**Assignment One**

**Report**

-

09020328 王亮

(Note: If you have any trouble understanding the first three parts, jump to the fourth)

# **I. Motivation/Aim**

(1) Further explore finite automata.

(2) Implement a lexical analyzer to analyze C++ program (This can be easily switched to other languages by modifying the recognizable strings.)

# **II. Content Description**

Input: File name of cpp file.

Output: Sequence of tokens

Recognizable strings:

const keywords: &[&str] = &[

    "int",      "alignas",      "const\_cast", "static\_assert",

    "long",     "alignof",      "continue",   "static\_cast",

    "and",      "mutable",      "co\_await",   "struct",

    "and\_eq",   "co\_return",    "switch",     "namespace",

    "co\_yield", "new",          "template",   "asm",

    "decltype", "this",         "auto",       "noexcept",

    "default",  "bitand",       "not",        "thread\_local",

    "bitor",    "delete",       "not\_eq",     "throw",

    "bool",     "do",           "nullptr",    "true",

    "double",   "break",        "operator",   "try",

    "case",     "dynamic\_cast", "or",         "typedef",

    "catch",    "or\_eq",        "else",       "typeid",

    "char",     "private",      "typename",   "enum",

    "char8\_t",  "explicit",     "union",      "protected",

    "char16\_t", "public",       "unsigned",   "export",

    "using",    "char32\_t",     "register",   "extern",

    "class",    "false",        "virtual",    "reinterpret\_cast",

    "float",    "void",         "compl",      "requires",

    "for",      "volatile",     "concept",    "return",

    "friend",   "wchar\_t",      "short",      "const",

    "signed",   "while",        "consteval",  "goto",

    "sizeof",   "if",           "xor",        "constexpr",

    "inline",   "xor\_eq",       "constinit",  "static"

];

const separators: &[char] = &[

    ',', ';', '.', '(',

    ')', '[', ']', '{',

    '}', '#'

];

const operators: &[&str] = &[

    "+",  "-",  "++", "--",

    "\*",  "/",  "<",  "<=",

    ">",  ">=", "=",  "==",

    "!=", "<<", ">>", "&&",

    "||"

];

const spaces: &str = " \n\r\t";

# **III. Ideas && Assumptions && Related FA Description**

FA && Assumptions:

# 

(Note that different keywords, separators and operators all have their exclusive code)

Key idea explained with crucial snippets (algorithm):

I. Read text from file.

*/\* Encapsulated file reader \*/*

    let content = read\_safe(&path.trim());

II. Loop until all tokens are detected; each time the function returns, we get (token we want, the id code suggesting its type, pointer to the current position of file, how many lines we have crossed) ( or panic at errors)

while i < end {

        (buffer, id, i, line) = scan(&content, i, line, &mut res);

        let token = format!("({}, {})", if buffer == "" {"Not supported"} else {&buffer}, id);

        println!("{}", token);

        res.push\_str(&(token + "\r\n"));

    }

III. Inside our core function – scan:

*/\**

*\* A token will be generated each time 'scan' gets invoked.*

*\* The program is originally designed to generate tokens out of C++ code text;*

*\* however, not all features are included in the target list*

*\* eg. ` ` - whitespace (skip), `'` - single quote, `''` - double quote, `//` - comment (Not supported)*

*\* comments, punctuation and string literals are not supported*

*\* unicode text is supported*

*\*/*

fn scan(content: &Vec<char>, pointer: usize, line: i32, res: &mut String) -> (String, i32, usize, i32) {

    let mut line = line;

    let end = content.len();

*/\* token id (indicate the category) \*/*

    let mut id = 1453\_05\_29;

*/\* token buffer \*/*

    let mut buffer = String::new();

*/\* Rust only accepts usize as type of indices \*/*

    let mut i: usize = pointer;

*/\* Scan, do nothing if space occurs, start new line if '\n' occurs \*/*

    while i < end && is\_space(&content[i]) {

        if is\_new\_line(&content[i]) {

            line += 1;

            new\_line(line, res);

        }

        i += 1;

    }

*/\* guarantee safe access to i in the following code \*/*

    if i == end {

        return (buffer, id, i, line);

    }

*/\* judge the type of character encountered, act according to the type \*/*

    match judge\_type(&content[i]) {

*/\* token starts with digit -> integer | float \*/*

        Type::Digit => {

*/\* accept both . and digits, try to get all the acceptable pushed to buffer \*/*

            while i < end && (is\_float(&content[i]) || is\_digit(&content[i])) {

                buffer.push(content[i]);

                i += 1;

            }

*/\* error handling for multiple float points \*/*

            if buffer.matches('.').count() > 1 {

                panic!("{}", "Reject: multiple float points".to\_string());

            }

*/\* error handling for digits followed by letter \*/*

            if i < end && is\_letter(&content[i]) {

                panic!("{}", "Reject: letter appeared right after digits".to\_string());

            } else if buffer.contains('.') {

                id = float\_code;

            } else {

                id = integer\_code;

            }

        },

*/\* token starts with letter -> identifier | keyword  \*/*

        Type::Letter => {

            while i < end && (is\_digit(&content[i]) || is\_letter(&content[i])) {

                buffer.push(content[i]);

                i += 1;

            }

            let code = is\_keyword(&buffer);

            id = if code == error\_code {identifier\_code} else {code};

        },

*/\* token starts with separator \*/*

        Type::Separator => {

            buffer.push(content[i]);

            id = is\_separator(&content[i]);

            i += 1;

        },

*/\* token starts with operator \*/*

        Type::Operator => {

            buffer.push(content[i]);

*/\* may take two positions \*/*

            if i + 1 < end && is\_operator(&(buffer.clone() + &content[i + 1].to\_string())) != error\_code {

                i += 1;

                buffer.push(content[i]);

            }

            id = is\_operator(&buffer);

            i += 1;

        },

*/\* ignore other occurrences \*/*

        Type::Other => {

            i += 1;

        }

    }

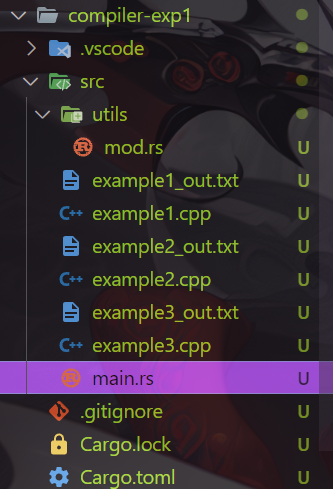
*/\* last sentence without ; will be returned in Rust \*/*

    (buffer, id, i, line)

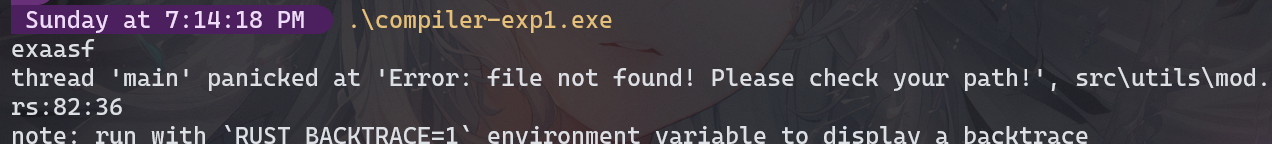
}

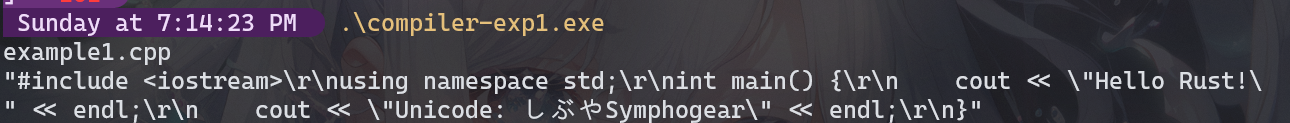
# **IV. Complement Explanation**

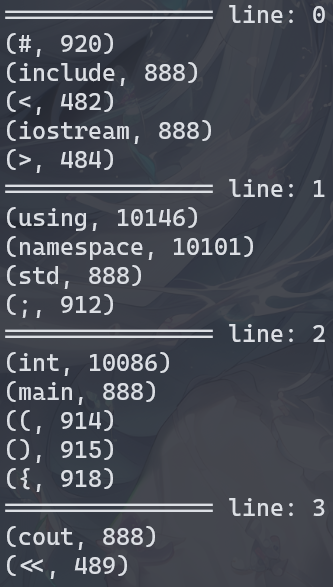
Project directories is provided as the image on the right side.

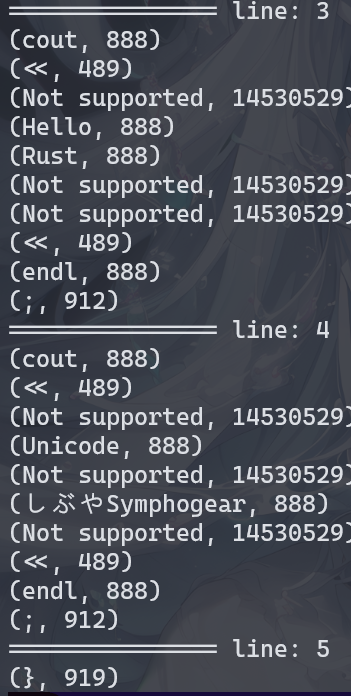
 ~~To run the the program, install rust and run “cargo run” under the src directory.~~

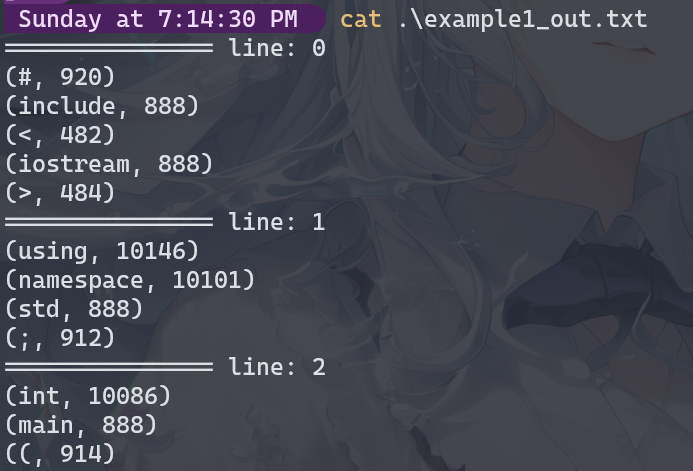
Or run the .exe file directly (no environment requirement), input the filename to analyze and see the result in {filename}\_out.txt. (but highly recommended to run in terminal so that error handling messages will be given)

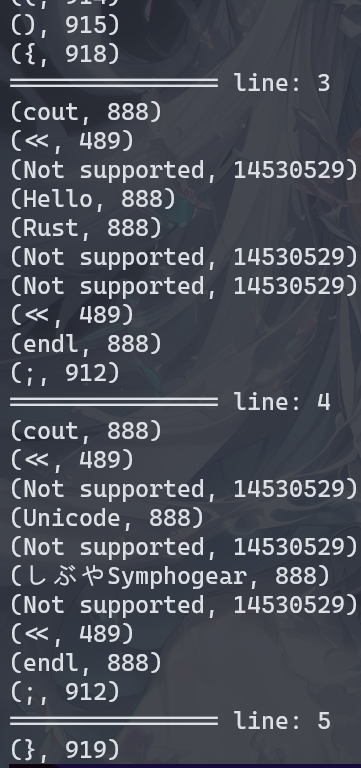












# **V. Example**

See full snippets of the program in the appendix (but read in source code files provided is highly recommended), here is one example to demonstrate the functionality of the program.

example1.cpp

#include <iostream>

using namespace std;

int main() {

    cout << "Hello Rust!" << endl;

    cout << "Unicode: しぶやSymphogear" << endl;

}

example1\_out.txt

=============== line: 0

(#, 920)

(include, 888)

(<, 482)

(iostream, 888)

(>, 484)

=============== line: 1

(using, 10146)

(namespace, 10101)

(std, 888)

(;, 912)

=============== line: 2

(int, 10086)

(main, 888)

((, 914)

(), 915)

({, 918)

=============== line: 3

(cout, 888)

(<<, 489)

(Not supported, 14530529)

(Hello, 888)

(Rust, 888)

(Not supported, 14530529)

(Not supported, 14530529)

(<<, 489)

(endl, 888)

(;, 912)

=============== line: 4

(cout, 888)

(<<, 489)

(Not supported, 14530529)

(Unicode, 888)

(Not supported, 14530529)

(しぶやSymphogear, 888)

(Not supported, 14530529)

(<<, 489)

(endl, 888)

(;, 912)

=============== line: 5

(}, 919)

# **VI. Review && Comments**

From my perspective, the code snippets are fully explained with comments. Should not take too much trouble understanding the program; whether the logic or the features on rust.

The key reason I adopted Rust is that where there is a possibility of error there is a must on error handling. Therefore, the program is unlikely to crash due to any unknown reason and only panic at the anticipated errors (like multiple float points).

For the program to implement the functionality, I do have another idea – we can actually make a class to abstract a finite automata, with state as its attribute, change as different characters occur.

# **Appendix**

Main.rs

*/\* Disable unnecessary warnings \*/*

#![allow(non\_upper\_case\_globals, unused\_mut)]

mod utils;

use utils::\*;

fn new\_line(line: i32, res: &mut String) {

    let string = format!("=============== line: {}", line);

    println!("{}", string);

    res.push\_str(&(string + "\r\n"));

}

*/\**

*\* A token will be generated each time 'scan' gets invoked.*

*\* The program is originally designed to generate tokens out of C++ code text;*

*\* however, not all features are included in the target list*

*\* eg. ` ` - whitespace (skip), `'` - single quote, `''` - double quote, `//` - comment (Not supported)*

*\* comments, punctuation and string literals are not supported*

*\* unicode text is supported*

*\*/*

fn scan(content: &Vec<char>, pointer: usize, line: i32, res: &mut String) -> (String, i32, usize, i32) {

    let mut line = line;

    let end = content.len();

*/\* token id (indicate the category) \*/*

    let mut id = 1453\_05\_29;

*/\* token buffer \*/*

    let mut buffer = String::new();

*/\* Rust only accepts usize as type of indices \*/*

    let mut i: usize = pointer;

*/\* Scan, do nothing if space occurs, start new line if '\n' occurs \*/*

    while i < end && is\_space(&content[i]) {

        if is\_new\_line(&content[i]) {

            line += 1;

            new\_line(line, res);

        }

        i += 1;

    }

*/\* guarantee safe access to i in the following code \*/*

    if i == end {

        return (buffer, id, i, line);

    }

*/\* judge the type of character encountered, act according to the type \*/*

    match judge\_type(&content[i]) {

*/\* token starts with digit -> integer | float \*/*

        Type::Digit => {

*/\* accept both . and digits, try to get all the acceptable pushed to buffer \*/*

            while i < end && (is\_float(&content[i]) || is\_digit(&content[i])) {

                buffer.push(content[i]);

                i += 1;

            }

*/\* error handling for multiple float points \*/*

            if buffer.matches('.').count() > 1 {

                panic!("{}", "Reject: multiple float points".to\_string());

            }

*/\* error handling for digits followed by letter \*/*

            if i < end && is\_letter(&content[i]) {

                panic!("{}", "Reject: letter appeared right after digits".to\_string());

            } else if buffer.contains('.') {

                id = float\_code;

            } else {

                id = integer\_code;

            }

        },

*/\* token starts with letter -> identifier | keyword  \*/*

        Type::Letter => {

            while i < end && (is\_digit(&content[i]) || is\_letter(&content[i])) {

                buffer.push(content[i]);

                i += 1;

            }

            let code = is\_keyword(&buffer);

            id = if code == error\_code {identifier\_code} else {code};

        },

*/\* token starts with separator \*/*

        Type::Separator => {

            buffer.push(content[i]);

            id = is\_separator(&content[i]);

            i += 1;

        },

*/\* token starts with operator \*/*

        Type::Operator => {

            buffer.push(content[i]);

*/\* may take two positions \*/*

            if i + 1 < end && is\_operator(&(buffer.clone() + &content[i + 1].to\_string())) != error\_code {

                i += 1;

                buffer.push(content[i]);

            }

            id = is\_operator(&buffer);

            i += 1;

        },

*/\* ignore other occurrences \*/*

        Type::Other => {

            i += 1;

        }

    }

*/\* last sentence without ; will be returned in Rust \*/*

    (buffer, id, i, line)

}

fn main() {

    let mut path = String::from("");

*/\* dis-comment the following statement to input given file name, pay attention to the location of file \*/*

    std::io::stdin().read\_line(&mut path).unwrap();

*/\* Encapsulated file reader \*/*

    let content = read\_safe(&path.trim());

    let file\_name: Vec<&str> = path.split(".").collect();

    let file\_name = file\_name[0].to\_string();

    let file\_name = file\_name + "\_out.txt";

    println!("{:?}", content);

*/\* String access with indices is not allowed in Rust, i.e "random words"[0] -> X \*/*

    let content: Vec<char> = content.chars().collect();

    let end = content.len();

    let mut buffer;

    let mut id;

    let mut i = 0;

    let mut line = 0;

    let mut res = String::new();

    new\_line(line, &mut res);

    while i < end {

        (buffer, id, i, line) = scan(&content, i, line, &mut res);

        let token = format!("({}, {})", if buffer == "" {"Not supported"} else {&buffer}, id);

        println!("{}", token);

        res.push\_str(&(token + "\r\n"));

    }

    write\_safe(&file\_name, &res);

}

Mod.rs

use std::{fs::File, io::{ErrorKind, Read, Write}};

#[derive(PartialEq)]

pub enum Type {

    Digit,

    Letter,

    Separator,

    Operator,

    Other

}

pub fn judge\_type(ch: &char) -> Type {

    if is\_digit(ch) {

        return Type::Digit;

    } else if is\_letter(ch) {

        return Type::Letter;

    } else if is\_separator(ch) != error\_code {

        return Type::Separator;

    } else if is\_operator(&ch.to\_string()) != error\_code {

        return Type::Operator;

    } else {

        return Type::Other;

    }

}

const keywords: &[&str] = &[

    "int",      "alignas",      "const\_cast", "static\_assert",

    "long",     "alignof",      "continue",   "static\_cast",

    "and",      "mutable",      "co\_await",   "struct",

    "and\_eq",   "co\_return",    "switch",     "namespace",

    "co\_yield", "new",          "template",   "asm",

    "decltype", "this",         "auto",       "noexcept",

    "default",  "bitand",       "not",        "thread\_local",

    "bitor",    "delete",       "not\_eq",     "throw",

    "bool",     "do",           "nullptr",    "true",

    "double",   "break",        "operator",   "try",

    "case",     "dynamic\_cast", "or",         "typedef",

    "catch",    "or\_eq",        "else",       "typeid",

    "char",     "private",      "typename",   "enum",

    "char8\_t",  "explicit",     "union",      "protected",

    "char16\_t", "public",       "unsigned",   "export",

    "using",    "char32\_t",     "register",   "extern",

    "class",    "false",        "virtual",    "reinterpret\_cast",

    "float",    "void",         "compl",      "requires",

    "for",      "volatile",     "concept",    "return",

    "friend",   "wchar\_t",      "short",      "const",

    "signed",   "while",        "consteval",  "goto",

    "sizeof",   "if",           "xor",        "constexpr",

    "inline",   "xor\_eq",       "constinit",  "static"

];

const separators: &[char] = &[

    ',', ';', '.', '(',

    ')', '[', ']', '{',

    '}', '#'

];

const operators: &[&str] = &[

    "+",  "-",  "++", "--",

    "\*",  "/",  "<",  "<=",

    ">",  ">=", "=",  "==",

    "!=", "<<", ">>", "&&",

    "||"

];

const spaces: &str = " \n\r\t";

pub const error\_code: i32 = -1;

pub const integer\_code: i32 = 666;

pub const float\_code: i32 = 777;

pub const identifier\_code: i32 = 888;

const keyword\_shift: i32 = 10086;

const separator\_shift: i32 = 911;

const operator\_shift: i32 = 476;

pub fn read\_safe(path: &str) -> String {

    let mut res = String::new();

    match File::open(path) {

        Ok(mut file) => match file.read\_to\_string(&mut res) {

            Ok(\_) => res,

            Err(err) => panic!("Error: error occurred while reading file: {:?}", err)

        },

        Err(err) => match err.kind() {

            ErrorKind::NotFound => panic!("Error: file not found! Please check your path!"),

            \_ => panic!("Error: error occurred while opening file: {:?}", err)

        }

    }

}

pub fn write\_safe(path: &str, content: &str) {

    match File::create(path) {

        Ok(mut file) => match file.write(content.as\_bytes()) {

            Ok(\_) => {},

            Err(err) => panic!("Error: error occurred while writing file, {:?}", err)

        },

        Err(err) => panic!("Error: error occurred while creating file, {:?}", err)

    }

}

pub fn is\_digit(ch: &char) -> bool {

    ch.is\_ascii\_digit()

}

*// bytes() would be adopted but for the support for Unicode*

pub fn is\_letter(ch: &char) -> bool {

    ch.is\_alphabetic()

}

pub fn is\_keyword(string: &str) -> i32 {

    match keywords.iter().position(|&r| r == string) {

        Some(pos) => keyword\_shift + pos as i32,

        None => error\_code

    }

}

pub fn is\_separator(ch: &char) -> i32 {

    match separators.iter().position(|&r| r == \*ch) {

        Some(pos) => separator\_shift + pos as i32,

        None => error\_code

    }

}

pub fn is\_operator(string: &str) -> i32 {

    match operators.iter().position(|&r| r == string) {

        Some(pos) => operator\_shift + pos as i32,

        None => error\_code

    }

}

pub fn is\_space(ch: &char) -> bool {

    spaces.contains(\*ch)

}

pub fn is\_new\_line(ch: &char) -> bool {

    \*ch == '\n'

}

pub fn is\_float(ch: &char) -> bool {

    \*ch == '.'

}