Principles of Programming Languages 2023
Assignment 3

Responsible TA: Eyal German

Submission Date: 21/5/2023

Part 0: Preliminaries

This assignment focuses on interpreters. It covers material from Chapter 2 of the course, with a focus on operational semantics and the environment model.

Structure of a TypeScript Project

Every TypeScript assignment will come with two very important files:

- `package.json` - lists the dependencies of the project.
- `tsconfig.json` - specifies the TypeScript compiler options.

Before starting to work on your assignment, open a command prompt in your assignment folder and run `npm install` to install the dependencies.

What happens when you run `npm install` and the file `package.json` is present in the folder is the following:

1. `npm` will download all required modules and their dependencies from the internet into the folder `node_modules`.
2. A file `package-lock.json` is created which lists the exact version of all the packages that have been installed.

What `tsconfig.json` controls is the way the TypeScript compiler (`tsc`) analyzes and typechecks the code in this project. We will use for all the assignments the strongest form of type-checking, which is called the "strict" mode of the `tsc` compiler.

Do not delete or change these files (*e.g.*, install new packages or change compiler options), as we will run your code against our own copy of those files, exactly the way we provide them.

If you change these files, your code may run on your machine but not when we test it, which may lead to a situation where you believe your code is correct, but you would fail to pass compilation when we grade the assignment (which means a grade of zero).

Testing Your Code

Every TypeScript assignment will have Jest as a global dependency for testing purposes (so no need to import it). In order to run the tests, save your tests in the `test` directory in a file ending with `.test.ts` and run `npm test` from a command prompt. This will activate the execution of the tests you have specified in the test file and report the results of the tests in a very nice format.

An example test file `assignmentX.test.ts` might look like this:

```
import { sum } from "../src/assignmentX";

describe("Assignment X", () => {
  it("sums two numbers", () => {
    expect(sum(1, 2)).toEqual(3);
  });
});
```

Every function you want to test must be `export`-ed, for example, in `assignmentX.ts`, so that it can be `import`-ed in the `.test.ts` file (and by our automatic test script when we grade the assignment).

```
export const sum = (a: number, b: number) => a + b;
```

You are given some basic tests in the `test` directory, just to make sure you are on the right track during the assignment.

What to Submit

You should submit a zip file called `<id1>_<id2>.zip` which has the following structure:

```
/
├── Part1.pdf
└── src/ ...  This directory should hold all
              the files needed.
```

Make sure that when you extract the zip (using `unzip` on Linux), the result is flat, *i.e.*, not inside a folder (the file `Part1.pdf` is in the root directory). This structure is crucial for us to be able to import your code to our tests. Also, make sure the file is a `.zip` file – not a RAR or TAR or any other compression format.

1 Part 1: Theoretical Questions [36 pts]

Submit the solution to this part as `Part1.pdf`. We can't stress this enough: the file *has to be a PDF file*.

1. What is the purpose of `valueToLitExp` and what problem does it solve? [4 pts]
2. `valueToLitExp` is not needed in the normal order evaluation strategy interpreter (L3-normal.ts). Why? [4 pts]
3. What are the two strategies for evaluating a `let` expression? [4 pts]
4. List four types of **semantic errors** that can be raised when executing an L3 program - with an example for each type. [4 pts]
5. What is the difference between a *special form* and a *primitive operator*? [4 pts]
6. What is the main reason for switching from the substitution model to the environment model? Give an example. [4 pts]
7. What is the main reason for implementing an environment using box? [4 pts]
8. Draw an environment diagram for the following computation. Make sure to include the lexical block markers, the control links and the returned values. [8 pts]

```
(define a 2)
(define goo
  (lambda (x)
    (lambda (y)
      (/ x y))))

(define foo
  (let* ((f (goo a))
         (g (lambda (x) (f x))))
    (lambda (x)
      (if (= x 0)
          x
          (g x)))))
(foo (foo 0))
```

2 Part 2: Enhancing L4[64 pts]

Introduction

In this part you are asked to modify and enhance L4 (L4 with Box environment). Most of this part involves programming, but a small part is theoretical, and you are asked to answer it in the PDF of part 1. In the template for this assignment, you are given the whole source code of L4. You may modify all the files in the template, except for the tests file. In addition, skeleton functions were added with their signature. We advise you to use them, but you are not required to do so.

2.1 Check if a variable was bound

In this section you are asked to add a `bound?` **special form**, that checks if a variable was bound in the current environment.

2.1.1 Syntax

Here's the addition to the syntax of L4:

```
<cexp> ::= ...
        | (bound? <var> )   / BoundExp(var: varRef)
```

2.1.2 Use Case

```
(L4
  (define foo 1)
  (bound? foo) ;; => #t
  (bound? goo) ;; => #f
  (define hoo
    (lambda(l)
      (if (bound? l) 0 1))) ;; => #t
)
```

2.1.3 Theoretical Question (answer in Part1.pdf)

Why is `bound?` expression has to be a *special form*, and cannot be a *primitive* or a *user function*?

2.1.4 Implementation Guidelines

- Add a new type of expression, `BoundExp` to the AST. with all the necessary constructors and predicates. Don't forget to check for syntax errors!
- Make sure it is defined as a compound expression by updating the necessary function.
- Add an `unparse` function.
- Add an evaluation rule for this special form.

2.2 Timing Expression Evaluation Like a Pro

In this part you are asked to implement a timing tool for L4. Timing is a profiling tool, and it is extremely useful for optimizing code. The timing tool mimics the `time` tool availabe in Racket (read here for more info.).

2.2.1 Syntax

Here's the addition to the syntax of L4:

```
<cexp> ::= ...
        | (time <cexp> )   / TimeExp(exp: cexp)
```

2.2.2 Theoretical Question (answer in Part1.pdf)

Can it be implemented as a user function, primitive or special form?

2.2.3 How Timing Works

The time is calculated for the evaluation of given expression only. i.e. the parse time is not calculated.

The value of a `time` expression is the `Pair(v, t)` where `v` is the value of the expression, and `t` is the time in milliseconds.

2.3 Examples

For the following program in L4:

```
(L4
  (define list-len
    (lambda(l)
      (if (eq? l (list))
          0
          (+ 1 (list-len (cdr l))))))
  (time (list-len '(1 2 3))
)
```

This value of the last expression should be:

```
  '(3 . 2)
```

2.3.1 Implementation Guidelines

- Add a new type of expression, `TimeExp` to the AST. with all the necessary constructors and predicates. Don't forget to check for syntax errors!
- Make sure it is defined as a compound expression by updating the necessary function.
- Add an `unparse` function.
- Add an evaluation rule for this special form.

Good Luck and Have Fun!