# Parserpalloza

Today, we'll implement a **few** recursive-descent parsers in groups

You'll have to figure this out yourself in Lab 5

I'll post this code online after we're done

Take 2 minutes to find 1-2 group mates (you can work by yourself, too, but if you do you have to commit to programming, not sitting there)

Everyone must touch the keyboard once today

If you get stuck, ask the group to your left / right first, **not me**

If two groups stuck, I will help

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

# 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

S -> A | B

A -> aAa

B -> bb

**What is FIRST for each nonterminal**

**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**

Let’s say I want to parse the following grammar

```
A -> aAa | bb
```

```
A -> aAa | B
B -> bb
```

To parse A, I check for either

FIRST(aAa)

FIRST(B)

```
A -> aAa | B
B -> bb
```

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

```
A -> aAa | B
B -> bb
```

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

A -> aAa | B

B -> bb

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

A -> aAa | B

B -> bb

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

```
A -> aAa | B
B -> bb
```

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

```
A -> aAa | B
B -> bb
```

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

# *A general comment*

You can often “follow your nose” for writing recursive descent parsers

In **this class** we want you to follow this **cookbook** method. Make sure your parser follows the grammar

(If you implement a parser for a different grammar that still works you will **still lose points in lab**)

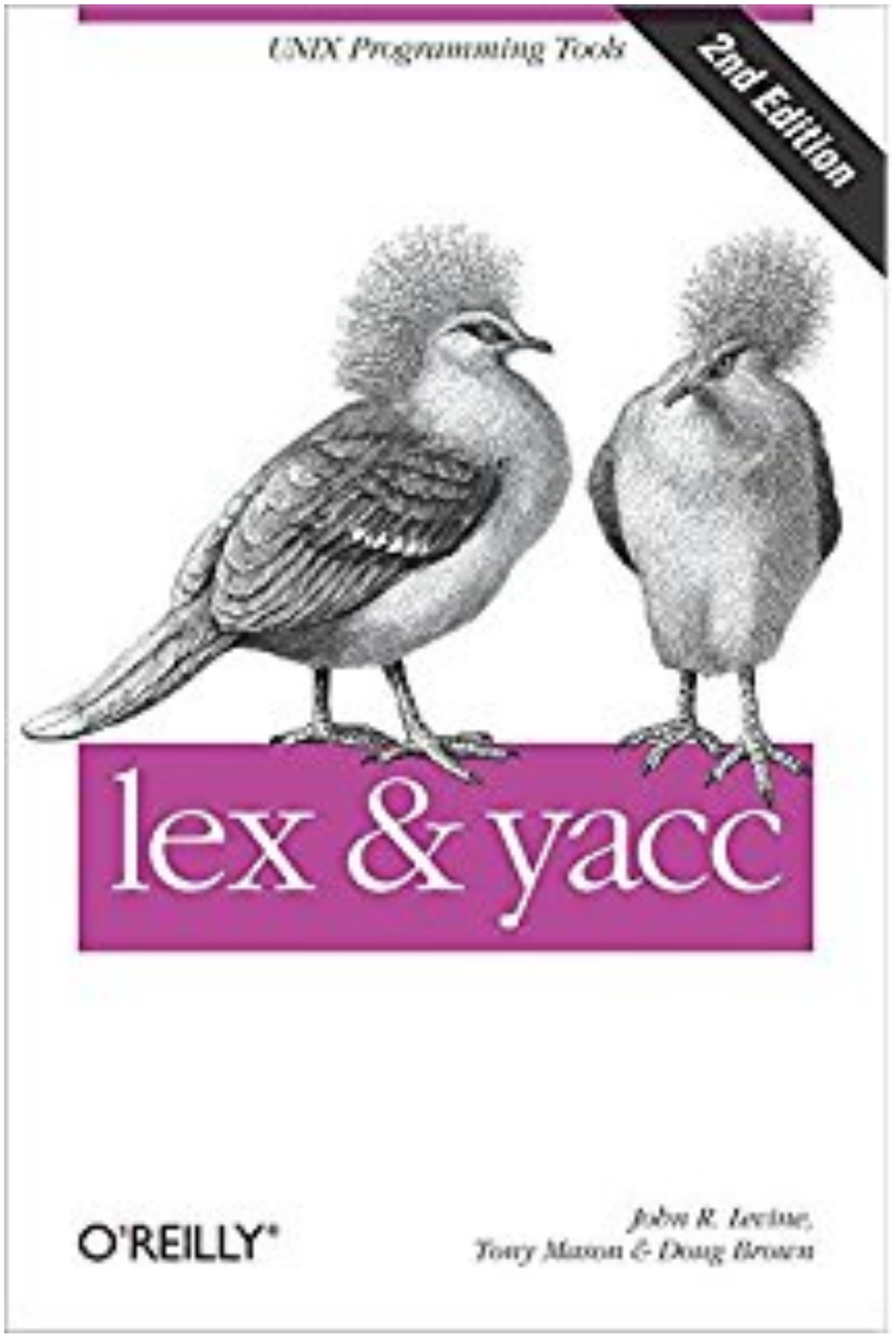
***Comment* each production** (I *didn’t* do in slides for space)

Challenge 1: Produce 2 strings in the language and one string out of the language

Demonstrate how to parse them (or show parsing error)

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 **messing around with grammar**

# More practice with parsers

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

How would you do it?

(**Hint:** Think about NULLABLE)

Let’s think through this one on the board in pseudo-code

Plus -> num MoreNums

MoreNums -> + num MoreNums | ε

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

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

Yet another (this one in the C++ files)

```
START -> E ε
E -> number
E -> identifier
E -> ( E_IN_PARENS )
E_IN_PARENS -> OP E E
OP -> +|-|*
```

Now yet another….

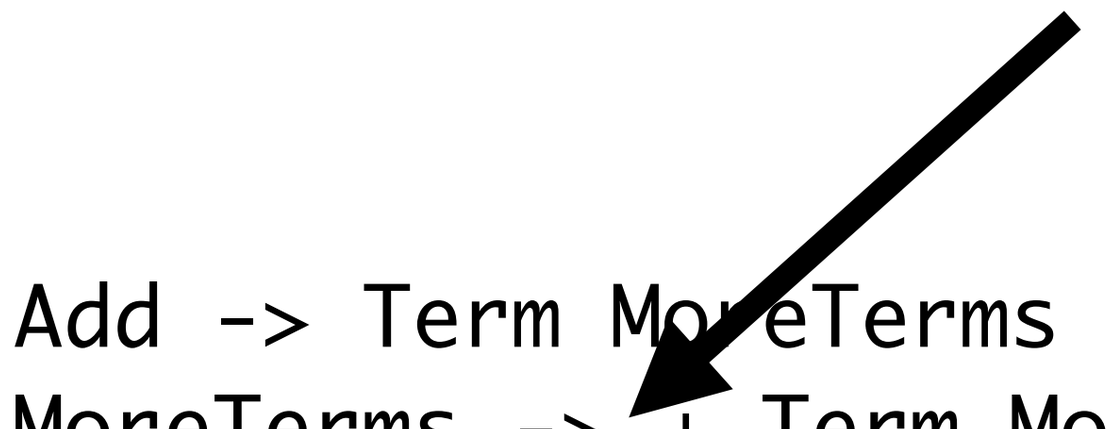
This will use the intuition from FOLLOW

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

Consider how we would implement `MoreTerms`

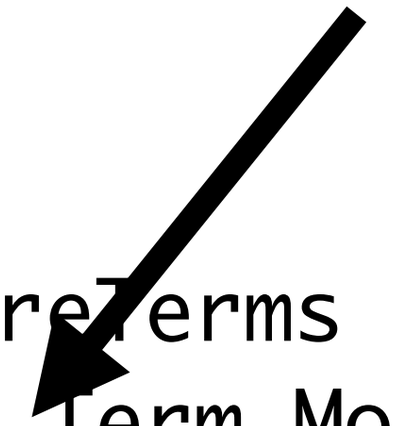
```
Add -> Term MoreTerms
MoreTerms -> + Term MoreTerms
MoreTerms -> ε
Term -> num MoreNums
MoreNums -> * num MoreNums | ε
```

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)

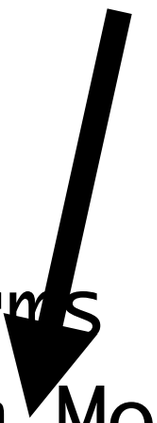


```
Add -> Term MoreTerms
MoreTerms -> + Term MoreTerms
MoreTerms -> ε
Term -> num MoreNums
MoreNums -> * num MoreNums | ε
```

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)
```

# Model my AST…

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

**Now, modify your parser to generate this AST**

# More Recursive-descent practice…

(We’ll skip this for now and you can do it by yourself)

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

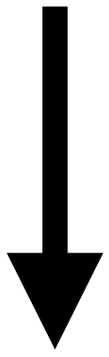
As humans, **we** naturally guess which derivation to use (for small examples)

**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 -> Aα | bB
```

Rewrite to…

```
A -> bB A'
A' -> α 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
```

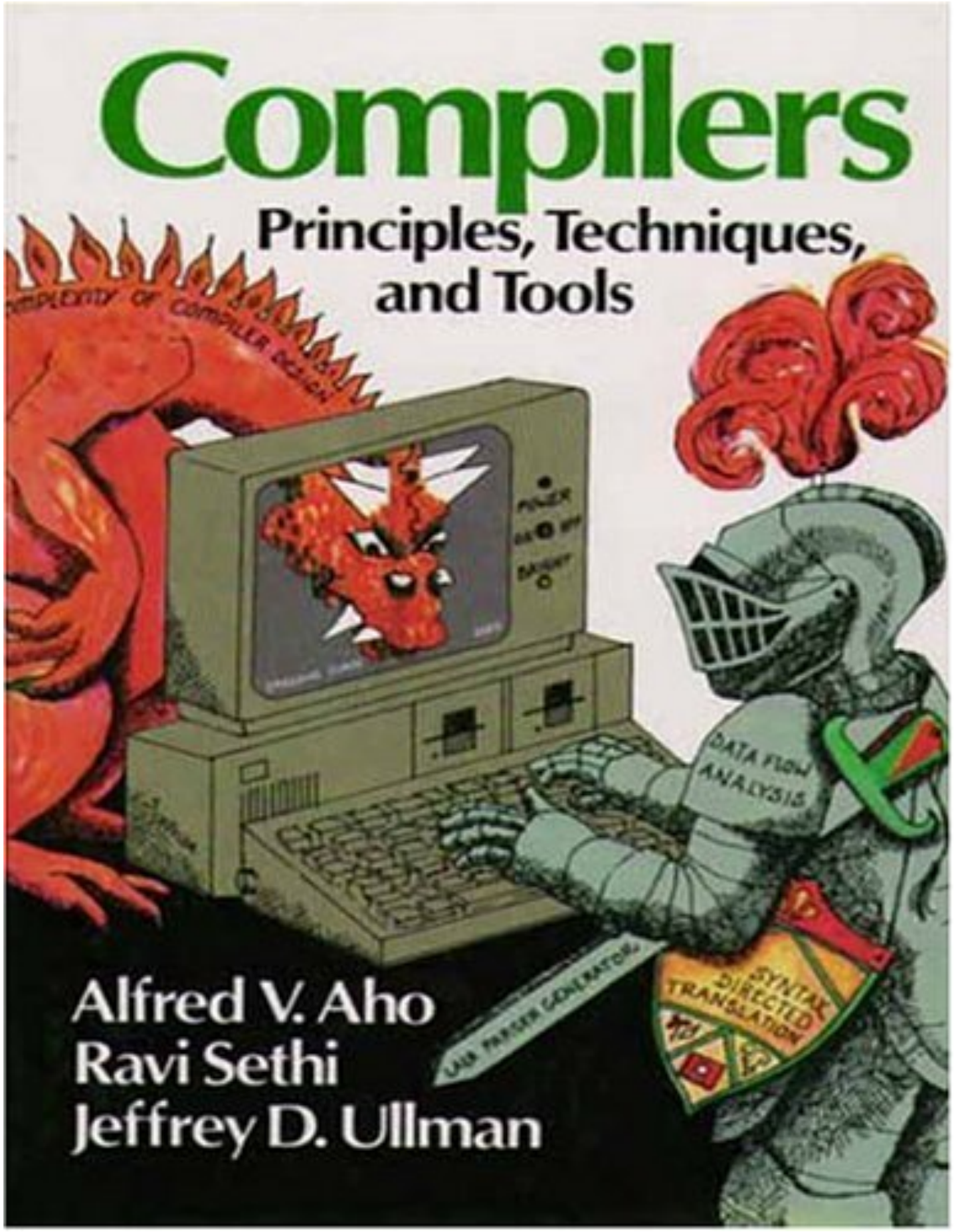
If you want to get **rightmost** derivation, you need to use an LR parser

```
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