Grammar Examples

In this section we illustrate several grammars for some small languages. In PLCC, we can write both the lexical and syntactical specifications for a language in a single file. The lexical specification comes first, it is followed by a single `%` on a line by itself, and this is followed by the syntactic specification.

English Example

Here is one variant of our recurring over-simplified English example.

```
skip WS '\s+'
token WORD '\w+'
token PERIOD '\.'
token QMARK '\?'
%
<sentence> ::= <words> <endMark>
<words> ::= WORD <zeroOrMoreWords>
<zeroOrMoreWords> **= WORD
<endMark> ::= PERIOD
<endMark> ::= QMARK
```

This matches the following "programs":

```
I eat candy.
Monkeys.
Many many monkes.
Sentence is this?
alpha bravo charlie delta echo foxtrot?
asldkjf qoiwer zmnb.
```

The following "programs" would be rejected:

```
.
I ate candy, popcorn, and peaches.
```

Many of the "programs" this language accepts would not be accepted as valid sentences by English speakers. And some that would be valid to an English speaker would not be accepted by this language specification. Specifying a grammar for most human languages is quite challenging, and are beyond the scope of this class. The point of this and other examples in this section is to begin to understand what a particular grammar does and does not allow so that you can begin to design your own.

CSV File Example

A comma separate values files are a common data format. For example, here is sample of some open data from the US Government about birds in California Dataset.

```
X,Y,OBJECTID,SEGMENT,POINT,LATDD,LONDD,SEGPTID
-13526921.5265,4801858.9325,1,19574,1,39.5575829,-121.5143903,195741
-13526977.2754,4802475.7977,2,19574,2,39.5618551,-121.5148911,195742
```

Let's define a grammar based on this example file's structure.

A CSV files consists of one or more lines. The first line contain headers. Each header is separated by a comma. After the header line comes data lines. Each data line contains values separated by comma.

```
skip WS '\s+'
token FIELD '[^,]*'
token COMMA ','
%
<csv> ::= <headers> <data>
<headers> **= FIELD +COMMA
<data> **= <lines>
<lines> **= FIELD +COMMA
```

Note that this definition is not a complete CSV definition; our specification is more more simplistic. Also note that our specification will not prevent some invalid structures (e.g., lines with different number of fields).

Expression Example

Let's design a very simple expression language. We want to be able to write expressions in prefix notation. Below are some examples of expressions in infix notation (what we are used to) and their equivalent in prefix notation. Notice that in prefix notation, we don't need parentheses nor implicit order of operations (e.g., "times comes before addition") to determine order an expression is evaluated!

```
Infix                   Prefix
3 + 4                   + 3 4
3 + 4 * 8               + 3 * 4 8
(3 + 4) * 8             * + 3 4 8
```

Now let's define a grammar for prefix expressions.

```
skip WS '\s+'
token NUM '[0-9]+'
token PLUS '\+'
token MINUS '-'
token TIMES '\*'
token DIVIDE '/'
%
<prefixExpression> ::= NUM
<prefixExpression> ::= <operator> <prefixExpression> <prefixExpression>
<operator> ::= PLUS
```

```
<operator> ::= MINUS
<operator> ::= TIMES
<operator> ::= DIVIDE
```

Exercises

1. Augment the prefix expression grammar to allow unary minus (i.e., - 3).
2. Augment the prefix expression grammar to allow negative numeric literals.
3. Augment the prefix expression grammar to allow operations to have two or more operands (e.g., `+ 3 2 4` is equivalent to `3 + 2 + 4`).