Recursive-Descent Parsing

First, a digression on lexing

Let’s assume the **get-token** function will give me the next token

```
(define lex
 (lexer
    ; skip spaces:
    [#\space     (lex input-port)]
    ; skip newline:
    [#\newline   (lex input-port)]

    [#\+         'plus]
    [#\-         'minus]
    [#\*         'times]
    [#\/         'div]

    [(:: (:? #\-) (:+ (char-range #\0 #\9)))
      (string->number lexeme)]
    ; an actual character:
    [any-char    (string-ref lexeme 0)]))
```

Assume current token is `curtok`

`(accept c)` matches character c

```
(define curtok (next-tok))

(define (accept c)
  (if (not (equal? curtok c))
      (raise 'unexpected-token)
      (begin
        (printf "Accepting ~a\n" c)
        (set! curtok (next-tok)))))
```

L eft to right

L eft derivation

1 token of lookahead

Let’s say I want to parse the following grammar

S -> aSa | bb

First, a few questions

```
S -> aSa | bb
```

Is this grammar ambiguous?

If I were matching the string **bb,** what would my derivation look like?

If I were matching the string **abba**, what would my derivation look like?

First, a few questions

S -> aSa | bb

Key idea: if I look at the next input, at most one of these productions can “fire”

If I see an **a** I **know** that I **must** use the first production

If I see a b, I know I must be in second production

This is called a **predictive** parser. It uses lookahead to determine which production to choose

(My friend Tom points out that **predictive** is a dumb name because it is really “determining”, no guess)

In this class, we'll restrict ourselves to grammars that require only **one** character of lookahead

Generalizing to k characters is straightforward

I need two characters of lookahead

S -> aaS | abS | c

I need three characters of lookahead

S -> aaaS | aabS | c

I need four characters of lookahead

S -> aaaaS | aaabS | c

...

Slight transformation..

```
S  ->  A  |  B
A  ->  aSa
B  ->  bb
```

Slight transformation..

S -> A | B
A -> aSa
B -> bb

Now, I write out **one function** to parse **each** nonterminal

```
S -> A | B
A -> aSa
B -> bb
```

Intuition: when I see **a**, I call `parse-A`

when I see **b**, I call `parse-B`

```
(define (parse-A)
  (match curtok
    [#\a
     (begin
       (accept #\a)
       (parse-A)
       (accept #\a))]
    [#\b (parse-B)]))
```

```
(define (parse-B)
  (begin
    (accept #\b)
    (accept #\b)))
```

Livecoding this parser in class

Three parsing-related pieces of trivia

FIRST(A)

FIRST(A) is the **set** of terminals that
could occur **first** when I recognize A

NULLABLE

Is the set productions which could generate ε

FOLLOW(A)

FOLLOW(A) is the set of terminals that appear immediately to the right of A in some form

Why learn these?

A: They help your intuition for building parsers
(as we’ll see)

What is FIRST for each nonterminal

S -> A | B
A -> aAa
B -> bb

What is NULLABLE for the grammar

What is FOLLOW for each nonterminal

More practice…

```
E  → TE'
E' → +TE'
E' → ε
T  → FT'
T' → *FT'
T' → ε
F  → (E)
F  → id
```

What is FIRST for each nonterminal

What is NULLABLE for the grammar

What is FOLLOW for each nonterminal

We use the **FIRST** set to help us design our recursive-descent parser!

LL(1)

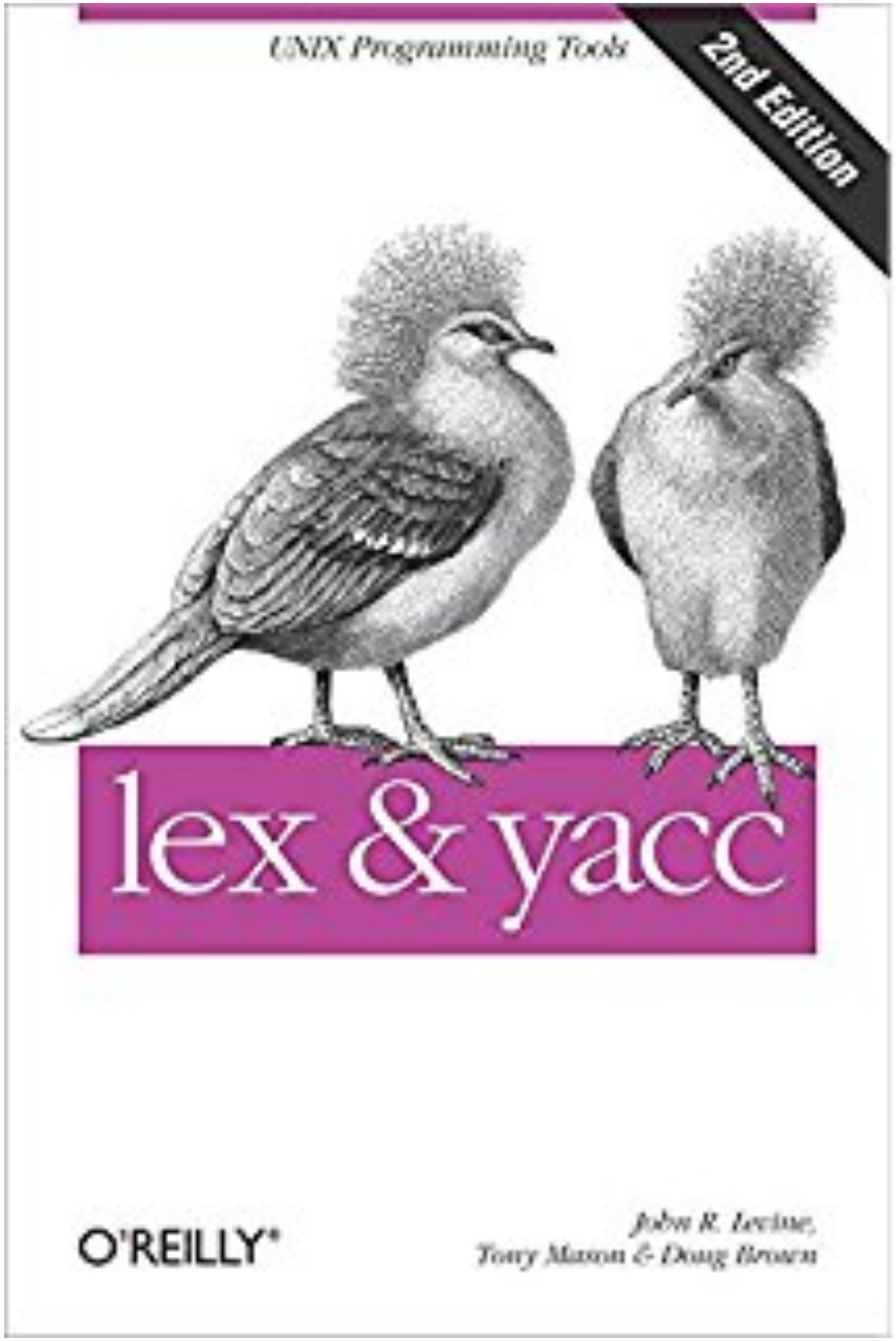
A grammar is LL(1) if we only have to look at the **next** token to decide which production will match!

I.e., if S -> A | B, FIRST(A) ∩ FIRST(B) must be empty

Recursive-descent is called **top-down** parsing because you build a parse tree from the root down to the leaves

There are also **bottom-up** parsers,
which produce the rightmost derivation

Won’t talk about them, in general they’re
impossibly-hard to write / understand, easier to use


UNIX Programming Tools
2nd Edition
lex & yacc
O'REILLY®
John R. Levine,
Tony Mason & Doug Brown

Basically everyone uses lex and yacc to write real parsers

Recursive-descent is easy to implement, but requires lots of messing around with grammar

More practice with parsers

This one is more tricky!!

```
Plus -> num MoreNums
MoreNums -> + num MoreNums | ε
```

How would you do it?

(**Hint:** Think about NULLABLE)

Code up collectively….

```
(define (parse-Plus)
  (begin
    (parse-num)
    (parse-MorePlus)))

(define (parse-MorePlus)
  (match curtok
    ['plus
     (begin
       (accept 'plus)
       (parse-num)
       (parse-MorePlus))]
    ['eof (void)]))
```

Key rule: At each step of the way, if I see some token next, what rule production **must** I choose

Now yet another….

This will use the intuition from FOLLOW

```
Add -> Term MoreTerms
MoreTerms -> + Term MoreTerms
MoreTerms -> ε
Term -> num MoreNums
MoreNums -> * num MoreNums | ε
```

Consider how we would implement `MoreTerms`

```
Add -> Term MoreTerms
MoreTerms -> + Term MoreTerms
MoreTerms -> ε
Term -> num MoreNums
MoreNums -> * num MoreNums | ε
```

If you’re at the beginning of `MoreTerms` you **have** to see a +

```
Add -> Term MoreTerms
MoreTerms -> + Term MoreTerms
MoreTerms -> ε
Term -> num MoreNums
MoreNums -> * num MoreNums | ε
```

If you’ve just seen a + you have to see FIRST(Term)

```
Add -> Term MoreTerms
MoreTerms -> + Term MoreTerms
MoreTerms -> ε
Term -> num MoreNums
MoreNums -> * num MoreNums | ε
```

After Term you recognize something in FOLLOW(Term)

```
Add -> Term MoreTerms
MoreTerms -> + Term MoreTerms
MoreTerms -> ε
Term -> num MoreNums
MoreNums -> * num MoreNums | ε
```

Because MoreTerms is NULLABLE, have to account for null

```
Add -> Term MoreTerms
MoreTerms -> + Term MoreTerms
MoreTerms -> ε
Term -> num MoreNums
MoreNums -> * num MoreNums | ε
```

Code up collectively….

Let’s say I want to generate an AST

Model my AST…

```
(struct add (left right) #:transparent)
(struct times (left right) #:transparent)
```

More Recursive-descent practice…

Write recursive-descent parsers for the following….

A grammar for S-Expressions

Parsing mini-Racket / Scheme

```
datum ::= number
        | string
        | identifier
        | ‘SExpr
SExpr ::= (SExprs)
        | datum
SExprs ::= SExpr SExprs
        | ε
```

```
S -> a C H | b H C
H -> b H | d
C -> e C | f C
```

```
E -> A
E -> L
A -> n
A -> i
L -> ( S )
S -> E S’
S’ -> , S
S’ -> ε
```

So far, I’ve given you grammars that are amenable to LL(1) parsers…

(Many grammars are **not**)

(But you can manipulate them to be!)

What about this grammar?

```
E -> E - T | T
T -> number
```

This grammar is **left recursive**

```
E -> E - T | T
T -> number
```

What happens if we try to write recursive-descent parser?

This grammar is **left recursive**

```
E -> E - T | T
T -> number
```

We really **want** this grammar, because it corresponds to the **correct** notion of associativity

```
E -> E - T | T
T -> number
```

5 - 3 - 1

Infinite loop!

```
E -> E - T | T
T -> number
```

```
5 - 3 - 1
```

A recursive descent parser will first call `parse-E`

And then crash

```
E -> E - T | T
T -> number
```

```
5 - 3 - 1
```

Draw the **rightmost derivation** for this string

If we could only have the **rightmost** derivation, our problem would be solved

The problem is, a recursive-descent parser needs to look at the **next input immediately**

Recursive descent parsers work by looking at the next token and making a decision / prediction

Rightmost derivations require us to delay making choices about the input until later

As humans, **we** naturally guess which derivation to use (for small examples)

Thus, LL(k) parsers cannot generate rightmost derivations :(

We can **remove left recursion**

```
E -> E - T | T
T -> number
```

Factor!

```
E  -> T E’
E’ -> - T E’
E’ -> ε
```

In general, if we have

```
A -> Aa | bB
```

Rewrite to…

```
A -> bB A'
A' -> a A' | ε
```

Generalizes even further

https://en.wikipedia.org/wiki/LL_parser#Left_Factoring

But this still doesn’t give us what we want!!!

```
E   -> T E’
E’  -> - T E’
E’  -> ε
```

```
E -> T E’
  -> T - T E’
  -> T - T - T E’
  -> T - T - T
```

So how do we get left associativity?

Answer: Basically, hack in implementation

```
Sub -> num Sub’
Sub’ -> + num Sub’ | epsilon
```

Is basically...

```
Sub -> num Sub’ (+ num)*
```

Intuition: treat this as while loop, then when building parse tree, put in left-associative order

```
Sub -> num Sub’ (+ num)*
```

```
Sub -> num Sub’
Sub’ -> + num Sub’ | epsilon
```

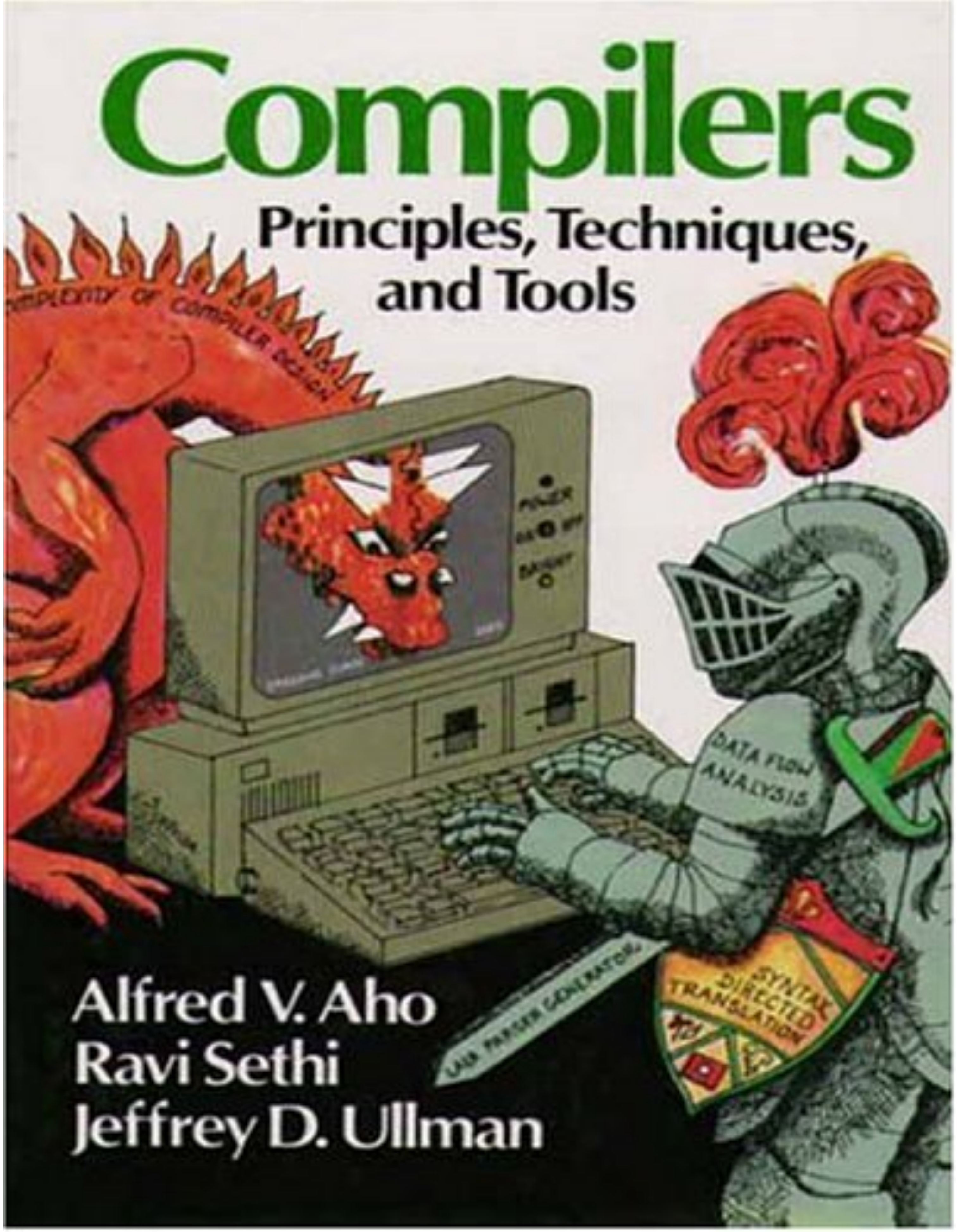
If you want to get **rightmost** derivation, you need to use an LR parser

```
input:    /* empty */
        | input line
;

line:     '\n'
        | exp '\n'  { printf ("\t%.10g\n", $1); }
;

exp:      NUM             { $$ = $1;         }
        | exp exp '+'     { $$ = $1 + $2;    }
        | exp exp '-'     { $$ = $1 - $2;    }
        | exp exp '*'     { $$ = $1 * $2;    }
        | exp exp '/'     { $$ = $1 / $2;    }
      /* Exponentiation */
        | exp exp '^'     { $$ = pow ($1, $2); }
      /* Unary minus    */
        | exp 'n'         { $$ = -$1;        }
;
```

Parsing is lame, it’s 2017



Compilers

Principles, Techniques, and Tools

Alfred V. Aho
Ravi Sethi
Jeffrey D. Ullman

If you can, just use something like JSON / protobufs / etc…

Inventing your own format is probably wrong

For small / prototypical things, recursive-descent

For real things, use yacc / bison / ANTLR