Lab 3: Grammars and Scheme

Learning objectives:

- Understand the difference between internal and external representations
- Understand how recursive rules can be used to derive valid syntax
- Understand how grammar rules relate to the structure of an internal representation
- Gain experience writing Scheme code
- Gain experience with recursion

The concepts covered in this lab will carry over directly into project 2. In project 2, you will be building a parser for Scheme, which takes in an external representation of a program and converts it to a structured, internal representation. You will also need to consult the Scheme language specification and grammar to be able to correctly implement the parser.

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

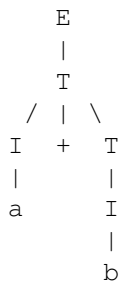
Use the following commands to download and unpack the distribution code:

```
$ wget https://eecs390.github.io/lab/lab03/starter-files.tar.gz
$ tar xzf starter-files.tar.gz
```

1. *Context-free grammars.* Consider the following CFG, with start symbol E:

$$E \to T \mid T - E$$
$$T \to I \mid I + T$$
$$I \to a \mid b$$

The following is a derivation of the string $a + b$:



What are the derivation trees produced for each of the following fragments?

a) $a + b + a$

b) $a + b - a$

c) $a - b + a$

d) $a - b - a$

2. *Parsing.* 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.

List Parsing Code

Starter code has been provided for you in `list_parser.py`. The `parse_list()` function is the entry point for the parser -- it takes in a `Stream` object over a string (external representation) and returns a Python list representing the parsed string (internal representation).

Your parser should use the recursive descent strategy for parsing, with a function for each nonterminal in the grammar. We have provided function stubs for each nonterminal, as well as an implementation for `parse_list()`, which corresponds to the `List` nonterminal. The functions make use of the provided `Stream` class to traverse the string, as it is a similar interface to the one you will use in project 2.

To test your implementation, run the doctests:

```
$ python3 -m doctest list_parser.py
```

3. *List manipulation.* Implement the following procedures in `lists.scm`. Simple tests for each procedure are included in the `lists.scm` starter file. To run the tests:

```
$ plt-r5rs lists.scm
```

You may only use the following special forms and primitive procedures: `define` at global scope, `let`, `let*`, `lambda`, `if`, `cond`, `and`, `or`, `quote`, `null?`, `list?`, `not`, `cons`, `car`, `cdr`, `list`, and `equal?`. Your functions do not have to be tail recursive.

a) Implement the `list-append` procedure, which takes two lists as parameters and returns a new list that is the second list appended to the first list.

```
(list-append '(1 2) '(2 3))  ; returns (1 2 2 3)
(list-append '() '(2 3))     ; returns (2 3)
```

b) Now implement the `deep-reverse` procedure, which takes a list as a parameter and returns a new list that is the reverse of the input list. If the input list contains a list, that list must be reversed as well. You may use your implementation of `list-append`, if necessary.

```
(deep-reverse '(1 2 3 4 5))    ; returns (5 4 3 2 1)
(deep-reverse '((2 4) (1 3)))  ; returns ((3 1) (4 2))
```

c) Implement the `contains` procedure, which takes a list and a value and returns true if the value is found in the list and false otherwise.

```
(contains '(1 2 4) 4)  ; returns #t
(contains '(1 2 4) 3)  ; returns #f
```

4. *Sorting.* In this question, you will implement selection sort, which sorts a list of numbers by repeatedly finding the smallest element in a sublist and moving it to the beginning of that sublist. To sort `(6 4 5 3 9 3)`, for instance, the smallest element `3` is placed at the beginning of the list, and then the rest of the list is sorted. The

full sorting process on the list is as follows, where the portion of the list being examined is underlined, and the smallest element is bolded (e.g. **3**).

(6 4 5 **3** 9 3)
(3 6 4 5 9 **3**)
(3 3 6 **4** 5 9)
(3 3 4 6 **5** 9)
(3 3 4 5 **6** 9)
(3 3 4 5 6 **9**)
(3 3 4 5 6 9)

You may only use the following special forms and primitive procedures: `define` at global scope, `let`, `let*`, `lambda`, `if`, `cond`, `and`, `or`, `quote`, `null?`, `list?`, `not`, `cons`, `car`, `cdr`, `list`, `append`, and number comparisons (=, <=, etc.). Your functions do not have to be tail recursive.

Write your implementations in `sort.scm`. Simple tests are included in the `sort.scm` starter file. To run the tests:

```
$ plt-r5rs sort.scm
```

a) Write the `smallest` procedure, which given a list of numbers and a value as arguments, returns the smallest number in either the list or the given value. The following are examples of using `smallest`:

```
(smallest '(4 3 7) -1)  ; returns -1
(smallest '(4 3 7) 5)   ; returns 3
```

b) Write the `remove` procedure that, given a list and a value, returns a new list that is the same as the original but with the first occurrence of the value removed. If the value does not occur in the list, the new list is the same as the original. The following are examples of using `remove`:

```
(remove '(4 3 7 3) -1)  ; returns (4 3 7 3)
(remove '(4 3 7 3) 3)   ; returns (4 7 3)
```

c) Write the `sort` procedure, which given a list of numbers, returns a copy of the list in sorted, increasing order. Implement selection sort using the `smallest` and `remove` procedures from the previous parts.

The following is an example of using `sort`:

```
(sort '(6 4 5 3 9 3))  ; returns (3 3 4 5 6 9)
```