CSE341 FALL 2024 – Programming Languages Assignment 1 Report

-Cemal BOLAT-

- **Production Rules that I followed on my assignment.**

```
;; Production Rules:
<program> ::= <line>*

<line> ::= <if-statement>
         | <for-loop>
         | <while-loop>
         | <variable-assignment>
         | <function-definition>
         | <function-declaration>
         | <return-statement>
         | <variable-re-assignment>
         | <end-block>
         | <empty-line>
         | <logical-expression>
         | <arithmetic-expr>
         | <function-call>

<if-statement> ::= "if" <space> "(" <logical-expression> ")" <space> "{" <line>* "}"

<for-loop> ::= "for" <space> "(" <variable-assignment> <space> ";" <logical-expression> <space> ";" <arithmetic-expr> <space> ")" <space> "{" <line>* "}"

<while-loop> ::= "while" <space> "(" <logical-expression> ")" <space> "{" <line>* "}"

<variable-assignment> ::= <data-type> <space> <param-name> <space> "=" <space> <arithmetic-expr> <space> ";"
                        | <data-type> <space> <param-name> <space> "=" <space> <func-name> <space> "(" <function-call-params> ")" <space> ";"

<function-definition> ::= <data-type> <space> <func-name> <space> "(" <function-declaration-params> ")" <space> "{" <line>* "}"

<function-declaration> ::= <data-type> <space> <func-name> <space> "(" <function-declaration-params> ")" <space> ";"

<return-statement> ::= "return" <space> <arithmetic-expr> <space> ";"

<variable-re-assignment> ::= <param-name> <space> "=" <space> <arithmetic-expr> <space> ";"

<end-block> ::= "}"

<empty-line> ::= <space> | <new-line> | <tab>

<logical-expression> ::= <literal> <space> <logical-operator> <space> <literal>
                       | <literal> <space> <logical-operator> <space> <logical-expression>

<logical-operator> ::= "&&" | "||" | "!"

<literal> ::= <param-name> | <number>

<arithmetic-expr> ::= <term> <space> "+" <space> <term>
                    | <term> <space> "-" <space> <term>
                    | <term>

<term> ::= <factor> <space> "*" <space> <factor>
         | <factor> <space> "/" <space> <factor>
         | <factor>

<factor> ::= <number>
           | <param-name>
           | <space> "(" <space> <arithmetic-expr> <space> ")"

<function-declaration-params> ::= <data-type> <space> <param-name>
                                | <data-type> <space> <param-name> "," <space> <function-declaration-params>

<function-call-params> ::= <param-name>
                         | <param-name> "," <space> <function-call-params>

<data-type> ::= "int" | "float" | "double" | "char" | "void"

<param-name> ::= <letter>
               | <letter> <param-name>
               | <letter> <number>
               | <letter> <number> <param-name>

<func-name> ::= <letter>
              | <letter> <func-name>
              | <letter> <number>
              | <letter> <number> <func-name>

<number> ::= <digit>
           | <digit> <number>

<digit> ::= "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9"
<letter> ::= "a" | "b" | "c" | ... | "z" | "A" | "B" | ... | "Z"
<space> ::= " " | "\t" | "\n"
```

- **Main Function**

```lisp
(defun main (converted-lines index)
    ;; main function that reads the file and converts the lines
    ;; if the line is null then writes the converted lines to the file
    ;; else converts the line and adds it to the converted lines
    (let* ((next-line (read-file "input.c" index))
           (line-before (read-file "input.c" (- index 1))))
        (cond
            ((null line-before) (write-file (reverse converted-lines)))
            (t
                (let* ((type-of-line (line-type line-before))
                       (conversion-func (conversion-foo type-of-line))
                       (converted-line (add-till-space (funcall conversion-func line-before next-line) line-before 0)))
                    (main (cons converted-line converted-lines) (+ index 1)))))))

(main '() 1)
```

The main function recursively processes and converts lines from a file, storing each converted line in a list until all lines are processed, then writes the reversed list to an output file.

- **Read-file Function**

```lisp
(defun read-file-helper (stream index line-index)
    ;; reads the given file and returns the lines as a string
    (let ((line (read-line stream nil)))
        (if line
            (if (= index line-index)
                line
                (read-file-helper stream (+ index 1) line-index)
            )
            nil )))

(defun read-file (fileName index)
    ;; reads the given file and returns the lines as a list
    ;; :direction the type of the stream, :input for reading, :output for writing
    (with-open-file (stream fileName :direction :input)
        (read-file-helper stream 0 index)))
```

The read-file function opens a file and, using read-file-helper, recursively reads lines until it retrieves the specified line by index.

- **Write-file Function**

```lisp
(defun write-file-helper (stream lines)
    ;; writes the given lines to a file
    (if lines
        (progn
            (format stream "~a~%" (car lines))
            (write-file-helper stream (cdr lines))
        )
        nil ))

(defun write-file (lines)
    ;; writes the given lines to a file
    ;; :direction the type of the stream, :input for reading, :output for writing
    ;; if-exists :supersede to overwrite, :append to append
    (with-open-file (stream "output.lisp" :direction :output :if-exists :supersede)
        (write-file-helper stream lines)))
```

The write-file function opens "output.lisp" in write mode and, using write-file-helper, recursively writes each line from the provided list to the file.

- **Line-type Function**

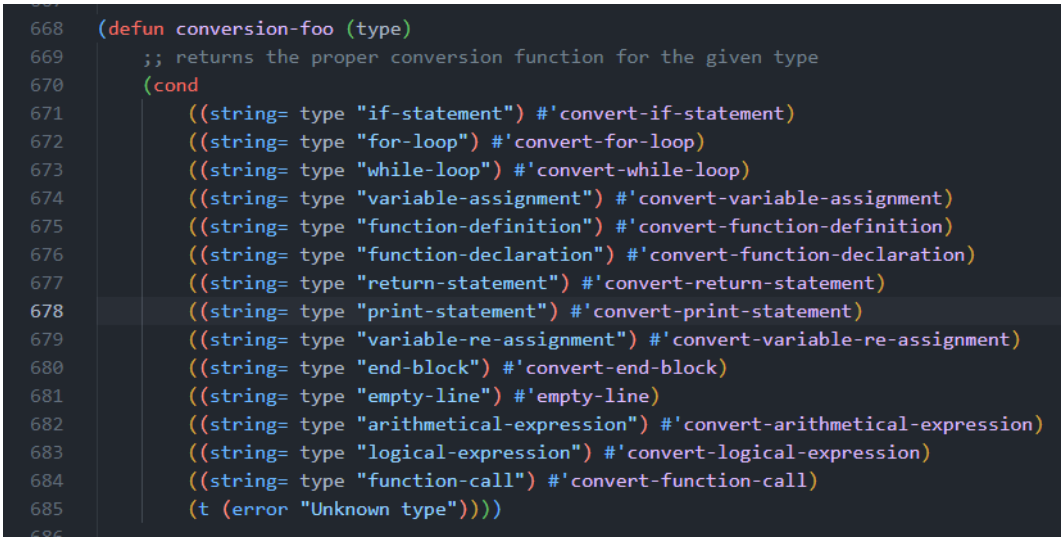
```
(defun line-type (line)
    ;; returns the type of the given line according to the production rules
    (let* ((line-without-space (remove-whitespace line)) (line-len (length line-without-space)))
        (cond
            ((string= line-without-space "") "empty-line")                  ;; empty line
            ((and (< 4 line-len)
                  (string= (subseq line-without-space 0 2) "if")            ;; if the line starts with if and ends with {
                  (string= (subseq line-without-space (- line-len 1) line-len) "{")) "if-statement")
            ((and (< 5 line-len)
                  (string= (subseq line-without-space 0 3) "for")           ;; if the line starts with for and ends with {
                  (string= (subseq line-without-space (- line-len 1) line-len) "{")) "for-loop")
            ((and (< 6 line-len)
                  (string= (subseq line-without-space 0 5) "while")         ;; if the line starts with while and ends with {
                  (string= (subseq line-without-space (- line-len 1) line-len) "{")) "while-loop")
            ((and (= 1 line-len)                                            ;; if the line is }
                  (string= line-without-space "}") "end-block"))
            ((and (< 6 line-len)                                            ;; if the line starts with return
                (string= (subseq line-without-space 0 6) "return") "return-statement"))
            ((and (< 6 line-len)                                            ;; if the line starts with printf
                (string= (subseq line-without-space 0 6) "printf") "print-statement"))
            ((and (is-start-with-data-type line-without-space)              ;; if the line starts with a c-type data type and has a '=' sign
                (search "=" line-without-space) "variable-assignment"))
            ((and (is-start-with-data-type line-without-space)              ;; if the line starts with a c-type data type and ends with '{'
                (string= (subseq line-without-space (- line-len 1) line-len) "{") "function-definition"))
            ((and (is-start-with-data-type line-without-space)              ;; if the line starts with a c-type data type and ends with ';'
                  (string= (subseq line-without-space (- line-len 1) line-len) ";") "function-declaration"))
            ((and (search "=" line-without-space) "variable-re-assignment")) ;; if the line has a '=' sign

            ((or (search "+" line-without-space) (search "-" line-without-space)  ;; if the line has an arithmetical operator
                (search "*" line-without-space) (search "/" line-without-space)
                (search "%" line-without-space)) "arithmetical-expression")
            ((or (search "&&" line-without-space) (search "||" line-without-space)  ;; if the line has a logical operator
                (search "==" line-without-space) (search "!=" line-without-space)
                (search "<" line-without-space) (search ">" line-without-space)) "logical-expression")
            ((or (search "()" line-without-space) (search "," line-without-space)) "function-call") ;; if the line has a function call

        )))
```

The line-type function determines the type of a given line based on production rules by removing whitespace and checking patterns for structures like if, for, while loops, return statements, variable assignments, and function calls, returning a corresponding type string for each match.

- **Conversion-Foo Function**



```
(defun conversion-foo (type)
    ;; returns the proper conversion function for the given type
    (cond
        ((string= type "if-statement") #'convert-if-statement)
        ((string= type "for-loop") #'convert-for-loop)
        ((string= type "while-loop") #'convert-while-loop)
        ((string= type "variable-assignment") #'convert-variable-assignment)
        ((string= type "function-definition") #'convert-function-definition)
        ((string= type "function-declaration") #'convert-function-declaration)
        ((string= type "return-statement") #'convert-return-statement)
        ((string= type "print-statement") #'convert-print-statement)
        ((string= type "variable-re-assignment") #'convert-variable-re-assignment)
        ((string= type "end-block") #'convert-end-block)
        ((string= type "empty-line") #'empty-line)
        ((string= type "arithmetical-expression") #'convert-arithmetical-expression)
        ((string= type "logical-expression") #'convert-logical-expression)
        ((string= type "function-call") #'convert-function-call)
        (t (error "Unknown type"))))
```

The Conversion-foo function returns the appropriate conversion function based on the given type, mapping each type (like if-statement, for-loop, variable-assignment, etc.) to a specific conversion function; if type is unrecognized, it raises an error.

- **Convert-if-statement Function**

```lisp
(defun convert-if-statement (line next-line)
    ;; line example: "if (a < 10) {"
    ;; first remove unnecessary whitespaces and if + paranthesis
    ;; then add space before and after the logical delimiters
    ;; then split the line by ' ' to get the parts
    ;; then evaluate the infix logical expression
    (let* ((trimmed-line (list-to-string (split-string " " line 0 '() "" nil) 0 " ")) ;; '(#\Newline) "(progn"
           (logical-expression (subseq trimmed-line 3 (- (length trimmed-line) 1)))
           (splitted-logical-expression (split-string " " (add-space-before-after-logical-delimiters logical-expression 0) 0 '() "" nil)))
      (concatenate 'string "(if " (evaluate-infix-expression (reverse splitted-logical-expression) '() '() 0) '(#\Newline) (add-till-space "(progn" line 0))
    )
)
```

The convert-if-statement function trims unnecessary whitespace from an if statement line, adds spacing around logical delimiters, splits it into parts, evaluates the logical expression in infix notation, and formats it for output.

- **Convert-while-loop Function**

```lisp
(defun convert-while-loop (line next-line)
    ;; line example: "while (a < 10) {"
    ;; first remove unnecessary whitespaces and while + paranthesis
    ;; then add space before and after the logical delimiters
    ;; then split the line by ' ' to get the parts
    ;; then evaluate the infix logical expression
    (let* ((trimmed-line (list-to-string (split-string " " line 0 '() "" nil) 0 " "))
           (logical-expression (subseq trimmed-line 6 (- (length trimmed-line) 1)))
           (splitted-logical-expression (split-string " " (add-space-before-after-logical-delimiters logical-expression 0) 0 '() "" nil)))
      (concatenate 'string "(loop while " (evaluate-infix-expression (reverse splitted-logical-expression) '() '() 0) " do" '(#\Newline) (add-till-space "(progn" line 0))
    )
)
```

The convert-while-loop function trims unnecessary whitespace from an while loop line, adds spacing around logical delimiters, splits it into parts, evaluates the logical expression in infix notation, and formats it for output.

- **Convert-for-loop Function**

```lisp
(defun convert-for-loop (line next-line)
    ;; line example: "for (int a = 0; a < 10; a++) {"
    ;; first remove unnecessary whitespaces and for + paranthesis
    ;; then split the line by ';' to get the parts
    ;; then extract the variable name, start value, end value, relational operator and increment value
    ;; then return the lisp equivalent of the for loop
    (let* ((temp-line (remove-whitespace line))
           (trimmed-line (subseq temp-line 3 (- (length temp-line) 1)))
           (parantesis-deleted-line (subseq trimmed-line 1 (- (length trimmed-line) 1)))
           (splitted-line (split-string ";" parantesis-deleted-line 0 '() "" nil))
           (first-expr (subseq (nth 0 splitted-line) 0 (position #\= (nth 0 splitted-line))))
           (variable-name (extract-var-name first-expr))
           (start-value (subseq (nth 0 splitted-line) (+ 1 (position #\= (nth 0 splitted-line)))))
           (relational-operator (extract-relational-operator (nth 1 splitted-line)))
           (end-value (extract-end-value (nth 1 splitted-line) variable-name relational-operator))
           (increment-value (extract-increment-value (nth 2 splitted-line) variable-name))
        )
        (cond
            ((string= relational-operator "<") (concatenate 'string "(loop for " variable-name " from " start-value
                " below " end-value " by " increment-value " do" '(#\Newline) (add-till-space "(progn" line 0)))
            ((string= relational-operator ">") (concatenate 'string "(loop for " variable-name " from " start-value
                " downto " "( + " end-value " 1)" " by " increment-value " do" '(#\Newline) (add-till-space "(progn" line 0)))
            ((string= relational-operator ">=") (concatenate 'string "(loop for " variable-name " from " start-value
                " downto " end-value " by " increment-value " do" '(#\Newline) (add-till-space "(progn" line 0)))
            ((string= relational-operator "<=") (concatenate 'string "(loop for " variable-name " from " start-value
                " to " end-value " by " increment-value " do" '(#\Newline) (add-till-space "(progn" line 0)))
        )
    )
)
```

The convert-for-loop function removes unnecessary whitespace from a C-style for loop line, splits it by ‘;’ to extract components like the variable name, start value, end value, relational operator, and increment value, and then constructs an equivalent Lisp loop expression based on the relational operator.

- **Convert-variable-assignment Function**

```lisp
(defun convert-variable-assignment (line next-line)
    ;; line example: "int a = 5;"
    ;; first remove unnecessary whitespaces
    ;; then data type is the first word
    ;;      variable name is the second word till the first '='
    ;;      value is the words from the first '=' to the end except the ';'
    ;;      if the value is a function call then split the parameters by ',' and convert them to lisp data types
    ;;      if the value is a function call without parameters then just return the function call
    ;;      if the value is an arithmetic expression then split the expression by ' ' and evaluate the expression
    (let* ((trimmed-line (list-to-string (split-string " " line 0 '() "" nil) 0 " "))
           (data-type (subseq trimmed-line 0 (position #\space trimmed-line)))
           (line-without-data-type (subseq trimmed-line (+ 1 (position #\space trimmed-line))))
           (param-name (subseq line-without-data-type 0 (position #\= line-without-data-type)))
           (arithmetic-expr (remove-whitespace (subseq line-without-data-type (+ 1 (position #\= line-without-data-type)) (- (length line-without-data-type) 1)))))
           (if (search "()" arithmetic-expr)
               (let ((func-name (subseq arithmetic-expr 0 (- (length arithmetic-expr) 2))))
                   (if (string= (line-type next-line) "variable-assignment")
                       (concatenate 'string "(" (string-trim " " param-name) " (" (string-trim " " func-name) "))")
                       (concatenate 'string "(" (string-trim " " param-name) " (" (string-trim " " func-name) ")))")
                   )
               )
               (if (string= (line-type arithmetic-expr) "function-call")
                   (let* ((func-name (subseq arithmetic-expr 0 (position (code-char 40) arithmetic-expr)))
                          (func-params (subseq arithmetic-expr (+ 1 (position (code-char 40) arithmetic-expr)) (- (length arithmetic-expr) 1))))
                          (if (string= (line-type next-line) "variable-assignment")
                              (concatenate 'string "(" (string-trim " " param-name) " (" (string-trim " " func-name) " "
                                           (list-to-string (split-string "," func-params 0 '() "" nil) 0 " "))")
                              (concatenate 'string "(" (string-trim " " param-name) " (" (string-trim " " func-name) " "
                                           (list-to-string (split-string "," func-params 0 '() "" nil) 0 " ")))")
                          )
                   )
                   (if (string= (line-type next-line) "variable-assignment")
                       (concatenate 'string "(" (string-trim " " param-name) " " (evaluate-infix-expression
                           (reverse (split-string " " (add-space-before-after-delimiters arithmetic-expr 0) 0 '() "" nil)) '() '() 0) ")")
                       (concatenate 'string "(" (string-trim " " param-name) " " (evaluate-infix-expression
                           (reverse (split-string " " (add-space-before-after-delimiters arithmetic-expr 0) 0 '() "" nil)) '() '() 0) "))")
                   )
               ))))
```

The convert-variable-assignment function processes a variable assignment line by extracting the data type, variable name, and assigned value, then generates a corresponding Lisp expression based on the type of value (function call with or without parameters, or arithmetic expression), handling each case to construct the correct format for setq or nested assignments.

- **Convert-variable-re-assignment Function**

```lisp
(defun convert-variable-re-assignment (line next-line)
    ;; line example: "a = 5;"
    ;; first remove unnecessary whitespaces
    ;; then variable name is the first word
    ;;      value is the words from the first '=' to the end except the ';'
    ;;      if the value is a function call then split the parameters by ',' and convert them to lisp data types
    ;;      if the value is a function call without parameters then just return the function call
    ;;      if the value is an arithmetic expression then split the expression by ' ' and evaluate the expression
    (let* ((trimmed-line (list-to-string (split-string " " line 0 '() "" nil) 0 " "))
               (param-name (subseq trimmed-line 0 (position #\= trimmed-line)))
               (arithmetic-expr (remove-whitespace (subseq trimmed-line (+ 1 (position #\= trimmed-line)) (- (length trimmed-line) 1)))))
           (if (search "()" arithmetic-expr)
               (let ((func-name (subseq arithmetic-expr 0 (- (length arithmetic-expr) 2))))
                   (concatenate 'string "(setq " (string-trim " " param-name) " (" (string-trim " " func-name) "))")
               )
               (if (string= (line-type arithmetic-expr) "function-call")
                   (let* ((func-name (subseq arithmetic-expr 0 (position (code-char 40) arithmetic-expr)))
                          (func-params (subseq arithmetic-expr (+ 1 (position (code-char 40) arithmetic-expr)) (- (length arithmetic-expr) 1))))
                          (concatenate 'string "(setq " (string-trim " " param-name) " (" (string-trim " " func-name) " "
                                       (list-to-string (split-string "," func-params 0 '() "" nil) 0 " ") "))")
                   )
                   (concatenate 'string "(setq " (string-trim " " param-name) " " (evaluate-infix-expression
                          (reverse (split-string " " (add-space-before-after-delimiters arithmetic-expr 0) 0 '() "" nil)) '() '() 0) ")")

               )
           )
    )
)
```

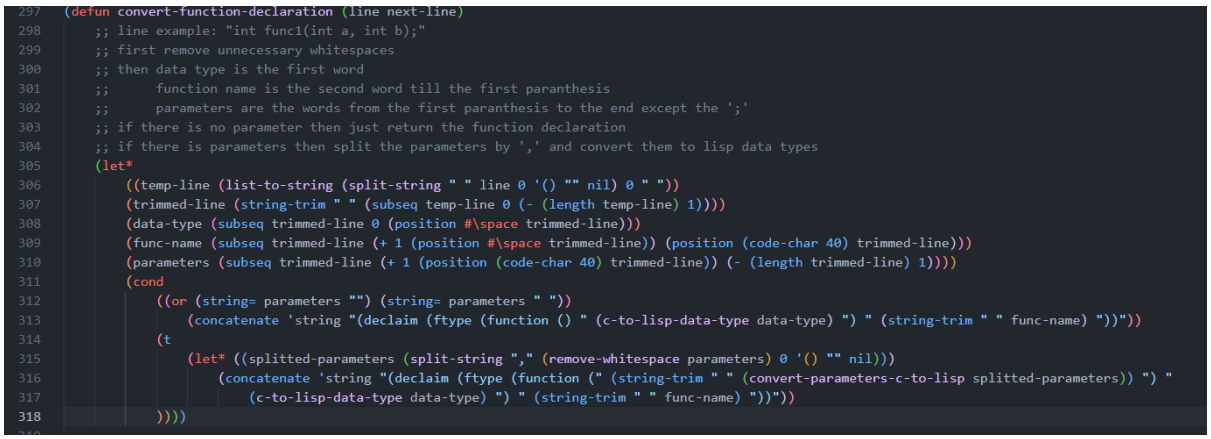
The convert-variable-re-assignment function processes a variable reassignment line by removing whitespace, extracting the variable name and value, and then generating a Lisp setq expression based on the value type: it handles function calls with or without parameters, and evaluates arithmetic expressions.

- **Convert-function-definition Function**

```lisp
(defun convert-function-definition (line next-line)
    ;; line example: "int func1(int a, int b) {"
    ;; first remove unnecessary whitespaces
    ;; then data type is the first word
    ;;      function name is the second word till the first paranthesis
    ;;      parameters are the words from the first paranthesis to the end except the '{'
    ;; if there is no parameter then just return the function definition
    ;; if there is parameters then split the parameters by ',' and convert them to lisp data types
    (let* ((temp-line (list-to-string (split-string " " line 0 '() "" nil) 0 " "))
           (trimmed-line (string-trim " " (subseq temp-line 0 (- (length temp-line) 1))))
           (func-return-type (subseq trimmed-line 0 (position #\space trimmed-line)))
           (func-name (subseq trimmed-line (+ 1 (position #\space trimmed-line)) (position (code-char 40) trimmed-line)))
           (parameters (subseq trimmed-line (+ 1 (position (code-char 40) trimmed-line)) (- (length trimmed-line) 1)))
      )
        (cond
            ((or (string= parameters "") (string= parameters " "))
                (if (string= (line-type next-line) "variable-assignment")
                    (concatenate 'string "(defun " (string-trim " " func-name) " () " '(#\Newline) (add-till-space "(let* (" line 0))
                    (concatenate 'string "(defun " (string-trim " " func-name) " () " '(#\Newline) (add-till-space "(let* ()" line 0))
                )
            )
            (t
                (let* ((splitted-parameters (split-string "," (remove-whitespace parameters) 0 '() "" nil)))
                    (if (string= (line-type next-line) "variable-assignment")
                        (concatenate 'string "(defun " (string-trim " " func-name) " (" (string-trim " " (definition-func-parameters splitted-parameters)) ") "
                            '(#\Newline) (add-till-space "(let* (" line 0))
                        (concatenate 'string "(defun " (string-trim " " func-name) " (" (string-trim " " (definition-func-parameters splitted-parameters)) ") "
                            '(#\Newline) (add-till-space "(let* ()" line 0))
                    )
                )
                ))))
```

The convert-function-definition function processes a function definition line by removing extra whitespace, extracting the return type, function name, and parameters, then generating the Lisp equivalent defun expression. If there are no parameters, it creates a function with an empty parameter list; if there are parameters, it converts them to a Lisp-compatible format. It also checks if the next line contains a variable assignment; if so, it adds a let* expression in the function body to accommodate potential variable declarations within the function.

- **Convert-function-declaration Function**



```lisp
(defun convert-function-declaration (line next-line)
    ;; line example: "int func1(int a, int b);"
    ;; first remove unnecessary whitespaces
    ;; then data type is the first word
    ;;      function name is the second word till the first paranthesis
    ;;      parameters are the words from the first paranthesis to the end except the ';'
    ;; if there is no parameter then just return the function declaration
    ;; if there is parameters then split the parameters by ',' and convert them to lisp data types
    (let*
        ((temp-line (list-to-string (split-string " " line 0 '() "" nil) 0 " "))
        (trimmed-line (string-trim " " (subseq temp-line 0 (- (length temp-line) 1))))
        (data-type (subseq trimmed-line 0 (position #\space trimmed-line)))
        (func-name (subseq trimmed-line (+ 1 (position #\space trimmed-line)) (position (code-char 40) trimmed-line)))
        (parameters (subseq trimmed-line (+ 1 (position (code-char 40) trimmed-line)) (- (length trimmed-line) 1))))
        (cond
            ((or (string= parameters "") (string= parameters " "))
                (concatenate 'string "(declaim (ftype (function () " (c-to-lisp-data-type data-type) ") " (string-trim " " func-name) "))"))
            (t
                (let* ((splitted-parameters (split-string "," (remove-whitespace parameters) 0 '() "" nil)))
                    (concatenate 'string "(declaim (ftype (function (" (string-trim " " (convert-parameters-c-to-lisp splitted-parameters)) ") "
                        (c-to-lisp-data-type data-type) ") " (string-trim " " func-name) "))"))
            ))))
```

The convert-function-declaration function processes a C-style function declaration line by removing extra whitespace, extracting the return type, function name, and parameters, and then generating a Lisp declaim expression for the function type. If there are no parameters, it creates a declaration with an empty parameter list; if parameters are present, it converts them to a Lisp-compatible format, ensuring the data types are properly translated.

- **Convert-return-statement Function**

```
(defun convert-return-statement (line next-line)
    ;; first remove the return keyword and the semicolon
    ;; then add space before and after the delimiters then split the string to get tokens
    ;; then evaluate the infix arithmetic expression
    (let* ((trimmed-line (list-to-string (split-string " " line 0 '() "" nil) 0 " "))
           (return-value (subseq trimmed-line 6 (- (length trimmed-line) 1)))
           (splitted-line (split-string " " (add-space-before-after-delimiters return-value 0) 0 '() "" nil)))
       (if splitted-line
           (concatenate 'string (evaluate-infix-expression (reverse splitted-line) '() '() 0))
           "nil"
       )
    )
)
```

The convert-return-statement function processes a return statement by removing the return keyword and semicolon, adding spaces around delimiters, splitting the line into tokens, and then evaluating any infix arithmetic expression, returning the Lisp-compatible result. If there's no expression, it returns "nil"

- **Convert-print-statement Function**

```
(defun convert-print-statement (line next-line)
    ;; example input : "printf("Hello World %d\n", x);"
    ;; example output : "(format t "Hello World ~a~%" x)"
    ;; Between the double quotes, the string is written as it is.
    ;; The variables in printf is refered by %and the type of the variable.
    ;; first remove the printf keyword and the semicolon
    ;; than get the string between the double quotes
    (let* ((trimmed-line (list-to-string (split-string " " line 0 '() "" nil) 0 " "))
           (print-value (subseq trimmed-line 7 (- (length trimmed-line) 2)))
           (str-between-quotes (get-string-between-quotes print-value))
           (new-line-converted (extract-formatters str-between-quotes 0))
           (parameters (split-string "," (subseq print-value (+ (length str-between-quotes) 2)) 0 '() "" nil)))
       (if (null parameters)
           (concatenate 'string "(format t \"" new-line-converted "\")")
           (concatenate 'string "(format t \"" new-line-converted "\" " (list-to-string parameters 0 " ") ")")
       )))

```

The convert-print-statement function converts a C-style printf statement to a Lisp format expression by removing the printf keyword and semicolon, extracting the string inside the double quotes, replacing format specifiers (e.g., %d) with Lisp format placeholders (e.g., ~a), and then appending any parameters to the format call if they exist.

- **Convert-arithmetic-and-logical-expr Functions**

```
(defun convert-arithmetical-expression (line next-line)
    ;; example input : "b + c;"
    ;; example output : "(+ b c)"
    ;; first remove the semicolon
    ;; then add space before and after the delimiters then split the string to get tokens
    ;; then evaluate the infix arithmetic expression
    (let* ((trimmed-line (list-to-string (split-string " " line 0 '() "" nil) 0 " "))
           (arithmetical-expr (subseq trimmed-line 0 (- (length trimmed-line) 1)))
           (splitted-line (split-string " " (add-space-before-after-delimiters arithmetical-expr 0) 0 '() "" nil)))
       (if splitted-line
           (concatenate 'string (evaluate-infix-expression (reverse splitted-line) '() '() 0))
           "nil"
       )
    )
)

(defun convert-logical-expression (line next-line)
    ;; example input : "a && b;"
    ;; example output : "(and a b)"
    ;; first remove the semicolon
    ;; then add space before and after the logical delimiters then split the string to get tokens
    ;; then evaluate the infix logical expression
    (let* ((trimmed-line (list-to-string (split-string " " line 0 '() "" nil) 0 " "))
           (logical-expr (subseq trimmed-line 0 (- (length trimmed-line) 1)))
           (splitted-line (split-string " " (add-space-before-after-logical-delimiters logical-expr 0) 0 '() "" nil)))
       (if splitted-line
           (concatenate 'string (evaluate-infix-expression (reverse splitted-line) '() '() 0))
           "nil"
       )
    )
)
```

The convert-logical-expression and convert-arithmetical-expression functions remove the semicolon, add spaces around delimiters, split the expression into tokens, and convert it from infix to Lisp's prefix notation, returning "nil" if no expression is found.

- **Convert-function-call/end-block and empty-line functions**

```
(defun convert-function-call (line next-line)
    ;; example input : "func1(a, b);"
    ;; example output : "(func1 a b)"
    ;; first remove the semicolon
    ;; then split the string by ',' and convert the parameters to lisp data types
    (let* ((trimmed-line (remove-whitespace line))
           (func-name (subseq trimmed-line 0 (position (code-char 40) trimmed-line)))
           (parameters (subseq trimmed-line (+ 1 (position (code-char 40) trimmed-line)) (- (length trimmed-line) 2)))
           (splitted (split-string "," parameters 0 '() "" nil)))
        (if (null splitted)
            (concatenate 'string "(" func-name ")")
            (concatenate 'string "(" func-name " " (list-to-string splitted 0 " ") ")")
        )
    )
)

(defun convert-end-block (line next-line)
    ;; returns the end of the block
    (concatenate 'string "))")
)

(defun empty-line (line next-line)
    "" )
```

The convert-function-call function converts a C-style function call by removing the semicolon, extracting the function name and parameters, splitting parameters by commas, and formatting them in Lisp's function call notation (e.g., converting func1(a, b); to (func1 a b)); if no parameters are found, it returns only the function name with parentheses.

The convert-end-block function returns the Lisp equivalent of ending a block by adding closing parentheses "))", while the empty-line function returns an empty string, effectively skipping blank lines.

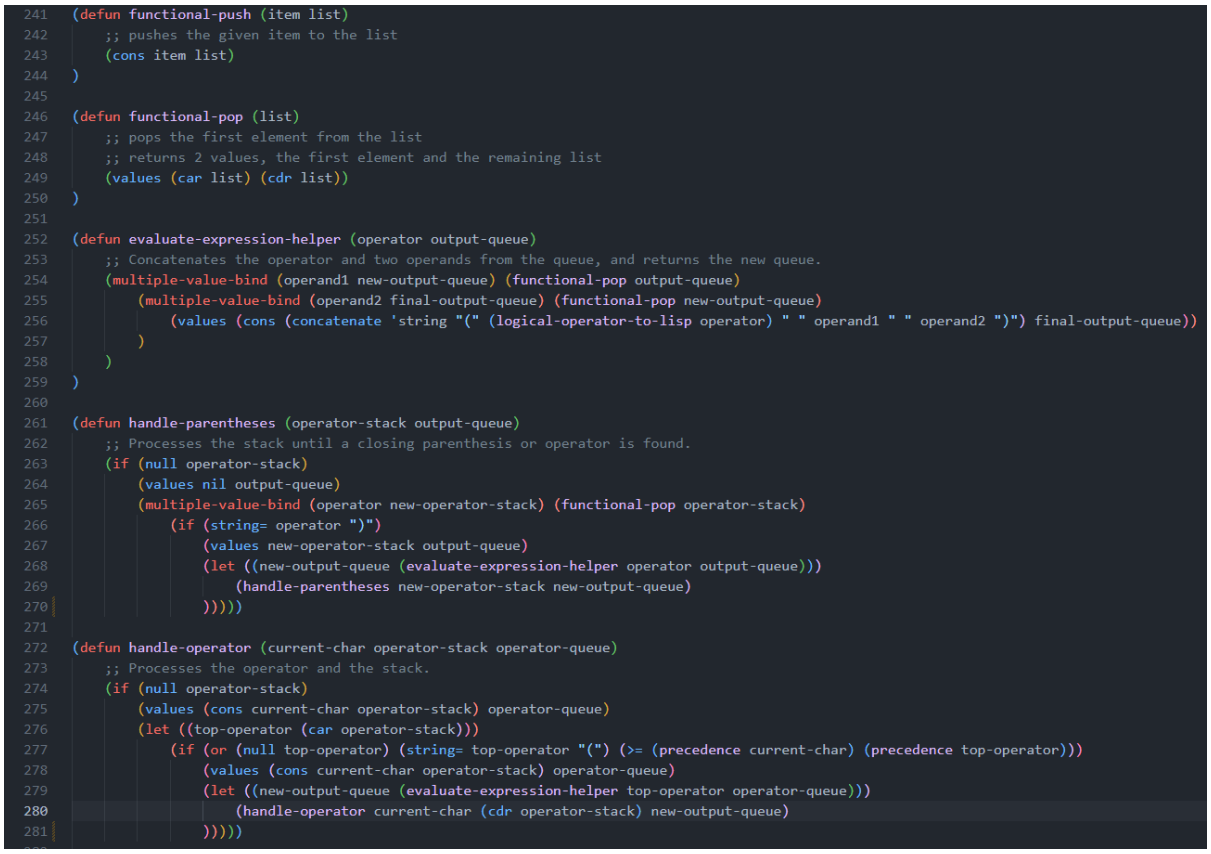
- **Additional Helper Functions**
    - **evaluate-infix-expression Function**

```
(defun evaluate-infix-expression (infix operator-stack output-queue index)
    ;; Evaluates the infix expression and returns the lisp equivalent.
    ;; The infix expression is reversed and processed from right to left.
    ;; The operator stack is used to store the operators.
    ;; The output queue is used to store the operands.
    (if (>= index (length infix))
        (multiple-value-bind (op remaining-operator-stack) (functional-pop operator-stack)
            (if (null op)
                (list-to-string output-queue 0 "")
                (let ((new-output-queue (evaluate-expression-helper op output-queue)))
                    (if (null remaining-operator-stack)
                        (list-to-string new-output-queue 0 "")
                        (evaluate-infix-expression infix remaining-operator-stack new-output-queue (+ index 1))
                    )
                )
            )
        )
        (let ((current-char (nth index infix)))
            (cond
                ((string= current-char ")") (evaluate-infix-expression infix (cons current-char operator-stack) output-queue (+ index 1)))
                ((string= current-char "(") (multiple-value-bind (new-operator-stack new-output-queue) (handle-parentheses operator-stack output-queue)
                                              (evaluate-infix-expression infix new-operator-stack new-output-queue (+ index 1)))
                )
                ((is-operator current-char) (multiple-value-bind (new-operator-stack new-output-queue) (handle-operator current-char operator-stack output-queue)
                                              (evaluate-infix-expression infix new-operator-stack new-output-queue (+ index 1)))
                )
                (t (evaluate-infix-expression infix operator-stack (functional-push current-char output-queue) (+ index 1)))
            )
        )
    )
)
```

This function evaluates an infix expression by iterating over its elements from right to left, managing operators in a stack and operands in a queue, and recursively building a Lisp-

formatted output. To make it purely functional, I used multiple-value-bind function in lisp which allows me to return multiple variables from functions

- ## Functions that I used in evaluate-infix-expression Function



```lisp
(defun functional-push (item list)
    ;; pushes the given item to the list
    (cons item list)
)

(defun functional-pop (list)
    ;; pops the first element from the list
    ;; returns 2 values, the first element and the remaining list
    (values (car list) (cdr list))
)

(defun evaluate-expression-helper (operator output-queue)
    ;; Concatenates the operator and two operands from the queue, and returns the new queue.
    (multiple-value-bind (operand1 new-output-queue) (functional-pop output-queue)
        (multiple-value-bind (operand2 final-output-queue) (functional-pop new-output-queue)
            (values (cons (concatenate 'string "(" (logical-operator-to-lisp operator) " " operand1 " " operand2 ")") final-output-queue))
        )
    )
)

(defun handle-parentheses (operator-stack output-queue)
    ;; Processes the stack until a closing parenthesis or operator is found.
    (if (null operator-stack)
        (values nil output-queue)
        (multiple-value-bind (operator new-operator-stack) (functional-pop operator-stack)
            (if (string= operator ")")
                (values new-operator-stack output-queue)
                (let ((new-output-queue (evaluate-expression-helper operator output-queue)))
                    (handle-parentheses new-operator-stack new-output-queue)
                )))))

(defun handle-operator (current-char operator-stack operator-queue)
    ;; Processes the operator and the stack.
    (if (null operator-stack)
        (values (cons current-char operator-stack) operator-queue)
        (let ((top-operator (car operator-stack)))
            (if (or (null top-operator) (string= top-operator "(") (>= (precedence current-char) (precedence top-operator)))
                (values (cons current-char operator-stack) operator-queue)
                (let ((new-output-queue (evaluate-expression-helper top-operator operator-queue)))
                    (handle-operator current-char (cdr operator-stack) new-output-queue)
                )))))
```

functional-push: Returns a new list with item added to the front of list, similar to the push operation but without mutating the original list. **functional-pop:** Returns the first element and the rest of the list as two separate values, like pop, but without modifying the original list. **evaluate-expression-helper:** Pops two operands from output-queue, applies the operator, and returns a new queue with the evaluated expression at the front. **handle-parentheses:** Processes elements in operator-stack until encountering a closing parenthesis, then evaluates and adds expressions to output-queue recursively. **handle-operator:** Manages current-char (an operator) by comparing precedence with operators in operator-stack. It evaluates higher-precedence operators first, then recursively continues the process. "**values**" allows a function to return multiple values at once, while "**multiple-value-bind**" binds each returned value to a variable in the receiving function.

- ## List-to-string Function

```lisp
(defun list-to-string (list index separator)
    ;; converts a list to a string by concating element with given separator recursively
    (if (>= index (length list))
        ""
        (string-trim " " (concatenate 'string (nth index list) separator (list-to-string list (+ index 1) separator)))
    )
)
```

The list-to-string function recursively converts a list to a string by concatenating its elements with a given separator, trimming any extra spaces along the way.

- ## Remove-whitespace Function

```lisp
(defun remove-whitespace (str)
    ;; removes the whitespaces from the given string
    (remove-if #'(lambda (char)
                   (find char " "))
               str)
)
```

The `remove-whitespace` function removes all whitespace characters from a given string by filtering out any spaces.

- ## Add-till-space Function

```lisp
(defun add-till-space (line original-line index)
    ;; add till first character that is not a space for alignment
    (if (>= index (length original-line))
        ""
        (let* ((current-char (subseq original-line index (+ index 1))))
            (if (string= current-char " ")
                (concatenate 'string current-char (add-till-space line original-line (+ index 1)))
                line
            ))))
```

The add-till-space function recursively adds spaces from the start of original-line to line until it encounters a non-space character, helping to align line with the original's leading spaces.

- ## Add-space-before-after-(logical)-delimiters Functions

```lisp
(defun add-space-before-after-delimiters (line index)
    ;; adds space before and after the delimiters
    (if (>= index (length line))
        ""
        (let* ((current-char (subseq line index (+ index 1))))
            (if (or (string= current-char "(") (string= current-char ")")
                    (string= current-char "+") (string= current-char "-") (string= current-char "*")
                    (string= current-char "/") (string= current-char "%"))
                (concatenate 'string " " current-char " " (add-space-before-after-delimiters line (+ index 1)))
                (concatenate 'string current-char (add-space-before-after-delimiters line (+ index 1)))))))
```

```lisp
(defun add-space-before-after-logical-delimiters (line index)
    ;; adds space before and after the logical delimiters
    ;; first gets the current char and the next char
    ;; then checks if the two char is a logical delimiter
    ;; if it is a logical delimiter then adds space before and after the delimiter
    (if (>= index (length line))
        ""
        (let* ((current-char (subseq line index (+ index 1)))
               (next-char (if (< (+ index 1) (length line)) (subseq line (+ index 1) (+ index 2)) nil))
               (two-char-op (if next-char (concatenate 'string current-char next-char) nil)))
            (cond
                ((or (string= two-char-op "<=") (string= two-char-op ">=") (string= two-char-op "==")
                     (string= two-char-op "!=") (string= two-char-op "&&") (string= two-char-op "||"))
                    (concatenate 'string " " two-char-op " " (add-space-before-after-logical-delimiters line (+ index 2))))
                ((or (string= current-char "<") (string= current-char ">") (string= current-char "!")
                     (string= current-char "(") (string= current-char ")"))
                    (concatenate 'string " " current-char " " (add-space-before-after-logical-delimiters line (+ index 1))))
                (t (concatenate 'string current-char (add-space-before-after-logical-delimiters line (+ index 1))))))))
```

The add-space-before-after-logical-delimiters function adds spaces around logical delimiters in a line by recursively checking each character (and the next character when needed) to identify logical operators (<=, >=, ==, !=, &&, ||) and single-character delimiters (<, >, !, (,)), adding spaces before and after them for readability.

- ## Split-string Function

```lisp
(defun split-string (separator line index result token is-splittable)
    ;; splits a string by the given separator and returns a list of the parts
    (let ((separator-length (length separator)))
        (if (>= index (length line))
            (if (string= token "")
                result
                (append result (list token)))
            (let ((current-substr (subseq line index (min (+ index separator-length) (length line)))))
                (if (string= current-substr separator)
                    (split-string separator line (+ index separator-length)
                                  (if (string= token "")
                                      result
                                      (append result (list token)))
                                  "" nil)
                    (split-string separator line (+ index 1)
                                  result
                                  (concatenate 'string token (subseq line index (+ index 1))) t)
                )))))

```

The split-string function recursively splits a string by a given separator, building a list of parts by checking each substring for the separator and either appending the accumulated token to result (when the separator is found) or continuing to build the token otherwise. If there's no separator match by the end, the last token is appended to the result.

- ## is-start-with-data-type Function

```lisp
(defun is-start-with-data-type (line)
    ;; checks if the given line starts with a c-type data type
    (cond
        ((and (> (length line) 3) (string= (subseq line 0 3) "int")) t)
        ((and (> (length line) 5) (string= (subseq line 0 5) "float")) t)
        ((and (> (length line) 6) (string= (subseq line 0 6) "double")) t)
        (t nil)
    )
)
```

The is-start-with-data-type function checks if a given line starts with a C-type data type (int, float, or double) by examining the beginning of the string, returning t if it matches any of these types and nil otherwise.

- ## extract-var-name Function

```lisp
(defun extract-var-name (variable-name)
    ;; extracts the variable name from the given variable name
    ;; example: int a --> a
    (cond
        ((and (> (length variable-name) 3) (string= (subseq variable-name 0 3) "int")) (subseq variable-name 3 (length variable-name)))
        ((and (> (length variable-name) 5) (string= (subseq variable-name 0 3) "float")) (subseq variable-name 5 (length variable-name)))
        ((and (> (length variable-name) 6) (string= (subseq variable-name 0 3) "double")) (subseq variable-name 6 (length variable-name)))
    )
)
```

The extract-var-name function extracts the variable name from a declaration by removing the data type prefix (int, float, or double) from the given variable declaration string, returning only the name part (e.g., "int a" becomes "a").

- ## extract-end-value Function

```lisp
(defun extract-end-value (expr varname relational-operator)
    ;; 2nd expression in the for loop
    ;; example: a < 5 / a > 5 / a <= 5 / a >= 5
    ;; expr is a string that contains the relational operator and the value
    ;; extract the value from the expression
    (let* ((split-parts (split-string relational-operator expr 0 nil "" nil)))
        (if (string= (nth 0 split-parts) varname)
            (nth 1 split-parts)
            (nth 0 split-parts)
        )))
```

The extract-end-value function retrieves the end value from a relational expression (like a < 5 or a >= 5) by splitting the expression based on the relational operator. It then returns the part that is not the variable name, providing the comparison value in a for loop condition.

- **extract-increment-value Function**

```lisp
(defun extract-increment-value (expr varname)
    ;; 3rd parameter of for loop is an expression that contains the increment value
    ;; example: a++ / a-- / a + 5 / a - 5
    ;; expr is a string that contains the increment value
    (cond
        ((search "++" expr) "1")
        ((search "--" expr) "-1")
        ((search "+" expr) (progn
                                (let* ((split-parts (split-string "+" expr 0 nil "" nil)))
                                    (if (string= (nth 0 split-parts) varname)
                                        (nth 1 split-parts)
                                        (nth 0 split-parts)
                                    )
                                )
                            ))
        ((search "-" expr) (progn
                                (let* ((split-parts (split-string "-" expr 0 nil "" nil)))
                                    (if (string= (nth 0 split-parts) varname)
                                        (nth 1 split-parts)
                                        (nth 0 split-parts)
                                    )
                                )
                            ))
    )
)
```

The extract-increment-value function determines the increment value in expressions like a++, a--, a + 5, or a - 5. It returns "1" for a++, "-1" for a--, or splits the expression based on + or - to extract the numeric increment value, returning the part not matching the variable name.

- **extract-relational-operator Function**

```lisp
(defun extract-relational-operator (expr)
    ;; expr is a string that contains the relational operator and the variable name
    ;; we need to find which relational operator is used in the expression
    (cond
        ((search ">=" expr) ">=")
        ((search "<=" expr) "<=")
        ((search ">" expr) ">")
        ((search "<" expr) "<")
    )
)
```

The extract-relational-operator function identifies and returns the relational operator used in a given expression (>=, <=, >, or <) by checking for each operator in sequence.

- **Precedence Function**

```lisp
(defun precedence (operator)
    ;; returns the precedence of the given operator
    (cond
        ((string= operator "+") 1)
        ((string= operator "-") 1)
        ((string= operator "*") 2)
        ((string= operator "/") 2)
        ((string= operator "%") 2)
        ((string= operator "||") 0)
        ((string= operator "&&") 1)
        ((or (string= operator "==") (string= operator "!=") (string= operator "<") (string= operator ">")
            (string= operator "<=") (string= operator ">=")) 2)
        ((string= operator "!") 3)
        (t -1)))
```

The precedence function returns the precedence level of a given operator, assigning higher values for higher precedence with unknown operators defaulting to -1.

- **is-operator Function**

```lisp
(defun is-operator (operator)
    ;; checks if the given operator is an operator
    (cond
        ((string= operator "+") t)
        ((string= operator "-") t)
        ((string= operator "*") t)
        ((string= operator "/") t)
        ((string= operator "%") t)
        ((string= operator "||") t)
        ((string= operator "&&") t)
        ((string= operator "==") t)
        ((string= operator "!=") t)
        ((string= operator "<") t)
        ((string= operator ">") t)
        ((string= operator "<=") t)
        ((string= operator ">=") t)
        ((string= operator "!") t)
        (t nil)
    )
)
```

The `is-operator` function checks if a given string is a recognized operator (such as `+`, `-`, `*`, `/`, `%`, `||`, `&&`, `==`, `!=`, `<`, `>`, `<=`, `>=`, `!`), returning `t` if it is an operator and `nil` otherwise.

- **logical-operator-to-lisp Function**

```lisp
(defun logical-operator-to-lisp (operator)
    ;; converts the given logical operator to lisp equivalent
    (cond
        ((string= operator "&&") "and")
        ((string= operator "||") "or")
        ((string= operator "!") "not")
        ((string= operator "==") "eq")
        ((string= operator "!=") "/=")
        (t operator)
    )
)
```

The logical-operator-to-lisp function converts common logical operators to their Lisp equivalents, mapping && to "and", || to "or", ! to "not", == to "eq", and != to "/=", while returning the original operator if no match is found.

- **definition-func-parameters**

```lisp
(defun definition-func-parameters (parameters)
    ;; takes a list of parameters and returns the string except the data type
    ;; (int a, int b) --> "a b"
    (if (null parameters)
        ""
        (cond
            ((string= (subseq (car parameters) 0 3) "int")
                (concatenate 'string (subseq (car parameters) 3 (length (car parameters))) " " (definition-func-parameters (cdr parameters))))
            ((string= (subseq (car parameters) 0 5) "float")
                (concatenate 'string (subseq (car parameters) 5 (length (car parameters))) " " (definition-func-parameters (cdr parameters))))
            ((string= (subseq (car parameters) 0 6) "double")
                (concatenate 'string (subseq (car parameters) 6 (length (car parameters))) " " (definition-func-parameters (cdr parameters))))
        )
    )
)
```

The definition-func-parameters function processes a list of function parameters by removing the data type prefixes (int, float, double) and returning a concatenated string of parameter names (e.g., (int a, int b) becomes "a b"). It recursively removes each data type prefix and concatenates the parameter names with a space.

- **c-to-lisp-data-type**

```lisp
(defun c-to-lisp-data-type (data-type)
    ;; converts a string c-type data type to lisp-type data type
    (cond
        ((string= data-type "int") "integer")
        ((string= data-type "float") "single-float")
        ((string= data-type "double") "double-float")
    )
)
```

The c-to-lisp-data-type function converts C data types to their Lisp equivalents, mapping "int" to "integer", "float" to "single-float", and "double" to "double-float".

- **convert-parameters-c-to-lisp**

```lisp
(defun convert-parameters-c-to-lisp (parameters)
    ;; converts a list of c-type parameters to string of lisp-type parameters
    (if (null parameters)
        ""
        (cond
            ((string= (subseq (car parameters) 0 3) "int") (concatenate 'string "integer " (convert-parameters-c-to-lisp (cdr parameters))))
            ((string= (subseq (car parameters) 0 5) "float") (concatenate 'string "single-float " (convert-parameters-c-to-lisp (cdr parameters))))
            ((string= (subseq (car parameters) 0 6) "double") (concatenate 'string "double-float " (convert-parameters-c-to-lisp (cdr parameters))))))
)
```

The convert-parameters-c-to-lisp function converts a list of C-type parameters to a string of Lisp-type parameters by recursively mapping "int" to "integer", "float" to "single-float", and "double" to "double-float", appending each converted type with a space.

- **extract-formatters and get-string-between-quotes**

```lisp
(defun get-string-between-quotes (line)
    (let* ((quote-start (position #\" line))
           (after-first-quote (subseq line (+ quote-start 1) (length line)))
           (quote-end (position #\" after-first-quote)))
        (if (null quote-start)
            ""
            (subseq after-first-quote 0 quote-end)
        )
    )
)

(defun extract-formatters (line index)
    ;; extracts the new line character from the given line
    ;; if the new line character is not found then returns the line
    (if (>= index (length line))
        ""
        (let* ((current-char (subseq line index (+ index 1)))
               (next-char (if (< (+ index 1) (length line))
                              (subseq line (+ index 1) (+ index 2)) nil)))
            (cond
                ((and (string= current-char "\\") (string= next-char "n"))
                    (concatenate 'string "~%" (extract-formatters line (+ index 2))))
                ((and (string= current-char "%") (string= next-char "d"))
                    (concatenate 'string "~a" (extract-formatters line (+ index 2))))
                ((and (string= current-char "%") (string= next-char "f"))
                    (concatenate 'string "~a" (extract-formatters line (+ index 2))))
                (t (concatenate 'string current-char (extract-formatters line (+ index 1))))
            )
        )
    )
)
```

The get-string-between-quotes function extracts a substring enclosed in double quotes within the input string line. It finds the position of the first quote (quote-start) and creates a substring starting right after this first quote. Then it searches for the next quote within this substring (after-first-quote) and returns the content between them. If there’s no opening quote, it returns an empty string.

The extract-formatters function processes a line to convert C-style format specifiers (\n, %d, %f) into their Lisp equivalents (~%, ~a), recursively iterating through the line. If a specifier is not found, it simply returns the line character-by-character.