Lab 3: Scheme and Grammars

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 and using recursion
- Gain experience reading a language specification

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.).

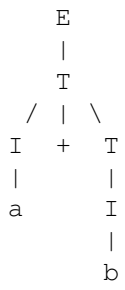
Use the following commands to download and unpack the distribution code:

```
$ 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. *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)
```

4. *Literals and operators.* The file `BitVector.hpp` contains an implementation of a `BitVector` class, which represents a growable sequence of booleans. The class supports all the operations provided by `std::vector<bool>`. In addition, it supports the following operations:

   - The bitwise operators `&`, `|`, and `^`, when applied to two `BitVectors`, should result in a `BitVector` that consists of the `AND`, `OR`, and `XOR` of the two `BitVectors`, respectively. If they differ in size, then the operators should treat the smaller as if it were padded on the right with zeros.
   - String literals that end with the `_bv` suffix construct a `BitVector` from the string with bits in order from left to right. A `0` character specifies `false` and a `1` character specifies `true`. Example:

     ```
     "1010"_bv  -->  [ true, false, true, false ]
     ```

     In other words, the literal (*external representation*) `"1010"_bv` gets converted to the data structure (*internal representation*) of a `BitVector` object that contains the elements `true, false, true, false`.

     You may assume that only the characters `0` or `1` appear in a string literal with the `_bv` suffix.
   - The `<<` stream insertion operator when applied to a `BitVector` should insert a literal (external) representation of the `BitVector` into the stream, complete with enclosing double quotes and the `_bv` suffix. Example:

     ```
     cout << "1010"_bv;  -->  prints "1010"_bv
     ```

     (This converts from the internal representation to the external representation.)

   Write your code in `BitVector.hpp`. We have provided a test case in `BitVector_test.cpp` and expected output in `BitVector_test.correct`.

   You may find this reference helpful.

   To compile and run tests, use the included Makefile:

   ```
   $ make bv
   ```