COP4020 Scheme Programming Assignment

In this programming assignment we design and implement a small imperative programming language “Micro-PL”. To execute **Mirco-PL** code we translate the code to Scheme and execute it in Scheme. This translator/compiler for Micro-PL will be written in **Scheme**. The objective of this assignment is to apply programming language design principles, implement program translation, and practice Scheme programming. For the Scheme programming we use the most common Scheme programming constructs discussed in class and some extras that will be explained in this document. **Acknowledgement:** This assignment is developed by *Dr. Robert Van Engelen* (FSU Tallahassee).

A New Programming Language Micro-PL

Our Micro-PL programming language is imperative. It adopts a procedural style programming with side effects (explicit state changes by variable assignment). Our Micro- PL programming language supports assignments to variables, function calls (including functions to perform arithmetic), programming statement sequences, if-then-else control flow, and loops. Micro-PL is dynamically typed, so any type of value can be assigned to a variable as long as the variable is declared in the outer scope.

The syntax of Micro-PL is relatively simple and is designed to simplify reading and parsing of Micro-PL programs by our translator/compiler. By keeping the syntax simple we do not need to implement advanced parsing techniques for our translator. Our translator looks for **keyword** commands to trigger a translation to Scheme. As a consequence, Micro-PL programs have a command-based appearance.

Each programming statement starts with a keyword command. The following commands are supported.

- The **set** command assigns a value to a variable. For example:

  set greeting = ”Hello, world!”

  assigns a string value to foo.

- The **begin** command is used to sequence a series of statements in a **begin**. . . **end** block. For example:

  begin
   set foo = 0
   set bar = 1
end

which is similar to all Algol-like programming languages (C/C++ and Java use **{ . . . }** instead). In our Micro-PL programming language we do not use semicolons to separate or terminate statements (taking a neutral position in the semicolon wars). Statements are simply chained together by white space (blanks, tabs, and newlines).

- Local variables are declared with the **declare** command that is followed by a **begin**. . . **end** block in which the variables are visible (are "in scope"). Thus, variables have a local scope, limited to the scope of the block. Each variable must be initialized. This requirement prevents any accidental use of uninitialized variables and values. (The use of uninitialized values is a typical programming error that can sometimes, but not always, be flagged by a compiler.) For example:

```
declare
  foo = 0
  bar = 1
begin
  set foo = bar
end
```

  where foo and bar have a local scope and are initialized to 0 and 1, respectively. The value of foo is reassigned in the block. The local scope means that the values of foo and bar are not accessible outside of the scope of the **declare**. . . **begin**. . . **end** block.

- The **if** command starts an if-then-else statement, where each **then** and **else** clause is a single statement (use **begin**. . . **end** when needed). For example:

```
declare
  foo = #f
  bar = -1
begin
  if foo then
    set bar = 0
  else
    set bar = 1
end
```

  which uses the Scheme value #f for *false* (and likewise we will use #t for *true*). This program assigns 1 to bar, since the condition is false and the else-clause is executed.

- The **for** command executes a Pascal-like loop with an integer-valued loop counter that ranges from a starting value to an ending value. The loop body is a single statement (use **begin**. . . **end** when needed). For example:

```
for i = 1 to 10
  set foo = i
```

Just as in many newer programming languages, such as Ada, the loop counter is locally declared and has a local scope in the loop body. This means that the loop counter is only visible in the loop body and its value cannot be accessed after the loop body.

- Finally, a statement can be an expression. An expression is either:
  - *A constant.* Any Scheme constant can be used, such as integer, float, #f and #t, string, and atom (a quoted name such as 'a and also '()).
  - *A variable.*
  - *A function call* of the form

    call *name* $arg_1 \ldots arg_n$ $

    where *name* is a function name or operator and arg_i, $i \geq 0$, are expressions.

For example:

```
if call = foo 0 $ then
  call display "foo is zero" $
else
  call display "foo is non-zero" $
```

where **display** is the Scheme function to display the argument on the terminal. All Scheme functions can be called this way. A sample of some useful built-in Scheme functions that can be called using this syntax are:

Function call	Description
call display *value* $	display value
call newline $	advance to new line
call list $value_1 \ldots value_n$ $	construct a list of *n* values
call cons *value list* $	construct a list node
call car *list* $	first value (head) of a list
call cdr *list* $	tail list
call cadr *list* $	second value of a list
call caddr *list* $	third value of a list
call length *list* $	length of a list
call + $value_1 \ldots value_n$ $	sum values (− subtracts)
call ∗ $value_1 \ldots value_n$ $	multiply values (/ divides)
call < $value_1 \ldots value_2$ $	compare values (<, <=, =, >, >=)
call eq? $value_1 \ldots value_2$ $	true when equal
call equal? $value_1 \ldots value_2$ $	true when structurally equal
call and $value_1 \ldots value_n$ $	logical and
call or $value_1 \ldots value_n$ $	logical or

The absence of parenthesis and commas to pass arguments is indicative of modern functional languages such as Haskell. Because our simple syntax does not need grouping with parenthesis, the use of the **call** needs a terminator $ to end the arguments. This is also the case for nested expressions, where we pass arguments that are function calls. This rule is consistently applied, though for arithmetic operations this can look a bit odd. For example, to compute $y = x^2 - x$ we write:

set y = **call** − **call** * x x $ x $

which is due to the prefix notation (also in Scheme we use prefix notation and write this as (- (* x x) x) which shows that **call** acts as the opening parenthesis and $ as the closing parenthesis in Micro-PL).

In the following sections we develop a translator to convert Micro-PL to an *s-expression*. The s-expression represents a valid Scheme program that can be executed.

Reading and Translating Micro-PL to Scheme

We need the following Scheme functions for file I/O:

- (set-current-input-port! (open-input-file *filename*)) opens *filename* for reading so that subsequent (read) calls scan the input file and return **token**s (atoms and constants).
- (read) returns the next "token" from the file, where a token is a Scheme atom or constant.
- eof-object? *token* is true of *token* is an end-of-file marker.

Here is an example use of these functions to read a file and construct a list of tokens:

```
; file2list takes a file name and returns a list of tokens
(define file2list (lambda
  (filename)
    (begin (set-current-input-port! (open-input-file filename))
           (read-list))))

; read-list returns a list of tokens read from the current input port
(define read-list
  (lambda ()
    (let ((token (read)))
      (cond ((eof-object? token) '())
            (else (cons token (read-list)))))))
```

If you are not familiar with the **begin**, **let**, **cond**, and the **if** special forms, then this is a good time to read up on Scheme using the lecture notes and textbook.

We observer the following:

- **file2list** returns the value of (read-list), but opens the file first and assigns it to the current input port before reading starts;
- the **let** in read-list assigns the current input token value (a Scheme atom or constant) to the token local variable (since let takes a list of variable-value pairs, we must use ((token read)) to enclose the pair in the list);
- if the token is an end-of-file marker then read-list returns an empty list **'()**. This is the base case of recursion where we have no (further) input and return an empty list;
- if the token is not an end-of-file marker then we construct a new list node with the token value and a tail that is recursively constructed by subsequent read-list calls.

The first function that we will write for our Micro-PL to s-expression translator is similar to file2list, except that it invokes statement to read and convert the token sequence of a Micro-PL program statement:

```
(define convert
  (lambda (filename)
    (begin (set-current-input-port! (open-input-file filename))
           (statement (read)))))
```

where statement takes the first token and continuous to read new tokens until the end of the statement and returns the s-expression converted from this token stream. Therefore, if we load our translator into Scheme and then enter the **convert** command from the Scheme command line (assuming our translator program is stored in file mytrans.scm):

```
1]=>        (load  "mytrans")
- - -
1]=>        (convert   "mytestprog")
```

we should see the s-expression representation of the Micro-PL program mytestprog.

How the statement function in convert should convert tokens to an s-expression is discussed next.

Micro-PL Program Statements

To translate a statement, we need to check the token for a matching **keyword**. If there is no match we assume the statement is an expression:

```
(define statement
  (lambda (token)
    (cond ((eq? token 'declare) (declare_statement))
          ((eq? token 'begin)   (begin_statement))
          ((eq? token 'for)     (for_statement))
          ((eq? token 'if)      (if_statement))
          ((eq? token 'set)     (set_statement))
          (else                 (expression token)))))
```

Each function is responsible for reading the entire sequence of tokens that represents that programming construct and to return the s-expression that is a valid Scheme program. In case a statement is an expression, we already read the first token and therefore pass it as an argument to the expression function for further processing.

We can implement the declare_statement function as follows:

```
(define declare_statement
  (lambda ()
    (let* ( (d (declarations))
            (s (statements)))
          (cons 'let* (cons d s)))))
```

where we used the Scheme let* to ensure that each variable-value pair is assigned in order. ***Scheme function arguments are not evaluated in order, nor are*** **let** ***pairs evaluated in order!*** *For example,* (list (read) (read) (read)) *does not ensure that the three values are stored in the list in the order they appear in the file!* In fact, most programming languages (C/C++ also) leave the evaluation of arguments unspecified, which allows the compiler to optimize the code more aggressively.

The **declarations** function should recursively convert the *variable = expression* pairs to a list of pairs (each pair is a list of two elements) until the **begin** keyword is reached. More formally we can write this as the inductive solution to the list *D* of declarations:

$$D = \begin{cases} '() & \text{if token} = \textbf{begin} \text{ or EOF} \\ (\text{cons } \textit{pair } D) & \text{otherwise} \end{cases}$$

where ***pair*** is a list of two elements produced by another function that takes the current token, considers it to be the variable name v, skips over the "=" in the input, and calls expression to convert the expression to an s-expression e to form the *pair* = (v e).

The **statements** function should recursively convert a sequence of statements into a list until the **end** keyword is reached. More formally, we can write this as the inductive solution to the list S of statements.

$$\boldsymbol{S} = \begin{cases} \text{'()} & \text{if token} = \textbf{end} \text{ or EOF} \\ (\text{cons } s\ S) & \text{otherwise} \end{cases}$$

where $\boldsymbol{s}$ is a statement converter for which we defined the statement function above (that takes the current token as argument).

Again, when necessary we must use **let*** in our implementation to ensure the proper ordering of function calls that perform I/O before we pass the results to other functions.

With the above, the aim is to convert

declare
  v_1 = e_1
  ⋮
  v_n = e_n
begin
  s_1
  ⋮
  s_m
end

into the s-expression

(let* ((v_1 e_1) . . . (v_n e_n)) s_1 . . . s_m)

where the e_i and s_j are converted from Micro-PL to s-expressions before we put them into the let* s-expression. All nested conversions should involve (**recursive**) function calls.

You should not use non-pure (destructive) functions such as set-car! *to modify data! You can simply call the conversion functions for expressions and statements and put the result in the s-expression under construction.*

Likewise, we can convert a **begin**. . . **end** block:

begin
  s_1
  .
  .
  s_m
end

into the s-expression

(begin s₁ . . . s$_m$)

We convert the conditional if-then-else

if e **then**
  s_1
else
  s_2

into the s-expression

(if e s_1 s_2)

where e, s_1, and s_2 are converted to s-expressions.

We convert the loop

for v = e_1 **to** e_2
  s

into the s-expression

(do ((v e_1 (+ v 1))) ((> v e_2) v) s)

where e_1, e_2, and s are converted to s-expressions.

We convert the assignment statement

set v = e

into the s-expression

(set! v e**)**

where e is converted to s-expression. Since this one is simple to implement. Here is a possible implementation:

```
(define set_statement
   (lambda ()
      (let ( (token (read)))
         (list 'set! token (expression (read-after '=))))))
```

Note the use of `let` to ensure we read the token before we call `read-after` and `expression`. The `read-after` function skips a token and reads the next:

```
(define read-after
   (lambda (token)
      (if (eq? (read) token)
         (read)
         (display "Syntax error"))))
```

so that an error message is displayed when there is no match.

Micro-PL Expressions and Function Calls

To convert Micro-PL expressions to s-expressions we only need to capture the **call** construct and convert it. Otherwise, we just return the token (which is an atom or constant) as the s-expression:

```
(define expression
   (lambda (token)
      (if (eq? token 'call)
         (call_expression)
         token)))
```

so that call_expression returns a list with the function name and s-expressions for the arguments. That is, we convert

call $f\ e_1 \ldots e_n$ $

into the s-expression

($f\ e_1 \ldots e_n$)

where the e_i are converted to s-expressions.

Converting a sequence of expressions to a list of s-expressions is similar to converting a sequence of statements to s-expressions, so it seems natural to exploit the same implementation approach.

Executing Micro-PL Programs

To convert the Micro-PL program and execute the resulting s-expression, we use eval from the Scheme command line:

```
(eval (convert ”mytestprog”) (the-environment))
```

If there is an error in the s-expression (meaning it is not a valid Scheme program) or if variables and functions are used that have not been declared, then an error message will be generated. Otherwise, the program executes and its result value is printed.

Code Examples

```
begin
  call display ”Hello, world!” $
  call newline $
end
```

Converted to s-expression:

```
(begin (display "Hello, world!")  (newline))
```

```
declare
   greeting = "Hello, world!"
begin
   call display greeting $
   call newline $
end
```

Converted to s-expression:

```
(let* ((greeting "Hello, world!")) (display greeting) = (newline))
```

```
declare
   x = 10
begin
   call display call * 2 x $ $
   call newline $
end
```

Converted to s-expression:

```
(let* ((x   10))  (display   (*  2  x)) (newline))
```

```
declare
   num = 10
   fac = 1
begin
   for i = 1 to num
      set fac = call * i fac $
   call display fac $
   call newline $
end
```

Converted to s-expression:

```
(let* ( (num 10) (fac 1)) (do ( (i 1 (+ i 1) ) )  ( (> i  num)  i )  (set!  fac (* i fac) ) )  (display fac)
(newline) )
```

```
if call eq? call read $ ’yes $ then
   call display ”OK” $
else
   call display ”Why not?” $
```

Converted to s-expression:

```
(if (eq? (read) (quote yes)) (display "OK") (display "Why
not?"))
```

```
declare
   weekdays = ’(mon tue wed thu fri)
   days = call append weekdays ’(sat sun) $
   party = ’()
begin
   set party = call member ’fri days $
   call display ”Going out on ” $
   for i = 0 to call –                    call length party $ 1 $
      declare
         day = call list-ref party i $
      begin
         call display day $
         call display ” ” $
      end
   call newline $
end
```

Converted to s-expression:

```
(let* ((weekdays (quote (mon tue wed thu fri)))  (days (append
weekdays (quote (sat   sun)))) (party  (quote    ()))) (set! party
(member   (quote fri)  days)) (display "Going out on") (do  ((I 0 (+ I
1 ))) ((> I (- (length  party) 1 )) i)  (let* ((day (list-ref party
i))) (display  day)  (display  "  "))) (newline))
```