PPL222

Assignment 2

Responsible Lecturer: Meni Adler
Responsible TA: Adham Jabarin

Submission Date: 1/5/2022

General Instructions

Submit your answers to the theoretical questions in a pdf file called id1_id2.pdf and your code for programming questions inside the provided q2.l3, and L31-ast.ts, q3.ts, q4.ts files of the *src* folder. ZIP those files together (including the pdf file, and only those files) into a file called *id1_id2.zip*. Make sure that your code abides by the Design By Contract methodology.
Do not send assignment related questions by e-mail, use the forum instead. For any administrative issues (milu'im/extensions/etc) please open a request ticket in the Student Requests system.

You are provided with the templates *ex2.zip*.
Unpack the template files inside a folder. From the command line in that folder, invoke `npm install`, and work on the files in that directory, preferably working in the Visual Studio Code IDE (refer to the Useful Links). In order to run the tests, run `npm test` from the command line.

<u>Important</u>: Do not add any extra libraries and do not change the provided package.json and tsconfig.json configuration files. **The graders will use the exact provided files**. If you find any missing necessary libraries, please let us know.

Question 1: Theoretical Questions [30 points]

1.1 Is a function-body with multiple expressions required in a pure functional programming? In which type of languages is it useful? [3 points]

Q1.2

a. Why are special forms required in programming languages? Why can't we simply define them as primitive operators? Give an example [3 points]
b. Can the logical operation 'or' be defined as a primitive operator, or must it be defined as a special form? Refer in your answer to the option of shortcut semantics. [4 points]

Q1.3 What is a syntactic abbreviation? Give two examples [4 points]

Q1.4

a. What is the value of the following L3 program? Explain. [2 points]

```
(define x 1)
(let ((x 5)
      (y (* x 3)))
  y)
```

b. Read about let* here.
What is the value of the following program? Explain. [2 points]

```
(define x 1)
(let* ((x 5)
       (y (* x 3)))
  y)
```

c. Annotate lexical addresses in the given expression [6 points]

```
(define x 2)
(define y 5)

(let
  ((x 1)
   (f (lambda (z) (+ x y z))))
  (f x))

(let*
  ((x 1)
   (f (lambda (z) (+ x y z))))
 (f x))
```

d. Define the let* expression in section c above as an equivalent let expression [3 points]
e. Define the let* expression in section c above as an equivalent application expression (with no let) [3 points]

Answers should be submitted in file id1_id2.pdf

Question 2: Programming in L3 [40 points]

Q2.1 Let us define the Result-OK-Error mechanism, met in assignment 1, in L3:

a. Write L3 procedures for supporting *result*, *ok* and *error* structures:

`make-ok` - gets a value and encapsulates it as an *ok* structure of type result
`make-error` - gets an error string and encapsulates it as an 'error' structure of type *result*
`ok?` - type predicate for *ok*
`error?` - type predicate for *error*
`result?` - type predicate for *result*
`result->val` - returns the encapsulated value of a given *result*: value for *ok result*, and the error string for *error result*

```
(define r1 (make-ok 3))
(ok? r1)
→ #t
(error? r1)
→ #f
(result? r1)
→ #t
(result->val r1)
→ 3

(define r2 (make-error "Error: key not found"))
(ok? r2)
→ #f
(error? r2)
→ #t
(result? r2)
→ #t
(result->val r2)
"Error: key not found"

(define r3 'ok)
(ok? r3)
→ #f
(error? r3)
→ #f
(result? r3)
→ #f

(define r3 'error)
(ok? r3)
→ #f
(error? r3)
→ #f
```

```
(result? r3)
→ #f
```

b. Implement the *bind* procedure, which gets a function from *non-result* parameter to *result* and returns this function from *result* to *result*.
For example: (bind `(lambda (x) (make-ok (* x x)))` should return a function which gets a `result` (`ok` with `x`, or `error`) and returns `(make-ok (* x x))` or `error` accordingly.
`bind` can simplify the composition of functions:

```
;; compose two given functions
(define compose
  (lambda (f g)
    (lambda (x)
       (f (g x)))))

;; Compose a list of functions
;; (pipe (list f1 f2 ... fn)) with fi: Ti->T(i+1)
;; returns the composition T1->T(n+1) \
;; fn(....(f1(x)))
;; (Functions are executed in the order in which they appear in
;; parameter)

(define pipe
  (lambda (fs)
    (if (empty? fs)
        (lambda (x) x)
        (compose (pipe (cdr fs)) (car fs)))))

(define square (lambda (x) (make-ok (* x x))))
(define inverse (lambda (x)
 (if (= x 0)
      (make-error "div by 0")
      (make-ok (/ 1 x)))))

(define inverse-square-inverse
 (pipe (list inverse (bind square) (bind inverse))))

(result->val (inverse-square-inverse 2))
→ 4

(result->val (inverse-square-inverse 0))
→ "div by 0"
```

Q2.2

Let us define a *dictionary* which maps unique keys to values.

`make-dict` - returns a new empty dictionary
`dict?` - type predicate for dictionaries
`put` - gets a dictionary, a key and a value, and returns a *result* of a dictionary with the addition of the given key-value. In case the given key already exists in the given dict, the returned dict should contain the new value for this key.
`get` - gets a dictionary and a key, and returns the value in dict assigned to the given key as an *ok result*. In case the given key is not defined in dict, an *error result* should be returned.

```
(define dict (make-dict))

(dict? dict)
→ true

(dict? '(2  4))
→ false

(result->val (get (result->val (put dict 3 4)) 3))
→ 4

(result->val (get (result->val (put (result->val (put dict 3 4)) 3
5)) 3))
→ 5

(result->val (get (result->val (put dict 3 4)) 4))
→ "Key not found"

(result->val (put '(1 2) 3 4))
→ "Error: not a dictionary"

(result->val (get '(1 2) 1))
→ "Error: not a dictionary"
```

Q2.3

a. Write an L3 procedure *map-dict*, which gets a dictionary and an unary function, applies the function of the values in the dictionary, and returns a result of a new dictionary with the resulting values.

   For example:

```
(result->val (get (result->val (map-dict (result->val (put
(result->val (put (make-dict) 1 #t)) 2 #f)) (lambda (x) (not x ))))
1))
→#f

(result->val (get (result->val (map-dict (result->val (put
(result->val (put (make-dict) 1 #t)) 2 #f)) (lambda (x) (not x ))))
2))
→#t
```

b. Write an L3 procedure *filter-dict*, which gets a dictionary and a predicate that takes (key value) as arguments, and returns a *result* of a new dictionary that contains only the key-values that satisfy the predicate.

   For example, let *even?* be the predicate that returns true when the number is even, then:

```
(define even-key-odd-value? (lambda (k v) (and (even? k) (odd? v))))

(result->val (get (result->val (filter-dict (result->val (put
(result->val (put (make-dict) 2 3)) 3 4)) even-key-odd-value?)) 2))
→ 3

(result->val (get (result->val (filter-dict (result->val (put
(result->val (put (make-dict) 2 3)) 3 4)) even-key-odd-value?)) 3))
→ "Key not found"
```

You may add auxiliary procedures to all questions.

<u>The code **(without comments)** should be submitted in file src/q2.l3</u>

Don't forget to write a contract for each of the above procedures.

```
; Signature:
; Type:
; Purpose:
; Pre-conditions:
; Tests:
```

<u>Write the contracts in file id1_id2.pdf.</u>

You can test your code with test/q2-tests.ts

Question 3: Syntactic Parsing & Transformations [15 points]

Let us define the L31 as L3 with the addition of a new special form *let** (see Q1.4c)

```
<program> ::= (L31 <exp>+)             / Program(exps:List(exp))
<exp> ::= <define> | <cexp>            / DefExp | CExp
<define> ::= ( define <var> <cexp> )   / DefExp(var:VarDecl,
val:CExp)
<var> ::= <identifier>                 / VarRef(var:string)
<cexp> ::= <number>                    / NumExp(val:number)
        |  <boolean>                   / BoolExp(val:boolean)
        |  <string>                    / StrExp(val:string)
        |  ( lambda ( <var>* ) <cexp>+ ) / ProcExp(args:VarDecl[],
                                        /         body:CExp[]))
        |  ( if <cexp> <cexp> <cexp> ) / IfExp(test: CExp,
                                              then: CExp,
                                              alt: CExp)
        |  ( let ( <binding>* ) <cexp>+ ) /
LetExp(bindings:Binding[],
                                                  body:CExp[]))
        |  ( let* ( <binding>* ) <cexp>+ ) /
                       LetPlusExp(bindings:Binding[],body:CExp[]))
        |  ( quote <sexp> )               / LitExp(val:SExp)
        |  ( <cexp> <cexp>* )             / AppExp(operator:CExp,
        |                                          operands:CExp[]))
<binding>  ::= ( <var> <cexp> )           / Binding(var:VarDecl,
                                                   val:Cexp)
<prim-op>  ::= + | - | * | / | < | > | = | not |  eq? | string=?
                | cons | car | cdr | list | pair? | list? | number?
                | boolean? | symbol? | string?
<num-exp>  ::= a number token
<bool-exp> ::= #t | #f
<str-exp>  ::= "tokens*"
<var-ref>  ::= an identifier token
<var-decl> ::= an identifier token
<sexp>     ::= symbol | number | bool | string | ( <sexp>* )
```

a. Add the new *let** special form to the parser of L31 (the file ‘src/L31-ast.ts’):
b. Implement the procedure *L31ToL3* (at file src/q3.ts), which gets an L31 AST and returns an equivalent L3 AST.

The code should be submitted in files src/q3.ts, src/L31-ast.ts
You can test your code with test/q3-tests.ts

Question 4: Code translation [15 points]

Let us define L30 as L3 excluding pairs and lists.
Write the procedure *l30ToJS* which transforms a given L30 program to a JavaScript program.

The procedure gets an L30 AST and returns a string of the equivalent JavaScript program.

For example:

```
(+ 3 5 7) ⇒  (3 + 5 + 7)

(= 3 (+ 1 2)) ⇒ (3 === (1 + 2))

(if (> x 3) 4 5) ⇒ ((x > 3) ? 4 : 5)

(lambda (x y) (* x y)) ⇒ ((x,y) => (x * y))

((lambda (x y) (* x y)) 3 4) ⇒  ((x,y) ⇒ (x * y))(3,4)

(define pi 3.14) ⇒ const pi = 3.14

(define f (lambda (x y) (* x y))) ⇒ const f = ((x,y) ⇒ (x * y))

(f 3 4) ⇒ f(3,4)

boolean? ⇒ ((x) => (typeof(x) === boolean))

symbol? ⇒ ((x) => (typeof (x) === symbol))

'red ⇒ Symbol.for("red")

"abc" ⇒ "abc"

(let ((a 1) (b 2)) (+ a b)) ⇒ ((a,b) => (a + b))(1,2)

(L3
(define b (> 3 4))
(define x 5)
(define f (lambda (y) (+ x y)))
(define g (lambda (y) (* x y)))
```

```
(if (not b) (f 3) (g 4))
(if (= a b) (f 3) (g 4))
(if (> a b) (f 3) (g 4))
((lambda (x) (* x x)) 7)
)
```

⇒

```
const b = (3 > 4)
const x = 5
const f = ((y) => (x + y))
const g = ((y) => (x * y))
((not b) ? f(3) : g(4))
((a === b) ? f(3) : g(4))
((a > b) ? f(3) : g(4))
((x) => (x * x))(7)
```

To make things simpler, you can assume that the body of the lambda and let expressions contains <u>one</u> expression.

You can use functions given in class, in particular `rewriteLet`.

<u>Notes</u>:

- The primitive operators of L30 are: `+, -, *, /, <, >, =, number?, boolean?, eq?, and, or, not, symbol?, string?, string=?`
  You can see their exact semantics in the applyPrimitive function in the interpreter of L3.
- L30 contains expressions for symbols (`'red`) and strings (`"abc"`).

<u>Hint</u>: Take a look at the unparse procedure.

<u>The code should be submitted in file src/q4.ts</u>
You can test your code with test/q4-tests.ts

Good Luck!