Syntax (vs. Semantics)

- syntax - form
- semantics - meaning
- specification - what we want formally
- implementation - how to get it (code)
- ex:
  - "colorless green ideas sleep furiously" - good syntax but bad semantics
  - "Ireland has leprechauns galore" - semantically correct but the adj. 'galore' is in the wrong place
    - can't be done with code, will not compile (usually)
  - "time flies"
    - time (noun); flies (verb) - declarative
    - time (verb); flies (noun) imperative
    - Both are correct which causes ambiguity
- C++ ambiguity:

```
int a, b;
...
return a + + + + + b;

possible interpretations:
((a++)++))b);
(a++)+(++b);
a+(++(++b));
```

- We cant be sure how it will be interpreted
- Good syntax:
  - its what we know already (inertia)
  - its simpler
  - its unambiguous
- its readable
- its writable
- its redundant
  - counterexample:

```
[I-N)*int
all else float
```

- grammar is a syntax specification
- tokens:
  - individual unit of a language
    - ex: words in English are tokens, but not entirely because I can make up words
  - built out of characters
    - character - a single keystroke
  - a program is a byte sequence, that represents a character sequence, which represents a token sequence
  - sequence of chars

```
ex:
char foo[] = "hello';
```

  - ^ not a proper token, unterminated string

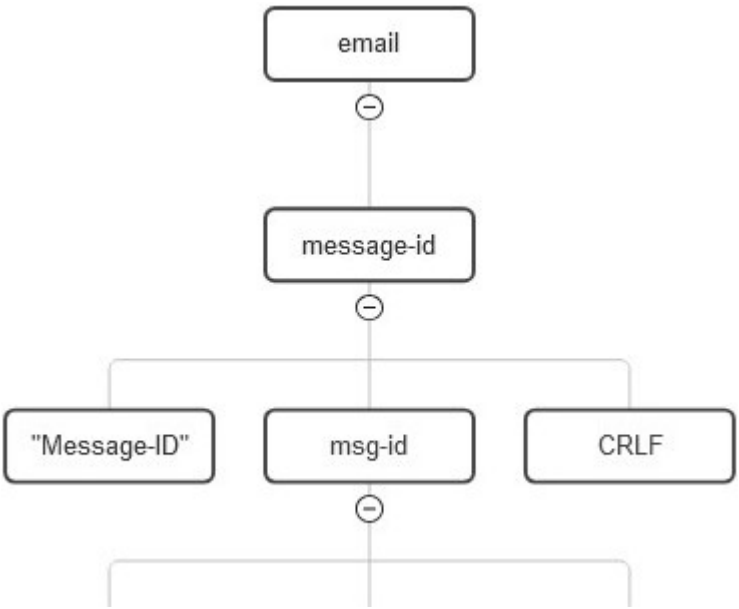
Internet RFC 5322

- uses EBNF
- format:
  - header
  - body
- every header has a message id

```
EBNF:
message-id = "Message-ID:" msg-id CRLF
msg-id = "<" dot-atom-text "@" id-right ">"
id-right = dot-atom-text | no-fold-literal
no-fold-literal = "[" *dtext "]"

BNF example:
id-right = dot-atom-text
id-right = no-fold-literal
sdtext =
sdtext = dtext sdtext
no-fold-literal = "[" sdtext "]"
```

```
dtext = %od33-90 | %od94-126
dot-atom-text = 1*atext *("." 1*atext)
atext = ALPHA | DIGIT | "!" | "#" | ...
-----------------------------------------------------------
Tree: (N) - nonterminal, (T) - terminal
```



- notation:
  - | - OR; not a nonterminal symbol, meta notation added with EBNF for convenience
  - repeat 0+ times
  - 1* - repeat 1+ times
  - *(. . .) - 0+ of the parentheses body
- grammar rule (BNF): LHS RHS
  - LHS - nonterminal symbol
  - RHS - finite sequence of symbols (can be empty)
- Why are grammars easy to parse?
  - You can count parentheses
  - so you can use grep like:

```
#!bin/sh
pat = '_____'
grep "$pat" filp
```

- BNF - context free grammar

- s = "(" s ")"
  - can't turn into regex, can't count number of parentheses
- s = "a"
  - can only be a regex if there is no recursion

XML 1.1

- a good way to represent data
- can't write a regex to match XML data

```
lowercase: recursively defined nonterminals e.g. element
LHS := RHS

| - OR
(P) - P
[a-z] - |"a"|"b"|...|"c"|
[^ ] - negation
X? - 0 or 1 X's
P* - 0+ P's
```