**Assignment Two**

**Report**

-

09020328 王亮

# **I. Motivation/Aim**

(1) Further explore syntactic analysis.

(2) Implement a syntactic analyzer to analyze given string with predefined CFG using LL(1) tactics.

# **II. Content Description**

Input:

Stream of characters

CFG(Combination of CFGs of some classes of sentences)

Output:

Sequence of derivations if top-down syntax analyzing methods are used.

Predefined CFG:

*/\* u8 for 'char' replacement in C, as char in rust takes four bytes \*/*

struct Prod<'a>(u8, &'a str);

*/\* E' -> R    T' -> Y \*/*

const productions: [Prod; 8] = [

    Prod(b'E', "TR"), *// 0 E->TE'*

    Prod(b'R', "+TR"), *// 1 E'->+TE'*

    Prod(b'R', "ε"), *// 2 E'->ε*

    Prod(b'T', "FY"), *// 3 T->FT'*

    Prod(b'Y', "\*FY"), *// 4 T'->\*FT'*

    Prod(b'Y', "ε"), *// 5 T'->ε*

    Prod(b'F', "(E)"), *// 6 F->(E)*

    Prod(b'F', "i"), *// 7 F->i*

];

LL(1) table

 match (stack\_top, input\_first) {

        (b'E', b'(') => 0,

        (b'E', b'i') => 0,

        (b'R', b'+') => 1,

        (b'R', b')') => 2,

        (b'R', b'$') => 2,

        (b'T', b'(') => 3,

        (b'T', b'i') => 3,

        (b'Y', b'+') => 5,

        (b'Y', b'\*') => 4,

        (b'Y', b')') => 5,

        (b'Y', b'$') => 5,

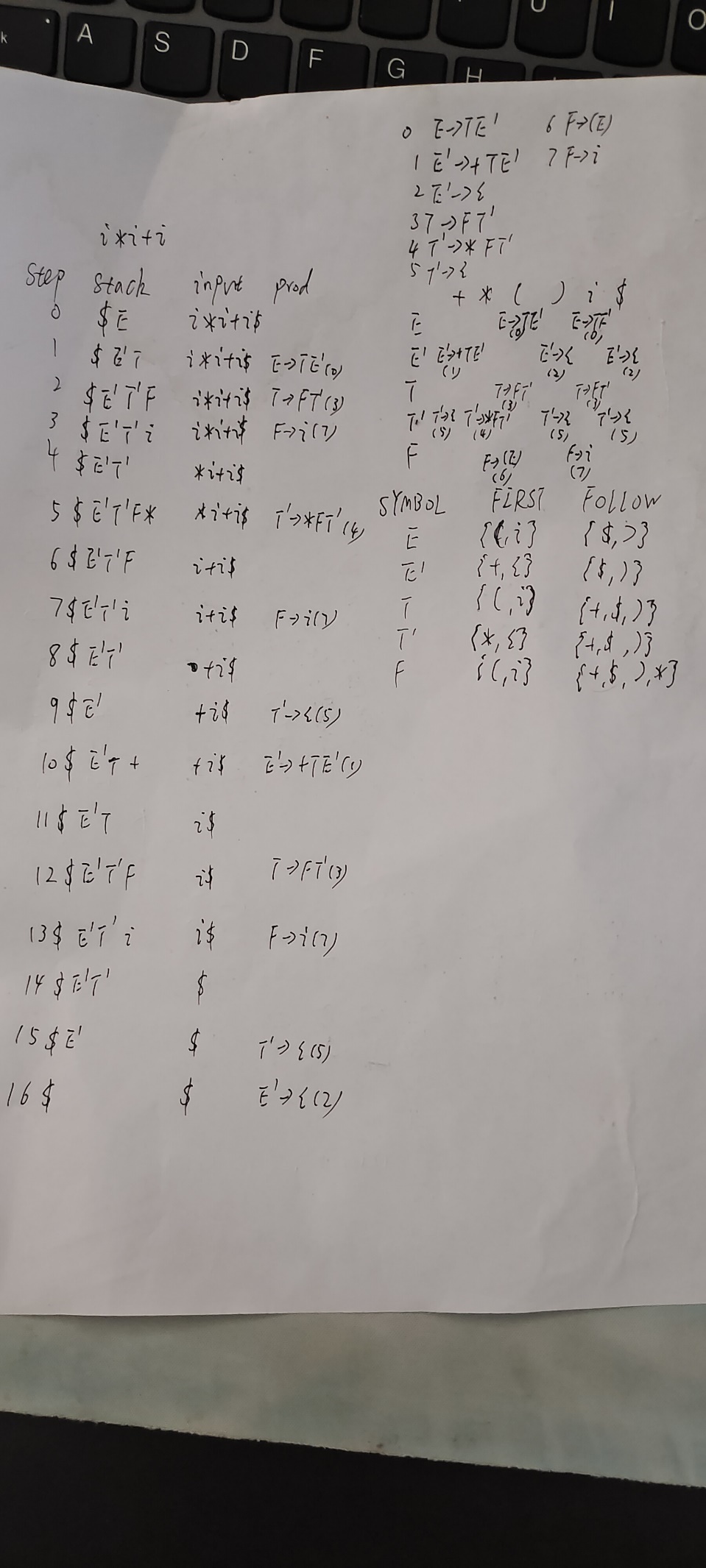
        (b'F', b'(') => 6,

        (b'F', b'i') => 7,

        \_ => panic!("Failure in grammatical analysis: no correct production available")

    }

Hand-written version:



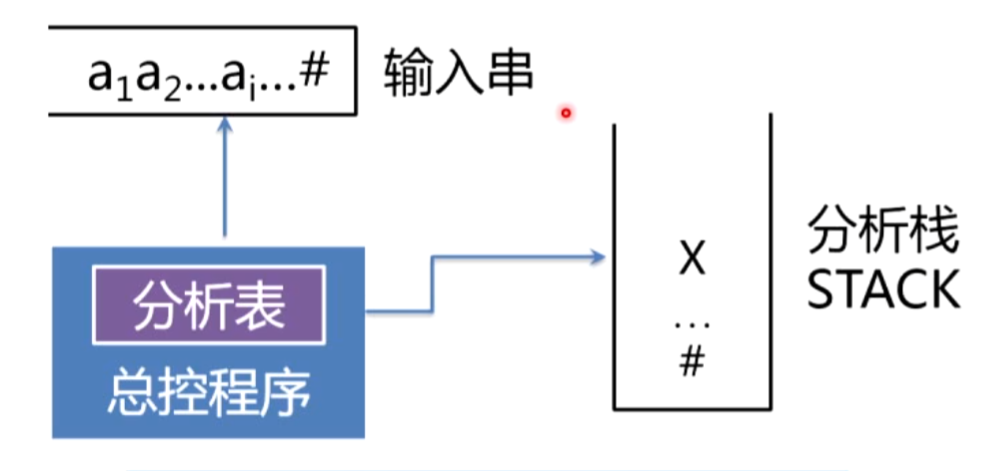
# **III. Ideas && Assumptions**

Assumptions:

Lexical analysis was finished beforehand.

Key idea explained with crucial snippets (algorithm):

I. Prepare a stack and a pointer to the input string.



    let mut stack = vec![b'$', b'E'];

    let input\_string = input\_string.to\_string() + "$";

    let mut prod = MATCH;

    let input\_string = input\_string.as\_bytes();

    let mut input\_pointer = 0usize;

II. Character on the top of the stack is popped out each time. Compare it with the current character the pointer to the input string references. Move forward the pointer if the popped character is a terminal and matched with the character from the input string. Push to stack in a reversed order the corresponding righthand part of the chosen production (decision made with LL(1) table).

    while stack.len() != 0 {

        res.push(print\_current(&stack, input\_string, input\_pointer, step, prod));

        print!("{}", res.last().unwrap());

        step += 1;

        prod = ll\_1\_match(stack.pop().unwrap(), input\_string[input\_pointer]);

        if prod == MATCH {

            input\_pointer += 1;

        } else {

            let to\_push = productions[prod].1;

            if to\_push == "ε" {

                continue;

            }

            let mut to\_push = String::from(to\_push);

            let to\_push = unsafe { to\_push.as\_bytes\_mut() };

            to\_push.reverse();

            to\_push.iter().for\_each(|el| stack.push(\*el));

        }

    }

Details in LL(1) table search.

*/\**

*\* return MATCH if terminal is given and matched with the first input string character*

*\* else the index of productions*

*\* panic if the entry is blank (error - failure in grammatical analysis)*

*\*/*

fn ll\_1\_match(stack\_top: u8, input\_first: u8) -> usize {

*/\* terminal \*/*

    if stack\_top == input\_first {

        return MATCH;

    }

    if stack\_top.is\_ascii\_lowercase() {

        panic!("Failure in grammatical analysis: terminal not matched");

    }

*/\* maybe not good for efficiency, but fine for readability \*/*

    match (stack\_top, input\_first) {

        (b'E', b'(') => 0,

        (b'E', b'i') => 0,

        (b'R', b'+') => 1,

        (b'R', b')') => 2,

        (b'R', b'$') => 2,

        (b'T', b'(') => 3,

        (b'T', b'i') => 3,

        (b'Y', b'+') => 5,

        (b'Y', b'\*') => 4,

        (b'Y', b')') => 5,

        (b'Y', b'$') => 5,

        (b'F', b'(') => 6,

        (b'F', b'i') => 7,

        \_ => panic!("Failure in grammatical analysis: no correct production available")

    }

}

# **IV. Complement Explanation**

Project directories is provided as the image on the right side.

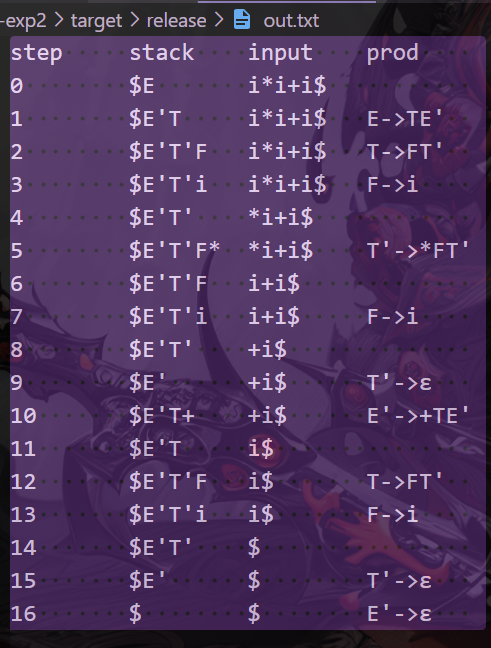
 ~~To run the the program, install rust and run “cargo run” under the src directory.~~

Or run the .exe file directly (no environment requirement), input the string to analyze and see the result in out.txt. (but highly recommended to run in terminal so that error handling messages will be given)

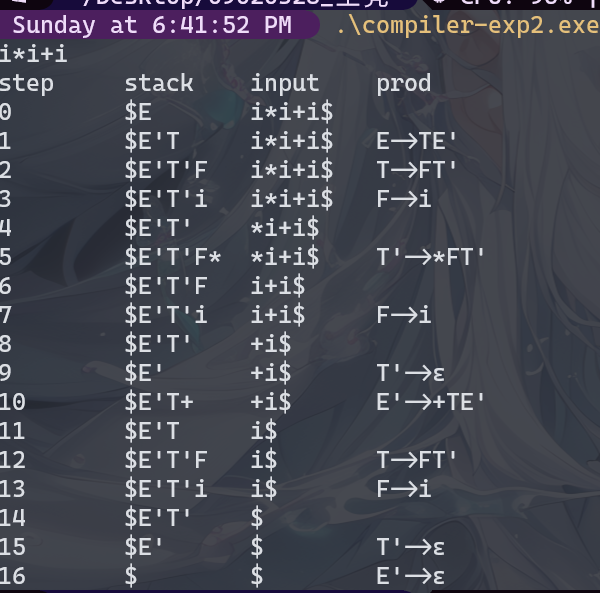
Directly:

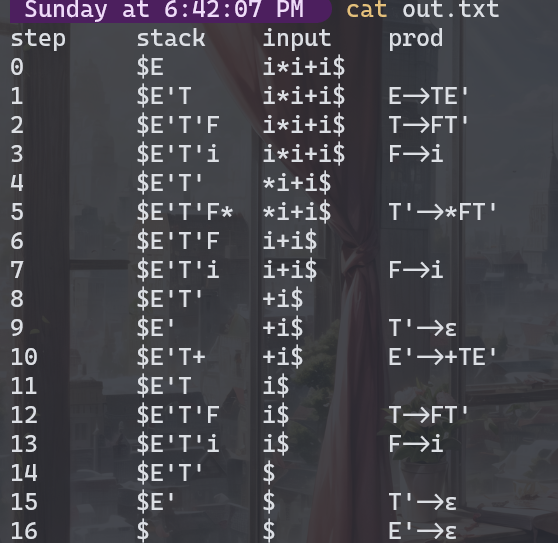


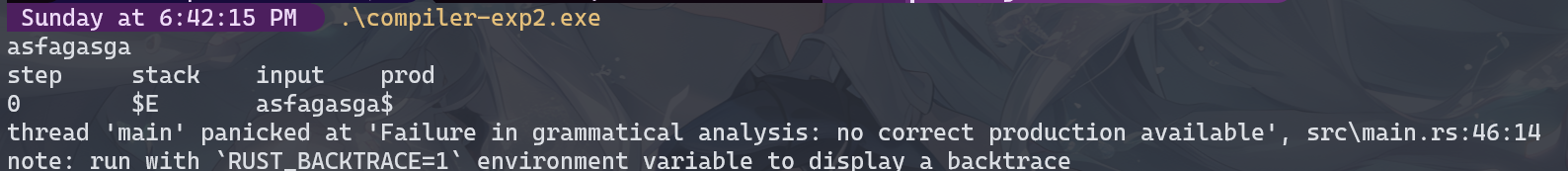




Terminal:







# **V. Example on running**

See full snippets of the program in the appendix.

Here is one example to demonstrate the functionality of the program.

Input sting:

"i\*i+i"

CFG:

const productions: [Prod; 8] = [

    Prod(b'E', "TR"), *// 0 E->TE'*

    Prod(b'R', "+TR"), *// 1 E'->+TE'*

    Prod(b'R', "ε"), *// 2 E'->ε*

    Prod(b'T', "FY"), *// 3 T->FT'*

    Prod(b'Y', "\*FY"), *// 4 T'->\*FT'*

    Prod(b'Y', "ε"), *// 5 T'->ε*

    Prod(b'F', "(E)"), *// 6 F->(E)*

    Prod(b'F', "i"), *// 7 F->i*

];

Inside out.txt:

step     stack    input    prod

0        $E       i\*i+i$

1        $E'T     i\*i+i$   E->TE'

2        $E'T'F   i\*i+i$   T->FT'

3        $E'T'i   i\*i+i$   F->i

4        $E'T'    \*i+i$

5        $E'T'F\*  \*i+i$    T'->\*FT'

6        $E'T'F   i+i$

7        $E'T'i   i+i$     F->i

8        $E'T'    +i$

9        $E'      +i$      T'->ε

10       $E'T+    +i$      E'->+TE'

11       $E'T     i$

12       $E'T'F   i$       T->FT'

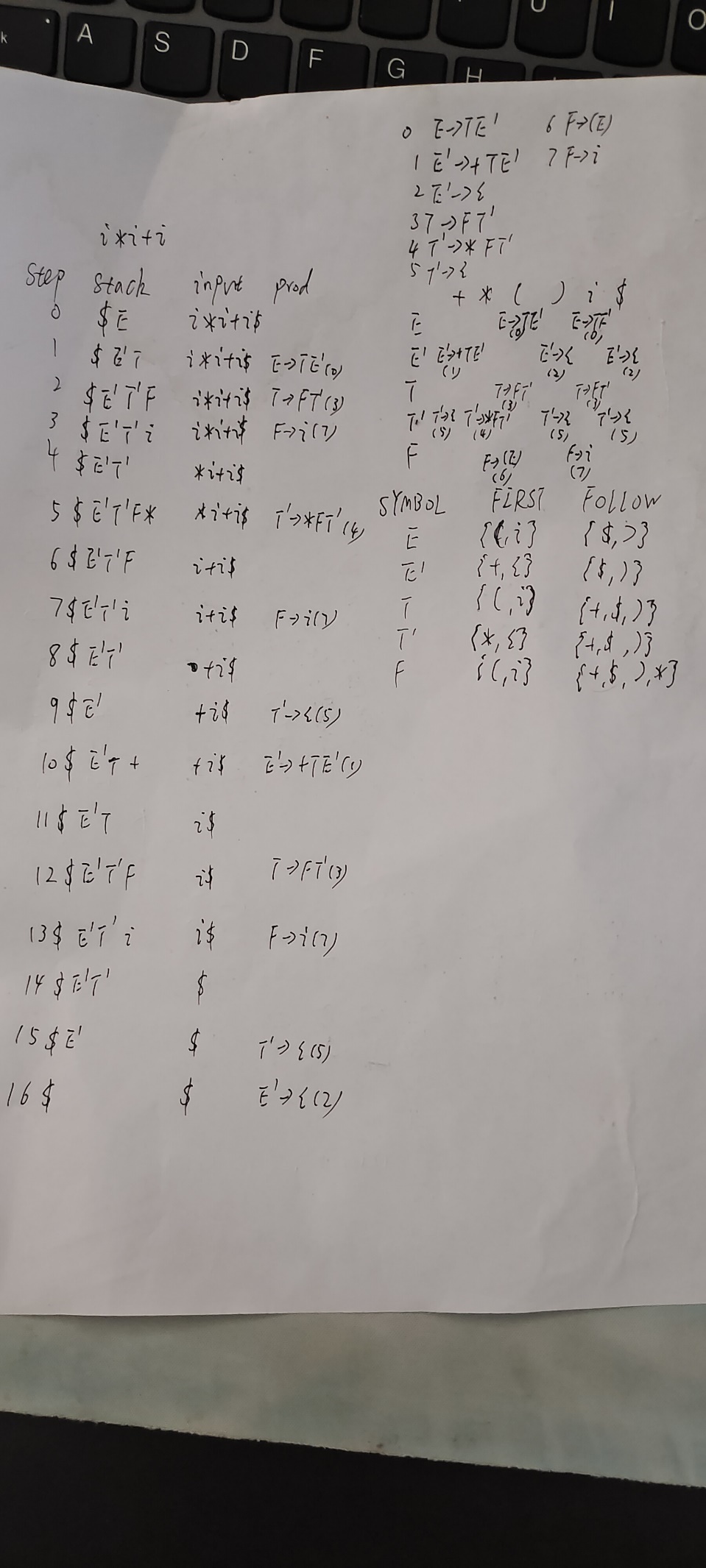
13       $E'T'i   i$       F->i

14       $E'T'    $

15       $E'      $        T'->ε

16       $        $        E'->ε

For comparison:



# **VI. Review && Comments**

The process of syntactic is quite plain. The main focus resolves around how to construct a correct LL(1) table and judge if the given grammar satisfies LL(1) limitations. Out of simplicity, the work is done by hand. The hardest part in the program should be credited to how rust tackles ascii.

# **Appendix**

use std::{fs, io};

*/\* u8 for 'char' replacement in C, as char in rust takes four bytes \*/*

struct Prod<'a>(u8, &'a str);

*/\* E' -> R    T' -> Y \*/*

const productions: [Prod; 8] = [

    Prod(b'E', "TR"), *// 0 E->TE'*

    Prod(b'R', "+TR"), *// 1 E'->+TE'*

    Prod(b'R', "ε"), *// 2 E'->ε*

    Prod(b'T', "FY"), *// 3 T->FT'*

    Prod(b'Y', "\*FY"), *// 4 T'->\*FT'*

    Prod(b'Y', "ε"), *// 5 T'->ε*

    Prod(b'F', "(E)"), *// 6 F->(E)*

    Prod(b'F', "i"), *// 7 F->i*

];

const MATCH: usize = usize::MAX;

*/\**

*\* return MATCH if terminal is given and matched with the first input string character*

*\* else the index of productions*

*\* panic if the entry is blank (error - failure in grammatical analysis)*

*\*/*

fn ll\_1\_match(stack\_top: u8, input\_first: u8) -> usize {

*/\* terminal \*/*

    if stack\_top == input\_first {

        return MATCH;

    }

    if stack\_top.is\_ascii\_lowercase() {

        panic!("Failure in grammatical analysis: terminal not matched");

    }

*/\* maybe not good for efficiency, but fine for readability \*/*

    match (stack\_top, input\_first) {

        (b'E', b'(') => 0,

        (b'E', b'i') => 0,

        (b'R', b'+') => 1,

        (b'R', b')') => 2,

        (b'R', b'$') => 2,

        (b'T', b'(') => 3,

        (b'T', b'i') => 3,

        (b'Y', b'+') => 5,

        (b'Y', b'\*') => 4,

        (b'Y', b')') => 5,

        (b'Y', b'$') => 5,

        (b'F', b'(') => 6,

        (b'F', b'i') => 7,

        \_ => panic!("Failure in grammatical analysis: no correct production available")

    }

}

fn format\_output(str: String) -> String {

    str.replace("R", "E'").replace("Y", "T'")

}

fn print\_current(stack: &Vec<u8>, input\_string: &[u8], input\_pointer: usize, step: u8, prod: usize) -> String {

    let production = if prod == MATCH {

        "".to\_string()

    } else {

        let prod = &productions[prod];

        let prod = (prod.0 as char).to\_string() + "->" + prod.1;

        format\_output(prod)

    };

    format!("{:<8} {:8} {:8} {:8}\n", step, format\_output(String::from\_utf8\_lossy(stack).to\_string()), String::from\_utf8\_lossy(&input\_string[input\_pointer..]), production)

}

fn analyze(input\_string: &str) -> Vec<String> {

    let mut res = vec![];

    let mut stack = vec![b'$', b'E'];

    let input\_string = input\_string.to\_string() + "$";

    let mut prod = MATCH;

    let input\_string = input\_string.as\_bytes();

    let mut input\_pointer = 0usize;

    let mut step = 0u8;

    res.push(format!("{:8} {:8} {:8} {:8}\n", "step", "stack", "input", "prod"));

    print!("{}", res[0]);

    while stack.len() != 0 {

        res.push(print\_current(&stack, input\_string, input\_pointer, step, prod));

        print!("{}", res.last().unwrap());

        step += 1;

        prod = ll\_1\_match(stack.pop().unwrap(), input\_string[input\_pointer]);

        if prod == MATCH {

            input\_pointer += 1;

        } else {

            let to\_push = productions[prod].1;

            if to\_push == "ε" {

                continue;

            }

            let mut to\_push = String::from(to\_push);

            let to\_push = unsafe { to\_push.as\_bytes\_mut() };

            to\_push.reverse();

            to\_push.iter().for\_each(|el| stack.push(\*el));

        }

    }

    res

}

fn main() {

    let mut input = String::new();

    io::stdin().read\_line(&mut input).expect("Fail to read input string");

    fs::write("out.txt", analyze(&input.trim()).join("")).unwrap();

}