# Principles of Programming Languages 2022
# Assignment 3

Responsible TA: Ariel Grunfeld

Submission Date: 19/5/2022

## Part 0: Preliminaries

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

### 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. Is `let` expression in L3 a *special form*? elaborate. [4 pts]
2. Is a closure created during the evaluation of a `let` expression? Refer to the various strategies discussed in class and in the practical session. [4 pts]
3. List four types of **semantic errors** that can be raised when executing an L3 program - with an example for each type. [4 pts]
4. What is the purpose of `valueToLitExp`? What problem does it solve? [4 pts]
5. `valueToLitExp` is not needed in the normal order evaluation strategy interpreter (L3-normal.ts). Why? [4 pts]
6. What is the difference between a *special form* and a *primitive operator*? [4 pts]
7. What is the reason for switching from the substitution model to the environment model? Give an example. [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: Tracing L4 Recursions [64 pts]

## 2.1 Introduction

In this part you are asked to implement a tracing facility for L4. Tracing is a debugging tool, and it is extremely useful for recursive functions. The tracing facility mimics the tracing facility availabe in Racket (read here for more info.).

## 2.2 How Trace Works

The `trace` expression syntax is:

```
(trace id)
```

*id* must be bound to a closure in the environment of the trace expression. *id* is set to a new closure that traces calls and returns by printing the arguments and results of the call.

The result of a `trace` expression is `void`.

## 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))))))
  (trace list-len)
  (list-len '(1 2 3))
)
```

This output will be printed:

```
> (list-len (1 2 3))
> > (list-len (2 3))
> > > (list-len (3))
> > > > (list-len '())
< < < < 0
< < < 1
< < 2
< 3
```

## 2.4 Guidelines

In the template, 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, we have added skeleton functions with their signature. We advise you to use them.

### 2.4.1 Syntax [24 pts]

Add the `TraceExp` expression to the syntax of L4. Think of what components does it have, and which disjoint type it should be part of. Add all the necessary constructor and predicates.

#### 2.4.2 Value & Semantics [40 pts]

When a `trace` expression is evaluated, a new type of closure is created: `TracedClosure`. A new type is needed because when a `TracedClosure` is applied it needs to execute meta-language code, which is not what happens when a `Closure` is applied. The new `TracedClosure` replaces the existing `Closure` in the environment, so when a recursive call is made, the `TracedClosure` is actually applied.

Think how `TracedClosure` is composed, and implement the constructor and predicates for this new type. Implement the evaluation rule for `TracedExp`. Find the place in the code where `TraceClosure` must be used and update the code accordingly.

# Good Luck and Have Fun!