GEBZE TECHNICAL UNİVERSITY

Department of Computer Engineering

CSE-341 HW3

G++ Simple Language Interpreter Documentation

31.12.2023

Salih Karagöllü

210104004069

Features

The G++ interpreter supports:

- **Basic Arithmetic**: Including addition, subtraction, multiplication, and division.
- **Variables**: Dynamically typed variables for integers, floats, and lists.
- **Functions**: User-defined functions with parameters and return values.
- **Conditional Statements**: Including the **if** statement for conditional execution.
- **Boolean Logic**: Logical operators like **and**, **or**, and **not**.
- **Special Forms**: Special forms like **exit** for program control.

Internal Structure

- **Symbol Table**: Stores variable names and values for reference.
- **Function Definitions**: Maintains user-defined functions for execution.
- **Tokenization**: Converts input into tokens for parsing.
- **Parsing**: Creates abstract syntax trees (ASTs) for expressions and statements.
- **Evaluation**: Executes ASTs, resolving variables, performing operations, and evaluating functions.
- **Function Calls**: Locates function definitions and manages argument binding and execution.

Usage

You can use the interpreter by providing input as a string or by loading code from a file. It tokenizes, parses, and executes the code, displaying results on the console.

- Firstly, load the program using the sbcl:
  (load “gpp_interpreter.lisp”)
  *Note: There might be a bug that shows some warnings, if you encounter them, ignore it and run the command again, second time there will be no error, it is about sbcl itself.

- Now that the program is running, before whatever that you are trying to call, use this syntax: (gppinterpreter “<your command>”)

```
0[3] (gppinterpreter "(set x 42b4)")

"21b2"
```

- Every time you define a new variable or a function, the information is stored:

```
0[3] (gppinterpreter "(set x 42b4)")

"21b2"
0[3] (gppinterpreter "(set new_variable (+ 3b11 x))")

"237b22"
```

  Also, as you can see in the example, you can use complex expressions too.

- You can also define functions:

```
0[3] (gppinterpreter "(def sum x y (+ x y))")

"((\"sum\" (\"x\" \"y\") (BINARY-OP OP_PLUS (IDENTIFIER \"x\") (IDENTIFIER \"y\"))))"
```

  And use them later on with the variables you defined before:

```
0[4] (gppinterpreter "(sum new_variable x )")

"237b11"
```

As you can see, all required functionality is working perfectly.

Expression Evaluation Strategy

The expression evaluation strategy in the G++ Simple Language Interpreter follows a conventional approach, where expressions are evaluated using a combination of recursive parsing and function calls. Here's a brief overview:

1. **Tokenization**: The input code is first tokenized, splitting it into meaningful units such as keywords, operators, identifiers, and values (numbers).
2. **Parsing**: The tokenized code is parsed to create an Abstract Syntax Tree (AST) representation of the expressions and statements in the code. The parser handles expressions, function definitions, function calls, special forms, and more.
3. **Evaluation**: The interpreter evaluates the AST by traversing it recursively. During this process, it resolves variables, performs operations, evaluates functions, and handles special forms.
4. **Function Calls**: When a function is called, the interpreter locates the corresponding function definition, binds the arguments to parameter names, and executes the function body.
5. **Variable Scope**: Variables are stored in a symbol table, allowing them to have local scope within functions. Each function call creates a new scope, and variables defined within a function are only accessible within that scope.
6. **Error Handling**: The interpreter includes error handling mechanisms to detect syntax errors, division by zero, undefined variables, and other issues.

For example, here is one of the main functions:

```
(defun evaluate (exp)
  "Evaluates the parsed expression."
  (cond
    ;; Handling function definitions
    ((eq (first exp) 'FUNCTION)
     (store-function-definition exp))

    ;; Handling function application
    ((eq (first exp) 'FUNCTION-CALL)
     (evaluate-function-call exp))

    ;; Handling identifiers (variables)
    ((eq (first exp) 'IDENTIFIER)
     (get-variable (second exp)))

    ;; Handling set expressions
    ((eq (first exp) 'SET)
     (let ((var-name (second exp))
           (value (evaluate (third exp))))
       (add-variable var-name value)
       value))

    ;; Handling binary operations
    ((eq (first exp) 'BINARY-OP)
     (let ((operator (second exp))
           (operand1 (evaluate (third exp)))
           (operand2 (evaluate (fourth exp))))
       (case operator
         (OP_PLUS (+ operand1 operand2))
         (OP_MINUS (- operand1 operand2))
         (OP_MULT (* operand1 operand2))
         (OP_DIV (if (zerop operand2)
                     (error "Division by zero.")
                     (/ operand1 operand2)))
         (otherwise (error "Unknown operator: ~A" operator)))))
```

Evaluates and distributes to corresponding functions.

```
(defun parse-exp (tokens)
;; (print tokens)
  "Parses an expression from the tokens."
  (if (null tokens)
      (error "Unexpected end of input while parsing expression.")
      (let ((first-token (car (first tokens))) (second-token (car (second tokens))))
        (cond
          ;; Handling function definitions
          ((and (eq second-token 'KW_DEF) (eq first-token 'OP_OP))
           (parse-function tokens))

          ;; Handling function application
          ((and (eq first-token 'OP_OP) (eq (car (second tokens)) 'IDENTIFIER))
           (parse-function-application tokens))

          ;; Handling identifiers (which could be variables)
          ((eq first-token 'IDENTIFIER)
           (values (list 'IDENTIFIER (second (first tokens))) (rest tokens)))

          ;; Handling set expressions
          ((and (eq first-token 'OP_OP) (eq (car (second tokens)) 'KW_SET))
           (parse-set tokens))

          ;; Handling binary operations (+ exp exp), (- exp exp), (* exp exp), (/ exp exp)
          ((and (eq first-token 'OP_OP)
                (member (car (second tokens)) '(OP_PLUS OP_MINUS OP_MULT OP_DIV)))
           (parse-binary-op tokens))

          ;; Handling values
          ((eq first-token 'VALUEF)
           (values (list 'VALUEF (second (first tokens))) (rest tokens)))

          ;; Syntax Error for other cases
          (t (error "Syntax error in expression."))))))
```

Parses expressions with other helper functions.

```lisp
(defun tokenize (input &optional (tokens '()))
  ;; Tokenize input while skipping whitespace within lines
  (loop while (and input (not (equal input "")))
        do (setq input (skip-whitespace-and-tokenize-comments input))
        when (and input (not (equal input ""))) ;; Check here to ensure input is not empty
          do (cond
               ;; Number or Negative Number
               ((or (is-digit (char input 0)) (and (> (length input) 1) (char= (char input 0) #\b)))
                (multiple-value-bind (token-type value rest-input) (tokenize-number input)
                  (if (eq token-type 'SYNTAX_ERROR) ; Handle syntax errors directly
                      (return (reverse (cons (list token-type value) tokens)))
                      (progn
                        (setq input (skip-whitespace rest-input))
                        (push (list token-type value) tokens)  ; Wrap in list
                      )))
               )
               ;; Keyword or Identifier
               ((is-alphabet (char input 0))
                (multiple-value-bind (identifier rest-input) (tokenize-identifier input)
                  (setq input (skip-whitespace rest-input))
                  (let ((keyword-token (is-keyword identifier)))
                    (if keyword-token
                        (push (list (cdr keyword-token)) tokens)  ; Wrap keyword in list
                        (push (list 'IDENTIFIER identifier) tokens)  ; Wrap identifier in list
                    )))
               )
               ;; Operator
               ((is-operator (string (char input 0)))
                (let ((operator-token (is-operator (string (char input 0)))))
                  (push (list (cdr operator-token)) tokens)  ; Wrap operator in list
                  (setq input (skip-whitespace (subseq input 1)))
                )
               )
               ;; Syntax Error for any other character
               (t
                (format t "Syntax error ~a cannot be tokenized.~%" (string (char input 0)))
                (return (reverse (cons (list 'SYNTAX_ERROR (string (char input 0))) tokens)))
               )
          ))
  ;; Return the reversed tokens list
  (reverse tokens))
```

Main tokenizer for the program.

```lisp
(defvar *symbol-table* '())

(defvar *function-definitions* '())

(defun store-function-definition (func-def)
  "Stores a function definition."
  (let ((func-name (second func-def))
        (params (third func-def))
        (body (fourth func-def)))
    (let ((existing-def (assoc func-name *function-definitions* :test #'equal)))
      (if existing-def
          (setf (cdr existing-def) (list params body)) ; Update existing definition
          (push (list func-name params body) *function-definitions*))))) ; Store new definition

(defun add-variable (name value)
  "Adds or updates a variable in the symbol table."
  (let ((existing-entry (assoc name *symbol-table* :test #'equal)))
    (if existing-entry
        ;; Update the value of the existing entry
        (setf (cdr existing-entry) value)
        ;; If the entry doesn't exist, add a new one
        (push (cons name value) *symbol-table*))
        ;; (print *symbol-table*)
        ))

(defun get-variable (name)
  "Retrieves the value of a variable from the symbol table."
  (let ((entry (assoc name *symbol-table* :test #'equal)))
    (if entry
        (cdr entry)  ; Return the value of the variable
        (error "Variable ~A not found." name))))  ; Error if the variable is not found
```

Using symbol and function table with other helper functions to store the definitions of symbols and functions to use them later on.

Scope

The scope in the G++ Simple Language Interpreter is primarily determined by the way variables and functions are defined and accessed:

- **Local Scope**: Variables defined within a function have local scope and are only accessible within that function. This means that they are not visible outside the function, and different functions can have variables with the same names without conflicts.
- **Global Scope**: Variables defined outside of any function are considered global and can be accessed and modified from anywhere in the program. Global variables persist throughout the program's execution. These are the symbols that are defined with KW_SET.
- **Function Definitions**: Functions are defined globally and can be called from anywhere in the program. However, they have their own local scope for parameters and local variables defined within the function body.
- **Variable Resolution**: When a variable is referenced, the interpreter first checks the local scope (within the current function) and then the global scope (for global variables). This ensures that variables are resolved correctly based on their scope.
- **Scope Isolation**: Each function call creates a new isolated scope, allowing variables with the same names to exist in different function calls without interfering with each other.

```lisp
(defun apply-function (body param-names args)
  "Applies the function body with given arguments."
  (let ((*symbol-table* (copy-alist *symbol-table*))) ;; Backup the current symbol table
    ;; Map each parameter name to the corresponding argument
    (loop for param in param-names
          for arg in args
          do (add-variable param (evaluate arg)))
    ;; Evaluate the function body
    (let ((result (evaluate body)))
      (setq *symbol-table* (copy-alist *symbol-table*)) ;; Restore the original symbol table
      result))) ;; Return the result of the function call
```

By using this function, when a function is called, arguments of that function are added to the symbol table temporarily. After the evaluation is complete, the variables are cleared.