User Guide for Guy And Lior's Interpreter

Welcome to Guy And Lior's Interpreter! This guide will walk you through the setup, usage, and features of the interpreter. This interpreter allows you to define and execute lambda expressions, function definitions, and other expressions in a custom language.

Table Of Content-

Setup

Prerequisites

Ensure you have Python installed on your system. You can download it from [python.org](https://www.python.org/).

Installation

1. Clone or download the project repository to your local machine.

2. Navigate to the project directory.

Running the Interpreter

To run the interpreter, use the following command:

--python main.py <path_to_your_script>

Lexer

The lexer converts the input source code into tokens, which are the building blocks of the language syntax.

Token Types

- **INTEGER**: Numeric literals (e.g., `1`, `42`)
- **BOOLEAN**: Boolean literals (`True`, `False`)
- **PLUS**: Addition operator (`+`)
- **MINUS**: Subtraction operator (`-`)
- **MUL**: Multiplication operator (`*`)
- **DIV**: Division operator (`/`)
- **MOD**: Modulo operator (`%`)
- **AND**: Logical AND (`&&`)
- **OR**: Logical OR (`||`)
- **NOT**: Logical NOT (`!`)
- **EQ**: Equality operator (`==`)
- **NEQ**: Not equal operator (`!=`)
- **GT**: Greater than operator (`>`)
- **LT**: Less than operator (`<`)
- **GTE**: Greater than or equal operator (`>=`)
- **LTE**: Less than or equal operator (`<=`)
- **LPAREN**: Left parenthesis (`(`)
- **RPAREN**: Right parenthesis (`)`)
- **DEFUN**: Function definition keyword (`defun`)
- **LAMBDA**: Lambda expression keyword (`lambda`)
- **IDENTIFIER**: Identifiers (e.g., variable names)
- **EOF**: End of file

Example

Input: `(lambda x. (x + 1))(5)`

Output:

```

Token(LPAREN, '(')

Token(LAMBDA, 'lambda')

Token(IDENTIFIER, 'x')

Token(PERIOD, '.')

Token(LPAREN, '(')

Token(IDENTIFIER, 'x')

Token(PLUS, '+')

Token(INTEGER, 1)

Token(RPAREN, ')')

Token(RPAREN, ')')

Token(LPAREN, '(')

Token(INTEGER, 5)

Token(RPAREN, ')')
```

Parser

The parser takes tokens produced by the lexer and builds an Abstract Syntax Tree (AST) that represents the program structure.

Grammar

The language is defined by the following grammar:

```
<program> ::= <statement>*
<statement> ::= <function_definition>
        | <if_statement>
        | <expr>
        | <function_application>

<function_definition> ::= "defun" <identifier> "(" <parameters> ")" "{"
<statement>* "}"
<lambda_expression> ::= "lambda" <parameters> "." <expr>
<parameters> ::= <identifier> ("," <identifier>)*
<function_application> ::= (<identifier> | <lambda_expression>) "(" <arguments>
")"
<arguments> ::= <expr> ("," <expr>)*
<if_statement> ::= "if" "(" <expr> ")" "{" <statement>* "}" ("else" "{" <statement>*
"}")?
<expr> ::= <term> (("+" | "-" | "&&" | "||" | "==" | "!=" | ">" | "<" | ">=" | "<=") <term>)*
<term> ::= <factor> (("*" | "/" | "%") <factor>)*
<factor> ::= <integer>
        | <boolean>
        | "(" <expr> ")"
        | "!" <factor>
        | <function_application>
        | <identifier>
        | <lambda_expression>
        | <if_statement>
        | <function_definition>

<identifier> ::= [a-zA-Z_][a-zA-Z0-9_]*
<integer> ::= [0-9]+
<boolean> ::= "True" | "False"
```

Interpreter

Purpose

The interpreter executes the AST produced by the parser and returns the result of the program.

Supported Operations

- **Arithmetic operations**: `+`, `-`, `*`, `/`, `%`
- **Logical operations**: `&&`, `||`, `!`
- **Comparison operations**: `==`, `!=`, `>`, `<`, `>=`, `<=`
- **Function definitions**: `defun`
- **Lambda expressions**: `lambda`
- **Conditional statements**: `if`, `else`
- **Function applications**: Applying functions and lambda expressions

Writing and Running Programs

Programs can be written in a plain text file with the appropriate syntax. Save the file with a `.txt` or `.lambda` extension.

Running Programs

To run a program, use the following command:

--python main.py <path_to_your_script>

Supported Syntax

Function Definitions

Define a function using `defun` keyword:

```
defun factorial(n) {
  if (n == 0) {
    1
  } else {
    n * factorial(n - 1)
  }
}
factorial(5)
```

Lambda Expressions

Define a lambda expression using `lambda` keyword:

```
(lambda x. (x + 1))(5)
```

Conditional Statements

Use `if` and `else` for conditional logic:

```
if (n == 0) {
  1
} else {
  n * factorial(n - 1)
}
```

Function Applications

Apply functions or lambda expressions to arguments:

```
factorial(5)
(lambda x. (x + 1))(5)
```

Error Handling

Lexer Errors

Lexer errors occur when the lexer encounters an invalid character:

-Lexer error at position <pos>: Invalid character <char>

Parser Errors

Parser errors occur when the parser encounters unexpected tokens:

-Parser error: Expected token <expected>, got <actual> at token <token>

Interpreter Errors

Interpreter errors occur during the execution of the AST:

-Runtime error: <message>

Example Programs

```
# Factorial Function
defun factorial(n) {
   if (n == 0) {
      1
   } else {
      n * factorial(n - 1)
   }
}
factorial(5)
# Nested Lambda Expressions
(lambda x. (lambda y. (x + y)))(3)(4)
# Conditional Statement
if (3 > 2) {
   True
} else {
   False
}
# Arithmetic Operations
(5 + 3) * 2
# Logical Operations
True && False || True
```

By following this guide, you should be able to write and run programs using Guy And Lior's Interpreter. Happy coding!