[Github project link.](https://github.com/ItayChabra/finalProject/tree/master/final_project)

Itay Chabra  
208292458

Neta Ben Mordechai  
211451901

Programming Languages

**Documentation of the BNF**

**The BNF grammar:**

<program> ::= <statement> | <statement> <program>

<statement> ::= <function\_def> | <lambda\_expr> | <lambda\_call> | <function\_call> | <boolean\_expr> | <arithmetic\_expr> | <conditional\_expr>

<function\_def> ::= "Defun" "{" "name" ":" <identifier> "," "arguments" ":" <params> "," "}" <expression>

<lambda\_expr> ::= "Lambd" <params> "." <expression>

<lambda\_call> ::= <lambda\_expr> "(" <args> ")"

<function\_call> ::= <identifier> "(" <args>? ")"

<expression> ::= <arithmetic\_expr> | <boolean\_expr> | <function\_call> | <lambda\_expr> | <lambda\_call> | <identifier> | <literal> | <conditional\_expr> | <comparison\_expr>

<arithmetic\_expr> ::= <term> | <term> <arithmetic\_op> <arithmetic\_expr>

<boolean\_expr> ::= <term> | <term> <boolean\_op> <boolean\_expr> | "!" <term>

<comparison\_expr> ::= <term> | <term> <comparison\_op> <comparison\_expr>

<params> ::= <identifier> | <identifier> "," <params>

<args> ::= <expression> ("," <expression>)\*

<arithmetic\_op> ::= "+" | "-" | "\*" | "/" | "%"

<boolean\_op> ::= "&&" | "||"

<comparison\_op> ::= "==" | "!=" | ">" | "<" | ">=" | "<="

<conditional\_expr> ::= <boolean\_expr> "?" <expression> ":" <expression>

<literal> ::= <integer> | <boolean>

<identifier> ::= /[a-zA-Z\_][a-zA-Z\_0-9]\*/

<integer> ::= /-?\d+/

<boolean> ::= "TRUE" | "FALSE"

<term> ::= <identifier> | <literal> | "(" <expression> ")"

This document outlines the syntax and key features of the custom programming language. The language is designed with a focus on functional programming, supporting first-class functions, lambda expressions, arithmetic and boolean operations, conditional expressions, and more. Below, we detail the syntax rules using Backus-Naur Form (BNF) and describe the features supported by the language.  
The BNF grammar defines the structure of programs, while the described features and limitations give insight into the design choices and potential use cases for the language.

The language supports:

* **Functional Programming**: First-class functions, higher-order functions, and lambdas.
* **Lambda Expressions:** Define anonymous functions using Lambd, which can be passed as arguments or used in place of named functions.
* **Arithmetic Operations:** Basic arithmetic with precedence control, even without parentheses.
* **Boolean Logic**: Logical operations, comparison operators, and conditional expressions.
* **Custom Functions**: Define named functions using Defun with parameterized arguments.
* **Recursion**: Functions can call themselves or other functions, enabling recursive solutions.

**Language syntax and features**

**Program structure:**

A program consists of one or more statements. Each statement can be a function definition, lambda expression, function or lambda call, arithmetic or boolean expression, comparison, or a conditional expression.

**Statements:**

Statements are the fundamental building blocks of a program, representing a single unit of execution.

**Function definitions:**

Defines a named function using the Defun keyword, with a set of parameters and a body consisting of an expression.

**Expressions:**

Expressions include arithmetic and boolean operations, function and lambda invocations, variables, literals, and conditional or comparison operations.

**Function and Lambda Calls:**

Supports invoking a lambda or a named function with arguments. Lambda calls involve invoking an inline lambda expression directly.

**Parameters and Arguments:**

Parameters are used in function and lambda definitions to define placeholders for input values, while arguments are the actual values passed when calling functions or lambdas.

**Literals, Identifiers, and Terms:**

* Literals: Include integers and boolean values (TRUE, FALSE).
* Identifiers: Names for variables and functions.
* Terms: Basic elements of expressions, including identifiers, literals, or expressions wrapped in parentheses.

**Trade-offs and Limitations**

**Trade-offs:**

* **Simplicity vs. Expressiveness**: The language focuses on simplicity, providing essential features for functional programming while maintaining a minimal syntax.

**Limitations:**

* **No Native Loop Constructs**: Recursion is the primary mechanism for iteration, as traditional loops (for, while) are not supported.
* **Limited Data Types**: The language currently supports only integers and booleans as literals.
* **No Error Handling**: The language does not include error-handling mechanisms, so runtime errors must be managed externally or avoided through careful coding.

**User guide for running the interpreter**

**Interactive mode**

Interactive mode allows users to run commands line by line in a REPL (Read-Eval-Print Loop) environment. This mode is ideal for testing small code snippets, debugging, and experimenting with the language's features.

**How to Use Interactive Mode:**

* **Enter the terminal:** Navigate to the directory “final\_project” by entering the following command: “cd final\_project”.
* **Start the Interpreter**: Run the interpreter by executing the main.py file without specifying any additional files: “python main.py”.
* **Input Commands**: You can type expressions, function definitions, or any other valid statement directly into the command line.
* **Immediate Feedback**: After pressing Enter, the interpreter evaluates the input and immediately displays the result or output.

**File execution mode**

File execution mode is designed for running entire programs written in a .lambda file. This mode is useful for executing complete scripts and programs that have been pre-written and saved.

**How to Use File Execution Mode:**

* **Create a Script**: Write your program in a .lambda file using the language's syntax.
* **Run the Script**: Execute the interpreter by entering the file name as an argument. For example: “python main.py your\_program.lambda”.
* **Program Output**: The interpreter reads the file, evaluates all the statements, and outputs the results.

**Design decisions**

The language was designed with several core features in mind:

* **First-Class Functions:** Functions and lambda expressions are treated as first-class citizens, meaning they can be passed as arguments and returned from other functions.
* **Lambda Expressions:** The language supports anonymous functions through lambda expressions, which can be defined and called inline.
* **Arithmetic and Boolean Operations:** The language includes support for standard arithmetic operations (+, -, \*, /, %) and boolean operations (&&, ||, !).
* **Conditional Expressions:** The language supports conditional expressions using a ternary-like syntax (condition ? true\_expr : false\_expr).
* **Function and Lambda Calls:** The language allows for both named function calls and the immediate invocation of lambda expressions.

**Parser**

The parser is responsible for converting a sequence of tokens into an AST. The design of the parser focused on the following key areas:

* **Token Consumption and Error Handling:** The parser includes mechanisms for consuming tokens and handling unexpected tokens through the eat method, which advances the token stream and raises exceptions for syntax errors.
* **Recursive Descent Parsing:** The parser uses a recursive descent approach, where each non-terminal in the grammar has a corresponding method in the parser class.
* **Operator Precedence:** The parser ensures that arithmetic and boolean operations respect the correct precedence without needing explicit parentheses.
* **Lambda Expressions and Calls:** The parser handles lambda expressions and immediately following calls, allowing for concise function definitions and usage.

**Interpreter**

The interpreter executes the AST produced by the parser. The design of the interpreter includes:

* **Closure Representation:** Lambda expressions are executed within their defining environment using closures, ensuring proper scope management.
* **Global Scope Management:** The interpreter maintains a global scope to store function definitions.
* **Call Stack for Function Calls:** A call stack is implemented to manage function calls and recursion, preserving the state of variables during function execution.
* **Support for Higher-Order Functions:** The interpreter supports higher-order functions, allowing functions to be passed as arguments and returned from other functions, including the use of lambda expressions with functions.

**Challenges and solutions**

**Scope Management in Lambda Expressions**

**Challenge:**

* Variable Capture:  
  When a lambda expression is defined, it captures variables from its enclosing scope. These captured variables are stored in the closure. However, managing these captured variables and their lifetimes can be tricky.
* Nested Scopes:  
  Consider nested lambda expressions or nested function calls. Each level of nesting introduces a new scope. Tracking which variables are accessible at each level becomes complex.

**Solution:**

* Closures:  
  When encountering a lambda expression, your interpreter creates a closure that consists of:  
  The lambda expression itself.  
  A copy of the global scope (or the environment where the lambda was defined).  
  This closure ensures that the lambda can access the variables it captured even after the original scope has exited.
* Environment:  
  The environment (a dictionary) holds variable bindings.  
  When a function (including lambdas) is called, a new environment is created by copying the global scope (or the parent environment).  
  The environment is updated with the arguments passed to the function.  
  The call stack maintains a stack of environments, allowing nested scopes to be managed.

**Handling Function and Lambda Calls**

**Challenge:** Parsing and interpreting function and lambda calls, especially when lambda expressions are followed immediately by their arguments, required careful handling to differentiate between the definition and invocation.

**Solution:** The parser was designed to check for an open parenthesis immediately following a lambda expression to identify a call. The interpreter then evaluated the arguments before executing the lambda within the appropriate environment.

**Operator Precedence Without Parentheses**

**Challenge:** The language supports arithmetic operations with precedence control even without parentheses. Ensuring that operations like multiplication and division are prioritized over addition and subtraction without explicit grouping was complex.

**Solution:** The parser was designed with separate methods for parsing terms (parse\_term) and factors (parse\_factor), ensuring that multiplication and division are parsed before addition and subtraction, thereby maintaining the correct precedence.

**Error Handling**

**Challenge:** The interpreter needed to handle various runtime errors, such as division by zero, undefined variables, and incorrect argument counts.

**Solution:** The interpreter includes exception handling mechanisms that provide meaningful error messages when runtime errors occur, aiding in debugging and user feedback.