SER 502 - Class project

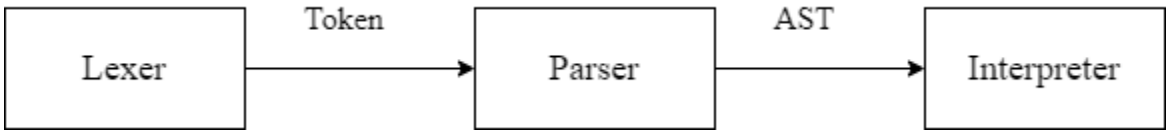
Team 5

Programming Language : Kiwi

Team members

1. Naga Venkata Dharani Vishwanadh Chinta (1228137224)
2. Vignesh Venkatachalam Iyer (1227792802)
3. Sri Harsha Gajavalli (1225891175)
4. Anish Nair (1225689179)
5. Vamsi Krishna Yadav Loya (1226303691)

Diagram:



Description:

- Kiwi is an imperative programming language that follows a minimalistic design for writing simple programs.
- The name given to the language emphasizes the simplicity and ease of use of the language.
- It has a syntax that is easy to learn, with familiar constructs for control flow, variables and functions found in most modern programming languages.
- Kiwi also includes a number of built-in data types(numbers, booleans, strings), logic operators(AND, OR, NOT). Additionally, it also supports the ternary operator.

Note: Information below is subject to change in future based on the implementation.

Language Grammar:
Program Structure: A program is composed of a series of functions and a main block containing declarations and commands.

Blocks: A block consists of a series of function definitions followed by a series of declarations and commands. Function definitions are optional and can be omitted if not required.

Declarations: Variables can be declared with a specific data type (let, int, float, or string) and an optional initial value. Ternary expressions can also be used as initial values.

Commands: The language supports various commands, including:

Print command for outputting expressions.
Variable assignment using expressions or ternary expressions.
Conditional statements (if-else) with Boolean expressions.
While loops with a Boolean condition.
For loops with two forms: traditional counter-based loops and range-based loops.
Functions: Functions are defined with a name, input parameters, a block of declarations and commands, and a return statement using the "give" keyword.

Boolean Expressions: Booleans can be represented by true, false, equality comparisons between expressions, or negated Boolean expressions.

Expressions: Expressions can be formed using basic arithmetic operations (addition, subtraction, multiplication, division), parentheses for grouping, variables, or numbers.

Ternary Expressions: The language supports ternary expressions in the form "Identifier == Expression ? Expression : Expression".

Data Types: The language supports four data types: let (for type inference), int, float, and string.

Identifiers: Identifiers are used for naming variables and functions and must begin with a letter or underscore, followed by any combination of letters, digits, or underscores.

Numbers: Numbers are sequences of digits.

Comparisons: Comparison operators include less than, greater than, less than or equal to, greater than or equal to, and equality.

This grammar defines a versatile imperative programming language with support for modern programming constructs such as functions, loops, and conditional statements.

Grammar:

Program ::= Block.

Block ::= Function ; Block | Declaration ; Command.

Declaration ::= Datatype Identifier = Expression ; Declaration | Datatype Identifier ; Declaration | Datatype Identifier = Ternary ; Declaration | Φ

Command ::= print Expression ; Command |
    Identifier = Expression ; Command |
    Identifier = Ternary ; Command |
    if Boolean { Command } else { Command } ; Command |
    while Boolean { Command } ; Command |
    for (Identifier = Expression ; Identifier Comparison Expression ; Identifier Update) { Command } ; Command |
    for Identifier in range (Expression, Expression) { Command } ; Command | Φ

Function ::= fn Identifier : (Identifier, Identifier) { Declaration ; Command ; give Expression }

Boolean ::= true | false | Expression == Expression | not Boolean

Expression ::= Expression + Expression | Expression - Expression | Expression * Expression | Expression / Expression | (Expression) | Identifier | Number

Ternary ::= Identifier == Expression ? Expression : Expression

Datatype ::= let | int | float | string

Identifier ::= [a-z A-Z_][a-z A-Z 0-9_]*

Number ::= Digit Number | Digit

Digit ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

Comparison ::= < | > | <= | >= | ==

Components:

1. Lexical Analyzer

- The lexical analyzer is responsible for reading the input Kiwi program and converting it into a stream of tokens.
- We plan to implement the lexical analyzer using the ANTLR.
- The stream of tokens will be an input to the parser program again implemented using ANTLR.

2. Parser

- The parser of the Kiwi programming language will be implemented using ANTLR.
- It will be a Definite Clause Grammar(DCG) for the Backus Naur Form(BNF) Grammar written above.
- It is responsible for checking the syntax of the Kiwi program and generates the parse tree.
- The output to the parser is an abstract syntax tree which will be converted to an intermediate code which will be passed as input to the interpreter.

3. **Interpreter**

- The interpreter is responsible for executing the code that has been parsed and translated into an intermediate form, such as an abstract syntax tree.
- The interpreter will walk through the intermediate form and execute each node, producing a result.
- It will take in the parsed code, which has been translated into an abstract syntax tree, and walk through the tree, executing each node and producing a result.
- It will handle tasks like Variable management, Expression evaluation, Control flow, and Input/Output.
- The interpreter will be implemented in Python and will use the Python interpreter to execute the intermediate form of the code.