GTU Department of Computer Engineering

CSE 341 - Fall 2024

Homework 1 Report

ALPER TAVŞANOĞLU

210104004142

In this assignment, i implement a functional recursive program in Lisp that reads a C file **(input.c)** line by line, converts each line into Lisp code, and writes the fully converted code to a new file **(output.lisp)**. The last page includes **usage and test results.**

Functions

read_file

This function, using recursion to read each line of the file into a list. If the file is not found, it returns nil.

write_file

This function, recursively writes each line to the file. If the file exists, it is overwritten; if not, it creates a new file.

```
(defun read_file (filename)
  (format t "~%[DEBUG] Reading from file: ~A~%" filename)
  (with-open-file (stream filename :if-does-not-exist nil)
    (when stream
      (labels ((read-lines-recursively (stream)
                 (let ((line (read-line stream nil nil)))
                   (if line
                       (cons line (read-lines-recursively stream))
                       nil))))
        (read-lines-recursively stream)))))

(defun write_file (filename lines)
  (format t "~%[DEBUG] Writing to file: ~A~%" filename)
  (with-open-file (stream filename :direction :output :if-exists :supersede :if-does-not-exist :create)
    (labels ((write-lines-recursively (lines)
               (when lines
                 (write-line (first lines) stream)
                 (write-lines-recursively (rest lines)))))
      (write-lines-recursively lines))))
```

line_type

This function, identifies the line according to type of the line. For example, if statement, for/while loop, variable declaration, etc..., based on specific keywords and patterns in the line.

conversion_foo

This function, receives the line type and returns the corresponding conversion function and apply the correct transformation.

convert

This function, trims whitespace, tab,newline and determines the line type using line_type, retrieves the correct conversion function using conversion_foo, and applies the appropriate conversion function to generate the corresponding Lisp code.

Conversion Functions

convert_function_prototype

This function, converts a C function prototype to a Lisp declaim statement.

convert_function_definition

This function, converts a C function definition to a Lisp defun , extracting and formatting parameter names.

convert_if

This function, converts a C if statement to a Lisp if expression with a progn block for inner lines.

convert_for

This function, transforms a C for loop into a Lisp loop construct with progn for nested lines.

convert_while

This function, converts a C while loop into a Lisp loop, converting the condition and nesting inner lines with progn.

```lisp
(defun convert_function_definition (line)                                    ;; To handle functin definitions > defun
  (let* ((name-and-params (subseq line 0 (search "(" line)))
         (name (second (split_by_char #\space name-and-params)))
         (params (subseq line (1+ (search "(" line)) (search ")" line)))
         (param-names (if (string= params "")
                          '()
                          (mapcar (lambda (param)
                                    (second (split_by_char #\space (string-trim '(#\Space #\Tab) param))))
                                  (split_by_char #\, params)))))
    (format nil "(defun ~a (~{~a~^ ~})" name param-names)))

(defun convert_if (line inner_lines)                                         ;; To handle if statements
  (let* ((condition (subseq line (1+ (search "(" line)) (search ")" line)))
         (formatted-condition (convert_condition condition))
         (converted_inner_lines (mapcar #'convert inner_lines))
         (inner-body (format nil "(progn~%    ~{~a~%    ~})" converted_inner_lines)))
    (format nil "(if ~a~%  ~a)" formatted-condition inner-body)))

(defun convert_for (line inner_lines)                                        ;; To handle for loops
  (let* ((loop-body (subseq line (search "(" line) (search ")" line)))
         (parts (split_by_char #\; loop-body))
         (init (first parts))
         (condition (second parts))
         (increment (third parts))
         (var-name (second (split_by_char #\space init)))
         (start-value (string-trim '(#\Space #\Tab) (subseq init (1+ (search "=" init)))))
         (target-value (string-trim '(#\Space #\Tab) (subseq condition (1+ (search "<" condition)))))
         (converted_inner_lines (mapcar #'convert inner_lines))
         (inner-body (format nil "(progn~%    ~{~a~%    ~})" converted_inner_lines)))
    (format nil "(loop for ~a from ~a below ~a do~%  ~a)" var-name start-value target-value inner-body)))

(defun convert_while (line inner_lines)                                      ;; To handle while loops
  (let* ((condition (subseq line (1+ (search "(" line)) (search ")" line)))
         (formatted-condition (convert_condition condition))
         (converted_inner_lines (mapcar #'convert inner_lines))
         (inner-body (format nil "(progn~%    ~{~a~%    ~})" converted_inner_lines)))
    (format nil "(loop while ~a do~%  ~a)" formatted-condition inner-body)))
```

convert_return

This function, convert C return statement to Lisp, translating return expression and closing let block.

convert_declaration

This function, converts variable declarations to variable bindings for use in let blocks, formatting each as (var value).

convert_assignment

This function, converts assignment statements to setf.

convert_print

This function, converts printf statements to format, replacing C format specifiers (%d, %s) with Lisp equivalents.

convert_increment_decrement

This function, converts C increment (x++) or decrement (x--) to Lisp (incf x) or (decf x).

convert_logical_operation

This function, convert C logical operations (&&, ||) to Lisp equivalents (and, or).

convert_unknown

This function, to handle unknown or unsupported lines.

```
(defun convert_return (line)
  (let* ((return-value (string-trim '(#\Space #\Tab)
                                    (subseq line (+ (search "return" line) 7) (search ";" line))))
         (operator (find-if (lambda (op) (search op return-value)) '("+" "-" "*" "/"))))
    (if operator
        (let* ((parts (split-by-char (char operator 0) return-value))
               (formatted-parts (mapcar (lambda (part) (string-trim '(#\Space #\Tab) part)) parts)))
          (format nil "  (~a ~a ~a))" operator (first formatted-parts) (second formatted-parts)))
        (format nil "  )~%~a)" return-value))))

(defun convert_declaration (line)
  (let* ((trimmed-line (string-trim '(#\Space #\Tab) line))
         (normalized-line (replace-regexp-in-string " *= *" "=" trimmed-line))
         (parts (split-by-char #\= normalized-line))
         (var-part (first parts))
         (value-part (second parts))
         (var-name (second (split-by-char #\space var-part)))
         (cleaned-value (string-trim '(#\Space #\Tab #\;) value-part))
         (formatted-value (if (search "(" cleaned-value)
                              (format-function-call cleaned-value)
                              cleaned-value)))
    (format nil "(~a ~a)" var-name formatted-value)))

(defun convert_assignment (line)
  (let* ((parts (split-by-char #\= (string-trim '(#\Space #\Tab) line)))
         (var-name (string-trim '(#\Space #\Tab) (first parts)))
         (value (string-trim '(#\Space #\Tab #\;) (second parts)))
         (formatted-value (if (search "(" value)
                              (format-function-call value)
                              value)))
    (format nil "(setf ~a ~a)" var-name formatted-value)))
```

Helper Functions

replace_c_format_specifiers

This function, helps to replaces C format specifiers (%d, %s) with equivalent Lisp format specifiers (~d, ~a, and ~%) in printf conversions.

replace_c_logical_operators

This function, helps to replaces C logical operators (&&, ||) with Lisp equivalents (and, or).

replace_regexp_in_string

This function, helps to replaces all occurrences of a given regular expression in a string with a specified replacement.

convert_condition

This function, helps to converts C-style infix conditions to Lisp-style prefix conditions (e.g., a > b to (> a b)). Specially helper for if and while functions.

convert_declaration_for_let

This function, helps to convert a declaration line to a let binding form.

format_function_call

Formats function calls by separating the function name and arguments.

format_let_block

Format multiple declarations as a let block.

split_by_char

This function, helps to splits strings by a given character. Splits strings by a specified character into a list of parts.

```
;;Helper Functions
(defun replace_c_format_specifiers (format-string)                    ; Helper function for print to c specifiers to lisp specifiers
  (labels ((replace_recursively (str replacements)
             (if (null replacements)
                 str
                 (let* ((from (caar replacements))
                        (to (cdar replacements))
                        (pos (search from str)))
                   (if pos
                       (concatenate 'string
                                    (subseq str 0 pos)
                                    to
                                    (replace_recursively (subseq str (+ pos (length from))) replacements))
                       (replace_recursively str (cdr replacements)))))))
    (replace_recursively format-string '(("%d" . "~d")
                                         ("%s" . "~a")
                                         ("\\n" . "~%")))))

(defun replace_c_logical_operators (expr)                                    ; Helper function for logical operations
  (replace_regexp_in_string "&&" "and" (replace_regexp_in_string "||" "or" expr)))

(defun replace_regexp_in_string (regexp replacement string)                  ; Helper function regular expressions
  (labels ((replace_recursively (str start)
             (let ((pos (search regexp str :start2 start)))
               (if pos
                   (concatenate 'string
                                (subseq str 0 pos)
                                replacement
                                (replace_recursively str (+ pos (length regexp))))
                   str))))
    (replace_recursively string 0)))
```

c_to_lisp_recursive

This recursive function, for processing the C code line-by-line. It uses the identified line types to determine if it should apply a specific construct conversion (like for, if, or while). It also identifies consecutive declarations to format them in a single let block.

```
(defun c_to_lisp_recursive (lines)
  (when lines
    (let ((line (first lines))
          (rest_lines (rest lines)))
      (cond
        ((and (search "int" line) (search "(" line) (not (search ";" line)))
         (cons (convert_function_definition line) (c_to_lisp_recursive rest_lines)))

        ((and (search "for" line) (search "{" line))
         (multiple-value-bind (inner_lines remaining-lines) (collect_inner_lines rest_lines)
           (cons (convert_for line inner_lines) (c_to_lisp_recursive remaining-lines))))

        ((and (search "if" line) (search "{" line))
         (multiple-value-bind (inner_lines remaining-lines) (collect_inner_lines rest_lines)
           (cons (convert_if line inner_lines) (c_to_lisp_recursive remaining-lines))))

        ((and (search "int" line) (search "=" line))
         (multiple-value-bind (declarations remaining-lines) (collect_declarations lines)
           (cons (format_let_block declarations) (c_to_lisp_recursive remaining-lines))))

        ((and (search "while" line) (search "{" line))
         (multiple-value-bind (inner_lines remaining-lines) (collect_inner_lines rest_lines)
           (cons (convert_while line inner_lines) (c_to_lisp_recursive remaining-lines))))

        (t (cons (convert line) (c_to_lisp_recursive rest_lines)))))))
```

collect_inner_lines

Collects lines within a code block (e.g., inside if, for, while), until it encounters a closing brace }.

collect_declarations

Collects consecutive variable declarations to format them into a single let block.

c_to_lisp

Runs the conversion by reading lines from the input file, applying c-to-lisp-recursive, and writing the converted output to the specified file.

```
(defun c_to_lisp (input_file output_file)
  (let ((lines (read_file input_file)))
    (if lines
        (let ((converted-lines (c_to_lisp_recursive lines)))
          (write_file output_file converted-lines)
          (format t "~%[DEBUG] Conversion completed successfully~%"))
        (format t "~%[DEBUG] No lines read from input file~%"))))

(let ((args ext:*args*))
  (if (= (length args) 2)
      (c_to_lisp (first args) (second args))
      (format t "Usage: clisp code.lisp input.c output.lisp~%")))
```

Usage and Test Results

```
alper@alper-VirtualBox:~/Desktop$ clisp converter.lisp input.c output.lisp

Reading from file: input.c

Writing to file: output.lisp

Conversion completed successfully
alper@alper-VirtualBox:~/Desktop$
```

For this input.c example

input.c

```c
int sum(int a, int b);

int sum(int a, int b) {
    return a + b;
}

int main() {
    int x = 10;
    int y = 20;
    int result = sum(x, y);

    if (result > 25) {
        printf("Result is greater than 25\n");
                x = 5;
    }

    for (int i = 0; i < 10; i++) {
        printf("%d\n", i);
    }

    return 0;
}
```

output.lisp

output.lisp

```lisp
(declaim (ftype (function (integer integer) integer) sum))

(defun sum (a b)
  (+ a b))


(defun main ()
(let (    (x 10)
    (y 20)
    (result (sum x y)))

(if (> result 25)

(progn
    (format t "Result is greater than 25~%")
    (setf x 5)
    ))

(loop for i from 0 below 10 do

(progn
    (format t "~d~%" i)
    ))

  )
0)
```

Another input.c example

```c
int sum(int a, int b);

int sum(int a, int b) {
    return a + b;
}

int main() {
    int x = 10;
    int y = 20;
    int result = sum(x, y);
    int z = 23;

    if (result < 30) {
        printf("Result is less than 30\n");
        z=40;
    }
    while(x>7){
    printf("X greater thant 7\n");
    x=7;
    }
    for (int i = 0; i < 10; i++) {
        printf("%d\n", i);
    }

    return 0;
```

output.lisp

```lisp
(declaim (ftype (function (integer integer) integer) sum))

(defun sum (a b)
  (+ a b))


(defun main ()
(let (    (x 10)
    (y 20)
    (result (sum x y))
    (z 23))

(if (< result 30)

(progn
    (format t "Result is less than 30~%")
    (setf z 40)
    ))
(loop while (> x 7) do

(progn
    (format t "X greater thant 7~%")
    (setf x 7)
    ))
(loop for i from 0 below 10 do

(progn
    (format t "~d~%" i)
    ))

  )
0)
```