1 2004 Paper 4 Question 1

https://www.cl.cam.ac.uk/teaching/exams/pastpapers/y2004p4q1.pdf

(a) A context-free grammar can be formally defined as a 4-tuple. Give a precise statement of what the components are

$$G = (N, T, P, S)$$

- G is the grammar
- N is the set of nonterminals

  A Nonterminal is an internal symbol. These represent concepts such as expressions or statements.
- T is the set of terminals

  A Terminal is a token passed to the parser by the lexer. These may correspond to an individual literal or a sequence of literals. Terminals are indivisible. The input to any PDA is a sequence of terminals.
- $P \subseteq N \times (N \cup T)*$ is the set of productions

  A production is of the form $A \to \alpha$ and says that it is legal for any occurrence of A to be replaced with α at any point.
- $S \in N$ is the start symbol

(b) Explain the difference between a grammar and the language it generates.

A grammar is a set of rules which is used to generate a language.

The language generated by a grammar is a set of strings.

Each grammar generates exactly one language, however a given language may be generated by many languages.

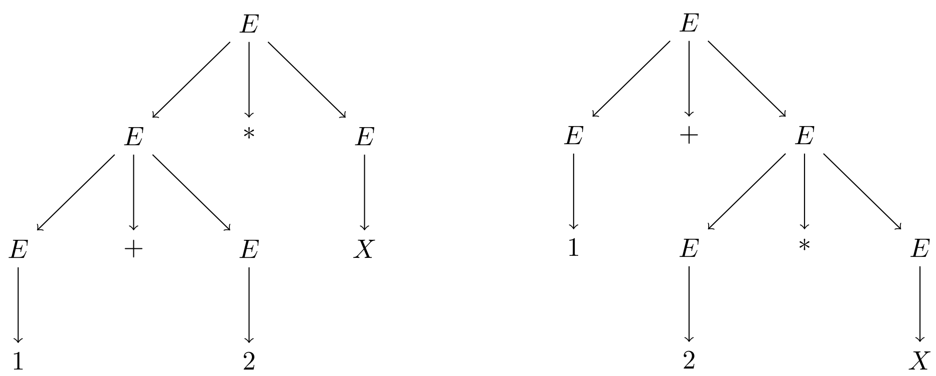
(c) Explain what makes a grammar ambiguous, with reference to the grammar which may commonly be expressed as a "rule"

$$E ::= 1 \mid 2 \mid X \mid E + E \mid E * E \mid -E$$

where X is an identifier

A grammar is ambiguous if there exists any string for which there are multiple ways the grammar can be used to generate that string. Consider the string $1 + 2 * X$ with the grammar above.

Under the grammar above, there are two possible parse trees for $1+2*X$ and therefore the grammar is ambiguous.



(d) For the "rule" in part (c), give a formal grammar containing this "rule" and adhering to your definition in part (a).

$$\begin{aligned}G = (&\{E\},\\ &\{1, 2, X\},\\ &\{(E, 1), (E', 2), (E', X), (E, E + E), (E, E * E), (E, -E)\}\\ &E)\end{aligned}$$

$$\begin{aligned}E &::= T\ E'\\ E' &::= +T\ E' \mid *T\ E' \mid \varepsilon\\ T &::= N \mid -N\\ N &::= 1 \mid 2 \mid X\end{aligned}$$

(e) Give non-ambiguous grammars each generating the same language as your grammar in part (d) for the cases:

(i) "−" is most tightly binding and "+" and "∗" have equal binding power and associate to the left.

$$\begin{aligned}G_1 = (&\{E, E', N, T\},\\ &\{1, 2, X\},\\ &\{(E, TE'), (E', E'T+), (E', E'T*), (E', \varepsilon), (T, N), (T, -N), (N, 1), (N, 2), (N, X)\}\\ &E)\end{aligned}$$

$$\begin{aligned}E &::= E'\ T\\ E' &::= E'\ T+ \mid E'\ T* \mid \varepsilon\\ T &::= N \mid -N\\ N &::= 1 \mid 2 \mid X\end{aligned}$$

(ii) "−" is most tightly binding and "+" and "∗" have equal binding power and associate to the right.

$$\begin{aligned}G_2 = (&\{E, E', N, T\},\\ &\{1, 2, X\},\\ &\{(E, TE'), (E', +TE'), (E', *TE'), (E', \varepsilon), (T, N), (T, -N), (N, 1), (N, 2), (N, X)\}\\ &E)\end{aligned}$$

$$\begin{aligned}E &::= T\ E'\\ E' &::= +T\ E' \mid *T\ E' \mid \varepsilon\\ T &::= N \mid -N\\ N &::= 1 \mid 2 \mid X\end{aligned}$$

(iii) "−" binds more tightly than "+", but less tightly than "∗", with "+" left-associative and "∗" right-associative so that "$-a + -b * c * c + d$" is associated as "$((-a) + (-(b * (c * d)))) + d$".

$$\begin{aligned}G_3 = (&\{E, E', A, T, T', N\},\\ &\{1, 2, X\},\\ &\{(E, E'A), (E', E'A+), (E', \varepsilon), (A, T), (A, -T), (T, NT'), (T', *NT'), (T', \varepsilon), (N, 1), (N, 2), (N, X)\}\\ &E)\end{aligned}$$

$$
\begin{aligned}
E &::= E' \ A \\
E' &::= E' \ A + \ \mid \varepsilon \\
A &::= T \mid -T \\
T &::= N \ T' \\
T' &::= *N \ T' \mid \varepsilon \\
N &::= 1 \mid 2 \mid X
\end{aligned}
$$

(f) Give a simple recursive descent parser for your grammar in part (e)(iii) above which yields a value of type `ParseTree`. You may assume operations *mkplus*, *mktimes*, *mkneg* acting on type `ParseTree`.

Firstly, note that the grammar (e)(iii) is by definition left-associative. A gramamr is left-recursive if and only if it is left-associative. So there exists no grammar which fulfils the criteria for (e)(iii) that is not left-recursive. Left-recursive grammars cannot be parsed by a recursive descent parser. My solution to this is to build a parse tree for the language and then assume *mkplus*, *mktimes* and *mkneg* rotate parse trees into the correct shape. This algorithm will build a valid parse tree for the grammar (e)(iii).

```
type n = E | E' | A | T | T' | N

type t = + | - | 1 | 2 | X | Epsilon

type parseTree = Branch of n * parseTree list | Leaf of t

let parse ts =
        let rec parse ts, n =
                match ts, n with
                | Plus::ts, E' ->
                        let pt1, ts = parse ts, T in
                        let pt2, ts = parse ts E' in
                        (Branch n, [pt1; Leaf Plus; pt2]), ts
                | Times::ts, E' ->
                        let pt1, ts = parse ts, T in
                        let pt2, ts = parse ts E' in
                        (Branch n, [pt1; Leaf Times; pt2]), ts
                | Minus::ts, E -> let pt, ts = parse (Minus::ts) T in
                        (Branch n, [Leaf Minus; pt]), ts
                | Minus::ts, T -> let pt, ts = parse ts N in
                        (T_P2 T, [Leaf Minus; pt]), ts
                | One::ts, N -> (Leaf One), ts
                | Two::ts, N -> (Leaf Two), ts
                | X::ts, N -> (Leaf X), ts
                | x::ts, T when x = 1 || x = 2 || x = X ->
                        let pt, ts = parse (x::ts), N in
                        (Branch n, [pt]), ts
                | x::ts, T when x = 1 || x = 2 || x = X ->
                        let pt1, ts = parse ts, T in
                        let pt2, ts = parse ts E' in
                        (Branch n [pt1; pt2]), ts
                | _, E' -> (Leaf Epsilon), ts
                | _ -> raise ParseException
        in
        match parse ts E with
        | pt, [] -> mkplus (mkminus (mktimes pt))
        | _ -> raise ParseException
```

2 2002 Paper 4 Question 2

https://www.cl.cam.ac.uk/teaching/exams/pastpapers/y2002p4q2.pdf

The specification for a pocket-calculator-style programming language is as follows:

- Valid inputs consist either of an Expression followed by the $\boxed{\text{enter}}$ button of of an Expression followed by $\boxed{\text{store}}$ Identifier $\boxed{\text{enter}}$;
- Expressions consist of Numbers and Identifiers connected with the binary operators $\boxed{+}$, $\boxed{\times}$ and $\boxed{\uparrow}$ (in increasing binding power), with the nary operators $\boxed{-}$ and $\boxed{\text{abs}}$, and possibly grouped with parentheses. Unary operators bind more strongly than $\boxed{+}$ but weaker than $\boxed{\times}$ so that $-a + b$ means $(-a) + b$ but $-a \times b$ means $-(a \times b)$.
- Numbers consist of a sequence of at least one digit, possibly interspersed with exactly one decimal point, and possibly followed by an exponential marker "e" followed by a signed integer, e.g. $6.023e + 22$. Identifiers are sequences of lower-case letters.

(a) Give a Context-Free Grammar for the set of valid input sequences using names beginning with an upper-case letter for non-terminals. It should be complete in that you should go as far as to define e.g.

$$\textbf{Letter} ::= \textbf{a} \mid \textbf{b} \mid \textbf{c} \mid \dots \mid \textbf{z}$$

$$\begin{array}{rl}
\textbf{Start} ::= & \textbf{Expression}\ \boxed{\text{enter}} \mid \textbf{Expression}\ \boxed{\text{store}}\ \textbf{Identifier}\ \boxed{\text{enter}} \\
\textbf{Expression} ::= & \textbf{Unary OptExpression} \\
\textbf{OptExpression} ::= & \boxed{+}\ \textbf{Unary OptExpression} \mid \varepsilon \\
\textbf{Unary} ::= & \textbf{Times} \mid \boxed{-}\ \textbf{Times} \mid \boxed{\text{abs}}\ \textbf{Times} \\
\textbf{Times} ::= & \textbf{Arrow OptTimes} \\
\textbf{OptTimes} ::= & \boxed{\times}\ \textbf{Arrow OptTimes} \\
\textbf{Arrow} ::= & \textbf{Value OptArrow} \\
\textbf{OptArrow} ::= & \boxed{\uparrow}\ \textbf{Value OptArrow} \\
\textbf{Value} ::= & \textbf{Identifier} \mid \textbf{Number} \\
\textbf{Identifier} ::= & \textbf{Letter OptIdentifier} \\
\textbf{OptIdentifier} ::= & \textbf{LetterOptIdentifier} \mid \varepsilon \\
\textbf{Letter} ::= & \textbf{a} \mid \textbf{b} \mid \textbf{c} \mid \dots \mid \textbf{z} \\
\textbf{Number} ::= & \textbf{Int OptInt OptDecimal OptSuffix} \mid \boxed{.}\ \textbf{Int OptInt OptSuffix} \\
\textbf{Int} ::= & 0 \mid 1 \mid \dots \mid 9 \\
\textbf{OptInt} ::= & \textbf{Int OptInt} \mid \varepsilon \\
\textbf{OptDecimal} ::= & \boxed{.}\ \textbf{OptInt} \mid \varepsilon \\
\textbf{OptSuffix} ::= & \textbf{e Sign Int OptInt} \\
\textbf{Sign} ::= & \boxed{+} \mid \boxed{-}
\end{array}$$

(b) Indicate, giving brief reasoning, which non-terminals are appropriate to be processed using lexical analysis and for which using syntax analysis is proper.

It's appropriate to process **Value**, **Identifier**, **OptIdentifier**, **Letter**, **Number**, **Int**, **OptInt**, **OptDecimal**, **OptSuffix** and **Sign** in lexical analysis. This is because the langauge which these non-terminals can match is regular and there is no binding tightness to consider. Therefore, it's appropriate to process them during lexing.

(c) Give yacc or CUP input describing those elements deemed in part (b) to be suitable for syntax analysis. You need not give "semantic actions".

```
%token Start Expression OptExpression Unary Times OptTimes Arrow OptArrow

%%

Start          : Expression 'enter'
               | Expression 'store' Identifier enter

Expression     : Unary OptExpression

OptExpression  : '+' Unary OptExpression
               | /* ε */

Unary          : Times
               | '−' Times
               | 'abs' Times

Times          : Arrow OptTimes

OptTimes       : '×' Arrow OptTimes

Arrow          : Value OptArrow

OptArrow       : '↑' Value OptArrow
               | /* ε */
```