The basic form of notation for a context-free grammar is the **Backus-Nour Form**.
These map non-terminal identifiers to their options for representation.
Consider the example BNF grammar:

S → (S)

S → a

A similar grammar can be expressed more efficiently in the form of a regular expression:

S → ab*

But sadly, regular expressions cannot be used on just any grammar.
For example, in attempting to emulate the BNF grammar above, the closest we can come is (*a)*.
This does not account for matching parenthesis count!
This is because regular expressions cannot handle nested recursion.

Let's try to extend REGEX by merging with standard notation.
Consider the example of email-RFC5322. This requires all emails contain a line similar to the following:

Message-ID:<eggert."93-5xx27"@cs.ucla.edu>

This uniquely identifies any email message to allow for easy filtering of duplicates.
The grammar takes the form of an extended BNF.
It uses / for OR, * for repeatable nonterminals, () for optional, and " for terminals:

msg-id = '<' dot-atom-text '@' id-right '>'
id-right = dot-atom-text / no-fold-literal
dot-atom-text = 1*atext (. 1*atext)
no-fold-literal = '[' *dtext ']'
atext = ALPHA | DIGIT | '!' | '#' | ...
dtext = %33-90 / %94-126 (the set of printable characters excluding '[', '/', and])

Since this is an EBNF grammar, it must be expressible in BNF form.
Consider the fourth rule in the list. It can be expressed in BNF form as the set of

no-fold-literal = '[' dtexts ']'
dtexts =
dtexts = dtext dtexts

We could perform similar expansion for the rest of the rules, but this would get unwieldy quick.
We thus need a more efficient form for expressing grammars.
Luckily, the International Standards Organization established a standard called **ISO BNF**

ISO-EBNF (a top-down view):

"terminal symbol"
[optional]
repeat zero or more times
*repeated
(* comment *)
A - B (* A except B, set division *)
A,B (* concatenate A and B *)
A|B (* A or B *)
A = BC; (* grammar rule *)

The gave an English summary of their grammar for grammars, then demonstrated it with their own notation:

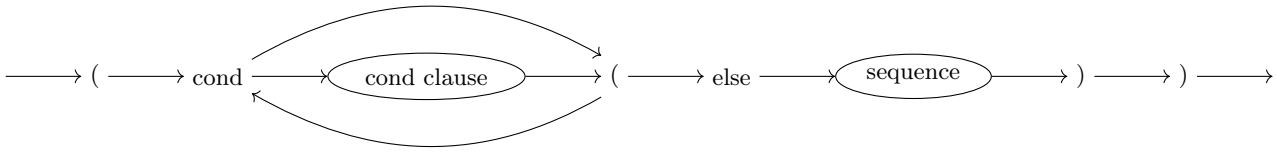
syntax = syntax-rule , syntax-rule;
syntax-rule = meta-id , '=' , defns-list , ';' ;
defns-list = defn , '|' , defn;
defn = term , ',' , term
term = factor , ['-' , exception];
exception = factor;
factor = [indeger , '*']

Note that exception is unnecessary, but was included for clarity.

Some grammars, like SQL data typing extensions, are best expressed through **syntax charts**.
Here is an impractically small example (in SQL notation):

<cond> *rightarrow* (<cond clause>*) | (cond <cond clause>* (else <sequence>))

This is equivalent to the following diagram:



We can use this to program a parser:

```
void cond(void) {
        eat("(");
        eat("cond");
        while (cond-clause()) continue;
        eat(")");
        ...
}
```

It would be nice to be able to express this as a finite state machine, but there is recursion.
Thus we would need to use a push-down automaton to express this.
Grammars progress into programs along grammar → design → parser code
But code can be buggy, and bugs in the parser would be bugs in the grammar, adding to complication!

Grammar Issues

1. Useless Rules
   S → Sa
   S → B
   C → Cd
   While this is a valid grammar, it defines a pointless subroutine, wasting space.
   These errors can further be broken down into

   (a) Nonterminal used but not defined
   S → Sa
   S → Bc
   S → d
   The second grammar rule above is analogous to calling an undefined subroutine.

   (b) Nonterminal defined but not used
   S → Sa
   S → d
   B → aS
   The third grammar rule above is analogous to defining an uncalled function.

   While these are both valid grammars, they are space-inefficient.

2. Uncaptured Constraints Consider the following subset of English:
   S → NP VP
   NP → N
   NP → Adj NP
   VP → V
   VP → VP Adv
   for tokens N, V, Adj, Adv
   This grammar has errors: for instance, it allows the sentence "Dog Bark".
   To account for subject-verb agreement, we must complicate the grammar.
   We would double the size of our grammar just to solve a boolean issue!

There are other types of errors, which would each cause exponential growth.
We can demonstrate some of these issues in OCaml:

```
let x=5 in x*y;; (* undefined y *)
5 / 3.0;; (* error: bad type float *)
```

How do we fix these? WE DON'T. DON'T DO IT.