Assignment 3

PPL192

Responsible staff: Ilya Kaufman, Yaron Gonen

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.
A folder named **part2**, which includes the following files:
* error.ts
* L4-ast.ts
* L4-env.ts
* L4-eval.ts
* L4-value.ts
* list.ts
* graph-ast.ts - which will include all your code for part 2.

A folder named **part3**, which includes the following files:
* error.ts
* L4-ast.ts
* L4-env-box.ts
* L4-eval-box.ts
* L4-value-box.ts
* L4-tests-box.ts
* list.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, otherwise, 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.

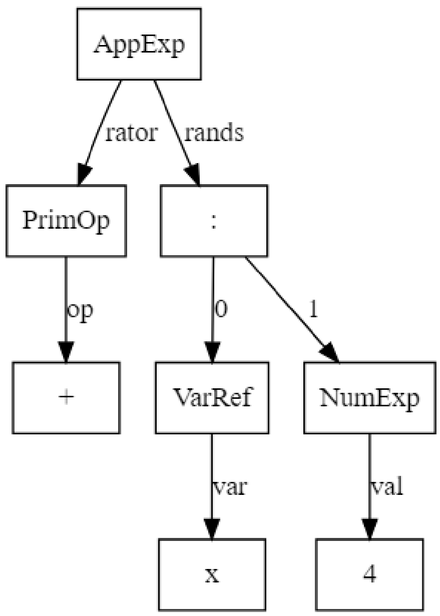
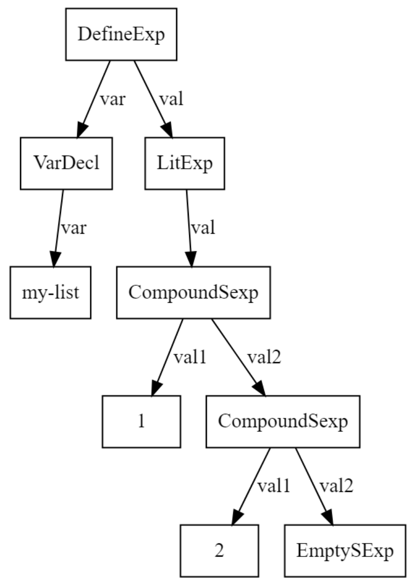
Part Ⅰ: Theoretical Questions

1. What is the difference between *special form* and *primitive operator*? Demonstrate your answer by pointing to code fragments in the interpreter of L3.
2. Two ways of representation for primitive operators were presented in class and in the practical session: `PrimOp` vs. `VarRef`. Explain what is the difference between them. Give example programming languages that use each of the two approaches.
3. Give code examples for equivalent and non-equivalent executions of *applicative order* and *normal order*.
4. What is the role of the function `valueToLitExp` in the substitution model?
5. The `valueToLitExp` function is not needed in the environment interpreter. Why?
6. What are the reasons that would justify switching from applicative order to normal order evaluation? Give an example.
7. What are the reasons that would justify switching from normal order to applicative order evaluation? Give an example.
8. Does the evaluation of `let` expression involve the creation of a closure? Refer to various strategies of evaluation of `let` in different interpreters discussed in class, and provide justification by showing code samples from the interpreter's code.

Part Ⅱ: Draw Beautiful ASTs

Introduction

In this part, you'll draw ASTs of expressions using the [DOT language](#).
Below are a couple of examples, showing the expected output:

<table>
<tr><td>Expression</td><td><code>(+ x 4)</code></td><td><code>(define my-list '(1 2))</code></td></tr>
<tr><td>Graph</td><td>
</td><td>
</td></tr>
<tr><td>DOT code</td><td><code>strict digraph {
_56ktlu5y3
[label=AppExp,shape=record]
_8cs2ofdbv
[label=PrimOp,shape=record]
_s5k6miaqb [label="+",shape=record]
_1dwy3rusj [label=":",shape=record]
_pbr842ggh
[label=VarRef,shape=record]
_atm1u5n0t [label=x,shape=record]
_air9u5n1i
[label=NumExp,shape=record]
_zjyha2n1s [label="4",shape=record]
_8cs2ofdbv -> _s5k6miaqb [label=op]
_56ktlu5y3 -> _8cs2ofdbv
[label=rator]
_pbr842ggh -> _atm1u5n0t [label=var]
_1dwy3rusj -> _pbr842ggh [label="0"]
_air9u5n1i -> _zjyha2n1s [label=val]
_1dwy3rusj -> _air9u5n1i [label="1"]</code></td><td><code>strict digraph {
  _j2z2ghq3b [label=DefineExp,shape=record]
  _gztjx7awx [label=VarDecl,shape=record]
  _ix25jbkcz [label="my-list",shape=record]
  _315k96o2b [label=LitExp,shape=record]
  _m66cjgakz [label=CompoundSexp,shape=record]
  _1g1e543q2 [label="1",shape=record]
  _1j8w63l6n [label=CompoundSexp,shape=record]
  _k4utje91u [label="2",shape=record]
  _emizgms98 [label=EmptySExp,shape=record]
  _gztjx7awx -> _ix25jbkcz [label=var]
  _j2z2ghq3b -> _gztjx7awx [label=var]
  _m66cjgakz -> _1g1e543q2 [label=val1]
  _1j8w63l6n -> _k4utje91u [label=val1]
  _1j8w63l6n -> _emizgms98 [label=val2]
  _m66cjgakz -> _1j8w63l6n [label=val2]
  _315k96o2b -> _m66cjgakz [label=val]
  _j2z2ghq3b -> _315k96o2b [label=val]
}</code></td></tr>
</table>

	_56ktlu5y3 -> _1dwy3rusj [label=rands] }	

We suggest playing around with the language using one of the external tools (see below) and get yourself comfortable with the language (no need to master it, just getting the idea) before continuing.

Instructions

Your Task

You are given a template file, `graph-ast.ts`. In this file, please complete the function `makeAST`, which receives an AST value (`Parsed` type) as an argument and outputs a `Tree` object. The returned `Tree` object will represent the graph of the input AST value as shown in the examples above. Once you have the `Tree` object, use the given wrapper function `expToTree` to run the whole process of converting a string expression to DOT string (see template file for example how to use it)

NOTE: In this part of the assignment you are only required to support expressions defined by the L3 language, even though the supplied files are of L4 (You don't need to support `set!` and `letrec` expressions)

External JavaScript Libraries

You do not need to learn the syntax or the semantics of the DOT language to the letter. We have written wrapper functions (see below) for you using the Javascript libraries graphlib and graphlib-dot. Please install those libraries using the following commands:

```
npm i graphlib
npm i graphlib-dot
```

External Tools

Once you generate the DOT code, you can visualize it using online services like Viz.js or Webgraphviz. If you do not wish to depend on the Internet, you can install GraphViz (available for all platforms), and use the `dot` tool like this

```
$ ts-node graph-ast.ts | dot -Tpng > temp && open temp
```

Given Code

We provide the following code for you:

- `generateId`: A function to generate random node IDs for the graph. In DOT, each node must have a unique node ID. However, you do not need to use this function directly.
- `Tree` type: A type to represent a drawable tree. It has two components: `rootId`, which is the node ID of the root node and `graph`, which is the `graphlib` object of the tree.
- `isTree`: A predicate for the Tree type.
- `makeTree`: A function to create a Tree value. This function receives the following parameter: (1) label: the string to display within the node, (2) nodes: array of `Tree`, which is an array of the children subtrees, (3) edgesLabels: array of strings that represents the edges labels from the root of the tree to its children.
- `makeLeaf`: A function to create a leaf in the tree.
- `astToDot`: Creates the DOT text out of a `Tree` object.
- `expToTree`: Wrapper function for the whole process, it receives an expression in the form of a string and outputs the DOT text that represents the corresponding graph.

Clarifications

- The label of a node is the value in the AST's tag field.
- Edges between nodes should be labeled with the name of the field in the AST. For example, 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`.
- A field of type array in an AST will be represented by an intermediate node with a label ":".
- An edge with the proper label will connect the AST node with its array node.
- The array node will be connected with an edge to all the nodes representing elements of that array. The edges coming out of an array node ":" will be labeled numerically in ascending order starting from 0. The order is determined by the order of the elements in the array.

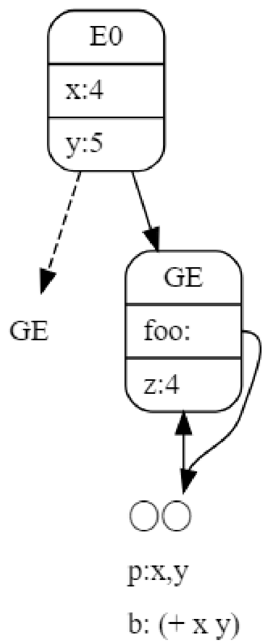
Part Ⅲ: Draw Beautiful Environment Diagrams

Introduction

In this part, you'll draw environment diagrams of programs using the DOT language (please read Part II for details and tools about the language). For example, the drawing of the AST of the program:

```
(L4 (define z 4) (define foo (lambda (x y) (+ x y))) (foo 4 5))
```

will look like this:



The expected DOT code of the drawing above:

```
strict digraph {
  GE_link [label="GE", shape=plaintext]
  B0 [
    label="{<0>◯◯\l|p:x,y\l| b: (+ x y)\l}",shape=record,color=white
  ]
  GE [label="{GE|<foo>foo:\l|z:4\l}",shape=Mrecord]
  E0 [label="{E0|x:4\l|y:5\l}",shape=Mrecord]
  B0:0 -> GE
  GE:foo -> B0:0
  E0 -> GE
  E0 -> GE_link [style=dashed]
}
```

A few notes about the DOT code:

- The circles for closures are gained by two adjacent Unicode characters ◯ (Unicode number U+25EF).
- The control link goes to a "pseudo" global environment, so the drawing will be less cumbersome. You will need such pseudo-environment for each control link.
- Direct link from a specific variable in a frame is gained via the `<...>` syntax. See [here]() for more details.

Instructions

The files for this part are located in the zip in the folder `part3`.
The implementation will be in two steps.

Step 1: Persistent Environment

At any given point in the execution, examining the current environment gives only a partial view of it, because there are no residues of environments no longer in use. So, to keep track of all the environments that were used during the computation, we'll move to a *persistent environment* model.
The idea is quite simple: use a mutable map (dictionary) to keep track of all the environments.
We have already defined this map for you in the code (in `L4-env-box.ts`):

```
export let persistentEnv = {}
```

The "key" in this map will be a unique environment id acquired by the function `generateEnvId`, and the value will be the environment itself (Env type: frame, enclosing environment, etc).
We have already defined the environment id generator and the persistent environment for you.
You are required to modify all the needed code. Hints:

- Change `ExtEnv` type
- Think what needs to change when an environment is extended
- Does it affect closure type?

Notice that no new files should be added. You only need to modify existing code. You can use the regular tests to make sure your code runs.

All the tests in L4-tests-box.ts should pass without modifying the file.
That means that the constructors of makeExtEnv and makeClosure should not be changed.

Step 2: Draw the Environment Diagram

Now you have all the environments in `persistentEnv`. Write the function `drawEnvDiagram` (in `L4-eval-box`) that accepts a persistent environment and draws its diagram. In addition, write the function `evalParseDraw` which is a wrapper function for the whole process: it runs `parseEval` and then `drawEnvDiagram`. It accepts a program string and prints the diagram
You are not required to implement drawing of lists and pairs, and will not be tested on this.

When creating a closure node, its label will be a unique body id acquired by the function `generateBodyId`.

Anonymous closures will not be tested (closures that are passed as parameters will be checked)

Hints:

- Use a single graph object and mutate it - pass it to all the functions
- You may modify any of the files, including making some definitions `export`.
- Use JavaScript's embedded expressions for simple string concatenation (not mandatory, but will make your life a lot easier)

Good luck!