Chris Fenton CNC - PLD Assignment 2

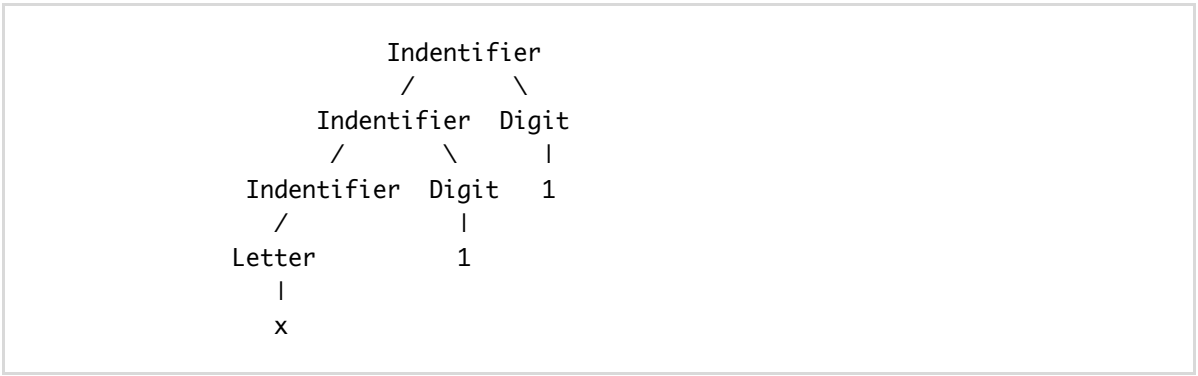
2.1 Using the grammar rules for Integer develop a leftmost derivation for the integer 4520. How many steps are required for this derivation? In general, how many steps are required to derive an integer with an arbitrary number, say d, of Digits?

```
Integer => Integer Digit => Integer Digit Digit => Integer Digit Digit Digit =>
Digit Digit Digit Digit => 4 Digit Digit Digit => 4 5 Digit Digit => 4 5 2 Digit
=> 4 5 2 0
```

8 steps in this case. In the general case, there 2 * d steps: d steps to peel of the digits from the right side of the integer and d steps to substitue the literals.

2.2 Develop a parse tree for the Identifier value x11, using the BNF syntaxgiven in Figure 2.3.

```
Indentifier => Identifier Digit => Identifier Digit Digit => Letter Digit Digit
=> x Digit Digit => x 1 Digit => x 1 1
```



2.5 Try to define the language {$a^n b^n$} using regular expressions. Discuss why this might not be possible.

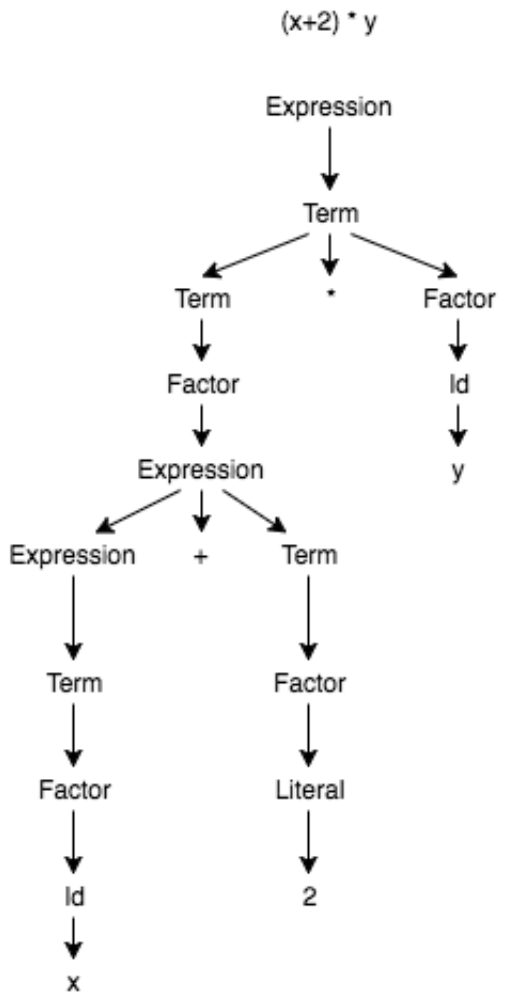
If L is a regular language then the pumping lemma must hold true:

```
Pumping Lemma:

There exists an integer p ≥ 1 depending only on L such that every string w in L
of length at least p (p is called the "pumping length") can be written as w =
xyz (i.e., w can be divided into three substrings), satisfying the following
conditions:

|y| ≥ 1,
|xy| ≤ p,
for all i ≥ 0, x(y^i)z ∈ L
```

- Claim: $a^n b^n$ is not regular
- Let p be the number from the pumping lemma
- Let w = $a^p b^p$
- Since w ∈ L and |w| >= p the pumping lemma applies:
    - w = xyz where y != {} and |xy| <= p
- Note: the xy part must occure in the first p characters:
    - Therefore y must occur in the firt p characters
    - y = a^k where 0 < k <= p
    - x = a^q wher 0 <= q < p
    - z = $a^{p-k-q} b^p$ (what's leftover)
- Per the pumping lemma: xyyz ∈ L:
    - xyyz = $a^q a^k a^k a^{p-k-q} b^p$
    - xyyz = $a^{q+k+k+p-k-q} b^p$
    - xyyz = $a^{k+p} b^p$
- $a^{k+p} b^p$ is not an element of L (contradiction)
- L is not regular

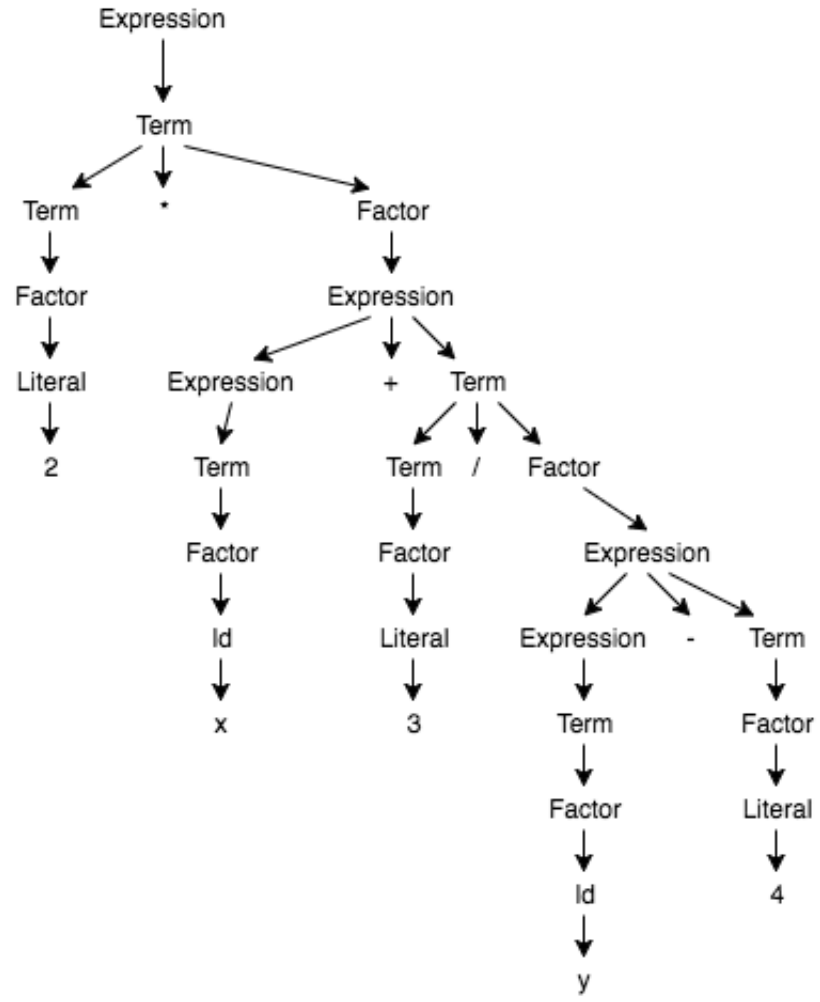
2.8 Develop parse tree for each of the following Expressions, as defined by the EBNF definition of Expression.

(a) (x+2)*y

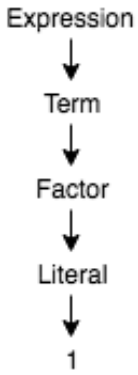

(x+2) * y
Expression
Term
Term
*
Factor
Factor
Id
Expression
y
Expression
+
Term
Term
Factor
Factor
Literal
Id
2
x

(b) 2*x+3/y-4

2 * x + 3 / y - 4


Expression
Term
Term
*
Factor
Factor
Expression
Literal
Expression
+
Term
2
Term
Term
/
Factor
Factor
Factor
Expression
Id
Literal
Expression
-
Term
x
3
Term
Factor
Factor
Literal
Id
4
y

(c) 1


Expression
Term
Factor
Literal
1

2.13 Complete the recursive descent parser by writing the methods that parse a Term and a Factor, guided by their EBNF syntax definitions and the alogrithm for constructing them given in Figure 2.18. Use the example methods for Assignment and Expression as models.

```
private Term term() {
  // Term -> Factor{ ['*' | /] Factor }*
  Binary b; Term t; Factor f;
  t = factor();
  while (token.value.equals('*') || token.value.equals('/')) {
    b = new Binary();
    b.term1 = t;
    b.op = new Operator(token.value);
    token = input.nextToken();
    b.term2 = factor();
    t = b;
  }
  return t;
}

private Factor factor() {
    // Factor -> Indentifer | Literal | (Expression)
    Expression e = null;
    if (token.getType().equals("Identifier")) {
        Variable v = new Variable();
        v.id = token.getValue();
        e = v;
        token = input.nextToken();
    } else if (token.getType().equals("Literal")) {
        Value val = new Value(token.getValue());
        e = val;
        token = input.nextToken();
    } else if (token.getType().equals("Expression")) {
        e = expression();
    }
    return e;
}
```