Compiler Final Project

In this final project, your goal is to craft a Mini-LISP interpreter. All resources you need are on the LMS, including the language specifications, a standard interpreter (called *smli*) and some Mini-LISP program examples for testing. Since you have learnt lex and yacc, it's good to use these tools to build your project, but you are allowed to use other languages and tools for this project.

Your tasks are to

1. Read the language specifications of Mini-LISP.
2. Read and run Mini-LISP program examples to understand the language behavior.
3. Write an interpreter for Mini-LISP, implement features of the language.

- How to run the standard interpreter?

```
$ ./smli example.lsp
```

Project Grade

For each features, there are 2 public test cases and 2 hidden test cases. You can get 80% of the score by passing public test cases, and 20% for hidden test cases.

Basic Features		
Feature	Description	Points
1. Syntax Validation	Print "syntax error" when parsing invalid syntax	10
2. Print	Implement `print-num` statement	10
3. Numerical Operations	Implement all numerical operations	25
4. Logical Operations	Implement all logical operations	25
5. `if` Expression	Implement `if` expression	8
6. Variable Definition	Able to define a variable	8
7. Function	Able to declare and call an anonymous function	8
8. Named Function	Able to declare and call a named function	6

Note: You have to finish feature 1~4 before other features.

Bonus Features		
Feature	Description	Points
1. Recursion	Support recursive function call	15
2. Type Checking	Print error messages for type errors	5
3. Nested Function	Nested function (static scope)	10

4. First-class Function	Able to pass functions, support closure	20

Feature Definitions

Basic Features

You should read the language specifications to understand basic features. After that, try to pass public test cases. Hidden test cases are used to avoid cheating, they are not harder than public test cases. In normal situation, once you passed the public one, you should be able to pass the hidden one, too.

Bonus Features

To implement these features, you are supposed to do some additional research. Actually, it is not very hard. Have fun!

1. **Recursion:** Make your interpreter be able to handle recursive function call. For example:

```
(define f
  (lambda (x) (if (= x 1)
                  1
                  (* x (f (- x 1))))))
(f 4) → 24
```

2. **Type Checking:** For type specifications of operations, please check out the table below:

Op	Parameter Type	Output Type
`+, -, *, /, mod`	**Number**(s)	**Number**
`>, <, =`	**Number**(s)	**Boolean**
`and, or, not`	**Boolean**(s)	**Boolean**
`if`	**Boolean**(s) for test-exp	Depend on then-exp and else-exp
`lambda`	Any	**Function**

Function call	Any	Depend on fun-body and parameters

Please print a message when detecting a type error. For example:

```
(> 1 #t)
  → Type Error: Expect ‘number’ but got ‘boolean’.
```

3. **Nested Function:** There could be a function inside another function. The inner one is able to access the local variables of the outer function. The syntax rule of fun-body should be redefined to:

   fun-body ::= def-stmt* exp

   For example:

```
(define dist-square
  (lambda (x y)
    (define square
      (lambda (x) (* x x)))
    (+ (square x) (square y))))
```

4. **First-class Function:** Functions can be passed like other variables. Furthermore, it can keep its environment. For more details, you can search for “First-class Functions” and “Closure”. For example:

```
(define chose
  (lambda (chose-fun x y)
    (if (chose-fun x y) x y)))
(chose (lambda (x y) (> x y)) 2 1) → 2
```

```
(define add-x
  (lambda (x)
    (lambda (y) (+ x y))))
(define f (add-x 5))
(f 3) → 8
```