LAKY

Team 33

Aaditya Muley(amuley2) | Li Che Hsu(lhsu16) | Karan Navin Javali(knjavali) | Yunhan Qiao(yqiao18)

Introduction:

Our programming language is called **LAKY (pronounced as lucky!!!)**, which is derived from the first character of each member's first name. We hope that by using LAKY, users will be fortunate enough to avoid encountering bugs and other issues.

LAKY is a programming language that supports numeric types (integers), boolean values, and string value assignments. It also supports variable assignment and relational and logical expressions.

The language includes constructs for if-else statements, for loops with both Java-style and Python-style syntax, and while loops. It also supports print statements.

The syntax for LAKY is defined using Backus-Naur Form (BNF), which is a formal notation used to describe programming language syntax.

Overall, LAKY is a simple programming language that can be used to perform basic calculations and conditional operations.

We are still deciding between Prolog or ANTLR and Java as our primary languages for our project. ANTLR is a popular choice for building parsers and supports many programming languages, making it a suitable choice for our needs. Additionally, Java is a powerful language that allows us to express complex structures and functionalities required for our project. But our familiarity with Prolog makes us more comfortable using it.

Interpreter:

We are debating upon 2 choices and which might be better to go ahead with:

1. **Prolog**:
   Prolog is a logic programming language that uses a formal system of rules to represent knowledge and reason about statements. It is commonly used for artificial intelligence, natural language processing, and other applications that require symbolic reasoning. Prolog has built-in support for backtracking, which makes it suitable for parsing and interpreting languages. In the case of LAKY, Prolog could be used to implement the interpreter by defining rules for each grammar rule in the BNF specification.
2. **ANTLR**:
   ANTLR (ANother Tool for Language Recognition) is a powerful parser generator that can be used to generate code for parsing and interpreting languages. It uses context-free grammar to generate a lexer and parser that can parse the input and generate an abstract syntax tree (AST) representing the input. ANTLR has support for multiple programming languages, including Java, C#, and Python. In the case of LAKY, ANTLR could be used to generate a parser and interpreter in a programming language of choice. The grammar rules specified in the BNF specification could be used to generate the lexer and parser, which would be responsible for parsing LAKY programs and generating an AST. The AST could then be used to interpret the program and execute it. Once the rules have been defined, ANTLR can be used to generate code for parsing LAKY programs and executing them in the target language.

In summary, both Prolog and ANTLR can be used to implement the LAKY language by defining rules for each grammar rule in the BNF specification. While Prolog is a logic programming language that is suitable for symbolic reasoning and familiar to us, ANTLR is a powerful parser generator that can be used to generate code in multiple programming languages.

Parsing Technique:

We are deciding upon one of these 2 parsing techniques for our language:

1. **LL(k) parsing**:
   LL(k) parsing is a top-down parsing technique that uses a lookahead of k tokens to decide which production rule to apply. This technique is commonly used in parser generators such as ANTLR and is efficient and scalable for most programming languages. LAKY's grammar is relatively simple, and LL(k) parsing is suitable for parsing this kind of grammar.
2. **Recursive descent parsing**:
   Recursive descent parsing is also a top-down parsing technique, but it uses recursive procedures to apply the production rules. This technique is easy to implement and understand, and it can be more efficient than LL(k) parsing for small and simple grammars like LAKY's.

Both techniques are suitable for implementing LAKY, and the choice between them depends on the specific requirements of the language implementation. However, if performance is a concern (which is not really a concern for such a simple language), LL(k) parsing is likely to be more efficient and scalable than recursive descent parsing.

Data Structures:

1. Primitive data types such as int, boolean, string
2. List - A collection of objects in []
3. Abstract syntax tree (AST): An AST is a tree-like data structure that represents the structure of a program's source code. It can be used to parse and represent the LAKY program, which makes it easy to traverse and manipulate. The nodes of the AST can represent different constructs of the LAKY language, such as variable declarations, assignment statements, and control flow statements.

Grammar Definition:

<program> ::= <block> '.'

<block> ::= 'begin' '#' <instructions> '#' 'end'

// This is the structure for the program
<instructions> ::= <variable_rule> '#' <instructions>
| <assignment_rule> '#' <instructions>
| <expression> ‘#’ <instructions>
| <ternary_rule> '#' <instructions>
| <if_else_rule> '#' <instructions>
| <for_javatype_rule> '#' <instructions>
| <for_pythontype_rule> '#' <instructions>
| <while_rule> '#' <instructions>
| <print_rule> '#' <instructions>
| ε

// Define rules for numeric type
<sign> ::= + | -
<digits> ::= <digit> | <digit> <digits>
<digit> ::= 0 | <non_zero_digit>
<non_zero_digit> ::= 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
<integer> ::= <digit>
| <non_zero_digit> <digits>
| <sign> <non_zero_digit> <digits>

// Define rules for boolean values
<booleans> ::= “true” | “false”

// Define rules for string value assignment
<string> ::= "<string_characters>"
<string_characters> ::= <string_character>

| <string_character> <string_characters>
<string_character> ::= <input_character> | <escape_character>
<input_character> ::= “all characters from the utf-8 character set"
<escape_character> ::= “ \n | \t | \ | "

// define rules for variable
<datatype> ::= <integer> | <booleans> | <string>
<variable_name> ::= [a-zA-Z] [a-zA-Z0-9]*
<variable_rule> ::= <datatype> <variable_name> | <variable_name>

// Define rules for arithmetic operators
<arithmatic_rule> ::= <arithmatic_rule1> <arithmatic_rule2>
<arithmatic_rule1> ::= * <variable_name> <arithmatic_rule1>
| / <variable_name> <arithmatic_rule1> |
| * <integer> <arithmatic_rule1>
| / <integer> <arithmatic_rule1>
| ε
<arithmatic_rule2> ::= + <variable_name> <arithmatic_rule2>
| - <variable_name> <arithmatic_rule2> |
| + <integer> <arithmatic_rule2>
| - <integer> <arithmatic_rule2>
| ε

// Define rules for relational operators
<relational_operators> ::= < | <= | > | >= | == | !=
<relational_rule> ::= <expression> <relational_operators> <expression>

// Deine rules for Logical operators
<logical_operators> ::= and | or
<logical_rule> ::= <expression> <logical_operators> <expression>

<expression> ::= <integer> | <variable_name> | (<expression>)
| <arithmatic_rule> ‘#’ <expression>
| <relational_rule> ‘#’ <expression>
| <logical_rule> ‘#’ <expression>
| ε

/ /Define rules for an assignment
<assignment_rule> ::= <variable_name> = <expression>
        | <expression> <assignment_rule1>
<assignment_rule1> ::= + <expression> <assignment_rule1>
        | - <expression> <assignment_rule1>
        | ε

// rule for the ternary operator
<ternary_rule> ::= <expression> <relational_rule> <expression> '?' <instructions>
':' <instructions>

// rule for if-else conditional
<if_else_rule> ::= if <relational_rule> # { # <instructions> # } # else # { #
<instructions> # }
        | if <relational_rule> # { # <instructions> # } #

// rule for Java-like for loop
<for_javatype_rule> ::= for <variable_name> = <integer> to <integer> # { #
<instructions> # } #

// rule for Python-like for loop
<for_pythontype_rule> ::= for <variable_name> in range (<digit> ,
<non_zero_digit>)

// rule for while construct
<while_rule> ::= while (<expression>) { <instructions> }