PPL232

Assignment 2

Responsible Lecturer: Meni Adler
Responsible TA: Eden Dayan

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, L31-ast.ts, q3.ts, q4.ts files in the correct places. 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.

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. In order to run the tests, run `npm test` from the command line.

Important: do not add any extra libraries in the supplied template files, otherwise, we will fail to compile and you will receive a grade of zero. If you find that we forgot to import necessary libraries, let us know.

Question 1: Theoretical Questions [30 points]

Q1.1 Why are special forms required in programming languages? Why can't we simply define them as primitive operators? Give an example [4 points]

Q1.2 Let us define the L0 language as L1 excluding the special form 'define'. Is there a program in L1 which cannot be transformed to an equivalent program in L0? Explain or give a contradictory example [4 points]

Q1.3 Let us define the L20 language as L2 excluding the special form 'define'. Is there a program in L2 which cannot be transformed to an equivalent program in L20? Explain or give a contradictory example [4 points]

Q1.4 In practical session 5, we deal with two representations of primitive operations: *PrimOp* and *Closure*. List an advantage for each of the two methods [2 points].

Q1.5 For the following high-order functions in L3, which gets a function and a list as parameters, indicate (and explain) whether the order of the procedure application on the list items should be sequential or can be applied in parallel:

- map
- reduce
- filter
- all (returns #t is the application of the given boolean function on each of the given list items returns #t)
- compose (compose a given procedure with a given list of procedures)

[10 points]

Q1.6 What is *lexical address*? Give an example which demonstrates this concept [2 points]

Q1.7 Let us define L31 as the L3 language with the addition of 'cond' special form (as described in practical session 4)

Note: The cond expression **must include at least one cond-clause**, and **must include an else-clause** (in contrast to the decription in practical session 4)

Complete the concrete and abstract syntax of L31:

```
<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[]))
        |  ______________________________ / ________________________

        |  ( 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>* )
```

[4 points]

Answers should be submitted in file id1_id2.pdf

Question 2: Programing in L3 [35 points]

Q2.1- In this part you can also use the procedures: empty?, length.

a. Implement in L3 the following procedures:
`take` - gets a $list$ and a number pos and returns a new list whose elements are the first pos elements of the $list$. If the $list$ is shorter then pos- return the $list$.
`take-map` - gets a $list$, a function $func$ and a number pos and returns a new list whose elements are the first pos elements mapped by $func$. If the $list$ is shorter then pos- return the mapped $list$.
`take-filter` - gets a $list$, a predicate $pred$ and a number pos and returns a new list whose elements are the first pos elements of the $list$ that satisfy $pred$. If the number of elements satisfy the predicate is less then pos- return the filtered $list$.
Examples:
```
(take (list 1 2 3) 2) → '(1 2)
(take '() 2) → '()

(take-map (list 1 2 3) (lambda (x) (* x x)) 2) → '(1 4)
(take-map (list 1 2 3) (lambda (x) (* x x)) 4) → '(1 4 9)

(take-filter (list 1 2 3 4) (lambda (x) (> x 1)) 2) → '(2 3)
(take-filter (list 1 2 3) (lambda (x) (> x 3)) 2) → '()
```

b. Implement in L3 the following procedures:
`sub-size` - gets a $list$ and a number $size$ and returns a new list of all the sublists of $list$ of length $size$.
`sub-size-map` - gets a $list$, a function $func$ and a number $size$ and returns a new list of all the sublists of $list$ of length $size$ that all their elements are mapped by $func$.

In both procedures, you can assume that $size \leq (length\ list)$. if the list is not empty you can assume $1 \leq size$.

Examples:

```
(sub-size '() 0) → '(())
(sub-size (list 1 2 3) 3) → '((1 2 3))
(sub-size (list 1 2 3) 2) → '((1 2) (2 3))
(sub-size (list 1 2 3) 1) → '((1) (2) (3))

(sub-size-map '() (lambda (x) (+ x 1)) 0) → '(())
(sub-size-map (list 1 2 3) (lambda (x) (+ x 1)) 3) → '((2 3 4))
(sub-size-map (list 1 2 3) (lambda (x) (+ x 1)) 2) → '((2 3) (3 4))
(sub-size-map (list 1 2 3) (lambda (x) (+ x 1)) 1) → '((2) (3) (4))
```

Q2.2

We can represent a binary tree in L3 using a list as follows: the first element in every nesting level represents the root of the sub-tree.
A leaf is represented by an atom (not a list/pair).
A missing child is represented as the empty list.
An empty tree is represented by an empty list.
For example:

```
     1
    / \
  #t    2
 / \
3   4
```

Is represented as:
`'(1 (#t 3 4) 2)`

a. Implement in L3 the following procedures:
   `root`- gets a list representing a tree and returns the value of the root.
   `left`- gets a list representing a tree and returns the subtree of the left son, or an empty list if there is no left son.
   `right`- gets a list representing a tree and returns the subtree of the right son, or an empty list if there is no right son.
   In all 3 procedures above, you can assume the tree is valid and contains the requested nodes.

   In the following two sections, to enforce the principle of encapsulation, your implementation of `count-node` and `mirror-tree` should only use the functions `root`, `left` and `right` when accessing the `tree` parameter.

b. Implement in L3 the procedure `count-node` which given a list representing a $tree$ and an atomic val, returns the number of nodes whose value is equal to val.
   For example:
   `(count-node '(1 (#t 3 #t) 2) #t) → 2`
   `(count-node '(1 (#t 3 #t) 2) 4) → 0`

c. Implement in L3 the procedure `mirror-tree` which given a list representing a $tree$, returns the mirrored tree.
   For example, the mirrored tree for the example above will be:

```
     1
    / \
   2   #t
       / \
      4   3
```

   Is represented as:
   `'(1 2 (#t 4 3))`

Q2.3

a. Implement in L3 the following procedures to support result, ok and error structures:
   `make-ok` - gets a $value$ and returns an ok structure for the $value$ of type result.
   `make-error` - gets an error $message$ and returns an error structure for the $message$ of type result.
   `ok?` - type predicate for ok.
   `error?` - type predicate for $error$.
   `result?` - type predicate for $result$.
   `result->val` - gets a $result$ structure and returns the value it represents, or the error message for error. If the given result is not a result, return an error structure with the message “Error: not a result”

   Examples:
```
(define ok (make-ok 1))
(ok? ok) → #t
(error? ok) → #f
(result? ok) → #t
(result->val ok) → 1

(define error (make-error “some error message”))
(error? error) → #t
(ok? error) → #f
(result? error) → #t
(result->val error) → “some error message”

(define not-ok ‘ok)
(ok? not-ok) → #f
(error? not-ok) → #f
(result? not-ok) → #f
(result->val (result->val not-ok)) → “Error: not a result”
```

b. Implement in L3 the procedure `bind` which given a function $func$ from a non-result to result, returns a new function which given a result, returns the activation of $func$ on its value or an error structure accordingly.
   For example:
```
(define inc-result (bind (lambda (x) (make-ok (+ x 1)))))

(define ok (make-ok 1))
(result->val (inc-result ok)) → 2

(define error (make-error “some error message”))
(result->val (inc-result error)) → “some error message”
```

You may add auxiliary procedures to all questions.

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

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

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

Write the contracts in file id1_id2.pdf.

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

Question 3: Syntactic Transformations [20 points]

a. Implement the parser of L31, as defied in **Q1.7** above.
b. Write the procedure *l31ToL3* which transforms a given L31 program to a L3 program.

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]

Write the procedure *l2ToPython* which transforms a given L2 program to a Python program.

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

For example:

```
(+ 3 5) ⇒ (3 + 5)

(if (> x 3) 4 5) ⇒ (4 if (x > 3) else 5)

(lambda (x y) (* x y)) ⇒ (lambda x, y : (x * y))

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

(define pi 3.14) ⇒ pi = 3.14

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

(f 3 4) ⇒ f(3,4)
```

```
boolean? ⇒ (lambda x : (type(x) == bool))

(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)
)
⇒
b = (3 > 4)
x = 5
f = (lambda y : (x + y))
g = (lambda y : (x * y))
(f(3) if (not b) else g(4))
(f(3) if (a == b) else g(4))
(f(3) if (a > b) else g(4))
(lambda x : (x * x))(7)
```

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

<u>Note</u>: The primitive operators of L2 are: +, -, *, /, <, >, =, number?, boolean?, eq?, and, or, not

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

Good Luck! 😊