Assignment 3

PPL202

Responsible staff: Or Hayat, Michael Elhadad

Submission Instructions

Submit your answers to the theoretical questions and your code for programming questions inside the provided files in the correct places. Zip those files together (including the pdf file, and only those files) into a file called `id1_id2.zip`.

The id1_id2.zip file should include the following files:

Q1.pdf - which will include all your answers for theoretical questions.

The folders shared and L3 (not be changed by you – with files from git).
The files package.json and tsconfig.json (not be changed by you – with files from git).

A folder named **part2**, which includes the following files:
Files not to be changed (from git):

- L4-ast.ts
- L4-env.ts
- L4-value.ts

Files for you to write:

- mermaid-ast.ts
- mermaid.ts

A folder named **part3**, which includes the following files:

Files not to be changed (from git):

- eval-primitive.ts
- test-define-normal.ts
- L4-ast.ts
- L4-env.ts
- L4-eval.ts

Files for you to write or update:

- L4-normal.ts
- L4-env-normal.ts
- L4-value.ts
- L4-normal-test.ts

Do not send assignment related questions by e-mail, use the forum only. For any administrative issues (milu'im/extensions/etc) please open a request ticket in the Student Requests system.

Important: do not add any extra libraries in the supplied template files. Your code must work with EXACTLY the same package.json we provide. If you change it, we will fail to run your code and you will receive a grade of zero. If you find that we forgot to import necessary libraries, let us know.

Part I: Theoretical Questions

1. Is **let** in L3 a *special form*? Justify your answer.
2. List four types of **semantic errors** that can be raised when executing an L3 program – with an example for each type.
3. Instead of using `valueToLitExp` in the substitution model (L3-eval.ts), we could extend the type definitions of the AST to accept values.
   3.1 Update the syntax specification of L3 accordingly.
   3.2 Which parts of the interpreter need to be updated if we adopt this change?
   3.3 Discuss: which of the two options is preferable? (use valueToLitExp or change types)

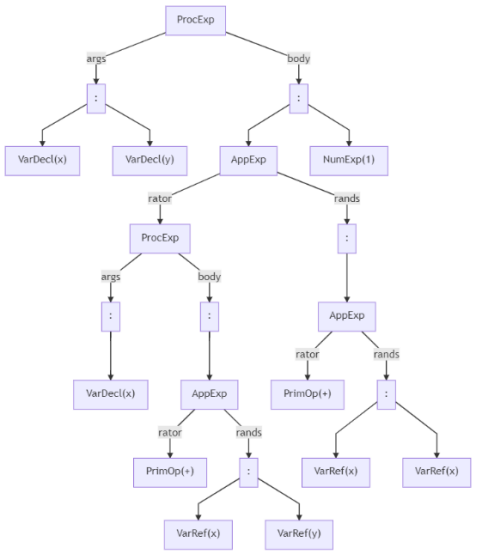
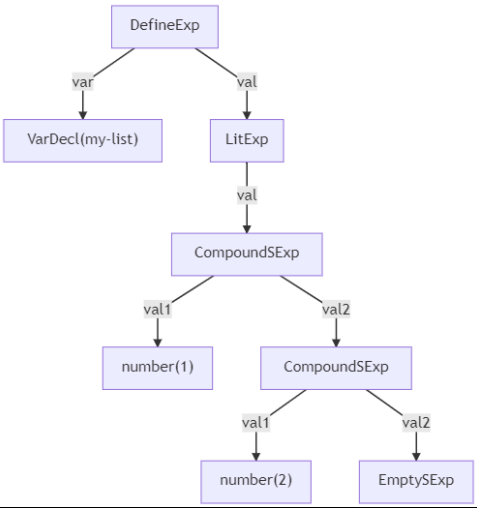
```
<program> ::= (L3 <exp>+) // Program(exps:List(Exp))
<exp> ::= <define> | <cexp>              / DefExp | CExp
<define> ::= ( define <var> <cexp> )     / DefExp(var:VarDecl, val:CExp)
<var> ::= <identifier>                   / VarRef(var:string)
<cexp> ::= <num-exp>                     / NumExp(val:number)
        |  <bool-exp>                    / BoolExp(val:boolean)
        |  <str-exp>                     / 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 |  and | or | eq? | string=?
                 | cons | car | cdr | pair? | number? | list
                 | boolean? | symbol? | string?      ##### L3
<num-exp>  ::= a number token
<bool-exp> ::= #t | #f
<str-exp>  ::= a sequence of characters between double quotes
<var-ref>  ::= an identifier token
<var-decl> ::= an identifier token
<sexp>     ::= symbol | number | bool | string |
               (<sexp>+ . <sexp>) | ( <sexp>* )       ##### L3
```

4. The `valueToLitExp` function is not needed in the normal evaluation strategy interpreter (L3-normal.ts). Why?
5. Show one program where normal order will execute faster than applicative order evaluation and another program where applicative is faster than normal.

Part Ⅱ: Draw Beautiful ASTs

Introduction

In this part, you will write code to draw ASTs of expressions using the Mermaid graphical language. Below are a couple of examples, showing the expected output:

Expression	`(lambda (x y) ((lambda (x) (+ x y)) (+ x x)) 1)`	`(define my-list '(1 2))`
Graph		

Mermaid diagrams are rendered from a graphical language which describes graphs. The Mermaid expression for the first example is:

```
graph TD
    ProcExp_1[ProcExp] -->|args| Params_1[:]
    ProcExp_1 -->|body| Body_1[:]
    Params_1 --> VarDecl_1["VarDecl(x)"]
    Params_1 --> VarDecl_2["VarDecl(y)"]
    Body_1 --> AppExp_1[AppExp]
    Body_1 --> NumExp_1["NumExp(1)"]
    AppExp_1 -->|rator| ProcExp_2[ProcExp]
    AppExp_1 -->|rands| Rands_1[:]
    ProcExp_2 -->|args| Params_2[:]
    Params_2 --> VarDecl_3["VarDecl(x)"]
    ProcExp_2 -->|body| Body_2[:]
    Body_2 --> AppExp_2[AppExp]
    AppExp_2 -->|rator| PrimOp_1["PrimOp(+)"]
```

```
    AppExp_2 -->|rands| Rands_2[:]
    Rands_2 --> VarRef_1["VarRef(x)"]
    Rands_2 --> VarRef_2["VarRef(y)"]
    Rands_1 --> AppExp_3[AppExp]
    AppExp_3 -->|rator| PrimOp_2["PrimOp(+)"]
    AppExp_3 -->|rands| Rands_3[:]
    Rands_3 --> VarRef_3["VarRef(x)"]
    Rands_3 --> VarRef_4["VarRef(x)"]
```

For the second expression `(define my-list '(1 2))` it is:

```
graph TD
    DefineExp_1[DefineExp] -->|var| Var_1["VarDecl(my-list)"]
    DefineExp_1 -->|val| LitExp_1[LitExp]
    LitExp_1 -->|val| CompoundSExp_1["CompoundSExp"]
    CompoundSExp_1 -->|val1| number_1["number(1)"]
    CompoundSExp_1 -->|val2| CompoundSExp_2["CompoundSExp"]
    CompoundSExp_2 -->|val1| number_2["number(2)"]
    CompoundSExp_2 -->|val2| EmptySExp_1["EmptySExp"]
```

The atomic expression “1” is mapped to the following graph:

```
graph TD
    number_1["number(1)"]
```

The program `(L4 1 #t “hello”)` is mapped to the following graph:

```
graph TD
    Program_1[Program] -->|exps| Exps_1[:]
    Exps_1 --> number_1["number(1)"]
    Exps_1 --> boolean_1["boolean(#t)"]
    Exps_1 --> string_1["string(hello)"]
```

You can experiment with the Mermaid language using the online editor:
https://mermaid-js.github.io/mermaid-live-editor/

In addition, when you install the package.json of the assignment, this will install the mermaid-cli command line tool https://github.com/mermaid-js/mermaid.cli. If you store these Mermaid expressions in a file (for example “graph1.mmd”), you can render them into files as follows:

```
./node_modules/.bin/mmdc -i graph1.mmd -o graph1.png
```

This will generate the image in the graph1.png file.

Your code must adopt the same functional programming style as used in the class code in git. (No mutation, no assignment, const only variables, => functions with expression body and no statements except in extreme conditions to be justified in comments).

2.1 Graphical Language Syntax

Let us relate to the Mermaid graphical language as a formal language and model its concrete and abstract syntax. The specific type of diagram we are interested in generating is called a "Flow Chart" in Mermaid specification. (Mermaid supports other types of diagrams, such as sequence diagrams or class diagrams.) A Flow Chart expression in Mermaid has the following structure:

```
<graph>          ::= <header> <graphContent> // Graph(dir: Dir, content: GraphContent)
<header>         ::= graph (TD|LR)<newline>  // Direction can be TD or LR
<graphContent>   ::= <atomicGraph> | <compoundGraph>
<atomicGraph>    ::= <nodeDecl>
<compoundGraph>  ::= <edge>+

<edge>       ::= <node> --><edgeLabel>? <node><newline> // <edgeLabel> is optional
                                        // Edge(from: Node, to: Node, label?: string)

<node>       ::= <nodeDecl> | <nodeRef>
<nodeDecl>   ::= <identifier>["<string>"] // NodeDecl(id: string, label: string)
<nodeRef>    ::= <identifier>             // NodeRef(id: string)
<edgeLabel>  ::= |<identifier>|           // string
```

A TD graph is rendered top-down, an LR graph is rendered left-to-right.
The label after the --> is optional in edges.

- Define the abstract syntax for this language and write the corresponding TypeScript type definitions as disjoint union types in the file mermaid-ast.ts
- TypeScript types for the mermaid AST must have the same names as the syntactic categories, constructors and type predicates must follow the same conventions as those we use in the interpreters code for Li languages (like `makeGraph, makeEdge, isNode` etc).
- All the procedures in this module must be exported – so that they can be used in other modules. Follow the conventions of this example:

```ts
export interface Program {tag: "Program"; exps: Exp[]; }
export const makeProgram = (exps: Exp[]): Program => ({tag: "Program", exps: exps});
export const isProgram = (x: any): x is Program => x.tag === "Program";
```

2.2 Map L4 ASTs to Mermaid diagrams

Your objective is to map the AST of an L4 program to a corresponding Mermaid AST.
The mapping rules are the following:

1. We traverse the L4 AST depth-first
2. The first time an L4 AST node is traversed, we create a corresponding Mermaid AST node and generate a unique name for it.
3. The label of the node must be either:
    a. the name of the type of the L4 AST for compound expressions (*e.g.,* `AppExp_2[AppExp]`)
    b. the atomic type with the value of the atomic leaf for atomic expressions (*e.g.,* `VarRef_4["VarRef(x)"] or number_1["number(1)"]`)
    c. the label ":" when a compound expression has a field with an array value. (*e.g.,* `Body_2[:]`)
4. The id of the node must be a unique identifier within the whole expression which is formed as the L4 AST type of the node – except for the case of [:] nodes which have for name the name of the edge that links to them (*e.g.,* `Body_2[:]` because the edge leading to this "pseudo node" comes from a field named "body" in the **ProcExp** type).
5. For composite nodes in the L4 AST node, we generate one edge for each child.
6. In the edges, the first time the node appears in the Mermaid, it appears as a **NodeDecl**, else as a **NodeRef**.
7. If the L4 AST compound expression has an Array value, we generate a [:] pseudo node.
8. The edge has for label the name of the field, except from [:] to each of the items in the array, in which case the edge has no label.
    a. For example, the AST of the expression `(+ x 4)` is of type **AppExp**, which has two fields, **rator** and **rands**. Thus, the edge between the **AppExp** node and the **PrimOp** node will be labeled as **rator**. The edge from **AppExp** to the operands will be an edge to a pseudo node **":"** with an id of the form **rands_<number>.**
9. To generate unique names for the Mermaid Node IDs, adopt the method used in `makeVarGen()` (https://github.com/bguppl/interpreters/blob/master/src/L3/substitute.ts#L42). Expect to create one such generator for each type you need in the L4 AST syntax.

You must cover all the AST of the L4-ast
https://github.com/bguppl/interpreters/blob/master/src/L4/L4-ast.ts#L27

Signature:

```
export const mapL4toMermaid = (exp: Parsed): Result<Graph>
```

2.3 Serialize the Mermaid AST into an executable diagram

Given the Mermaid AST, we need to generate a Mermaid expression that can be executed by the Mermaid rendering engine.

Adopt the same method as illustrated in the **unparse()** method of the https://github.com/bguppl/interpreters/blob/master/src/L4/L4-ast.ts#L328

Signature:

```
export const unparseMermaid = (exp: Graph): Result<string>
```

Write an end-to-end method to map the concrete syntax of an L4 expression into a Mermaid diagram expression:

Signature:

```
export const L4toMermaid = (concrete: string): Result<string>
```

To validate your solution, you should be able to paste the output of `L4toMermaid` into the Mermaid online editor https://mermaid-js.github.io/mermaid-live-editor/ or using the mermaid-cli:

```
./node_modules/.bin/mmdc -i graph1.mmd -o graph1.png
```

Part III: Normal Environment Evaluator

We reviewed in class two evaluation strategies:

- Applicative evaluation order
- Normal evaluation order

We also covered two distinct implementation methods:

- Substitution model
- Environment-based model

We studied three different implementations:

- Applicative-eval substitution model
  https://github.com/bguppl/interpreters/blob/master/src/L3/L3-eval.ts
- Normal-eval substitution model
  https://github.com/bguppl/interpreters/blob/master/src/L3/L3-normal.ts
- Applicative-eval environment-based model
  https://github.com/bguppl/interpreters/blob/master/src/L4/L4-eval.ts

In this question you will implement the last combination: a normal-eval environment-based model, based on L4-eval (without `set!` And `letrec`).

Your code must adopt the same functional programming style as used in the class code in git. (No mutation, no assignment, const only variables, => functions with expression body and no statements except in extreme conditions to be justified in comments).

3.1 Handling define in normal evaluation

DrRacket supports a version of Scheme which adopts the “normal evaluation strategy”. It is called the “lazy” language https://docs.racket-lang.org/lazy/index.html (developed by esteemed BGU alumni Dr. Eli Barzilay). It is invoked in DrRacket by replacing `#racket` with `#lazy`.

Consider the following programs:

```
#lang lazy
(define x (-))
x
```

and

```
#lang lazy
(define x (-))
1
```

Run them in DrRacket. Explain the behavior of each case.

Consider now the code found in file test-define-normal.ts:

```
import {bind} from "./shared/result";
import {evalNormalParse, evalNormalProgram} from './L3/L3-normal';
import {parseL3} from "./L3/L3-ast";

const pretty = (obj: any): void =>
    console.log(JSON.parse(JSON.stringify(obj, undefined, 2)));

const p2 = bind(parseL3(`(L3 (define x (-)) x)`), evalNormalProgram);
const p3 = bind(parseL3(`(L3 (define x (-)) 1)`), evalNormalProgram);

pretty(p2);
pretty(p3);
```

We observe that our implementation of normal evaluation strategy is incorrect when handling the `define` expression (it does not behave as the `#lazy` language).

Explain what the problem is and what should the solution be in one sentence.

3.2 Environment

We will implement a fixed version of normal evaluation starting from L4 using the environment model. Consider the type of the environment used in L4
https://github.com/bguppl/interpreters/blob/master/src/L4/L4-env.ts

```
export const applyEnv = (env: Env, v: string): Result<Value>
```

In normal evaluation order, variables are not mapped to **Values** but to **CExp**.

Write L4-env-normal.ts according to this change of types.

3.3 Normal Order Evaluation

Write L4-eval-normal.ts to implement the normal evaluation strategy on the L4 language (without `letrec` and `set!`).

Test your implementation with L4-normal-test.ts which is included in the assignment skeleton.

Add 3 additional tests demonstrating different results for normal and applicative order evaluation.

Good luck!