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.

  $N \cap T = \emptyset$
- $P \subseteq N \times (N \cup T)*$ 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.

  N,T,P must be finite
- $S \in N$ is the start symbol

  N, T, P needs to be finite to relate to the theoretical PDA equivalence.
  Infinite grammar woud be impossible to implement.

(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.

finite vs poss inf

structured vs flat

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

grammar is finite -- language is (potentially) infinite

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

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

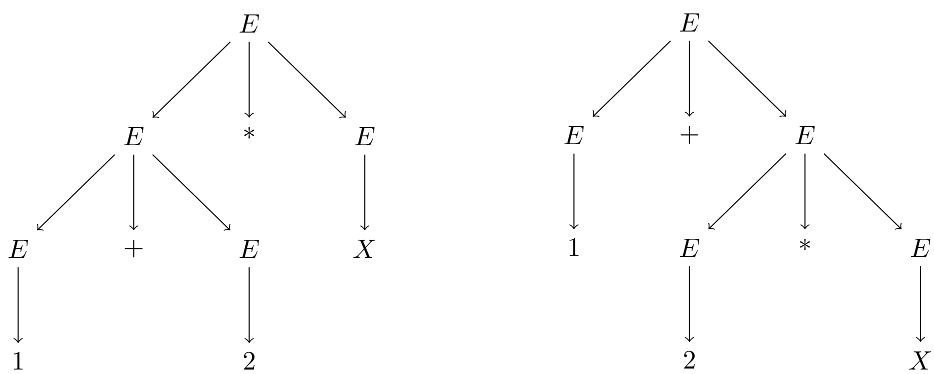
where X is an identifier

Always inspect the language -- don't consider it how you WANT to consider it.
Be very. careful

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.

must restrict to only counting leftmost derivations

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



A turing machine (type 0 grammar) is a CFG but

P \subseteq (N \cup T)^* N (N \cup T)^* \to (N \cup T)^*

Note that you can CHANGE terminals and delete them

A context sensitive grammar (type 1) is a CFG but

P \subseteq α N β \to α (N \cup T)^* β

English is a Context Sensitive Grammar

To process CFG you need an infinite stack -- but you ONLY need a stack
You don't need to invent a new data structure to parse a particular grammar

Type 2 grammars require memory proportional to
(1 + lookahead) ^ (|N| + |T|)

Regular languags are CFG but

Productions are of the form
N \to T | TN

Lexers have to process bytes. so we wnat to use constant memory and lineaar time. So we want to process them with regular expressions. After tokenizing, we can decrease the size of the input by 10x.

Lex as much as you can -- it's faster, constant memory.
Only parse things you physically cannot lex (pumping lemma)

If you use a regular expression to parse part of your string, it can turn an ambiguous grammar into an unambiguous one; it's just too simple to notice the ambiguity.

Sometimes we do this deliberately

Using a regular expression to lex cannot INRODUCE ambiguity

(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
```

surely not correct? this sets ts=T which prevents the next line from accessing the token stream

disagrees with type definition

otherwise, right idea, yes

You can get out of left/right recursive by making an abstract syntax tree.

- when you use the left recursion elimination algorithm, you only use it on small parts of the grammar
- in these cases, you don't really care about HOW this integer is parsed

Add a start symbol for EOF
Eliminate direct / indirect left-recursion
Left-factor the rules

EOF check is ESSENTIAL -- otherwise the language will accept strings which have a valid prefix

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 unary 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.

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

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

can't say "- abs X"

can't say 2*-1

big problems

can't say 2^-1

2. is also valid

otherwise OK

(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.

what about absence of binding makes a lexer appropriate?

(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 */
```

?

Same bugs as earlier.

Don't encoding precedence or binding in the yacc input.
The tool provides this so just use it; don't do yacc's job for it!

```
%left +
%left *
%noassoc -
%right ^

start symbol -> E : E
                  | E*E
                  | E+E
                  | -E
                  | E^E
```