Lab 4: Scoping, Parsing in Scheme, and Exceptions

Learning objectives:

- Understand the difference between static and dynamic scoping
- Gain experience writing parsing code in Scheme
- Understand how exception control flow works
- Gain experience reading a programming language specification

For all coding questions, you must use recursion and you may not use mutation (`set!`, `set-car!`, `set-cdr!`, etc.).

Use the following commands to download and unpack the distribution code:

```
$ wget https://eecs390.github.io/lab/lab04/starter-files.tar.gz
$ tar xzf starter-files.tar.gz
```

1. *Static and dynamic scope.* Consider the following code in a language with C-like syntax. Assume the `print()` function prints a value to standard output, and that `main()` is the entry point of the program.

```
int x = 42;
int y = 43;

void func1() {
  print(x);
  print(y);
}

void func2(int x) {
  func1();
}

void func3() {
  func2(y);
}

int main() {
  int y = 44;
  func3();
}
```

   a) Determine the output of the code if the language were to use **static** scope.

   b) Determine the output of the code if the language were to use **dynamic** scope.

2. *Parsing in Scheme.* Consider the following CFG that represents nested lists of positive integers, similar to Python's list syntax:

```
List        -> [ ElementsOpt ]
ElementsOpt -> ε | Elements
Elements    -> Element | Element , Elements
Element     -> List | Integer
Integer     -> Digit | Digit Integer
Digit       -> 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
```

Input String	Syntactically Valid?
`[]`	Yes
`[1,2,3]`	Yes
`[1, 2, 3]`	No (whitespace not allowed)
`[1,2,3,]`	No (trailing comma not allowed)
`[1,[2,3],[[4]]]`	Yes

Implement a parser for this grammar, this time in Scheme.

List Parsing Code

Starter code has been provided for you in `list_parser.scm`. The `parse-list` procedure is the entry point for the parser -- it reads characters from standard input (external representation) and returns a Scheme list representing the parsed string (internal representation).

Your parser should use the recursive descent strategy for parsing, with a procedure for each nonterminal in the grammar. We have provided procedure stubs for each nonterminal as well as an implementation for `parse-list`, which corresponds to the `List` nonterminal. The procedures make use of the provided `read-non-eof-char` and `peek-non-eof-char` procedures to traverse the string, as it is almost identical to the interface you will use in project 2.

When you want to test your implementation, run the parser tests:

```
$ plt-r5rs list_parser.scm < list_parser_test.in > list_parser_test.out
$ diff list_parser_test.out list_parser_test.correct
```

3. *Exceptions.* Consider the following Python program:

```
class NegativeException(Exception):
    def __init__(self, index):
        self.index = index * -1

class OutOfBoundsException(Exception):
    pass  # empty class

def element_at(arr, index):
    print('A', end='')
    if index < 0:
        raise NegativeException(index)
        print('B', end='')
    elif index >= len(arr):
        raise OutOfBoundsException()
    return arr[index]

def main():
    arr = [0,1,2,3,4]
    try:
        # read an integer from standard input
        index = int(input())
        print(element_at(arr, index), end='')
    except OutOfBoundsException:
        print('C', end='')
    except NegativeException as e:
        print('D', end='')
        print(element_at(arr, e.index), end='')
    print("E", end='')

if __name__ == '__main__':
    # entry point of the program is main()
    main()
```

For each of the following inputs on standard input, what does the program print? If nothing is printed, write "nothing" If the program crashes or returns nonzero, ignore any output and write "runtime error". Explain the control flow that causes each result.

a) -3

b) 10

c) -10

4. *Names and scope.* For each of the following snippets of C++17 code, determine what value is printed when the code is executed. If the result is indeterminate according to the C++17 standard, write "undefined". If the code should result in a compilation error, write "error". For each case, write a brief (1-2 sentence) explanation of why the resulting value is printed or is undefined, or why the code is erroneous.

   You may find section 6.3 [basic.scope] of the spec helpful for this question.

a)
```
int main() {
  int x = 3;
  {
    int y = x;
    int x = 4;
    cout << x + y;
  }
}
```

b)
```
int main() {
  int x = 3;
  {
    int y = x, x;
    cout << x;
  }
}
```

c)
```
int main() {
  int x = 3;
  {
    int y = x, x;
    cout << y;
  }
}
```

d)
```
int main() {
  int x = 3;
  {
    int y = x, x = y;
    cout << x + y;
  }
}
```

e)
```
int main() {
  int x = 3;
  {
    int x = x;
    cout << x;
  }
}
```

f)
```
using x = int;

int main() {
  x x = 3;
  cout << x;
```

```
}
```

g)
```
using x = int;

int main() {
  x x = x;
  cout << x;
}
```

h)
```
using x = int;

int main() {
  x x = 3, y = x;
  cout << x + y;
}
```

i)
```
using x = int;

int main() {
  x x = 3;
  x y = x;
  cout << x + y;
}
```