Grammars and Parsing

Forth mini-homework…

If there is a number on the stack, and we enter dup dup * *, what will be on the stack?

If there are three numbers on the stark, and we enter

`over -1 * over -1 * + + + *`,

what will be on the stack?

If we assume there are 2 values on the top of the stack, and we want to replace them with the sum of their squares, what would we type?

- If we assume there are at least 3 values on the top of the stack, and we want to replace the top three with two values, so that the new top is one less than the old top, and the number right below it is the product of the other two we removed, what should we type?

```
: iter 1 - rot rot * swap ;
```

If commands in FORTH

```
: maybeadd1 dup 42 = invert if 1 + then ;
23  ok
maybeadd1  ok
.s <1> 24  ok
drop  ok
42  ok
maybeadd1  ok
.s <1> 42  ok
```

An if will be true if -1 (true) is on the stack

```
if <handle-true> (else <handle-else>)? then
```

```
: maybeadd1 if 1 + then ;
23 -1  ok
maybeadd1
```

Grammars and Parsing

This allows us to write **interpreters**

```
(define my-tree
  '(+ 1 (* 2 3)))

(define (evaluate-expr e)
  (match e
    [`(+ ,e1 ,e2) (+ (evaluate-expr e1) (evaluate-expr e2))]
    [`(* ,e1 ,e2) (* (evaluate-expr e2) (evaluate-expr e2))]
    [else e]))
```

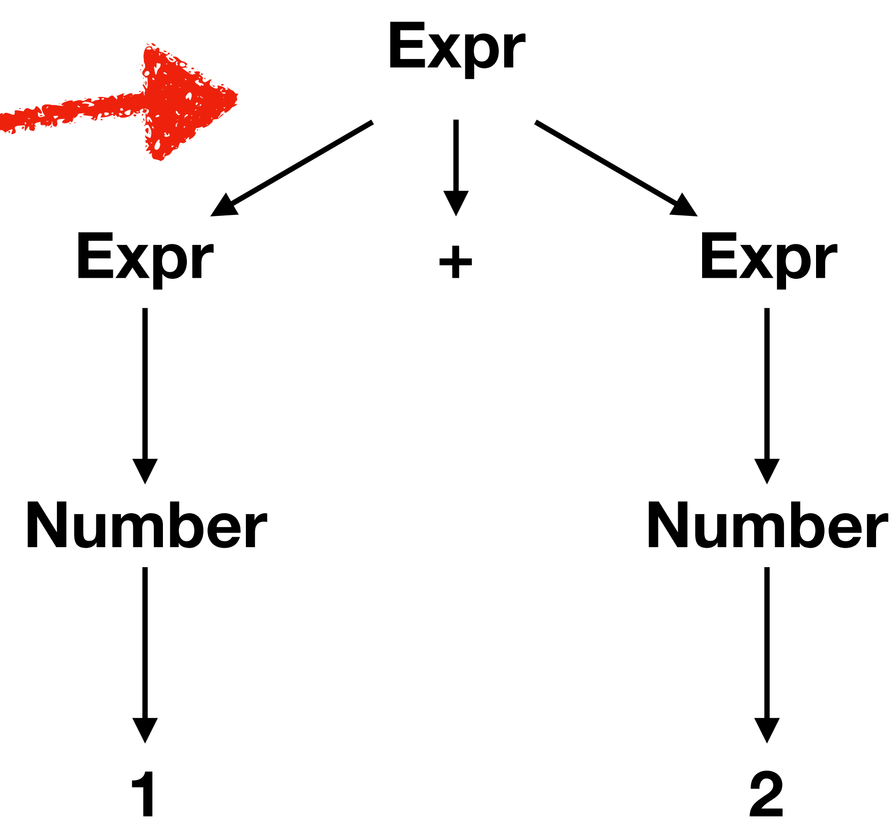
```
Expr -> number
Expr -> Expr + Expr
Expr -> Expr * Expr
```

```
1 + 2 * 3
```

```
Expr
-> Expr + Expr
-> Expr + Expr * Expr
-> number + Expr * Expr
-> number + number * Expr
-> number + number * number
```

```
Expr
-> Expr * Expr
-> Expr + Expr * Expr
-> number + Expr * Expr
-> number + number * Expr
-> number + number * number
```

```
Expr
-> Expr + Expr
-> number + Expr
-> number + number
-> 1 + number
-> 1 + 2
```



This parse tree is a **hierarchical representation** of the data

A **parser** is a program that automatically generates a parse tree

A parser will generate an **abstract syntax tree** for the language

Exercise: draw the parse trees for the following derivations

```
Expr
-> Expr + Expr
-> Expr + Expr * Expr
-> number + Expr * Expr
-> number + number * Expr
-> number + number * number
```

```
Expr
-> Expr * Expr
-> Expr + Expr * Expr
-> number + Expr * Expr
-> number + number * Expr
-> number + number * number
```

BNF

(Bakus-Naur Form)

```
<Expr> ::= <number>
<Expr> ::= <Expr> + <Expr>
<Expr> ::= <Expr> * <Expr>
```

Slightly different form for writing CFGs, superficially different

(BNF renders nicely in ASCII, but no huge differences)

I write colloquially in some mix of BNF and more math style

Two kinds of derivations

Leftmost derivation: The leftmost nonterminal is expanded first at each step

Rightmost derivation: The rightmost nonterminal is expanded first at each step

Work in groups

```
G -> GG
G -> a
```

Draw the **leftmost derivation** for…

aaa

Draw the **rightmost derivation** for…

aaa

```
G -> G + G
G -> G / G
G -> number
```

Draw a leftmost derivation for…

```
1 / 2 / 3
```

Now draw *another* leftmost derivation

Draw the parse trees for each derivation

What does each parse tree mean?

A grammar is **ambiguous** if there is a string with **more than one** leftmost derivation

(Equiv: has more than one parse tree)

Generally, we’re going to want our grammar to be **unambiguous**

```
G -> G + G
G -> G / G
G -> number
```

There’s another problem with this grammar (OOO)

We need to tackle **ambiguity**

Idea: introduce extra nonterminals that force you to get left-associativity

(Also force OOP)

```
Add -> Add + Mul | Mul
Mul -> Mul / Term | Term
Term -> number
```

Write derivation for 5 / 3 / 1

Draw the parse tree for 5 / 3 / 1

```
Add -> Add + Mul | Mul
Mul -> Mul / Term | Term
Term -> number
```

This grammar is **left recursive**

```
Add -> Add + Mul | Mul
Mul -> Mul / Term | Term
Term -> number
```

A grammar is left-recursive if any nonterminal A has a production of the form A -> A…

```
Add -> Add + Mul | Mul
Mul -> Mul / Term | Term
Term -> number
```

This will turn out to be bad for one class of parsing algorithms

Recursive-Descent Parsing

Recursive-descent parsing is a simple parsing algorithm

First, a digression on lexing

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

Let’s say I want to parse the following grammar

$$S \to aSa \mid bb$$

First, a few questions

$$S \to aSa \mid bb$$

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

Slight transformation..

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

Slight transformation..

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

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

FIRST(A)

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

Note: ε **cannot** be a member of FIRST because it is not a character

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

$$S \rightarrow A \mid B$$
$$A \rightarrow aAa$$
$$B \rightarrow bb$$

What is FIRST for each nonterminal

What is NULLABLE for the grammar

What is FOLLOW for each nonterminal

More practice…

$E \rightarrow TE'$

$E' \rightarrow +TE'$

$E' \rightarrow \varepsilon$

$T \rightarrow FT'$

$T' \rightarrow *FT'$

$T' \rightarrow \varepsilon$

$F \rightarrow (E)$

$F \rightarrow id$

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 S

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

I **look at the next token**, and I have *two possible choices*

If I see an **a**, I must parse an A

If I see a **b**, I must parse a B

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

Livecoding this parser in class

The recursive-descent parsers we will cover are generally called **predictive** parsers, because they use **lookahead** to predict which production to handle next

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

L eft to right

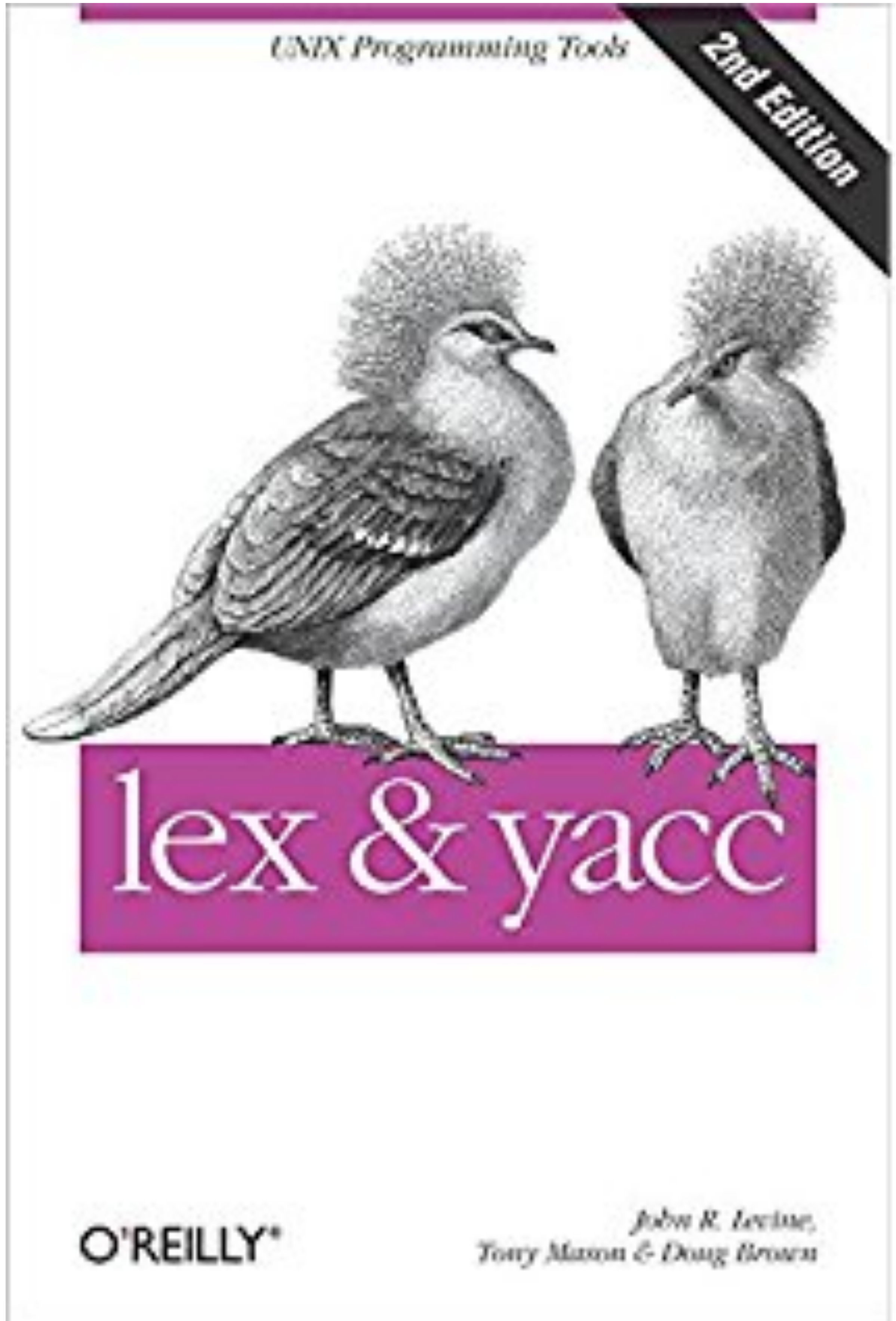
L eft derivation

1 token of lookahead

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

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?

Infinite loop!

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, stupid hack in implementation

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

Is basically…

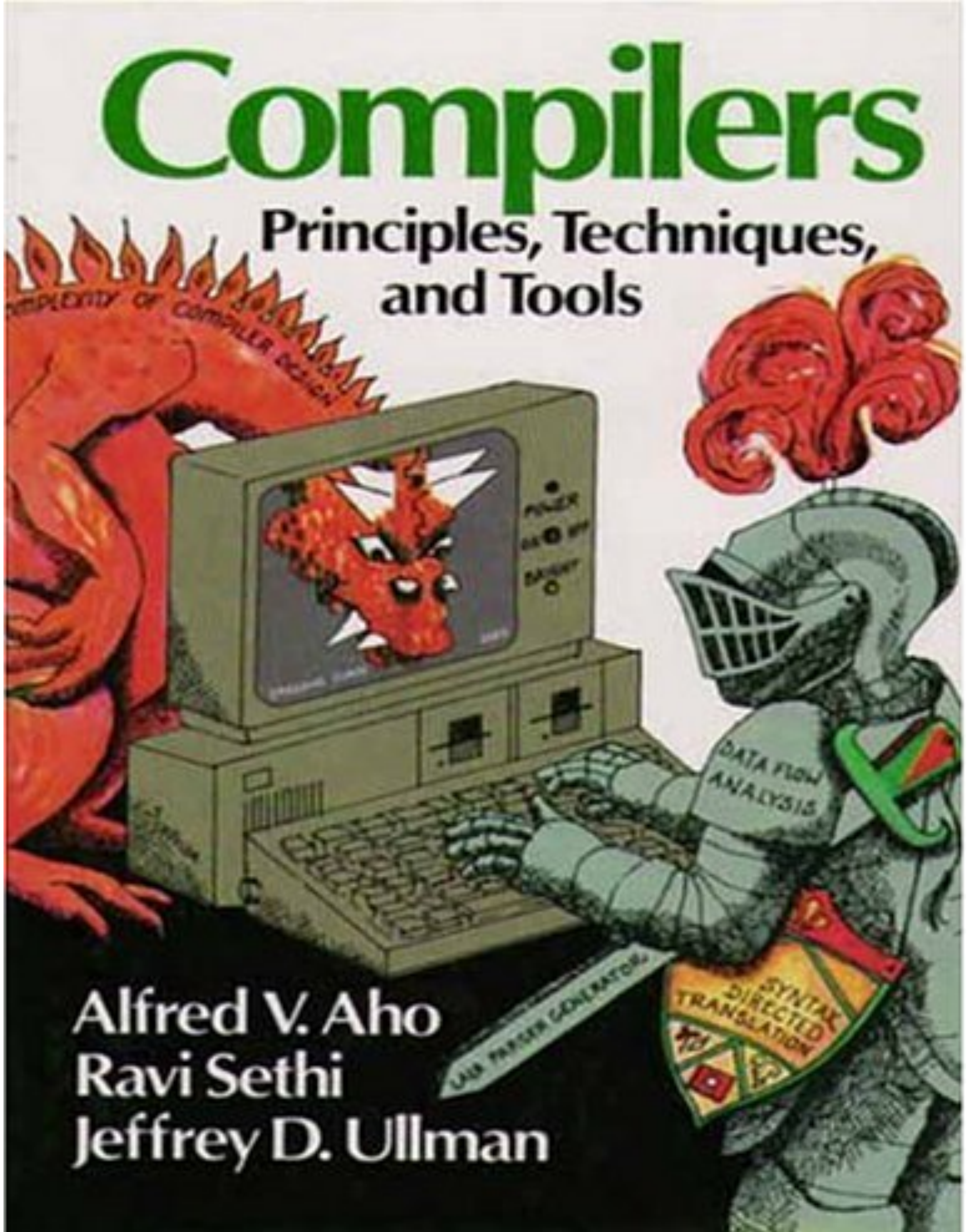
```
Sub -> num Sub’ (+ num)*
```

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
```

Parsing is lame, it’s 2017


Compilers
Principles, Techniques, and Tools
COMPLEXITY OF COMPILER DESIGN
DATA FLOW ANALYSIS
LALR PARSER GENERATOR
SYNTAX DIRECTED TRANSLATION
Alfred V. Aho
Ravi Sethi
Jeffrey D. Ullman

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

Inventing your own format is stupid

For small / prototypical things, recursive-descent

For real things, just use yacc