COMP 725/825 - FINAL PROJECT

RASHMI

February 25, 2025

GRAMMAR IMPLEMENTATION FOR RUSH

<RUSH> ::= start { <block> } done

<block> ::= <statement>
| <statement> <block>

<statement> ::= output(<expr>)
| output(<string>)
| take(<var>)
| <var> = <expr>
| if (<cond>) { <block> }
| if (<cond>) { <block> } else { <block> }
| <loop>
| <function_def>
| <function_call>

<expr> ::= <val> + <val>
| <val> - <val>
| <val> * <val>
| <val> / <val>
| <val> % <val>
| <val>
| <function_call>

<cond> ::= <val> == <val>
| <val> > <val>
| <val> < <val>
| <val> != <val>

<val> ::= <num>
| <var>
<num> ::= <dig>
| <dig><num>

<var> ::= <char>
| <char><rest>

<rest> ::= <dig>
| <char>

| <dig><rest>
| <char><rest>

<string> ::= "<text>"

<text> ::= <dig>
| <char>
| <sym>
| <text><text>

<dig> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

<char> ::= a | b | c | d | e | f | g | h | i | j | k | l | m | n | o | p | q | r | s | t | u | v | w | x | y | z

<sym> ::= . | _

<loop> ::= iterate from <expr> to <expr> by <expr> { <block> }

<function_def> ::= define <var>(<var_list>) { <block> }

<var_list> ::= <var>
| <var>, <var_list>

<function_call> ::= <var>(<arg_list>)

<arg_list> ::= <expr>
| <expr>, <arg_list>

About Language Behavior Discussion

Introduction to RUSH Programming Language:
The programming language I am developing is called "RUSH". RUSH is inspired by other programming language syntax. For example, It used the concept of class from Java. Every code block in the RUSH must be written inside the below block format

```
start {

    Code...
} done
```

RUSH incorporates arithmetic operations to enable mathematical computations. Users can perform basic mathematical operations, including addition, subtraction, multiplication, division, and modulus, using clear and concise syntax inspired from the C, C++, Java.

One of the key features of RUSH is its support for function calls and function definitions. This functionality allows User to define reusable blocks of code, known as functions, and invoke them multiple times within a program.

Overall, RUSH is designed to be a basic programming language. We can play around the RUSH language to understand the basics of lex and Yacc concepts using basic RUSH programming language.

Key Features

- **Structured Syntax**: RUSH adopts a structured syntax, for organizing code into blocks and statements.

- **Expressive Statements**: RUSH offers a set of statements for performing common programming tasks such as conditional branching (if-else) it uses the same keywaords so that it is easy to write RUSH codes, looping, and function definition.

- **Variable Storing**: Variables in RUSH can store alphanumeric values, including single characters, numbers, and alphanumeric strings.

- **Arithmetic Operations**: RUSH supports basic arithmetic operations, including addition, subtraction, multiplication, division, and modulus. These operations can be performed on numeric values stored in variables or provided directly in expressions.

- **Looping Constructs**: RUSH provides looping constructs for iterating over a range of values.

- **Function Definition**: RUSH allows user to define functions with custom parameters and code blocks. RUSH supports function calls with arguments.

Grammar Specification for RUSH Programming Language

Below is a detailed explanation of the grammar rules and constructs of RUSH:

1. Entry Point and Block Structure:

- **<RUSH>**: The entry point of a RUSH program is marked by the **start** keyword, followed by an open parenthesis. It defines the beginning of the program execution.
- **<block>**: A block is a sequence of statements enclosed within parentheses. It represents a logical unit of code execution.

```
program: START block DONE block:
LBRACE statement_list RBRACE
statement_list: statement | statement statement_list
```

2. Statements:

- **<statement>**: Statements are the building blocks of RUSH programs. They represent individual actions or operations that the program can perform.
    - **output**: The **output** keyword is used to print the result of an expression or a string to the console.
    - **take**: The **take** keyword the user for input and assigns the provided value to a variable. (I tried to implement the code but this is the only part which is not working)
    - **if**: The **if** keyword allows conditional execution of code blocks based on the evaluation of a condition.
    - **else**: The **else** statement complements the **if** statement, providing an alternative code block to execute if the condition is not met.
    - **<loop>**: Loops enable repetitive execution of code blocks. The **iterate** statement defines a loop that iterates over a range of values.
    - **Variable assignment:** Variables can be assigned values using the assignment operator =.

```
o  statement: OUTPUT LPAREN expr RPAREN
o        | OUTPUT LPAREN STRING RPAREN
o        | TAKE LPAREN IDENTIFIER RPAREN
```

- | ***IDENTIFIER ASSIGN expr***
- | ***if_stmt***
- | ***loop_stmt***
- | ***function_def***

3. Expressions and Conditions:

- **<expr>**: Expressions represent mathematical or functional computations that produce a value.

  <cond>: Conditions are expressions that evaluate to a Boolean value.

- ***expr: expr PLUS term | expr MINUS term | term***
  term: term TIMES factor | term DIVIDE factor | term MOD factor | factor
  factor: NUMBER | IDENTIFIER | LPAREN expr RPAREN | function_call

  condition: expr EQ expr | expr NEQ expr | expr GT expr | expr LT expr | expr GTE expr | expr LTE expr

4. Values and Variables:

- **<val>**: Values represent numerical or alpha data.
- **<num>**: Numeric values consist of digits (0-9) or a sequence of digits.
  <var>: Variables are identifiers that store values. They can consist of alphanumeric characters and underscores, allowing for descriptive variable names.

  - IDENTIFIER: [a-z]
    NUMBER: [0-9]+

5. Strings and Text:

- **<string>**: Strings are sequences of characters enclosed within double quotes.

  <text>: Text represents alphanumeric strings, including letters, digits, and symbols.

  - ***STRING: \"[^\"]*\"***

6. Arithmetic Operations:

RUSH supports basic arithmetic operations such as addition, subtraction, multiplication, division.

- PLUS: '+'
  MINUS: '-'
  TIMES: '*'
  DIVIDE: '/'
  MOD: '%'

7. Function Definitions and Calls:

- **<function_def>**: Function definitions allow user to define reusable blocks of code with custom parameters. Function must be declared at the start of the code.
  <function_call>: Function calls invoke functions with specific arguments.

  - function_def: DEFINE IDENTIFIER LPAREN param_list RPAREN block
    function_call: IDENTIFIER LPAREN arg_list RPAREN
    param_list: /* empty */ | *IDENTIFIER* | *IDENTIFIER COMMA param_list*
    arg_list: */* empty */ | expr | expr COMMA arg_list

8. Miscellaneous Symbols:

- **<dig>**: Digits represent numerical values (0-9).
- **<char>**: Characters represent alphanumeric characters (a-z).

Code Examples

1. Variable Assignment and Output:

```
start {
        a = 10
        output(a)
} done In th
```

In this case, **(output << (x))** outputs the value stored in variable **x** to the console.

```
rl1228@linus-ubuntu:~/comp/825/demo$ ./rush < test.rush
RUSH Compiler
10
```

Execution:

1. The program begins execution with the **start** keyword.
2. **(x = 10)**: Assigns the value **10** to the variable **x**.
3. **output(x)**: Outputs the value of variable **x**, which is **10**, to the console.
4. The program execution is completed with the **done** keyword.

Output:

The output displayed on the console is **10**, indicating that the value stored in variable **x** is **10**. This demonstrates successful variable assignment and output in the RUSH programming language.

2. Conditional Statements:

```
start {
   a = 10
   if (a > 5) {
      output("a is greater than 5")
   } else {
      output("a is not greater than 5")
   }
} done
```

```
rl1228@linus-ubuntu:~/comp/825/demo$ ./rush < test.rush
RUSH Compiler
a is greater than 5
```

***Explanation*:**

This code snippet demonstrates the usage of conditional statements to output different messages based on the value of **x**. Here's how it works:

- If **x** is greater than 5, it outputs **"x is greater than 5"**.
- If **x** is not greater than 5, it outputs **"x is not greater than 5"**.

***Execution*:**

- The program starts execution with the **start** keyword.
- As a is 10, it outputs **"x is greater than 5"**.

4. Looping:

```
start {
   iterate from 1 to 5 by 1 {
      output(i)
   }
} done
```

```
rl1228@linus-ubuntu:~/comp/825/demo$ ./rush < test.rush
RUSH Compiler
0
1
2
3
4
5
```

***Execution*:**

- The program starts execution with the **start** keyword.
- It iterates over the values of **i** from 1 to 5, outputting each value to the console.

5. Function Definition and Call:

```
start {
   define f(x) {
      output("Function called with value:")
      output(x)
   }

   f(10)
   f(20)
} done
```

```
rl1228@linus-ubuntu:~/comp/825/demo$ ./rush < test.rush
RUSH Compiler
Function called with value:
0
Function called with value
10
Function called with valu
20
```

6. Arithmetic Operations:

```
start {
   a = 10
   b = 3

   c = a + b
   d = a - b
```

```
    e = a * b
    f = a / b
    g = a % b

    output(c)
    output(d)
    output(e)
    output(f)
    output(g)
} done
```

```
rl1228@linus-ubuntu:~/comp/825/demo$ ./rush < test.rush
RUSH Compiler
13
7
30
3
1
```

These examples illustrate various features and capabilities of the RUSH programming language, showcasing its versatility and utility in solving a wide range of programming problems.

Conclusion

RUSH is a simple programming language which uses lex and Yacc to implement the code designee. With its structured syntax and simple features, RUSH empowers first time user to understand the lex and Yacc implementation through RUSH programming.

.