Sonoma State University
Computer Science Department
CS 460 – Fall 2023 – Watts

Exercise 3
Part 1 Date Due: Wednesday, 1 November 2023; 2:59 pm.
Part 2 Date Due: Monday, 5 November 2023; 6:59 am.

This tutorial is designed to give you an introduction to the hybrid language PL460. The answers to the questions in the steps preceded by an asterisk (*) in the first part of this tutorial should be stored in a file called *lastname*E3.txt. A template .txt file can be copied from the Exercise 3 pickup folder; it is called `lastnameE3.txt.`

Exercise 3 Part 1, **Getting Started**

1. Copy the directory Exercise3 from the course pickup folder to your account on the CS Department server. This directory contains the PL460 interpreter and several sample programs:

```
ex-1.cpp    hw-1.cpp    hw-2.cpp    hw-3.cpp    hw-4.cpp    pl460
ex-1.pl460  hw-1.pl460  hw-2.pl460  hw-3.pl460  hw-4.pl460
```

2. Using an editor, cat, more, or less, view the contents of the file hw-1.pl460:

```
;;; This is your first PL460 program

(define (main)
        (display "Hello World")
        (newline)
)

(main)
```

3. Run the PL460 interpreter using this program as input:

```
./pl460 hw-1.pl460
```

   Review, compile, and run the C++ program hw-1.cpp. How does the result differ from the PL460 result?

4. Like Python, PL460 needs to call the primary program function to have it interpreted. In hw-1.pl460, the name of the primary function is "main" and you can see the call to main at the end of the program: `(main)` The primary function does not need to be called main. In hw-2.pl460, you will see that the primary function is "helloWorld". Using the interpreter, run this program.

```
;;; This is your second PL460 program

(define (helloWorld)
        (display "Hello World")
        (newline)
)
```

```
(helloWorld)
```

Review, compile, and run the C++ program hw-2.cpp. How does the result differ from the PL460 result? Since we will be translating PL460 programs to C++ program, most of the examples in this exercise will call the primary program function “main”.

5. The interpreter will detect lexical, syntactical, and semantic errors in a program. The program hw-3.pl460 contains a syntactical error. A list of errors will be displayed when the program is interpreted. Additionally, lexical and syntactical errors will be inserted into the code listing (the .lst file.) Using the interpreter, run this program. View the errors in hw-3.lst.

```
;;; This PL460 program contains a syntax error
;;; The display command only expects one argument

(define (main)
        (display "Hello" "World")
        (newline)
)

(main)
```

*Review, compile, and run the C++ program hw-3.cpp. How does the result differ from the PL460 result?

6. The program hw-4.pl460 contains a semantic error. A list of semantic errors will be displayed when the program is interpreted. Using the interpreter, run this program.

```
;;; This PL460 program contains a semantic error
;;; The variable 'Hello' has not been bound to a value

(define (main)
        (display Hello)
        (newline)
)

(main)
```

*Review, compile, and run the C++ program hw-4.cpp. How does the result differ from the PL460 result?

PL460 Arithmetic Expressions

7. PL460 arithmetic expressions are expressed using the standard operators (`+ -* /`) and prefix notation. View and interpret the program ex-1.pl460.

```
;;; This PL460 program uses arithmetic expressions

(define (main)
        (+ 2 5)
```

```
)

(main)
```

You should not have received any output when you interpreted this program. The value 7 was calculated, but never displayed. Review, compile, and run the C++ program hw-4.cpp. How does the result differ from the PL460 result?

Modify thePL460 program to display the calculated value:

```
(display (+ 2 5))
```

This time, you should see the value 7 when you interpret the program. Further modify the program to display more useful information and an end of line:

```
(display "(+ 2 5) --> ")
(display (+ 2 5))
(newline)
```

This time, you should see the value `(+ 2 5) --> 7` when you interpret the program.

8. Add additional lines of code to ex-1.pl460 to calculate and display the results of the following prefix notation arithmetic expressions.
   a. `(- 5.0 0.5)`
   b. `(* 5. .5)`
   c. `(/ 1.5 2.3)`
   d. `(+ 3/4 5/3)`
   e. `(-(* 2 2/5)(/ 2 5))`

9. Using the program ex-1.pl460 as a model, write a PL460 program called ex-2.pl460 to perform each of the following calculations. You will need to convert the expressions from infix notation to prefix notation.
   a. 1.2 × (2 - 1/3) + -8.7
   b. (2/3 + 2/9)/(5/1 -2/3)
   c. 1 + 1/(2 + 1/(1 + 1/2))
   d. 1 × -2 × 3 × -2 × 5 × -6 × 7
   e. 3/4 + 5/8 * 1/2 / 4/5

10. * When a statement uses a variety of numerical types, the compiler or interpreter will determine the type of the result. Using the following PL460 addition statements in ex-3.pl460, the results of addition operations mixing integers with real and fraction types have been entered in the adjacent table.

```
(+ 3 5) --> 8
(+ 3 5/3) --> 14/3
(+ 3 5.3) --> 8.3
(+ 3/4 5) --> 23/4
(+ 3.4 5) --> 8.4
```

+	Integer	fraction	real
integer	Integer	fraction	real
fraction	Fraction		
real	Real		

Complete this table by testing addition operators that combine real and fraction operands.

11. * Create tables similar to the one in the previous step for the subtraction, multiplication, and division operators.

PL460 List Operations

12. The PL460 language is a descendent of LISP – a list processing language. A literal list is represented as '(list of items in list). The PL460 program ls-1.pl460 contains:

```
;;; This PL460 program uses list operations

(define (main)
    (display "(car '(a b c)) --> ")
    (display (car '(a b c)))
    (newline)
)

(main)
```

Produces the output:

```
(car '(a b c)) --> a
```

13. Modify the program ls-1.p460 to display the output for the following list statements:

```
a. (cdr '(a b c))
b. (cons 'a '(b c))
c. (cons '(a b) '(c d))
d. (car '((1 2) (3 4) (5 6)))
e. (cdr '((1 2) (3 4) (5 6)))
f. (cons '55 '((1 2) (3 4) (5 6)))
g. (cons 55 '((1 2) (3 4) (5 6)))
h. (cons '(5 5) '((1 2) (3 4) (5 6)))
i. (cons (car '(a b c)) (cdr '(x y z)))
```

14. The PL460 program ls-2.pl460 contains:

```
;;; This PL460 program produces list operation errors

(define (main)
    (display "(car '55) --> ")
    (display (car '55))
    (newline)
)

(main)
```

Produces the output:

```
(car '55) --> In ls-2.pl460:
ls-2.pl460:5:17: In procedure main:
```

```
ls-2.pl460:5:17: In procedure car: Wrong type argument in position
1 (expecting pair): 55
```

because ’55 is not a list.

15. By modifying the program ls-2.pl460, determine the error messages produced by each of the following statements (Note: you will only be able to interpret one erroneous statement at a time).

```
a. (car 'a)
b. (cdr (a))
c. (cdr 'a)
d. (car (a b c))
e. (cdr (a b c))
f. (cons '(b c) 'a)
g. (cons 5 (a b c))
```

16. *Based on the above responses, describe the function of each of the PL460 functions represented:

```
a. car:
b. cdr:
c. cons:
d. the ' symbol:
```

17. List operations can be combined in commands. Using ls-2.pl460 as a model, create a program called ls-3.pl460 that displays the results of the following list operations:

```
a. (car '(a b c))
b. (car (cdr '(a b c)))
c. (cadr '(a b c))
d. (cdr '((a b) c))
e. (cdr (car '((a b) c)))
f. (cdar '((a b) c))
g. (cons (car '(a1 b2 c3)) (cdr '(x4 y5 z6)))
h. (cons (cdr '(a6 b5 c4)) (car '((x3 y2) z1)))
```

Note that the results of b and c are the same; cadr returns the car of the cdr. Similarly, the results of e and f are the same; cdar returns the cdr of the car.

18. Predict the results of the following PL460 commands:

```
a. (cons (cadr '(a1 b2 c3a)) (cdar '((w x y) z)))
b. (car (cons 5 '(a b c)))
c. (cdr (cons 5 '(a b c)))
d. (cdr '((a b) c))
e. (cons (car '(a1 b2 c3)) (cdr '(x4 y5 z6)))
f. (cons (cdr '(a6 b5 c4)) (car '((x3 y2) z1)))
g. (cons (cdr '(a b c)) (car '(x y z)))
```

Write a program called ls-4.pl460 displaying the results of these commands to determine whether your predictions were correct.

The last result contains a ‘.’. This indicates that the second parameter passed to the cons function was not a list.

PL460 expressions may span more than one line. The PL460 interpreter knows when it has parsed an entire expression by matching double quotes and parentheses.

Writing PL460 Functions

19. Some PL460 arithmetic operations are defined as function calls. For example, the modulo (%) function is defined using the modulo function and rounding is done using the round function. Write a PL460 program called fn-1.pl460 that displays the results of each of the following PL460 commands. Note that the modulo function requires 2 integer arguments.
    ```
    a. (modulo 12 5)
    b. (modulo (- 25 2) 7)
    c. (modulo (round 12.4) 2)
    ```

20. PL460 also includes predicate functions. These functions provide an answer to a question. `number?` is a predicate function; predicate functions will return #t if the answer is true and #f if it is not. Add statements to fn-1.pl460 that display the result of each of the following PL460 commands. Note that the `number?` function requires a single argument.
    ```
    a. (number? 5)
    b. (number? (- 2 5.5))
    c. (number? 12/4)
    d. (number? '0)
    e. (number? 'abc)
    f. (number? '(a b c))
    g. (number? '())
    h. (number? "(a b c) ")
    ```

    *What conclusions have you drawn about the functionality of the `number?` function?

21. Write a PL460 program called fn-2.pl460 and enter the following PL460 function definition at the beginning of the program:

    ```
    ; This function will return the square
    ; of a numeric input value
    (define (square n)
        (* n n)
    )
    ```

    The procedure square computes the square of any number n (n^2). Thee PL460 command define creates a function and describes its required formal parameter(s).

    Add a main function to fn-2.pl460 that displays the result of (square 5). Add a call to the main function.

    ```
    (define (main)
        (display "(square 5) --> ") (display (square 5)) (newline)
    )

    (main)
    ```

    What is the output of this program?

22. Add statements to the main function of fn-2.pl460 that will display the results of:
    a. `(square -200)`
    b. `(square 0.5)`
    c. `(square -1/2)`

    Note: PL460 systems that do not support exact ratios internally may print 0.25 for `(square -1/2)`.

23. Write a PL460 program called fn-3.pl460 and enter the following PL460 function definition at the beginning of the program:

```
; This function will return the multiplicative reciprocal
; of a numeric input value
(define (reciprocal n)
      (/ 1 n)
)
```

    Write and call a main function that displays the results of the following calls to reciprocal.
    a. `(reciprocal 10)`
    b. `(reciprocal 1/10)`
    c. `(reciprocal 10.0)`
    d. `(reciprocal 11.1)`
    e. `(reciprocal (reciprocal 1/10))`
    f. `(reciprocal 0)`

    *Why does the last statement result in an error?

    Replace that statement with a statement that displays the result of:
    `(reciprocal 'abc)`

    *Why does this statement result in an error?

24. Add statements to fn-1.pl460 that display the result of each of the following PL460 commands. Note that the `zero?` function requires a single argument.
    a. `(zero? 5)`
    b. `(zero? (- 2 5.5))`
    c. `(zero? 12/4)`
    d. `(zero? '0)`
    e. `(zero? 'abc)`
    f. `(zero? '(a b c))`
    g. `(zero? '())`
    h. `(zero? "(a b c) ")`

    *What conclusions have you drawn about the functionality of the `zero?` function?

25. The last 2 test cases in the previous step caused errors. Modify your reciprocal function so that it only returns the reciprocal if the parameter is a non-zero numeric value:

```
; This function will return the multiplicative reciprocal
; of a non-zero numeric input value
(define (reciprocal n)
```

```
        (if (and (number? n) (not (zero? n)))
                (/ 1 n)
        )
)
```

Reload the file and rerun the last 2 test cases. What are the results now? Why? Modify the function to display an error message if the reciprocal cannot be calculated:

```
; This function will return the multiplicative reciprocal
; of a non-zero numeric input value
(define (reciprocal n)
        (if (and (number? n) (not (zero? n)))
                (/ 1 n)
                "invalid parameter"
        )
)
```

*What are the results now? What can you surmise about the use of an "if statement" in a PL460 function?

26. Modify your square function in fn-2.pl460 so that it will return "invalid parameter" if the value passed to square is not numeric. Add 2 test cases to the main function in fn-2.pl460 to test your modifications.

27. By adding commands to fn-1.pl460 that are similar to the commands for `number?` and `zero?`, explore the functionality of the `list?` function.

    *What conclusions have you drawn about the functionality of the `list?` function?

28. Similarly, explore the functionality of the `zero?` function.

    *What conclusions have you drawn about the functionality of the `zero?` function?

29. Similarly, explore the functionality of the `null?` function.

    *What conclusions have you drawn about the functionality of the `null?` function?

PL460 Selection Control Structures

30. You modified your square and reciprocal functions to us an 'if' control structure to avoid errors by determining the validity of the parameter passed to the function. A few more 'if' examples follow. Enter the following PL460 code into a file called sc-1.pl460

```
(define (if_ex_1 p)
      (if (= p 0)
              'equal
              (if (< p 0)
                  'negative
                  'positive
              )
```

```
      )
)

(define (if_ex_2 p)
      (if (= p 0)
            'equal
            (if (< p 0)
                'negative
            )
      )
)

(define (main)
      (display "(if_ex_1 5) --> ")
      (display (if_ex_1 5))
      (newline)
      (display "(if_ex_1 -5) --> ")
      (display (if_ex_1 -5))
      (newline)
      (display "(if_ex_1 0) --> ")
      (display (if_ex_1 0))
      (newline)
)

(main)
```

Add statements to the main function of sc-1.pl460 to test `if_ex_2`.

*How do the results produced by `if_ex_2` differ from those produced by `if_ex_1`? Why do you think these differences occur?

31. PL460 also contains a multi branch selection control structure. Enter the following PL460 code into a file called sc-2.pl460

```
(define (cond_ex_1 p)
        (cond ((= p 0) 'equal)
              ((< p 0) 'negative)
              (else 'positive)
        )
)

(define (cond_ex_2 param)
        (cond ((= param 1) "The value is 1")
              ((= param 2) "The value is 2")
              ((> param 52) "The value is greater than 52")
        ((= (modulo param 5) 2) "The value ends in 2 or 7")
              (else "none of the above")
        )
)

(define (main)
```

```
  (display "(cond_ex_1 5) --> ")
  (display (cond_ex_1 5))
  (newline)
  (display "(cond_ex_1 -5) --> ")
  (display (cond_ex_1 -5))
  (newline)
  (display "(cond_ex_1 0) --> ")
  (display (cond_ex_1 0))
  (newline)
)

(main)
```

Add statements to the main function of sc-2.pl460 to test `cond_ex_2`.

32. Copy your square and reciprocal functions into sc-2.pl460. Add a third function, called cond_ex_3 to your sc_2.pl460 file. This function should have 2 parameters: choice and value. If choice is 1, the function should return value. If choice is even, the function should return the square of value. If choice is divisible by 3 and value is not zero, the function should return the reciprocal of the value. Otherwise the function should return 0.

    Add statements to the main function to test `cond_ex_3` and test your newly added function.

    *Can a `cond` function call be translated to a C++ switch statement? Why or why not?

PL460 Recursion

33. Create a file called rc-1.pl460 containing the following functions:

```
(define (list_copy1 ls)
        (if (list? ls)
                ls
                "list_copy1 requires a list argument"
        )
)

(define (list_copy2 ls)
        (if (list? ls)
                (if (null? ls)
                        '() ;;; base case
                        (cons (car ls) (list_copy2 (cdr ls)))
                )
                "list_copy2 requires a list argument"
        )
)
```

34. Add a main function and commands to test these functions.

35. `list_copy1` and `list_copy2` should function identically. However, `list_copy2` illustrates a recursive approach to parsing a list and accessing the elements of the list. In `list_copy2`

36. Make a copy of the function `list_copy2` called `list_copy3`. Modify the line:
```
(cons (car ls) (list_copy2 (cdr ls)))
```
to:
```
(cons (car ls) (list_copy3 (cddr ls)))
```

Add statements to test this function.

*How does this change the functionality of `list_copy3` differ from the function of `list_copy2`?

37. Create a file called rc-2.pl460 containing the following function:
```
(define (list_sum ls)
        (if (list? ls)
                (if (null? ls)
                        0 ;;; base case
                        (+ (car ls) (list_sum (cdr ls)))
                )
                "list_sum requires a list argument"
        )
)
```

Add a main function and commands to test this function. Determine the results of
```
(list_sum '())
(list_sum 55)
(list_sum '(a 2 b 5))
```

Modify the function to only add the numeric values in the list. If there are no numeric value in the list it should return 0.

38. Copy your *lastname*E3.txt file to the course dropbox.

For **Exercise 3 Part 2**, you are to write the following PL460 functions. To write these functions, you should only use the PL460 primitives introduced in the Tutorial. Many of the functions you write will use recursion – but, if you can determine a non-recursive solution, that is also fine. If need be, you can create helper functions to accomplish these tasks. Place the well documented implementations of these functions in a file called *lastname*E3.pl460

1. Copy the functions `square` and `reciprocal` to your file.

2. Copy the functions `list_copy1` and `list_copy3` to your file. Rename `list_copy1` as `list_copy`. Rename `list_copy3` as `odd_copy`.

3. Write and test a recursive PL460 function called `even_copy` that will create a copy of the elements in the even numbered positions in a list starting with the second element in the list. The formal argument should be a list.
```
(define (even_copy mylist)
      . . .
)
```

4. Copy the function `list_sum` to your file. Modify this function to return 0 it the argument is not a list.

5. Write and test a recursive PL460 function called `list_product` that will return the product of the numeric, non-zero values in the list. This function should use recursion to calculate the product. The formal argument should be a list. If the argument is not a list or if there are no numeric values in the list, this function should return 1.

```
(define (list_product mylist)
      . . .
)
```

6. The PL460 command `car` returns the first value in a list. Write and test a PL460 function called last that will return the last item in an existing list. The formal argument should be a list.

```
(define (last mylist)
      . . .
)
```

7. The PL460 command `cons` inserts an item as the first item in an existing list. Write and test a PL460 function called insert_last that will insert a value into a list as the last element of the list. The formal arguments should be the value to be inserted into the list and the list.

```
(define (insert_last myvalue mylist)
      . . .
)
```

8. The PL460 command `cdr` returns a list after the first value has been removed. Write and test a PL460 function called remove _first that will remove the first element of a list and return the resulting list. The formal argument should be a list.

```
(define (remove_first mylist)
      . . .
)
```

9. The PL460 command `cdr` returns a list after the first value has been removed. Write and test a PL460 function called remove _last that will remove the last element of a list and return the resulting list. The formal argument should be a list.

```
(define (remove_last mylist)
      . . .
)
```

10. Write and test a PL460 function called list_reverse that will reverse the elements of a list. The formal argument should be a list.

```
(define (list_reverse mylist)
      . . .
)
```

11. Write and test a PL460 function called square_root. This function should calculate the square root of a numeric value to 5 decimal places of precision. The formal argument should be a non-negative (0 or positive) numeric value.

```
(define (square_root n)
   . . .
)
```

12. Comment out the last function call in your *lastname*E3.pl460 file (probably to main) and copy your file to the course dropbox.