# Lab 2: Grammars and Internal/External Representations

**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 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.

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

```
$ wget https://eecs390.github.io/lab/lab02/starter-files.tar.gz
$ tar xzf starter-files.tar.gz
```

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

$$\begin{aligned} E &\to T \mid T - E \\ T &\to I \mid I + T \\ I &\to a \mid b \end{aligned}$$

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

```
    E
    |
    T
  / | \
 I  +  T
 |     |
 a     I
       |
       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. *Literals and operators.* Implement a `BitVector` type in C++17, representing a growable sequence of booleans. Your `BitVector` must support the following operations:

   - 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 external representation `"1010"_bv` gets converted to the internal representation of a `BitVector` object that contains the elements `true, false, true, false`.
   - The `size()` member function returns the size of the `BitVector`. The return type should be `size_t`.
   - The `push_back()` member function takes in a `bool` and appends it to the end of the `BitVector`.
   - The `[]` operator returns the boolean at the given position. You may return by value or reference (i.e. you do not have to support modifying a `BitVector` using the `[]` operator).
   - 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.
   - The `<<` stream insertion operator when applied to a `BitVector` should insert each boolean as a `0` or `1`. Example:

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

   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
   ```