CMSI 488 Homework #3

Zane Kansil & Edward Bramanti

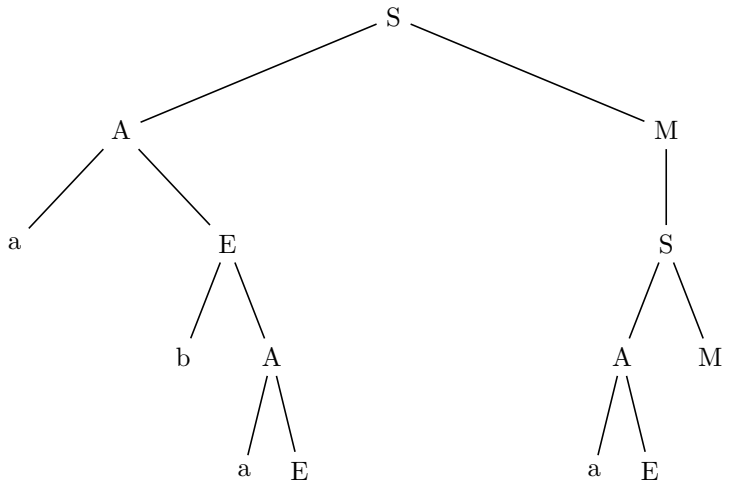
March 25, 2014

1. Here's a grammar:

```
S -> A M
M -> S?
A -> 'a' E | 'b' A A
E -> ('a' B | 'b' A)?
B -> 'b' E | 'a' B B
```

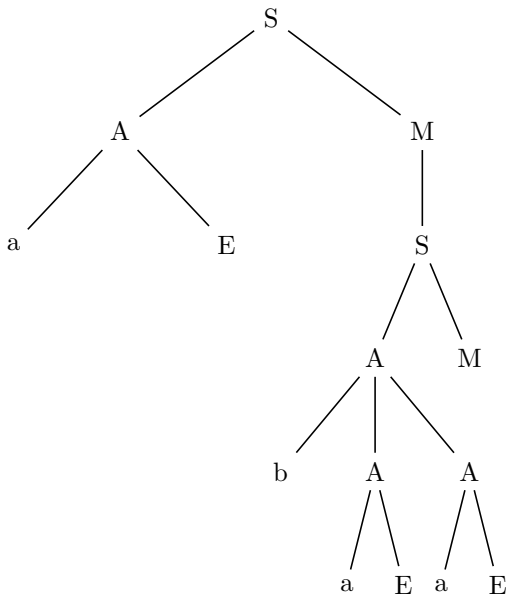
   (a) Describe in English, the language of this grammar.
   It is a language of strings over {a, b}. The start symbol generates one or more A's. Each A will expand to a string creating one more a than b, and each B will do the opposite: one more b than a. E will also have an equal number of a's and b's.

   (b) Draw a parse tree for the string abaa.



   (c) Prove or disprove: This grammar is LL(1).
   The grammar is not LL(1). When looking at E, you can either go from E to aB or the empty string, because an E can be followed by an 'a'. An E can end with an A followed by an A, and begin with an 'a'.

   (d) Prove or disprove: This grammar is ambiguous.
   The grammar is ambiguous. Here is another parse tree for abaa, different from the tree above.


S
A
M
a
E
S
A
M
b
A
A
a
E
a
E

2. Here's a grammar that's trying to capture the usual expressions, terms, and factors, while considering assignment to be an expression.

```
EXP         -> ID ':=' EXP | TERM TERM_TAIL
TERM_TAIL   -> ('+' TERM TERM_TAIL)?
TERM        -> FACTOR FACTOR_TAIL
FACTOR_TAIL -> ('*' FACTOR FACTOR_TAIL)?
FACTOR      -> '(' EXP ')' | ID
```

(a) Prove that this grammar is not LL(1).
The grammar is not LL(1) because you can expand an EXP when looking at ID in two ways:

```
EXP     -> ID ':=' EXP
EXP     -> ID           // from TERM to FACTOR
```

(b) Rewrite it so that it is LL(1).

```
EXP     -> ID( ':=' EXP | '(' EXP ')' ( '*' FACTOR )* ('+' FACTOR ( '*' FACTOR )* )* )
FACTOR  -> '(' EXP ')' | ID
```

3. Write an attribute grammar for the grammar in the previous problem. Your attribute grammar should describe the obvious run-time semantics.

```
EXP ->  ID -> {E.is = ID.value}
        (   (":=" EXP1 -> {E.is = assign(left, EXP1.value)})
        |   ("*"
                (  "(" EXP1 ")" -> {E.is = multiply(E.is, EXP1.value)})
                |  ID1 -> {E.is = multiply(E.is, ID1.value)}
            )*
        |   ("+"
                {E.left = E.is, E.right = null}
                (  "(" EXP1 ")" {E.right = EXP1.value}
                    ("*"
                        (  "(" EXP2 ")" {E.right = multiply(E.right, EXP1.value)})
                        |  ID1 {E.right = multiply(E.right, ID1.value)}  )
                    )*
                    {E.is = add(E.left, E.right)}
                |  ID1 -> {E.right = ID1.value}
                    ("*"
                        (  "(" EXP2 ")" {E.right = multiply(E.right, EXP1.value)})
                        |  ID1 {E.right = multiply(E.right, ID1.value)}  )
                    )*
                    {E.is = add(E.left, E.right)}
                )
            )*
        )

Original:
    EXP         -> ID ":=" EXP | TERM TERM_TAIL
    TERM_TAIL   -> ("+" TERM TERM_TAIL)?
    TERM        -> FACTOR FACTOR_TAIL
    FACTOR_TAIL -> ("*" FACTOR FACTOR_TAIL)?
    FACTOR      -> "(" EXP ")" | ID

Changed:
    EXP         -> ID ":=" EXP | FACTOR ("*" FACTOR)* ("+" FACTOR ("*" FACTOR)*)*
    FACTOR      -> "(" EXP ")" | ID
```

4. Write an attribute grammar for evaluation (using the notation introduced in this class), whose underlying grammar is amenable to LL(1) parsing, for polynomials whose sole variable is x and for which all coefficients are integers, and all exponents are non-negative integers. The following strings must be accepted.

   - `2x`
   - `2x^3+7x+5`
   - `3x^8-x+x^2`
   - `3x-x^3+2`
   - `-9x^5-0+4x^100`
   - `-3x^1+8x^0`

```
POLY   ->   ('-' {negative := true})?

            TERM {POLY.value := TERM.value}
            (
              ( '+' {op := '+'} | '-' {op := '-'} )
              TERM {POLY.value := op(POLY.value, TERM.value)}
            )*
            {P.value := negative ? -P.value : P.value}

TERM   ->   {EXPONENT := 0} (NUMBER {BASE := INT(NUMBER.lexeme)})
            ('x' {EXPONENT := 1}
                ('^' (NUMBER) {EXPONENT := NUMBER.value})?
            )?
```

5. Write a command-line application in Ruby, Clojure, JavaScript, or Python that evaluates polynomials from the language you defined above. The first argument should be the polynomial and the second is the value at which to evaluate the polynomial. Note that for this problem it is not necessary for you to build a seriously structured project with separate modules for scanning, parsing, error handling, abstract syntax tree construction and evaluation. Instead, make a very short and sweet script. There are no spaces in the polynomial strings so lexical analysis is no big deal; just bang out the code as a simple script.

```
var StringScanner = require("strscan").StringScanner
function evaluatePolynomial(scanner, x) {
    negative = scanner.scan(/-/);
    value = parseTerm(scanner, x);
    value = negative ? -value : value;
    while(scanner.scan(/[+-]/)){
        operator = scanner.getMatch()
        value += (operator === '+' ? 1 : -1) * parseTerm(scanner, x);
    }
    return value;
}
function parseTerm(scanner, x) {
    var coefficient = 1;
    var exponent = 0;
    var lexeme = scanner.scan(/\d+/);
    if (lexeme) {
        coefficient = lexeme;
    }
    if(scanner.scan(/x/)) {
        exponent = 1;
        if(scanner.scan(/\^/)) {
            exponent = scanner.scan(/\d+/);
        }
    }
    return coefficient * (Math.pow(x, exponent));
}
var s = new StringScanner(process.argv[2])
if (process.argv.length < 4) {
    console.log("Not enough arguments. Requires polynomial and x-value.");
} else if (process.argv.length > 4) {
    console.log("Too many arguments. Only requires polynomial and x-value.");
} else {
    console.log(evaluatePolynomial(s, process.argv[3]))
}
```