PPL182

Assignment 2

Responsible Lecturer: Meni Adler
Responsible TA: Morad Muslimany

Due date: 01/05/2018 @ 23:59

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 *q4.rkt,L3-ast.ts,L3-rewrite.ts,q6.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 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 refer to `iliaevd@post.bgu.ac.il`. Theoretical questions are marked with a **T** , and programming questions are marked with a **P**. Important: do not add any extra imports/includes/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/functions, let us know.

Please do not print this assignment unless you have to. If you do print it, please recycle the papers after you are done.
Have fun and may the λs always be by your side.

Question 1 - General Terminology **(20 points)**

1.1 - T (1 point) What is a special form?

1.2 - T (1 point) What is an atomic expression?

1.3 - T (1 point) What is a compound expression?

1.4 - T (1 point) What is a primitive expression?

1.5 - T (4 points) For each one of the following L3 expressions, decide whether it is atomic and/or primitive and/or compound or none

- **1.5.1** (1 point) `+`
- **1.5.2** (1 point) `5`
- **1.5.3** (1 point) `x`
- **1.5.4** (1 point) `((lambda (x) x) 5)`

1.6 - T (1 point) Complete the sentence: multiple expressions in the body of a procedure expression (lambda form) is useful mainly when those expressions have a _____

1.7 - T (1 point) Complete the sentence: we call an expression a "syntactic abbreviation" of another expression when _____

1.8 - T (3 points) What is the syntactic abbreviation of the following expression? (you do not have to evaluate the expression - simply write the form it translates to)

```
(let ((x (lambda (x) (+ x 1)))
      (y ((lambda (y) (- y 22)) 23))
      (z 6))
  (* (x z) y))
```

1.9 - T (3 points) Recall our definition of *shortcut semantics* (Practical Session 1 reading material). Do `and` expressions in `Racket` support *shortcut semantics*? Prove your answer (answers without proof will not be accepted).

1.10 - T (4 points) Recall our definition of *functional equivalence* (Practical Session 1 reading material). Given the following two procedures `foo` and `goo`, and assuming that the procedure `display` never fails and works as defined:

```
(define foo (lambda (x) (+ x 1)))
(define goo (lambda (x)
              (display 'hi-there)
              (+ x 1)))
```

- **1.10.1** (2 points) Are `foo` and `goo` *functionally equivalent* according to our definition in class? Explain (answers without an explanation will not be accepted).
- **1.10.2** (2 points) Are `foo` and `goo` *functionally equivalent* when considering side-effects as well as an addition to the definition we saw in class? Explain (answers without an explanation will not be accepted).

Question 2 - Rules of Evaluation (10 points)

This question refers to the language ***L3*** as seen in the lectures.
Show the steps of evaluating the following expressions according to the evaluation rules for ***L3*** as seen in class. At each step, describe what expression is being evaluated, what type of evaluation is it (atomic/compound special form/compound non-special form), and what is the resulting value. If the return value is received from the global environment, state that fact next to the return value. You are provided with an example and its' solution on the next page.

2.1 - T (2 points)

```
(define x 12)
((lambda (x) (+ x (+ (/ x 2) x))) x)
```

2.2 - T (1 point)

```
(define last
  (lambda (l)
    (if (empty? (cdr l))
        (car l)
        (last (cdr l)))))
```

2.3 - T (7 points)

```
(define last
  (lambda (l)
    (if (empty? (cdr l))
        (car l)
        (last (cdr l)))))
(last '(1 2))
```

Example

```
(define x 5)
(if (> x 0)
    x
    (- 0 x))
```

Example Solution

```
evaluate((define x 5)) [compound special form]
    evaluate(5) [atomic]
    return value: 5
    add the binding <<x>,5> to the GE
return value: void
evaluate((if (> x 0)
             x
             (- 0 x))) [compound special form]
        evaluate((> x 0)) [compound non-special form]
            evaluate(>) [atomic]
            return value: #<procedure:>>
            evaluate(x) [atomic]
            return value: 5 (GE)
            evaluate(0) [atomic]
            return value: 0
        return value: #t
        evaluate(x) [atomic]
        return value: 5 (GE)
return value: 5
```

Question 3 - Scopes **(10 points)**

For each of the following pieces of code, and for each binding instance (variable declaration), state its scope and in which lines it appears bound. Furthermore, state which variables are free. If a variable is bound only in some lambda body, state in parenthesis the line number at which the lambda expression begins.

Example

```
(define even? (lambda (n)                          ;1
                (eq? (modulo n 2) 0)))             ;2
(even? 5)                                          ;3
```

a possible answer can be:

Binding Instance	Appears first at line	Scope	Line #s of bound occurrences
even?	1	Universal Scope	3
n	1	Lambda Body (1)	2

3.1 - T (5 points)

```
(define fib (lambda (n)                                          ;1
              (cond ((= n 0) 0)                                  ;2
                    ((= n 1) 1)                                  ;3
                    (else (+ (fib (- n 1)) (fib (- n 2)))))))    ;4
(define y 5)                                                     ;5
(fib (+ y y))                                                    ;6
```

3.2 - T (5 points)

```
(define triple (lambda (x)                        ;1
                 (lambda (y)                      ;2
                   (lambda (z) (+ x y z)))))      ;3
(((triple 5) 6) 7)                                ;4
```

Question 4 - Fun with Scheme (20 points)

Implement the following procedures inside the file *q4.rkt* in the correct places. Add the contract for each method. You may **not** add user-defined helper procedures. You may, however, use built-in (primitive) procedures. You may find the following built-in procedures useful: `length`, `equal?`, `append`, `foldl`, `map`, `boolean?`, `not`, `take`, `last`. Remember not to import any modules.

4.1 - P (3 points) Implement the procedure `shift-left` that takes a list as an argument and evaluates the list that is its' shift-left by one place. For example:

```
> (shift-left '())
'()
> (shift-left '(5))
'(5)
> (shift-left '(1 2))
'(2 1)
> (shift-left '(1 2 3 4))
'(2 3 4 1)
> (shift-left '(1 (2 3) 4))
'((2 3) 4 1)
```

4.2 - P (2 points) Implement the procedure `shift-k-left` that takes a list and a number $k \geq 0$ and evaluates the list that is the given list's shift-left k times. You may use the procedure `shift-left` from the previous question. For example:

```
> (shift-k-left '() 5)
'()
> (shift-k-left '(1) 100)
'(1)
> (shift-k-left '(1 2 3) 0)
'(1 2 3)
> (shift-k-left '(1 2 3) 1)
'(2 3 1)
> (shift-k-left '(1 2 3) 2)
'(3 1 2)
> (shift-k-left '(1 2 3) 3)
'(1 2 3)
```

4.3 - P (5 points) Implement the procedure `shift-right` that takes a list and evaluates the list that is the given list's shift-right one time. For example:

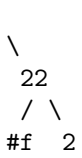
```
> (shift-right '())
'()
> (shift-right '(1))
'(1)
```

```
> (shift-right '(1 2 3))
'(3 1 2)
> (shift-right '(3 1 2))
'(2 3 1)
```

4.4 - P (2 points) Implement the procedure `combine` that takes two lists and combines them in an alternating manner starting from the first list. If one of the lists is empty, then the result of `combine` is the other list. For example:

```
> (combine '() '())
'()
> (combine '(1 2 3) '())
'(1 2 3)
> (combine '() '(4 5 6))
'(4 5 6)
> (combine '(1 3) '(2 4))
'(1 2 3 4)
> (combine '(1 3) '(2 4 5 6))
'(1 2 3 4 5 6)
> (combine '(1 2 3 4) '(5 6))
'(1 5 2 6 3 4)
```

The following is with no relation to previous parts:
We can represent a (possibly empty or non-complete) tree in `Scheme` as follows: the first element in every nesting level represents the root of the sub-tree. The rest of the elements are the children (each of them is a tree, of course). A leaf is represented by a list with only one element (the leaf value). A completely empty tree is represented by the empty list. For example:



is represented as:

```
'(5 (#t (5) (4)) (6 (3)) (22 (#f) (2)))
```

4.5 - P (4 points) Implement the procedure `sum-tree` that receives a tree whose nodes' data values are all numbers ≥ 0 and returns the sum of numbers present in all tree nodes. For example:

```
> (sum-tree '())
0
> (sum-tree '(5))
5
> (sum-tree '(5 (1 (2) (3))))
```

```
11
> (sum-tree '(5 (1 (2) (3) (6)) (7)))
24
> (sum-tree '(5 (1 (2) (3 (12) (12)) (6)) (7)))
48
```

4.6 - P (4 points) Implement the procedure `inverse-tree` that receives a tree whose nodes data values are numbers and booleans and returns the equivalent tree whose nodes satisfy the following:

- If the equivalent node of the original tree is a number, then the resulting tree's node is $-1\cdot$ that node value
- If the equivalent node of the original tree is a boolean, then the resulting tree's node is the logical *not* of that node value

For example:

```
> (inverse-tree '())
'()
> (inverse-tree '(5))
'(-5)
> (inverse-tree '(0))
'(0)
> (inverse-tree '(#f))
'(#t)
> (inverse-tree '(#t))
'(#f)
> (inverse-tree '(-5 (1 (-2) (3) (#f)) (#t)))
'(5 (-1 (2) (-3) (#t)) (#f))
```

Question 5 - Syntactic Abbreviation (20 points)

This question refers to the langauge `L3`.
At times, it is convenient to gradually define variables. Suppose we would like to say that x equals 5 and that y equals $3x$, then one would expect that y equals 15, but if we write the following code, y would be equal to 3 instead:

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

To achieve the result we want, we can nest `let` expressions as follows:

```
> (let ((x 5))
    (let ((y (* x 3)))
      y))
15
```

We can abbreviate the nesting of the `let` expression as `let*`:

```
> (let* ((x 5)
         (y (* x 3)))
    y)
15
```

You are asked to support `let*` expressions as a syntactic abbreviation. Meaning, `let*` expressions should be translated into nested `let` expressions. You should extend the parser shown in class (`L3-ast.ts`) and the rewrite methods in `L3-rewrite.ts`. You must:

5.1 - T (5 points) Extend the BNF of the language to support `let*` expressions – both the concrete BNF and abstract BNF (note that this part is theoretical – the solution should be in the PDF file.)

5.2 - P (15 points) Implement the `LetStarExp` AST datatype functions and add support for `LetStarExp` in the parser (in the file `L3-ast.ts`), implement the method `rewriteLetStar` (in the file `L3-rewrite.ts`), and implement the method `rewriteAllLetStar` (in the file `L3-rewrite.ts`).
The function `rewriteLetStar` should rewrite a single `let*` expression, and the function `rewriteAllLetStar` should rewrite all occurrences of `let*` expressions wherever the language allows them to appear. Note that all these procedures should follow the steps of adding support of new expressions as we have seen in practical session 4. The `tag` field inside the `LetStarExp` datatype **must** be `"LetStarExp"`, as can be seen from the examples below.
Warning: Do not alter the functions' types.

```
console.log(JSON.stringify(parseL3("(let* ((x 3) (y x)) x y)"), null, 4));
->
{
    "tag": "LetStarExp",
    "bindings": [
        {
            "tag": "Binding",
            "var": {
                "tag": "VarDecl",
                "var": "x"
            },
            "val": {
                "tag": "NumExp",
                "val": 3
            }
        },
        {
            "tag": "Binding",
            "var": {
                "tag": "VarDecl",
                "var": "y"
            },
            "val": {
                "tag": "VarRef",
                "var": "x"
            }
        }
    ],
    "body": [
        {
            "tag": "VarRef",
            "var": "x"
        },
        {
            "tag": "VarRef",
            "var": "y"
        }
    ]
}

console.log(JSON.stringify(
  rewriteLetStar(parseL3("(let* ((x 5) (y x) (z y)) (+ 1 2))")),null,4));

-> {
    "tag": "LetExp",
    "bindings": [
        {
            "tag": "Binding",
            "var": {
                "tag": "VarDecl",
```

```
            "var": "x"
        },
        "val": {
            "tag": "NumExp",
            "val": 5
        }
    }
],
"body": [
    {
        "tag": "LetExp",
        "bindings": [
            {
                "tag": "Binding",
                "var": {
                    "tag": "VarDecl",
                    "var": "y"
                },
                "val": {
                    "tag": "VarRef",
                    "var": "x"
                }
            }
        ],
        "body": [
            {
                "tag": "LetExp",
                "bindings": [
                    {
                        "tag": "Binding",
                        "var": {
                            "tag": "VarDecl",
                            "var": "z"
                        },
                        "val": {
                            "tag": "VarRef",
                            "var": "y"
                        }
                    }
                ],
                "body": [
                    {
                        "tag": "AppExp",
                        "rator": {
                            "tag": "PrimOp",
                            "op": "+"
                        },
                        "rands": [
                            {
                                "tag": "NumExp",
```

```
                                    "val": 1
                                },
                                {
                                    "tag": "NumExp",
                                    "val": 2
                                }
                            ]
                        }
                    ]
                }
            ]
        }
    ]
}

console.log(JSON.stringify(rewriteAllLetStar(parseL3
    ("(let* ((x (let* ((y 5)) y)) (z 7)) (+ x (let* ((t 12)) t)))")),
    null,4));
-> {
    "tag": "LetExp",
    "bindings": [
        {
            "tag": "Binding",
            "var": {
                "tag": "VarDecl",
                "var": "x"
            },
            "val": {
                "tag": "LetExp",
                "bindings": [
                    {
                        "tag": "Binding",
                        "var": {
                            "tag": "VarDecl",
                            "var": "y"
                        },
                        "val": {
                            "tag": "NumExp",
                            "val": 5
                        }
                    }
                ],
                "body": [
                    {
                        "tag": "VarRef",
                        "var": "y"
                    }
                ]
            }
        }
```

```
],
"body": [
    {
        "tag": "LetExp",
        "bindings": [
            {
                "tag": "Binding",
                "var": {
                    "tag": "VarDecl",
                    "var": "z"
                },
                "val": {
                    "tag": "NumExp",
                    "val": 7
                }
            }
        ],
        "body": [
            {
                "tag": "AppExp",
                "rator": {
                    "tag": "PrimOp",
                    "op": "+"
                },
                "rands": [
                    {
                        "tag": "VarRef",
                        "var": "x"
                    },
                    {
                        "tag": "LetExp",
                        "bindings": [
                            {
                                "tag": "Binding",
                                "var": {
                                    "tag": "VarDecl",
                                    "var": "t"
                                },
                                "val": {
                                    "tag": "NumExp",
                                    "val": 12
                                }
                            }
                        ],
                        "body": [
                            {
                                "tag": "VarRef",
                                "var": "t"
                            }
                        ]
```

```
                    }
                ]
            }
        ]
    }
]
}
```

Question 6 is on the following page.

Question 6 - Unparse (20 points)

This question refers to the language `L1`.
In this question, you are asked to implement a function called `unparse`, which takes as input an `L1` AST (a parsed expression) and transforms it back to its' concrete syntax form. The function should support unparsing all types of expressions that exist in `L1`.

6 - P (20 points) Implement the function `unparse` in the file `q6.ts` that takes as input an `L1` AST and outputs the concrete syntax form that represents the AST. In short, this function satisfies the equality:

```
unparse(parse(ProgramString)) = ProgramString
```

Note the signature of the function:

```
unparse = (x: Program | DefineExp | CExp | Error) : string | Error
```

We will always give a valid AST, however, you must still check the case of receiving an erroneous AST (and throw a relevant `Error`) as to comply with the parser's fashion. For convenience, we will ignore all whitespaces in the output string when we conduct tests.
Examples:

```
console.log(unparse(parseL1("1")));
-> 1 // notice that this is actually a string,
        console.log omits the string quotes
        when printing
console.log(unparse(parseL1("#t")));
-> #t
console.log(unparse(parseL1("#f")));
-> #f
console.log(unparse(parseL1("(+ x 5)")));
-> (+ x 5)
console.log(unparse(parseL1("(L1
    (define x 5)
    (+ x 5)
    (+ (+ (- x y) 3) 4)
    (and #t x))"))); // This is simply one long string.
                        It is split to multiple lines in
                        order to be readable in this document.
-> (L1 (define x 5) (+ x 5) (+ (+ (- x y) 3) 4) (and #t x))
```

Warning: Do not alter the parser in `L1-ast.ts`, since you will be only submitting the file `q6.ts`.

Review before submission

Before you submit, please review the following checklist:

- You are submitting a ZIP file called `id1_id2.zip` or `id.zip` where `id1` is the first partner's ID number and `id2` is the second partner's ID number, or where `id` is your ID number if you are submitting alone.
- The ZIP file includes a file called `id1_id2.pdf` or a file called `id.pdf` with answers to all questions marked with **T** (theoretical questions).
- The ZIP file includes a file called `q4.rkt` where you only implemented the given functions of question 4 and added contracts for each function. You did not add any further (user-defined) helper functions or includes/imports.
- The ZIP file includes a file called `L3-ast.ts` where you only *added* support for `let*` expressions as an addition to the original `L3-ast.ts` template file without adding any further imports/includes/libraries.
- The ZIP file includes a file called `L3-rewrite.ts` where you only implemented the two functions `rewriteLetStar` and `rewriteAllLetStar` without adding any further imports/includes/libraries.
- The ZIP file includes a file called `q6.ts` where you implemented the `unparse` function.
- The ZIP file includes no files other than the ones mentioned above.

You may assume that we will install `ramda,assert,s-expression` in the folder where we will place your files and run our tests. This folder will contain the original `L1-ast` file, the `value.ts` file required by `L3-ast`, as well as our testing files.

Have Fun and Good Luck