GEBZE TECHNICAL UNIVERSITY

COMPUTER ENGINEERING

AHMET YIGIT
200104004066

- Table of Contents

- Lisp Part

  - Information about Gpp_Lexer.lisp
  - Information about Gpp_Interpreter.lisp
  - Solve Approach
  - Function Definitions
    - Gppinterpreter
    - interpretLine
    - executeNormalStatement
    - decideParamethesisFunction
    - addValuef
    - minusValuef
    - multValuef
    - divValuef
    - getFloatList
    - remove-nth
    - add-nth
    - controlSyntax
    - controlSyntaxParamethesis
    - getLinesFromFile
  - Test Cases

- YACC Part

  - Information about gpp_interpreter.y
  - Information about gpp_interpreter.c/h
  - Solve Approach
  - Function Definitions
    - getFirstInt
    - getSecondInt
    - isFunctionParametherCountIsCorrect
    - doOperation
    - holdFunctionNoParameter
    - holdFunctionOneParameter
    - holdFunctionTwoParameter
    - printResult
    - addString
    - executeNoParametherFunction
    - executeOneParametherFunction
    - executeTwoParametherFunction
    - countParathesis
    - evaluateFunction
    - doOperationFunction
  - Test Cases

- **Lisp Part**
    - Information about Gpp_Lexer.lisp (it is first part of code command line defined)
        - **Purpose**
            - **Tokenization**:
                - It defines rules to identify different types of tokens in a programming language (keywords, operators, comments, identifiers, etc.).
            - **Parsing**:
                - It seems to perform a basic analysis of the input code, splitting it into meaningful units for further processing.

        - **Key Components**
            - **TokenList**
                - **KeywordList**: Contains keywords of the language.
                - **OperatorList**: Contains operators.
                - **DigitList**: Contains digits (for potential numbers).
                - **IgnoredList**: Contains characters to be ignored (whitespace, newline, etc.).
                - **Comment**: Defines the comment syntax.
            - **Tokenization and Processing**
                - **split**: Splits input string into words.
                - **decideWord**: Determines the type of token a word is (keyword, operator, identifier, etc.).
                - **getLine**: Reads user input and processes it word by word.

        - **Lexer Flow**
            - It reads input from the user line by line, then word by word.
            - It identifies tokens (keywords, operators, comments, identifiers, etc.) based on predefined rules and adds them to the token stack or other relevant stacks.
            - It handles comments and tracks exit conditions if there are syntax errors or 'exit' commands encountered.

    - **Information about Gpp_Interpreter.lisp**
        - **Main Functionalities**
            - **Token Management:**
                - Identifying tokens like identifiers, mathematical operators, function definitions (KW_DEF), open and close parentheses, and values (VALUEF).

        - **Function Definition and Calls:**
            - Distinguishing between function definitions and function calls, managing function parameters, and executing defined functions.

        - **Mathematical Operations:**
            - Handling addition, subtraction, multiplication, and division of fractional values.

        - **Syntax Checking:**
            - Verifying the correctness of parentheses and syntax in the input code.

        - **File Input Handling:**
            - Reading code from a file and interpreting its lines.

- **Solve Approach**
    - **Understanding Tokens**
        - Review the code that handles token recognition and storage.
        - Check how tokens are organized and utilized in the interpreter.

    - **Function Definitions and Calls**
        - Investigate how functions are defined and stored.
        - Trace the process of calling functions and handling their parameters.

    - **Mathematical Operations**
        - Examine the functions responsible for mathematical operations.
        - Validate the correctness of arithmetic calculations.

    - **Syntax Checking**
        - Identify how the code validates the syntax and structure of the input.
        - Ensure that errors and edge cases are handled appropriately.

    - **Testing**
        - Develop test cases to verify the functionality of different components.
        - Test arithmetic operations, function definitions, and various types of inputs.
        - Check for corner cases and potential sources of errors.

    - **Refactoring and Enhancements**
        - Refactor the code for readability and efficiency.

            - Add features or improve existing functionalities if necessary.
- Function Definitions
    - Gppinterpreter
        - Purpose:
            - The gppinterpreter function serves as the entry point to the Lisp interpreter.
            - It handles both file-based and interactive (user input) modes for interpreting Lisp-like code.
        - Parameters:
            - file-name (Optional):
        - Control Flow:
            - The function utilizes conditional statements (if) to determine the mode of operation (file-based or interactive) and appropriately processes the input.
        - Output:
            - In file-based mode, the interpreter processes the code from the file.
            - In interactive mode, the interpreter interacts with the user, interpreting input lines and providing output or error messages.
    - **interpretLine**
        - **Purpose**:
            - interpretLine is a function within the Lisp interpreter that processes and interprets a single line of Lisp-like code.
        - **Initialization:**
            - Initializes and resets several counters and flags (opCount, ccCount, identifierCount, index, isFunction, valuefCount, functionParametherCount, etc.) to manage the interpretation process.
        - **Token Processing:**
            - Looping through Tokens:
            - The function iterates through the tokens obtained from the tokenStack, processing each token in the line of code.
        - **Token Classification:**
            - Identifies different types of tokens (e.g., identifiers, keywords like KW_DEF, OP_OP, VALUEF, OP_CP, etc.) to execute specific actions based on the token type.

- **Function Identification:**
    - Recognizes function definitions (KW_DEF) and handles aspects related to function calls (IDENTIFIER) and their parameters.

- **Function Handling:**
    - Manages the process of defining functions, handling function calls, parameters, and execution by appropriately modifying various lists and counters (functionList, functionValueList, functionParametherNameList, etc.).

- **Execution and Token Removal:**
    - Executes actions based on the tokens encountered within the line of code.
    - Removes processed tokens from the tokenStack to manage the interpretation flow.

- **Syntax Control and Error Handling:**
    - Includes checks to ensure proper syntax, such as verifying parentheses balance, matching function calls, or reporting syntax errors.
    - Lacks detailed error messages or handling mechanisms for various encountered errors.

- **Function Termination and Output:**
    - End of Interpretation:
        - Terminates the interpretation process once it completes processing all tokens in the line.
    - Output/Result:
        - Prints the result of the interpretation or the success message of defining a function.

- **Data Management:**
    - Manipulates lists (functionNameList, functionCurrentValueList, executionFunction, parametherValues, etc.) to store, process, or remove function-related data during interpretation.

- **Recursive Calls:**
    - Calls other functions (decideParamethesisFunction, executeNormalStatement, etc.) to manage specific aspects of interpretation, often in a controlled recursive manner.

- **executeNormalStatement**
  - **Purpose**:
    - executeNormalStatement is responsible for executing operations present in the Lisp-like code's tokenized representation.
  - **Initialization**:
    - Sets the stage by resetting certain variables (isUsed, valuef, etc.) essential for the operation's execution.
  - **Operation Execution:**
    - **Operation Identification:**
      - Determines the specific arithmetic operation (e.g., addition, subtraction, multiplication, division) based on the tokens in the tokenStack.
    - **Valuef Computation:**
      - Carries out arithmetic operations on the valuefStack elements based on identified operations.
  - **Token Manipulation:**
    - **Updating Token Stack:**
      - Modifies the tokenStack to reflect the execution of operations and their results.
    - **Token and Stack Management:**
      - Removes processed tokens and updates the stack accordingly to maintain proper processing flow.
  - **Syntax Control and Error Handling:**
    - Syntax Validation:
      - Validates and ensures that the syntax of the expression being executed is correct.
      - Lacks detailed error messages for syntax-related issues.
  - **Result Handling:**
    - **Result Output:**
      - If operations are successfully executed, the result or intermediate computation outcome is reflected in the valuefStack.
  - **Loop Management:**
    - Utilizes looping structures to process operations iteratively and manage the token stack during execution.

    - **Data Management**:
        - Manages the valuefStack to store and manipulate values during arithmetic operations.

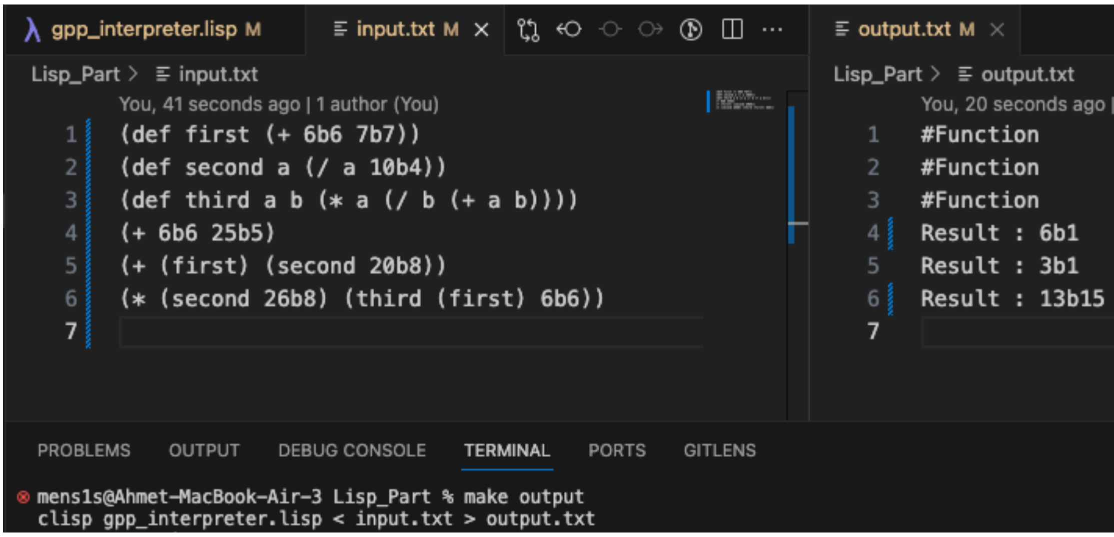
- **decideParamethesisFunction**
    - Purpose
        - Sets function ready to execute by executeNormalStatment.
        - It checks function name in list of function name list and if find it index it replace token with its information.
        - It checks paramethers and replace it to real values.
        - Call executeNormalStatement.

- **Basic Math Operation Functions**
    - addValuef
    - minusValuef
    - multValuef
    - divValuef

- **getFloatList**
    - It return fraction number as list [num1Den, num1In, num2Den, num2In)

- **remove-nth**
    - Removes nth element from list.
- **add-nth**
    - Adds element to nth index to list.
- **controlSyntax**
    - Checks syntax is correct or not.
- **controlSyntaxParamethesis**
    - Control of paramether count.
- **getLinesFromFile**
    - Read lines from file.

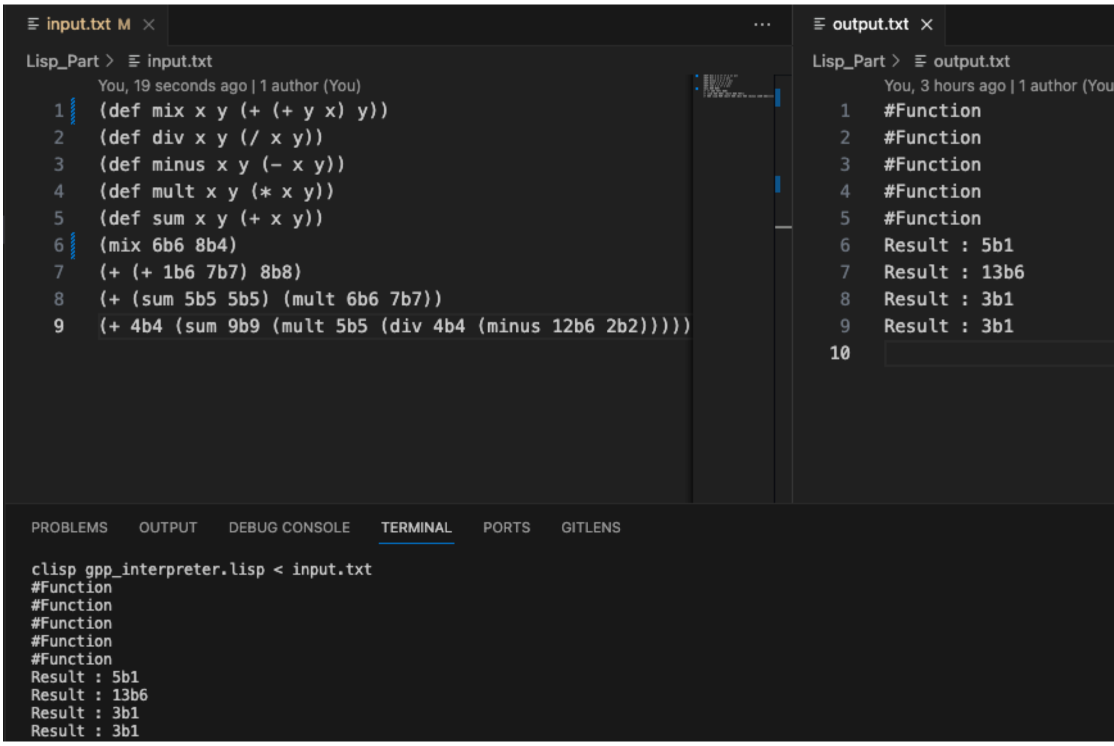
- **Test Cases**

  - 

```
(def first (+ 6b6 7b7))
(def second a (/ a 10b4))
(def third a b (* a (/ b (+ a b))))
(+ 6b6 25b5)
(+ (first) (second 20b8))
(* (second 26b8) (third (first) 6b6))
```

```
#Function
#Function
#Function
Result : 6b1
Result : 3b1
Result : 13b15
```

```
mens1s@Ahmet-MacBook-Air-3 Lisp_Part % make output
clisp gpp_interpreter.lisp < input.txt > output.txt
```

  - 

```
(def mix x y (+ (+ y x) y))
(def div x y (/ x y))
(def minus x y (- x y))
(def mult x y (* x y))
(def sum x y (+ x y))
(mix 6b6 8b4)
(+ (+ 1b6 7b7) 8b8)
(+ (sum 5b5 5b5) (mult 6b6 7b7))
(+ 4b4 (sum 9b9 (mult 5b5 (div 4b4 (minus 12b6 2b2)))))
```

```
#Function
#Function
#Function
#Function
#Function
Result : 5b1
Result : 13b6
Result : 3b1
Result : 3b1
```

```
clisp gpp_interpreter.lisp < input.txt
#Function
#Function
#Function
#Function
#Function
Result : 5b1
Result : 13b6
Result : 3b1
Result : 3b1
```

- ## YACC Part
  - Information about gpp_interpreter.c/h
    - It generates automatically by using interpreter.
  - **Information about gpp_interpreter.y**
    - **Includes and Definitions**
      - #include statements: Importing necessary C libraries like stdio.h, string.h, and stdlib.h.
      - extern FILE *yyin;: Declares a file pointer yyin that is likely used for input handling.
    - **Global Variables and Arrays**
      - functionArray: A two-dimensional array to store function information.
      - functionCount: An integer variable to keep track of the number of functions added to functionArray.
    - **Function Prototypes and Error Handling**
      - yylex() and yyerror(const char *error): Function prototypes; these are typically used in parser generators like Bison.
      - Error handling functions: yyerror() handles syntax errors and exits the program.
    - **Bison Declarations**
      - %union: Defines a union type to be used in the parser.
      - %token: Specifies token types for the lexer.
      - %start: Designates the starting rule of the grammar.
    - **Grammar Rules**
      - START, INPUT, EXP, FUNCTION, FUNCTION_INFO: Definitions of grammar rules for parsing the input language.
    - **main() Function**
      - main() function: Entry point of the program that initializes input handling and calls the parser generated by Bison (yyparse()).
  - Solve Approach
    - **Understanding the Grammar:**
      - Break down the grammar rules in the provided file (gpp_interpreter.y). Understand how the language constructs are defined (e.g., expressions, function definitions, operations).

    - **Reviewing the Parsing Logic:**
      - Analyze the Bison/Yacc parsing logic in the file.
      - Identify how tokens are recognized and parsed.
      - Understand the flow of the parsing process through rules and actions.

    - **Identify Key Functions:**
        - Examine the functions used within the grammar rules (doOperation, executeNoParametherFunction, etc.).
        - Understand their purposes and how they interact with the parsing logic.

    - **Testing and Debugging:**
        - Create sample inputs based on the defined grammar.
        - Run the program with these inputs to observe how the parser processes them.
        - Debug any issues encountered during parsing by checking the rules, actions, and function implementations.

    - **Expand Functionality (Optional):**
        - If required, enhance the grammar rules or add functionality to the interpreter (e.g., additional operations, control structures).

    - **Documentation and Refactoring:**
        - Document the grammar rules, functions, and any modifications made for future reference.
        - Refactor code if needed to improve readability, efficiency, or maintainability.

    - **Testing Comprehensive Inputs:**
        - Test the interpreter with a variety of inputs to ensure it handles various scenarios correctly (e.g., edge cases, complex expressions, function nesting).

    - **Error Handling and Robustness:**
        - Implement or improve error handling mechanisms to handle invalid inputs gracefully.
        - Ensure the interpreter doesn't crash on unexpected inputs but provides meaningful error messages.

- **Function Definitions**
    - **getFirstInt**
        - It returns first part of in 5b6 => 5

    - **getSecondInt**
        - It returns second part of in 5b6 => 6

    - **isFunctionParametherCountIsCorrect**
        - It checks paramether count and given paramether.

- **doOperation**
  - **Purpose**:
    - The doOperation function is designed to perform arithmetic operations on fractions and return the result as a string in the format "numerator/denominator".
  - **Parameters**:
    - a: String representing the first fraction in the format "a/b".
    - b: String representing the second fraction in the format "a/b".
    - op: Character representing the arithmetic operation (+, -, *, /).
  - **Operations Performed:**
    - The function performs arithmetic operations based on the provided operation character:
    - +: Addition of fractions a and b.
    - -: Subtraction of fraction b from fraction a.
    - *: Multiplication of fractions a and b.
    - /: Division of fraction a by fraction b.
- This functions stores function information in arrays by parameter count.
  - **holdFunctionNoParameter**
  - **holdFunctionOneParameter**
  - **holdFunctionTwoParameter**

- **printResult**
  - Print result to screen.

- **addString**
  - Add strings as executable format for doOperationFunction and executeXXParametherFunction.

- This functions gets ready to execute given expressions. Removes function name and replace function body replacing parameter names with real names.
  - **executeNoParametherFunction**
  - **executeOneParametherFunction**
  - **executeTwoParametherFunction**

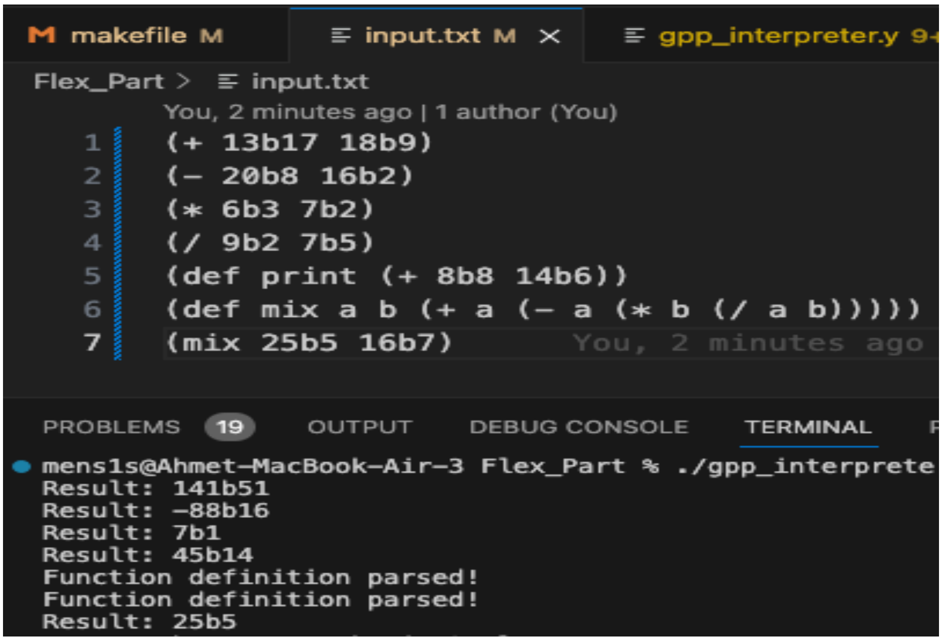
- **countParathesis**
  - Counts paranthesis.

- **evaluateFunction**
  - **Purpose:**
    - The evaluateFunction function processes arithmetic operations enclosed within parentheses and returns the result as a fraction in the format "numerator/denominator".
  - **Parameter**
    - expression: String representing an arithmetic operation enclosed within parentheses.
  - **Operation:**
    - The function primarily handles arithmetic operations within parentheses and computes the result.
  - **Implementation Overview:**
    - Parses the input expression to extract the operation within the parentheses.
    - Extracts the operator and two operands from the provided expression.
    - Calls the doOperation function to perform the arithmetic operation on the operands.
    - Returns the resulting fraction as a string in the format "numerator/denominator".
  - **Internal Process:**
    - Finds the operator (+, -, *, /) within the parentheses.
    - Extracts the two operands and performs the arithmetic operation using the doOperation function.
    - Returns the result of the operation as a simplified fraction.
- **doOperationFunction**
  - **Purpose:**
    - The doOperationFunction function is responsible for executing functions defined within the interpreter by substituting parameters with provided values and returning the resulting expression.
  - **Parameters:**
    - identifier: Identifier/name of the function to be executed.
    - parameterFirst: Value for the first parameter of the function (if applicable).
    - parameterSecond: Value for the second parameter of the function (if applicable).

- **Operation:**
    - The function aims to execute user-defined functions, potentially with zero, one, or two parameters, by substituting the parameters with provided values.

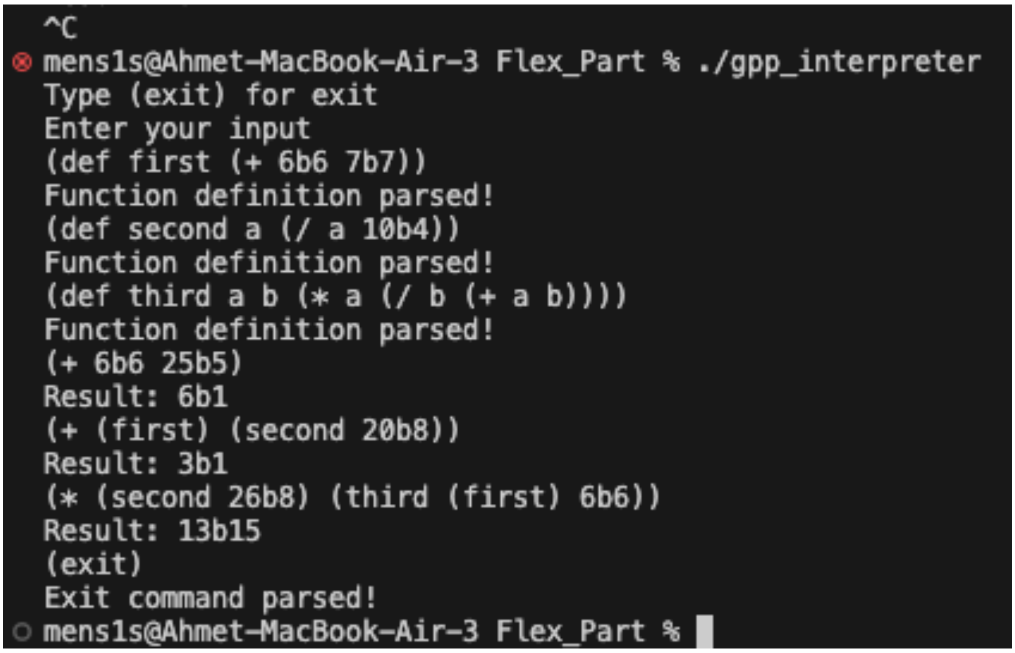
- **Implementation Overview:**
    - Searches for the specified function identifier within the interpreter's environment.
    - Replaces function parameters in the function's expression with the provided values.
    - Returns the resulting expression after parameter substitution.

- **Internal Process:**
    - Finds the function identified by identifier within the interpreter's environment.
    - Replaces occurrences of function parameters within the function's expression with provided parameter values.
    - Constructs and returns the resulting expression after parameter substitution.

- Test Cases
    - 

```
(+ 13b17 18b9)
(- 20b8 16b2)
(* 6b3 7b2)
(/ 9b2 7b5)
(def print (+ 8b8 14b6))
(def mix a b (+ a (- a (* b (/ a b)))))
(mix 25b5 16b7)          You, 2 minutes ago
```

```
mens1s@Ahmet-MacBook-Air-3 Flex_Part % ./gpp_interprete
Result: 141b51
Result: -88b16
Result: 7b1
Result: 45b14
Function definition parsed!
Function definition parsed!
Result: 25b5
```



```
^C
mens1s@Ahmet-MacBook-Air-3 Flex_Part % ./gpp_interpreter
Type (exit) for exit
Enter your input
(def first (+ 6b6 7b7))
Function definition parsed!
(def second a (/ a 10b4))
Function definition parsed!
(def third a b (* a (/ b (+ a b))))
Function definition parsed!
(+ 6b6 25b5)
Result: 6b1
(+ (first) (second 20b8))
Result: 3b1
(* (second 26b8) (third (first) 6b6))
Result: 13b15
(exit)
Exit command parsed!
mens1s@Ahmet-MacBook-Air-3 Flex_Part %
```