Assignment Number	2
Version	1
Print Date	28/09/17
Page	Page 1 of 14

Rust Report

SIGNATURE BLOCK				
Statement	I did my share of the work, and I have a general understanding of the contents of the assignment.			
Team Member	**Contribution**	**% of Total**	**Signature**	**Date**
Asher Mckoy	*Code/Report*	*25%*		*28/09/2017*
Raphael Cal	*Code/Presentation Slides*	*25%*		*28/09/2017*
Whitney Garbutt	*Code/Report*	*25%*		*28/09/2017*
Paul Yong	*Code/Report*	*25%*		*28/09/2017*

Table of Contents

Content **Page**

BNF Grammar

```
 <program>  → begin <stmt_list>  end
<stmt_list>  → <stmt> | <stmt> ; <stmt_list>
      <stmt> → <var> = <expression>
<expression> → <var> + <var> | <var> - <var> | <var>
        <var> → x | y | z
```

The BNF Grammar, also known as "**Backus-Naur form**," is used to describe the syntax of languages in programming languages. The BNF in this program represents the derivation and syntax that our program will print from the user entering when they enter a series of variables, expressions and statements. The input that the user enters must commence with the keyword *begin* and terminate with the keyword *end*. <stmt_list> can consist of a single, or multiple <stmt>s. A <stmt> must be in the form of <var> = <expression>. An <expression> can consist of two variables being added or subtracted, or just a variable by itself, and it must match the pattern <var>+<var> or <var>-<var> or <var>. A <var> can only be a limited set of values which are x, y, or z.

Examples:

Program
"begin x=y+z; y=z+x; z=x+y end"
Statement list – "x=y+z; y=z+x" or "y=z+x; z=x+y"
Statements – "x=y+z" or "y=z+x" or "z=x+y"
Expressions – "y+z" or "z+x" or "x+y"
Variables – x,y,z

Programming Language

Flavor

The programming language used here is the Rust programming language. Rust does not have a flavor, which is an early object-oriented extension to Lisp. Flavor's terminology of flavor means class in programming language. Rust does not provide a class nor does it allow the user to create a class, however, a standard library is provided for the user in Rust.

Version

The version of Rust that we are currently using is version 1.20.0. Rust is an open source programming language which means that its source code is made available with a license from the copyright holder. The community of people who created Rust, as well as other programmers, are making constant updates to make Rust's experience more rewarding.

Compiler/Interpreter

Rust programming language uses an interpreter to compile its code. The code can be edited on any text editor (such as notepad) that supports the Rust file extension, rs. When the file is saved, it needs to be saved as a Rust source file, which ends with the extension ".rs". For the interpreter to compile Rust's code, the user must open the command prompt and enter the path to the file that was created. After the path file has been set, the file must be linked and compiled. To do this, the user will have to enter the following line of command: "rustc {.rs file}". An executable will then be created if there are no compile errors, allowing the user to be able to access it manually via clicking on the executable or executing it via the command prompt. Should the user choose to access the executable via command prompt, the user will have to type in the following line of command: ".\executable.exe". The name of the executable file will be the same as your ".rs" file, but replaces the ".rs" extension with ".exe". To check if the executable was created, the user should type "ls" on linux or "dir" on windows. This will show all files within that folder for which the user has specified the path file.

Example:
"cd Desktop/RustProject" → This is the file of the path that was created
"rustc rustp.rs" → This complies the Rust source code file and creates an executable
".\rustp.exe" → This command runs the executable in the command line

Objective

The objective of this program set is to develop a program using the Rust programming language that will read the user's inputs and derivate them using the BNF programming language grammar. The program will accept what the user enters as a string and will verify that certain conditions are true or false using the BNF grammar rule. If conditions are true, then the string will be derivated based on the BNF rules, otherwise there will be a message showing that an error occurred. The string must begin with the word "begin" and end with the word "end". In the middle of both of these words are statements/statement lists/variables/expressions. If all conditions are met and the program grants its success, the program will proceed to do the string's right most derivation. Possible variables are x,y,z; possible operations are + or -.

Process flow

Upon starting the program, the BNF grammar will be displayed to the user, and a prompt to enter an input will be shown as well. Once the user has entered a string of their choice, the program will first evaluate if the string entered is equal to “quit”. If it is, then the program terminates. If the string is something else, the input is split by white-spaces and placed in a vector; this makes it easier to check and manipulate the sub-strings created. Then, the program will check that the keyword *begin* is the first word, and that the keyword *end* is the last word. If this is not the case, an error is displayed with an appropriate message, the BNF grammar along with the prompt is displayed again. Once the string includes the keywords in the right places, those are discarded so all that is left is the input that the derivation will be attempted on. The next check is to determine if semicolons are in the correct places, which should be at the end of each sub-string except the last one.
Since a rightmost derivation is to be performed, this program will start from the last element in the vector. The program will then look for a semi-colon and between the first semi-colons and “end” will be identified as a statement. How, if there is another semi-colon after the first, then what ever is between that semi-colon after the first one and end is identified as a statement list. Anything between “begin” and the second semi-colon will be identified as a statement. A BNF will be printed to the user that shows the rules and guidelines and asks the user to enter a string that follows the BNF’s rules. If an invalid or incomplete input is entered, an error represented as a message will be shown to the user. An error occurs if: “begin” and/or “end” is missing or spelled incorrectly; user enters an invalid operation; and if the user enters an invalid variable. The program will continue to prompt the user to enter a string until it matches the rules correctly.

Upon a successful input from the user, the program will perform 2 actions. The first of which is that the program will perform a RIGHTMOST derivation of the string. This means the program will derive the string starting from the back going left. The program will then display the process of the derivation. Subsequently, the program will then print a parse tree of how the derivation occurred. After everything that has occurred, the user will be prompted to continue with entering strings. If the user deemed so, they can proceed to do so or they can exit the program by entering “quit”.

Main

The main function starts with declaring arrays such as operations, variables. Thereafter, a loop is then created to do a series of procedures that receives and checks the input against the BNF grammar and performs a derivation of the input. These procedures are as follows: a function that displays BNF grammar; declarations of vectors; a great deal of "if" statements checking for validation at each step of the derivation process, outputs the derivation steps, and prints parse tree if a complete derivation is performed. Once all is done, the user will be prompted to enter a string again.

Subprograms/Functions

The following are subprograms/functions:

fn display_bnf_grammar(): This function displays the BNF grammar.

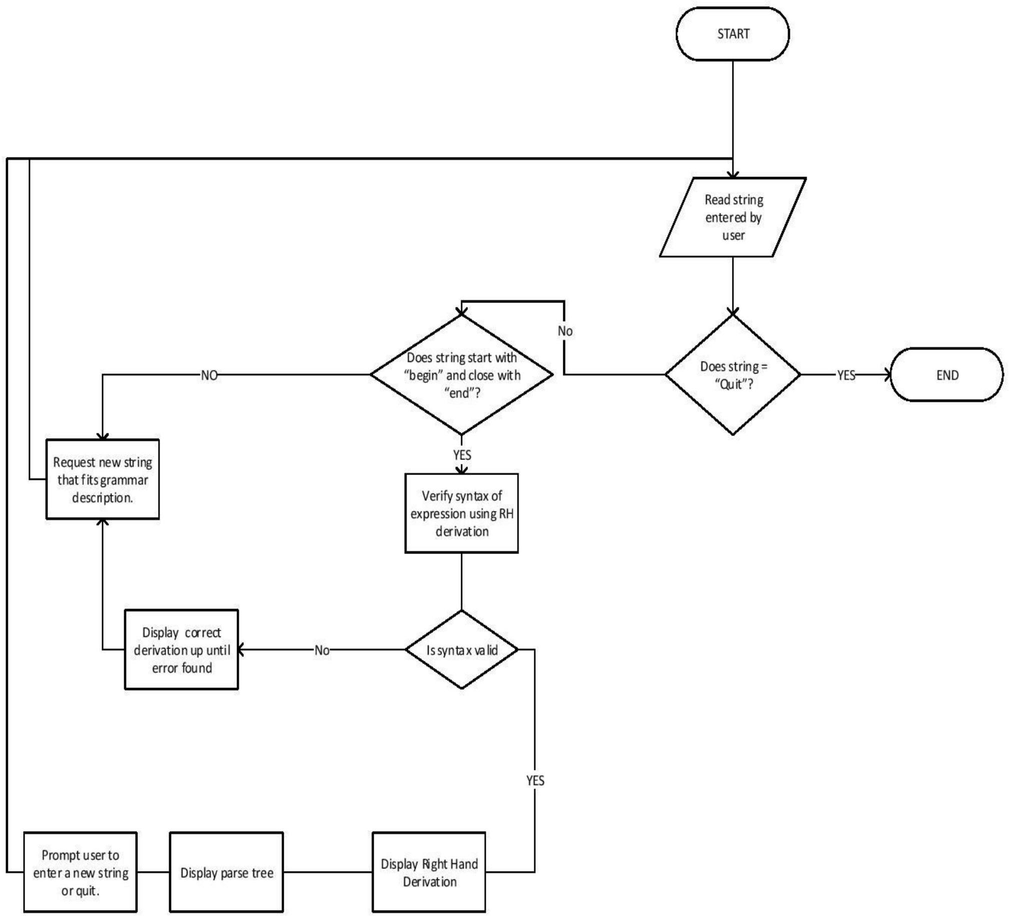
fn check_begin_end(): This function is a validation that checks if the first word if begin and checks if the last word is end.

Fn check_statement_list(): This function checks if a statement list is formed by looking for a semicolon. If a semicolon isn't where it should be, then this function returns false.

Fn check_statement(): This function checks if a statement is fully formed by first checking if the first character is not empty and if the second character is an equal sign. If either conditions or both aren't met, then this function returns false.

Fn print_parseTree(): This function prints a parse tree based on the string entered from the user only if the derivation was successful.

Flow Chart


START
Read string entered by user
Does string = "Quit"?
YES
END
No
Does string start with "begin" and close with "end"?
NO
YES
Request new string that fits grammar description.
Verify syntax of expression using RH derivation
Is syntax valid
No
Display correct derivation up until error found
YES
Display Right Hand Derivation
Display parse tree
Prompt user to enter a new string or quit.

Program Code

```
use std::io::prelude;
use std::env::*;
use std::io::{self, BufRead};
use std::*;


fn main()
{
let operations = ['+', '-']; //holds valid operations
let variables = ['x','y','z']; //holds valid variables
let initial_statement = "<stmt_list>";
let mut tree_char_vector:Vec<char> = Vec::new(); //vector of chars that is needed to create the
parse tree
let mut should_print_tree:bool = true; //will keep track to see if parse tree should be printed

loop
{
display_bnf_grammar();

println!("Please enter your string now: "); // prompts user for input
let input = io::stdin(); //gets input
let mut line = String::new(); //creates a string to hold the entire input
input.lock().read_line(&mut line).unwrap();

if(line.to_string().trim() == "quit") //user enters the word quit
{
println!("EXITING PROGRAM. THANK YOU!"); //displays message
std::process::exit(1); //exits the program
}

let split = line.split(" "); //splits the input into substrings by spaces
let mut vec = split.collect::<Vec<&str>>();//collects each substring generated and places them in
a vector

if(check_begin_end(vec[0], vec[vec.len()-1])) == true //checks for the word begin as the first
word and the word end as the last word
{
//keywords are in correct places
let mut final_vec = vec.clone(); //clones the input(makes a new identical one) but keeps the
original
final_vec.remove(0); //removes the word "begin"
let last_index = final_vec.len()-1; //gets the index of the input with the first word (begin)
removed
final_vec.remove(last_index);//removes the word "end", now all that is left is the substrings
that will attempt to derivate
let mut final_input = final_vec.clone(); //clones the input to consist of only substrings to
derivate
let mut temp_vec = final_vec.clone(); //create a new vector based on the final vector (without
begin and end)
let mut derivation_string = String::new().to_owned(); //holds substrings to derivate as one long
single string
```

```
let mut tree_vec:Vec<&str> = final_vec.clone();//vector needed to create the parse tree
let mut statement_string = String::new().to_owned(); //creates a new string
print!("program --> {}", line); //prints out what the user entered
statement_string.push_str(" --> ");
statement_string.push_str("begin <stmt_list> end");
print!("{}\n", statement_string); //prints begin <stmt_list> end, always the first thing that is
shown in derivation process

let vec_length = final_vec.len();
let mut current = 1;

/*
ATTEMPT AT PRINTING OUT APPROPRIATE LINES OF <STMT> & <STMT_LIST>
*/
/*
for i in current..vec_length
{
print!("{}", i);

if(i < vec_length)
{
statement_string.push_str("<stmt_list>");
}
else if(i == vec_length)
{
statement_string.replace("<stmt_list>", "<stmt>");
}
}
*/

if vec_length == 1
{
print!(" --> begin <stmt> end\n");
}
else if vec_length == 2
{
print!(" --> begin <stmt>;<stmt_list> end\n");
print!(" --> begin <stmt>;<stmt> end\n");
}
else if vec_length == 3
{
print!(" --> begin <stmt>;<stmt_list> end\n");
print!(" --> begin <stmt>;<stmt>;<stmt_list> end\n");
print!(" --> begin <stmt>;<stmt>;<stmt> end\n");
}
else if vec_length == 4
{
print!(" --> begin <stmt>;<stmt_list> end\n");
print!(" --> begin <stmt>;<stmt>;<stmt_list> end\n");
print!(" --> begin <stmt>;<stmt>;<stmt>;<stmt_list> end\n");
print!(" --> begin <stmt>;<stmt>;<stmt>;<stmt> end\n");
}
else if vec_length == 5
{
print!(" --> begin <stmt>;<stmt_list> end\n");
print!(" --> begin <stmt>;<stmt>;<stmt_list> end\n");
```

```
print!(" --> begin <stmt>;<stmt>;<stmt>;<stmt_list> end\n");
print!(" --> begin <stmt>;<stmt>;<stmt>;<stmt>;<stmt_list> end\n");
print!(" --> begin <stmt>;<stmt>;<stmt>;<stmt><stmt> end\n");
}
/*
END ATTEMPT
*/
// for loop travels down each element in the vector of strings to derivate and places them in one
string
for i in temp_vec
{
derivation_string.push_str(i.trim());
derivation_string.push(' ');
}
derivation_string.trim();

//for loop that travels down each character of the string from above and places each character as
an element of a char vector(includes whitespaces)
for e in derivation_string.chars()
{
tree_char_vector.push(e);
}

//for loop that removes all whitespaces from the char vector created above
for e in tree_char_vector.clone().iter() //this loop will remove spaces within the char vector
{
if let Some(index) = tree_char_vector.iter().position(|&i| i == ' ')
{
tree_char_vector.remove(index);
}
}
if check_statement_list(final_vec) //checks for correct placement of semicolons
{
//semicolons are where they belong
for element in (final_input).iter().rev()//start with rightmost substring (RIGHT DERIVATION)
{
let mut char_vec:Vec<char> = element.chars().collect();//creates a vector from the substring
(easier to index and check)
if let Some(index) = char_vec.iter().position(|&i| i == ';')//removes semicolon that exists in
the substrings
{
char_vec.remove(index); //remove ; (leaves with just characters to check for variables and
operations)
}

let length = char_vec.len(); //gets length of current vector
let last_index = length-1; //gets the last index of the current vector
println!("Current substring is: {:?}", char_vec);
if check_statement(element) //there is an equal sign and the character before the equal sign is
not empty
{
if char_vec.len() == 5 //there are 5 characters in the substring
{
println!(" --> <var> = <expr>");
if operations.contains(&char_vec[last_index-1]) //the second to last character is a valid
operation
```

```
{
println!(" --> <var> = <var> {} <var>",char_vec[last_index-1]);
if variables.contains(&char_vec[last_index]) //the last character is a valid variable
{
println!(" --> <var> = <var> {} {}", char_vec[last_index-1], char_vec[last_index]);
if variables.contains(&char_vec[last_index-2]) //the character before the operation is a valid
variable
{
println!(" --> <var> = {} {} {}", char_vec[last_index-2], char_vec[last_index-1],
char_vec[last_index]);
if variables.contains(&char_vec[0]) //the first character is a valid variable
{
println!(" --> {} = {} {} {}", char_vec[0], char_vec[last_index-2], char_vec[last_index-1],
char_vec[last_index]);
}
else
{
println!("ERROR! Invalid variable: {} in substring {}\nPlease review grammar and try again",
char_vec[0], element); //states error if variable is invalid at position and string
should_print_tree = false;
}
}
else
{
println!("ERROR! Invalid variable: {} in substring {}\nPlease review grammar and try again",
char_vec[last_index-2], element); //states error if variable is invalid at position and string
should_print_tree = false;
}
}
else
{
println!("ERROR! Invalid variable: {} in substring {}\nPlease review grammar and try again",
char_vec[last_index], element); //states error if variable is invalid at position and string
should_print_tree = false;
}
}
else
{
print!("ERROR! Invalid operation: {} in substring {}\nPlease review grammar and try again",
char_vec[last_index-1], element);//states error if operation is invalid at position and string
should_print_tree = false;
}
}
else if char_vec.len() == 3 //there are 3 characters in the substring (<var> op <var>)
{
println!(" --> <var> = <expr>");
println!(" --> <var> = <var>");
if variables.contains(&char_vec[char_vec.len()-1]) //the last character is a valid variable
{
println!(" --> <var> = {}", char_vec[char_vec.len()-1]);
if variables.contains(&char_vec[0]) //the first character is a valid variable
{
println!(" --> {} = {}", char_vec[0], char_vec[char_vec.len()-1]);
}
else
{
```

```
println!("ERROR! Invalid variable: {} in substring {}\nPlease review grammar and try again",
char_vec[0], element); //states error if variable is invalid at position and string
should_print_tree = false;
}
}
else
{
println!("ERROR! Invalid variable: {} in substring {}\nPlease review grammar and try again",
char_vec[char_vec.len()-1], element); //states error if variable is invalid at position and
string
should_print_tree = false;
}
}
else //input length is too short or too long
{
println!("ERROR! Invalid input string: {}\nPlease review grammar and try again", element);
should_print_tree = false;
}
}
}

if should_print_tree //if it is still true (no errors occured)
{
println!("\n\nPRINTING PARSE TREE\n-------------------------\n");
display_parse_tree(tree_vec);
}
}
else
{
println!("ERROR! Ensure that the semicolons are where they need to be\nPlease review grammar and
try again"); //semicolons are not found where they belong, state error and prompt for input again
}
}
else
{
println!("Failed to attempt derivation. Please make sure the input you entered starts with
\"begin\" and terminates with \"end\""); //keywords are not found where they belong, state error
and prompt for input again
}
}

}



fn display_bnf_grammar() //function prints the BNG grammar
{
print!("\n\n-------------------------------------------------\n");
println!(" BNF Grammar");
print!("\n");
println!(" <program> --> begin <stmt_list> end");
println!(" <stmt_list> --> <stmt>\n |<stmt> ; <stmt_list>");
println!(" <stmt> --> <var> = <expression>");
println!(" <var> --> x | y | z");
println!("<expression> --> <var> + <var>\n |<var> - <var>\n |<var>\n-----------------------------
--------------------\n");
```

```
}

fn check_begin_end(first: &str, last: &str) -> bool //checks that first word is begin and last
word is end
{
if (first.trim() == "begin") & (last.trim() == "end")
{
return true;
}
false
}

fn check_statement_list(fstring: Vec<&str>) -> bool //checks that all substrings EXCEPT the last
has a semicolon at the end
{
let mut temp = fstring.clone();
temp.remove(fstring.len()-1);
let mut last = fstring.last().clone();

for element in temp //checks that every substring ends with a semicolon except the last one
{
let char_vec:Vec<char> = element.chars().collect();
let ch = char_vec[char_vec.len()-1];
if ch != ';'
{
println!("A semicolon is missing at the end of statement: {}\nPlease review grammar and try
again.", element);
return false;
}
}

for element in last //checks that the last substring does not end with a semicolon
{
let char_vec:Vec<char> = element.chars().collect();
let ch = char_vec[char_vec.len()-1];
if ch == ';'
{
println!("A semicolon should not be at the end of the last statement: {}\nPlease review grammar
and try again", element);
return false;
}

}
true
}

fn check_statement(word: &str) -> bool // first character must not be empty and second character
has to be an equal sign
{
let char_vec:Vec<char> = word.chars().collect();
if char_vec[1] != '='
{
println!("An equal sign is missing at the second character of the substring {}\nPlease review
grammar and try again", word);
return false;
}
```

```
if char_vec[0].to_string().is_empty()
{
println!("The first character of the substring {} is empty\nPlease review grammar and try again",
word);
return false;
}
true
}

fn display_parse_tree(tree_elmts: Vec<&str>) //functions prints out the parse tree
{
let operations = ['+', '-'];
let variables = ['x','y','z'];
let mut final_string = String::new().to_owned();
final_string.push_str("program --> ");

  let mut index = tree_elmts.len() - 1;
  println!("<program>\n | ");
  println!("<stmt_list>\n |");
  for element in tree_elmts.iter()
  {
    let mut c_vec:Vec<char> = element.chars().collect();
    let mut last_index = c_vec.len()-1;

    if c_vec.len() == 5 //there are 5 characters in the substring
    {
      println!("<stmt> \n |");
      println!("<var> = <expr> \n | \t |");

      println!(" | \t<var> {} <var> \n | \t | \t |",c_vec[last_index-1]);
      println!(" {} \t {} \t {}",c_vec[0], c_vec[2], c_vec[last_index]);
    }
  }
}
```