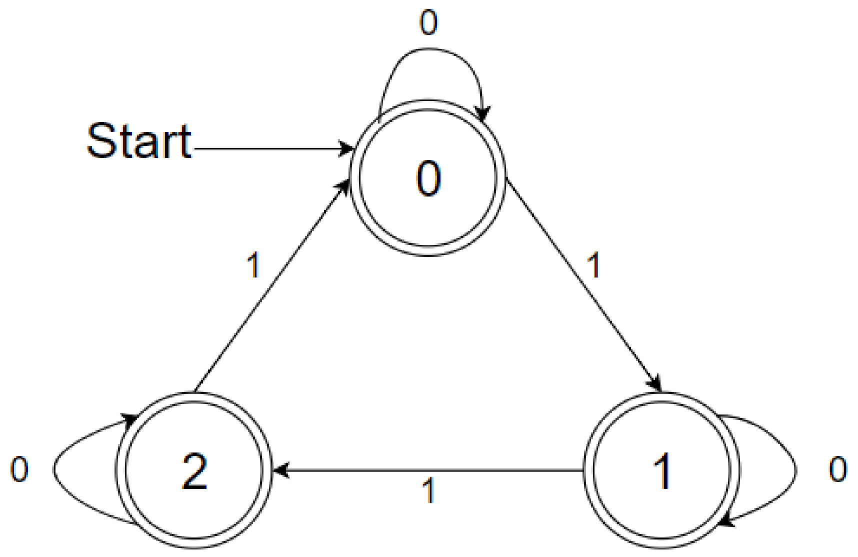
5.2 Regex (COMP2922)

Instructions to run the code

Section 5.2 is contained in parser.py and simply run parser.py to test section 5.2.

Steps

1. Read the input file and clean up the whitespaces.
2. Check if the input string starts with n: where n is one of the digits 0, 1 or 2 followed by a colon.
3. If not, then section 5.2 will not happen, and the input string will go straight to the parser.
4. If yes, parse the string after the colon to DFA.



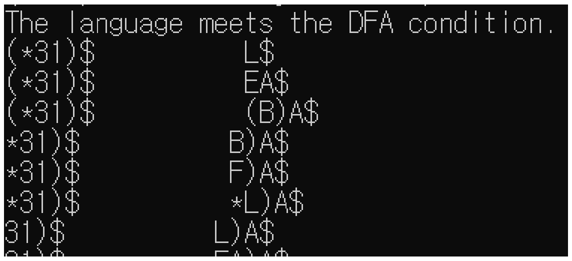
   1 is any arithmetic operations [+, -, *] and 0 is any other values.

5. Once the string reached the end, return the finishing accept state.
6. If returned state is different to n, then the language is rejected and programs finishes.
7. If returned state is equal to n, then parse the string after the colon to the parser and section 5.1 happens.

Example Outputs - ACCEPTED

Input: 1: (* 3 1)

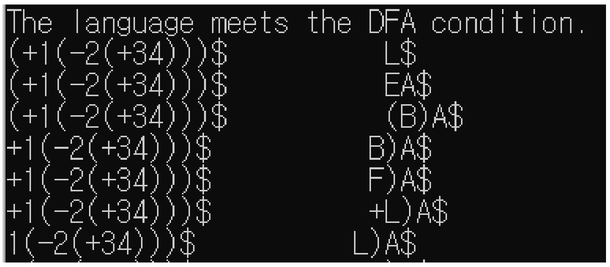
Output: language passes DFA, since there is one arithmetic operation and 3%1=1. Then the string will be parsed to the parser.



```
The language meets the DFA condition.
(*31)$          L$
(*31)$          EA$
(*31)$           (B)A$
*31)$           B)A$
*31)$           F)A$
*31)$           *L)A$
31)$           L)A$
```

Input: 0: (+ 1 (- 2 (+ 3 4)))

Output: language passes DFA, since there are three arithmetic operations and 3%3=0. Then the string will be parsed to the parser.

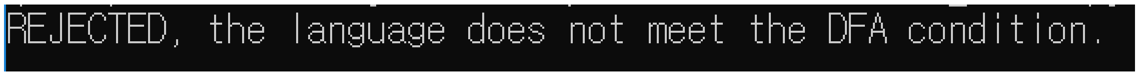


```
The language meets the DFA condition.
(+1(-2(+34)))$          L$
(+1(-2(+34)))$          EA$
(+1(-2(+34)))$           (B)A$
+1(-2(+34)))$           B)A$
+1(-2(+34)))$           F)A$
+1(-2(+34)))$           +L)A$
1(-2(+34)))$           L)A$
```

Example Outputs - REJECTED

Input: 2:

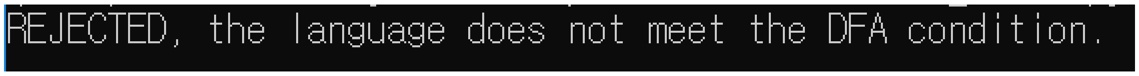
Output: REJECTED, since n=2 but string after the colon is empty, so the state will be 0.



```
REJECTED, the language does not meet the DFA condition.
```

Input: 0: (+3 1)

Output: REJECTED, since n=0 but string after the colon has one arithmetic operation.



```
REJECTED, the language does not meet the DFA condition.
```

6. EXTENSION PART A

The error recovery feature is expected to happen when there is no available rule to call with two inputs, one is the next element of unscanned list and another one is the top of the stack. When there was no error recovery feature implemented yet, the program should return the message "REJECTED" and terminate, but this extended feature allows the language that does not meet the grammar to be accepted. There are two possible situations, and I would like to explain them in details.

Instructions to run the extension

Extension is contained in the part where it should return REJECTED. To run the extension, give 'error' as the sys.argv[2]. i.e. python parser.py input_file error

1. Top of the stack is a terminal

This means, the top of the stack is one of [a, b, c, d, 0, 1, 2, 3, +, -, *, print, if, (,), $]. The expected input of the unscanned list is the terminal value at the top of the stack. This situation can be further divided into two cases (examples are not from the beginning);

- Unscanned list is empty but the stack is not

Then the program should ADD the value on the head of the unscanned list, and continue with the updated list. Input validation check happens to check whether the user gave the correct input.

```
2$              E)$
2$              V)$
2$              2)$
 $               )$
Error: got $, but expected {)}.
Add input? A
Wrong input given. Add input? )
)$               )$
 $                $
ACCEPTED
```

- Head of the unscanned list and the top of the stack are different

Then the program should DELETE the value on the head of the unscanned list. User input should be either Y or N, if Y is the input then the program should continue with the updated list, and if N is the input then the program should return REJECTED and terminate.

```
33$             E)$
33$             V)$
33$             3)$
 3$              )$
Error: got 3, but expected {)}.
Delete input? [Y/N] Y
  $              )$
Error: got $, but expected {}}.
Add input? )
 )$              )$
  $               $
ACCEPTED
```

2. Top of the stack is a variable

This means, the top of the stack is one of [L, A, E, B, C, D, F, V, T]. The expected input of the unscanned list is FIRST (top of the stack). This situation can be further divided into two cases;

- Unscanned list is empty but the stack is not

Then the program should ADD the value on the head of the unscanned list, and continue with the updated list. This works exactly the same as the above, but here the user input should be one of the terminals existing in the FIRST (top of the stack).

```
2+$                 2F$
 +$                  F$
 +$                 +L$
  $                  L$
Error: got $, but expected {a, ,b, c, d, 0,
1, 2, 3, (}.
Add input? h
Wrong input given. Add input? a
 a$                  L$
 a$                 EA$
 a$                 VA$
 a$                 aA$
  $                  A$
  $                   $
ACCEPTED
```

- Head of the unscanned list and the top of the stack are different

Then the program should REPLACE the value on the head of the unscanned list, and continue with the updated list. The user can only replace with one of the terminal values in the FIRST (top of the stack).

```
 f$                 A$
Error: got f, but expected {a, ,b, c, d, 0,
1, 2, 3, ε,  (}.
Replace input? a

 a$                  A$
 a$                  L$
 a$                 EA$
 a$                 VA$
 a$                 aA$
  $                  A$
  $                   $
ACCEPTED
```