1 A brief introduction to Extended Backus-Naur Form

Extended Backus-Naur Form (*EBNF*) is a language to specify programming languages in. The name is a tribute to John Backus who used it to describe the syntax of ALGOL58 and Peter Nauer for his work on ALGOL 60.

· Extended Backus-Naur Form
· EBNF

An EBNF consists of *terminal symbols* and *production rules*. Examples of typical terminal symbol are characters, numbers, punctuation marks, and whitespaces, e.g.,

· terminal symbols
· production rules

```
digit = "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9";
```

A production rule specifies a method of combining other production rules and terminal symbols, e.g.,

```
number = digit { digit };
```

A proposed standard for ebnf (proposal ISO/IEC 14977, `http://www.cl.cam.ac.uk/~mgk25/iso-14977.pdf`) is,

'=' definition, e.g.,

```
zero = "0";
```

here `zero` is the terminal symbol `0`.

'`,`' concatenation, e.g.,

```
one = "1";
eleven = one, one;
```

here `eleven` is the terminal symbol `11`.

'`;`' termination of line

'`|`' alternative options, e.g.,

```
digit = "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" |
  "9";
```

here `digit` is the single character terminal symbol, such as `3`.

'`[ ... ]`' optional, e.g.,

```
zero = "0";
nonZeroDigit = "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" |
  "9";
nonZero = [ zero ], nonZeroDigit;
```

here `nonZero` is a non-zero digit possibly preceded by zero, such as `02`.

'`{ ... }`' repetition zero or more times, e.g.,

```
digit = "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" |
   "9";
number = digit, { digit };
```

here `number` is a word consisting of 1 or more digits, such as `12`.

'`( ... )`' grouping, e.g.,

```
digit = "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" |
   "9";
signedNumber = ( "+" | "-" ) digit, { digit };
```

here `signedNumber` is a number with a mandatory sign, such as `+5` and `-3`.

'`" ... "`' a terminal string, e.g.,

```
string = "abc"';
```

"`' ... '`" a terminal string, e.g.,

```
string = 'abc';
```

'`(* ... *)`' a comment (* ... *)

```
 (* a binary digit *) digit = "0" | "1"; (* from this all numbers
   may be constructed *)
```

Everything inside the comments are not part of the formal definition.

'`? ... ?`' special sequence, a notation reserved for future extensions of EBNF.

```
codepoint = ?Any unicode codepoint?;
```

'`-`' exception, e.g.,

```
letter = "A" | "B" | "C" | "D" | "E" | "F" | "G" | "H"
   | "I" | "J" | "K" | "L" | "M" | "N" | "O" | "P" | "Q"
   | "R" | "S" | "T" | "U" | "V" | "W" | "X" | "Y" | "Z";
vowel = "A" | "E" | "I" | "O" | "U";
consonant = letter - vowel;
```

here `consonant` are all letters except vowels.

Rules for rewriting EBNF are:

Rule	Description
`s | t` ↔ `t | s`	| is commutative
`r | (s | t)` ↔ `(r | s) | t` ↔ `r | s | t`	| is associative
`(r, s) t` ↔ `r (s, t)` ↔ `r, s, t`	concatenation is associative
`r, (s | t)` ↔ `r, t | r, s`	concatenation is distributive over |
`(r | s), t` ↔ `r, t | r, t`	
`[s | t]` ↔ `[t] | [s]`	
`[[s]]` ↔ `[s]`	`[]` is idempotent
`{{s}}` ↔ `{s}`	`{}` is idempotent

where `r`, `s`, and `t` are production rules or terminals. Precedence for the EBNF symbols are,

Symbol	Description
`[]`, ...	Bracket and quotation mark pairs
`-`	except
`,`	concatenate
`|`	option
`=`	define
`;`	terminator

in order of precedence, such that bracket and quotation mark pairs has higher precedence than `-`.

The proposal allows for identifies that includes space, but often a reduced form is used, where identifiers are single words, in which case the concatenation symbol `,` is replaced by a space. Likewise, the termination symbol `;` is often replaced with the new-line character, and if long lines must be broken, then indentation is used to signify continuation. In this relaxed EBNF, the EBNF syntax itself can be expressed in EBNF as,

```
letter = "A" | "B" | "C" | "D" | "E" | "F" | "G" | "H"
  | "I" | "J" | "K" | "L" | "M" | "N" | "O" | "P" | "Q"
  | "R" | "S" | "T" | "U" | "V" | "W" | "X" | "Y" | "Z"
  | "a" | "b" | "c" | "d" | "e" | "f" | "g" | "h"
  | "i" | "j" | "k" | "l" | "m" | "n" | "o" | "p" | "q"
  | "r" | "s" | "t" | "u" | "v" | "w" | "x" | "y" | "z";
digit = "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9";
symbol = "[" | "]" | "{" | "}" | "(" | ")" | "<" | ">"
  | "?" | "'" | '"' | "=" | "|" | "." | "," | ";";
underscore = "_";
space = " ";
newline = ?a newline character?;
identifier = letter  { letter | digit | underscore };
character = letter | digit | symbol | underscore;
string = character  { character };
terminal = "'"  string  "'" | '"'  string  '"';
rhs = identifier
  | terminal
  | "["  rhs  "]"
  | "{"  rhs  "}"
  | "("  rhs  ")"
  | "?"  string  "?"
  | rhs  "|"  rhs
  | rhs  ","  rhs
  | rhs  space  rhs; (*relaxed ebnf*)
rule = identifier  "="  rhs ";"
  | identifier  "="  rhs newline; (*relaxed ebnf*)
grammar = rule { rule };
```

Here the comments demonstrate, the relaxed modification. Newline does not have an explicit representation in EBNF, which is why we use `?  ?` brackets