Write a parser today!

by Scott Vokes

vokes . s @ gmail . com

What's a parser?

First off, what IS a parser?

A parser is something that reads in a stream of symbols and assembles them into a structure, according to a grammar.

“Katherine Hawkins-Lee Metcalfe Volunteer”
Photo by USFWS Mountain-Prairie, CC-BY, scaled.

Looking at a diagram, trying to reassemble a skeleton out of bones.

Parse tree

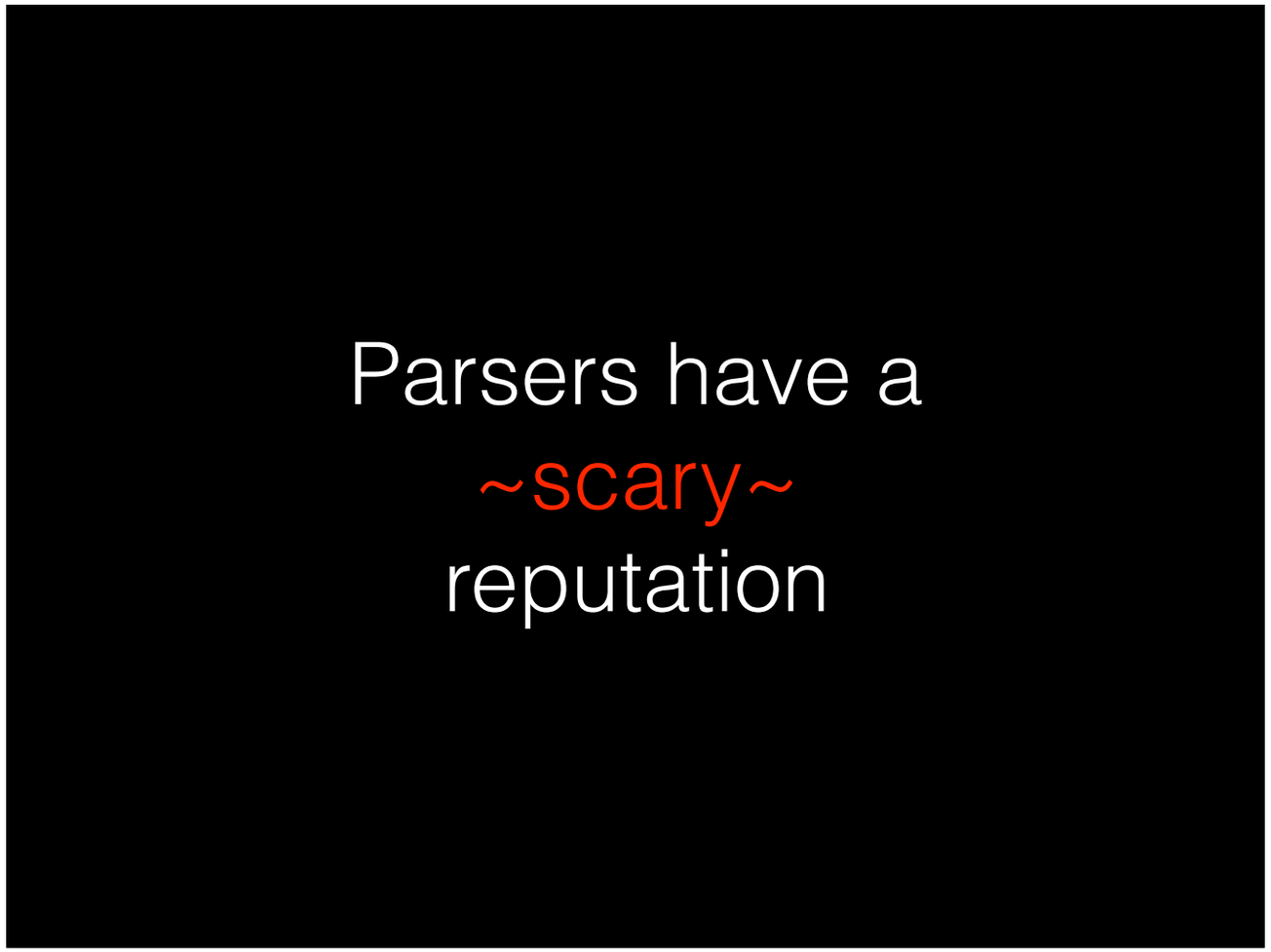
These structures are called “parse trees”, and…

“Correct Syntax Tree for a German Sentence”
Photo by Gunnar Grimnes, CC-BY.

It’s like diagramming sentences.

Structure in data

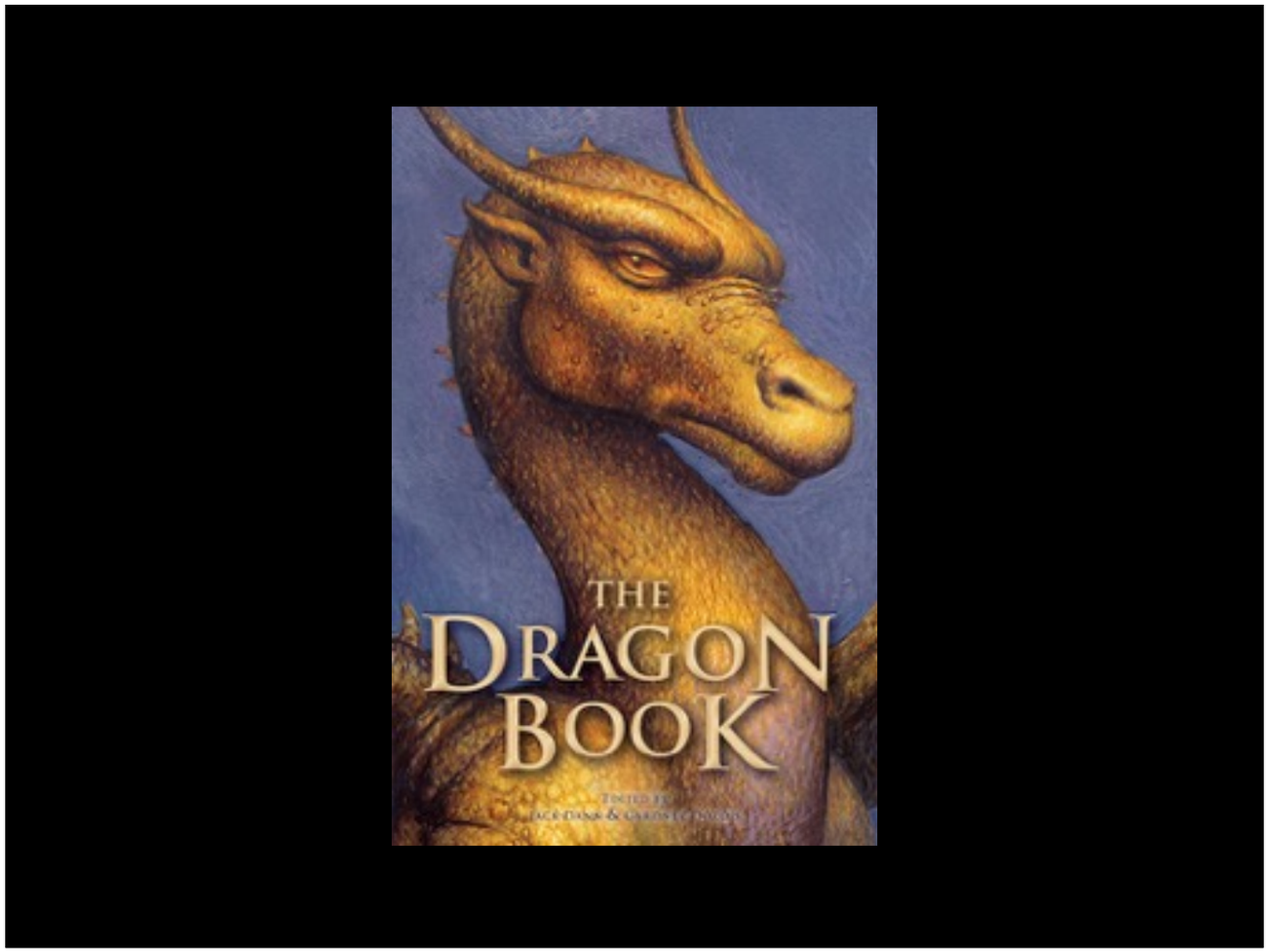
This is a pretty general tool, though: You've got data, what more could you do it if you could directly work with its structure?
Renaming/rewriting/refactoring/scraping
syntax highlighting, detecting certain kinds of errors



Unfortunately, parsers have a scary reputation.

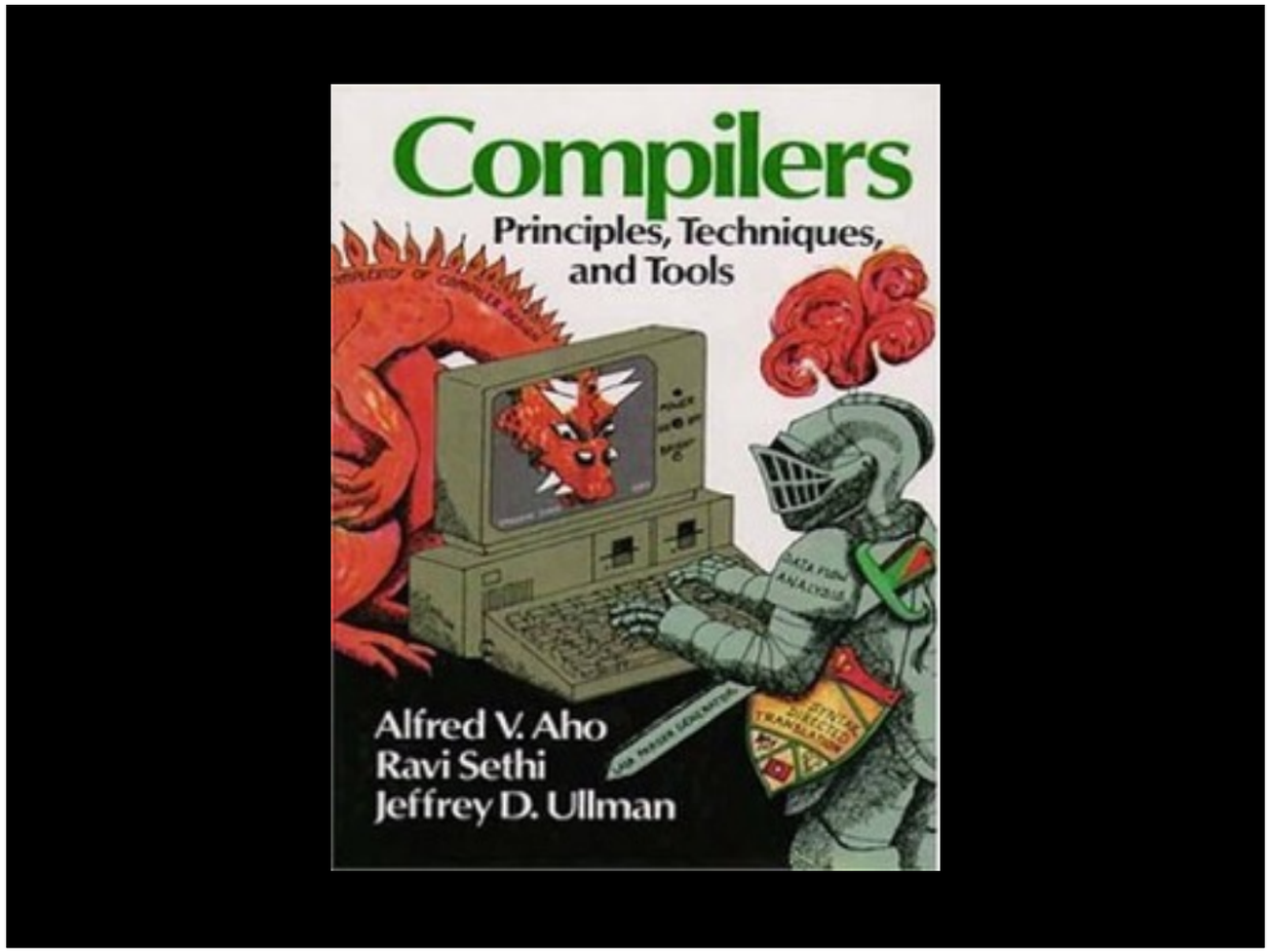
Some of the books and tools you may have encountered are pretty gnarly.

For instance, …



The Dragon Book

Okay, not this one.



This one.

[grumble about the dragon book]

yacc, bison, …

And then there’s stuff like yacc and bison

“shift-reduce errors”

which complain about things like “shift/reduce errors”. What are those?

These tools can be hard to use well without knowing the theory.

Regular Expressions

So people sometimes make do with regular expressions.

The problem is that they’re fundamentally incapable of doing parsing: they don’t have any sort of stack, so they struggle with nesting.

More Regular Expressions

But you can work around that by adding more regular expressions

And then you can work around that by adding more regular expressions

An easier way

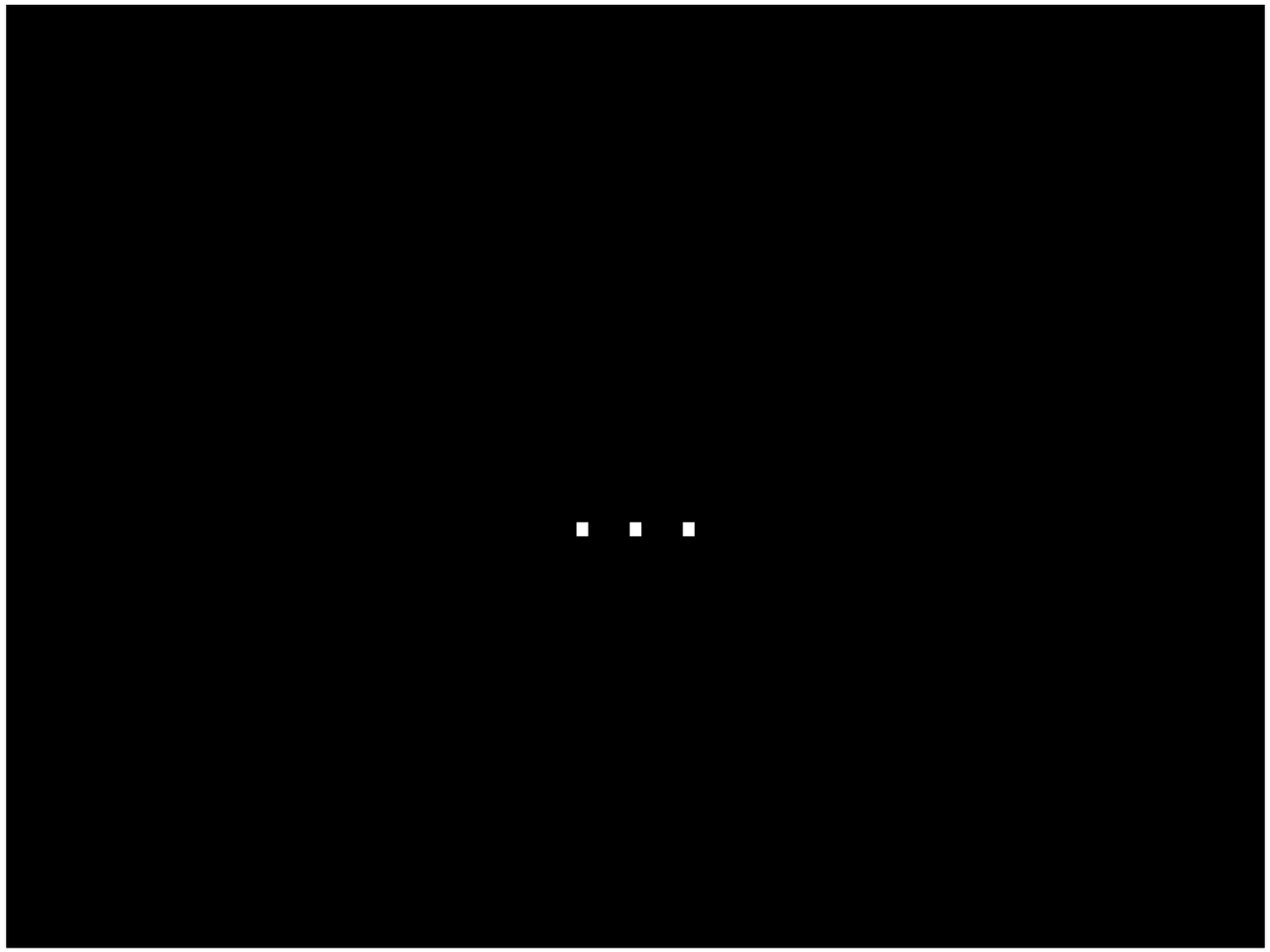
But there's an easier way. I'm going to talk about a couple parsing techniques that you can use, which don't depend on any particular tooling. (libraries only Haskell or Clojure has, etc)

One can be built up out of almost nothing. The other is more complicated, but can handle input that would tie most parsers in knots.

Recursive Descent

First, there's recursive descent parsing.

The idea here is to keep a 1:1 relationship between the structures in a grammar and the functions calls and other control flow in a program.



But before I get into that, I need to introduce a couple terms.

Terminals, Nonterminals

First, there are terminals and nonterminals.

These are the parser's building blocks.

Terminals

- x
- 12345
- 👻
- semicolon

Terminals are the individual words, numbers, punctuation marks, etc. that make up the input data.

A variable name, a number, a ghost emoji, a semicolon.

They often have position info saved with them, for “error at line X” messages.

Tokenizing (lexing)

While earlier I mentioned that you can't parse with regular expressions, they're good for categorizing these: does the next chunk of input look like a number? or a variable? or what?

This is called “tokenizing” or “lexing”. It’s good to do it as a separate pass, so surface details like whitespace or comments no longer complicate things.

Non-terminals

- Expression
- Function_Definition
- Mailing_Address

These are where the data builds up from individual symbols to meaningful structures.

e.g. mailing address <- name house_addr street city state zip

A raw number means more when you know it’s a part of street address.

Production

The rules for how these structures can fit together are called productions.

The reason they're called productions is because, historically, they were viewed as guides for generating a language rather than matching it.

```
Expr: var op Expr
    | var
    | num
```

In this case, an expression can be:

a var, an operator, an Expression
or a var
or a num

Grammar

And this set of productions is called a grammar

```
chunk ::= block

block ::= {stat} [retstat]

stat ::=  ';' |
         varlist '=' explist |
         functioncall |
         label |
         break |
         goto Name |
         do block end |
         while exp do block end |
         repeat block until exp |
         if exp then block {elseif exp then block} [else block] end |
         for Name '=' exp ',' exp [',' exp] do block end |
         for namelist in explist do block end |
         function funcname funcbody |
         local function Name funcbody |
         local namelist ['=' explist]

retstat ::= return [explist] [';']

label ::= '::' Name '::'

funcname ::= Name {'.' Name} [':' Name]

varlist ::= var {',' var}

var ::=  Name | prefixexp '[' exp ']' | prefixexp '.' Name

namelist ::= Name {',' Name}
```

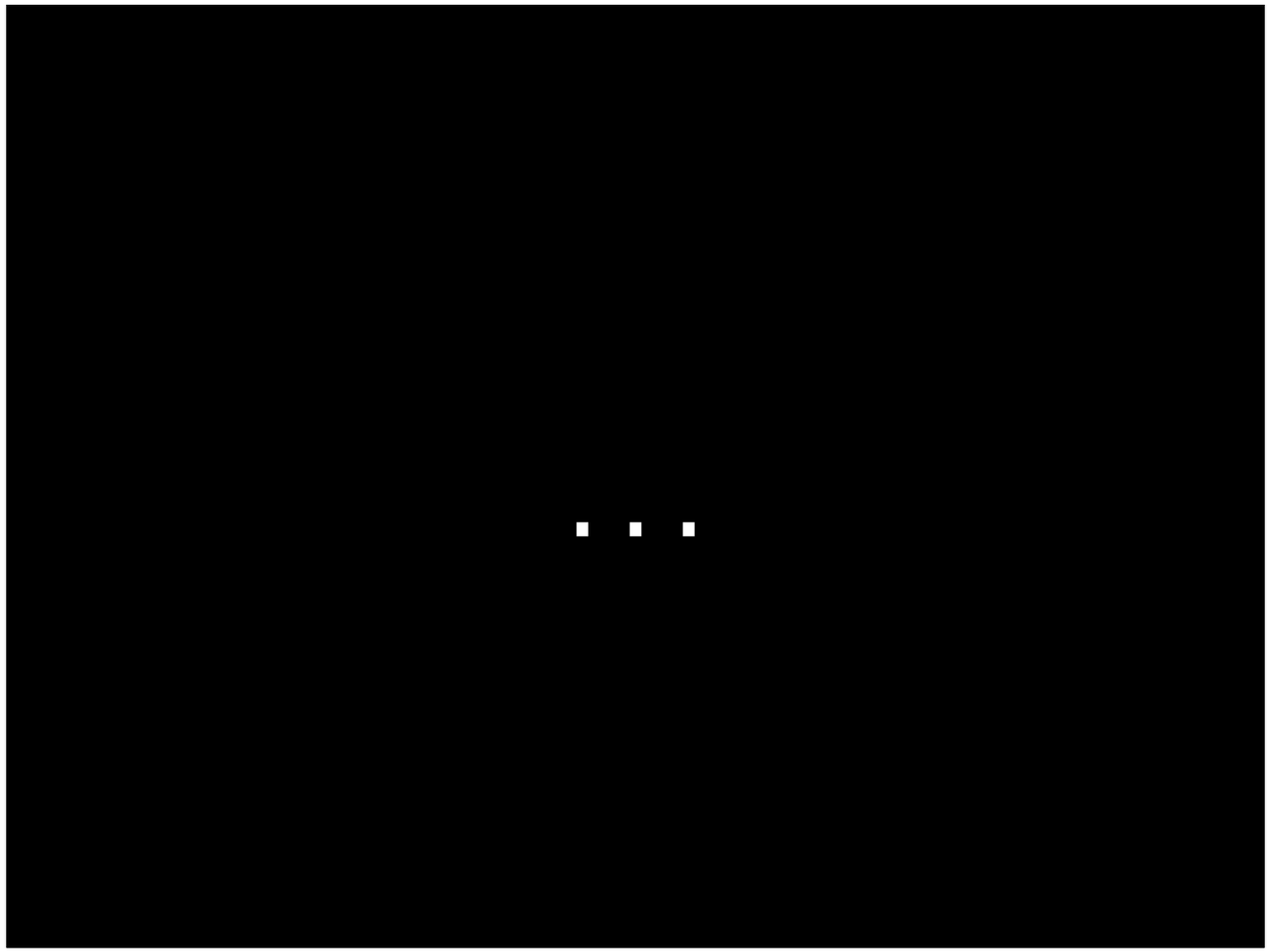
Here's a screenshot of part of the grammar for the Lua programming language.

While you might not be able to read it, just seeing the overall shape tells you several things about it.

Start symbol

The start symbol is a nonterminal that represents a complete parse for the grammar.

It will be the root of the parse tree.



Okay, so we were talking about how to write a real parser.

Recursive Descent

A recursive descent parser is a parser that uses function calls to represent nonterminals, and if statements to decide between alternatives, based on the next input terminal.

Intuitively: each token sets the context for a group of tokens following it.

```
Stmt: var op Expr
```

For a production like this, it’d translate into three steps.

I’m using a convention where terminals are lowercase and nonterminals are uppercase.

```
Stmt: var op Expr

stmt = Nonterminal:new()

input = token_stream()
```

We’re trying to match a nonterminal, stmt, so we start gathering the pieces that build it up.

Make an empty structure that will be the parsed statement, and get a handle to the input stream.

Stmt: var op Expr

```
if (input:next_type() != T_VAR) then
    error(“Expected var:” + input:pos())
else
    stmt:append(input:next_token())
    input:consume()
end
```

Check the type of the next token.
If it isn’t a variable, then raise an error, otherwise save it in the statement and move to the next input token.

(Fitting pseudocode on slides -> bad error messages.)

Stmt: var op Expr

```
if (input:next_type() != T_OP) then
    error(“Expected op: ” + input:pos())
else
    stmt:append(input:next_token())
    input:consume()
end
```

Next, check if we have an operator, do the same thing.

The code is nearly identical.

Stmt: var op Expr

```
expr = EXPR(input)

if (expr:type() != NT_EXPR) then
    error(“Exp. Expr: ” + expr:pos())
else
    stmt:append(expr)
end
```

Finally, we’re looking for a nonterminal, Expression, so call a function to try to build one of those. The parser uses the programming language stack as its stack, directly.

We either get an expression or we don’t, so do an error check. If we got one, add it to the statement, mark it as finished, and return .

Stmt: var op Expr

```
# We're done!
stmt:set_type(NT_STMT)
return stmt
```

We've matched an entire statement, so finish the nonterminal, and return it to the nonterminal that was trying to match it.

If it's the start symbol, we have a full parse tree.

Choices

Of course, there will be times during the parse where there is more than one option.

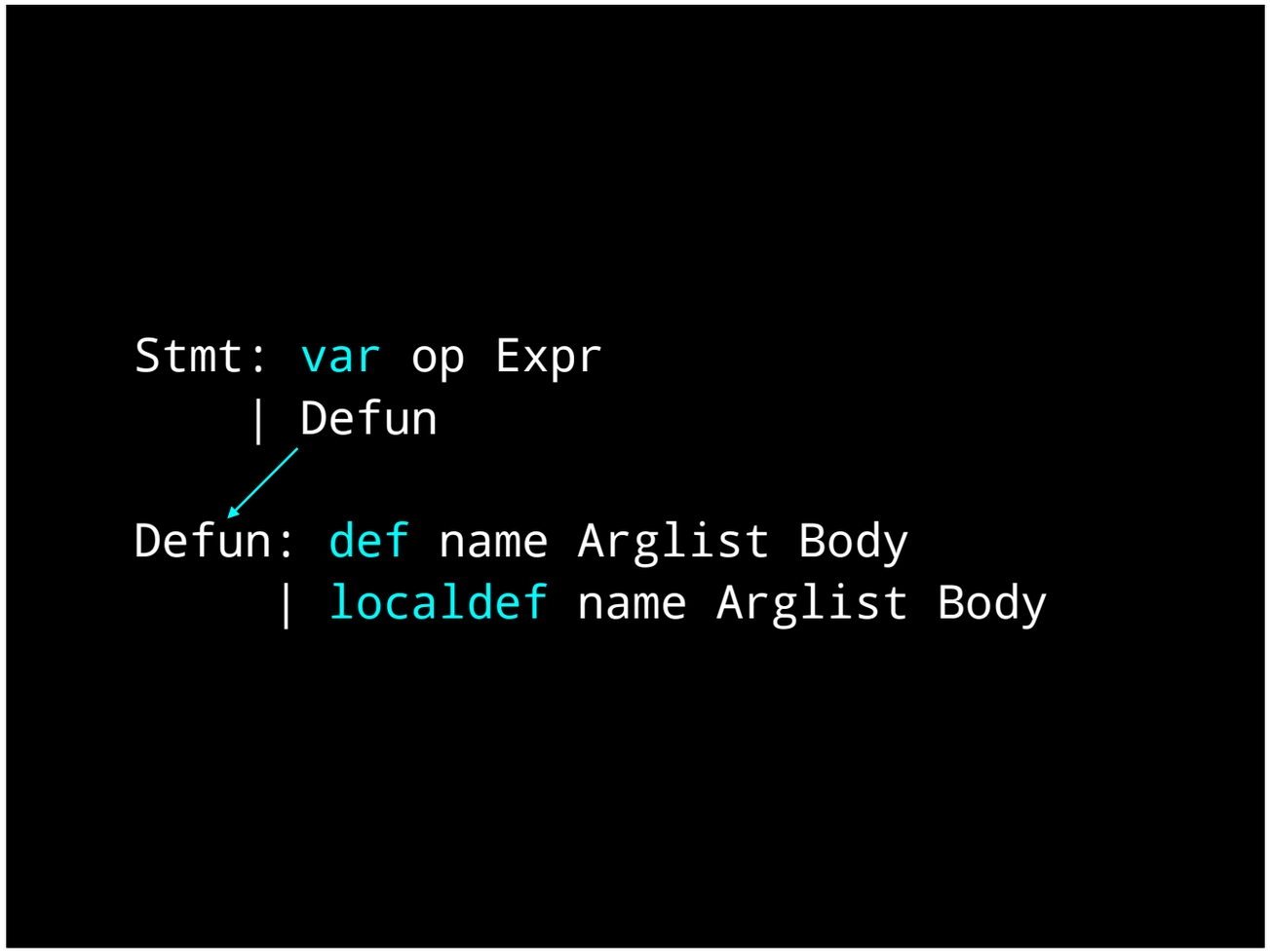
Then, we have to look at the first terminal that each production takes, and use that to decide.

If they start with nonterminals, figure out what they can start with, etc.

```
Stmt: var op Expr
    | Defun

Defun: def name Arglist Body
     | localdef name Arglist Body
```

Here are a couple productions



By looking at the grammar, we can see that the first Stmt production has to start with a variable, and the second either a “def” or “localdef”.

```
if (input:next_type() in {T_VAR}) then
    return STMT_1()
else if (input:next_type() in {T_LOCALDEF,
                               T_DEF}) then
    return STMT_2()
else
    error("Expected stmt @ " + input:pos())
end
```

So we check if the next input token is in the starting set for the first Statement production, or the second's starting set, or we error out.

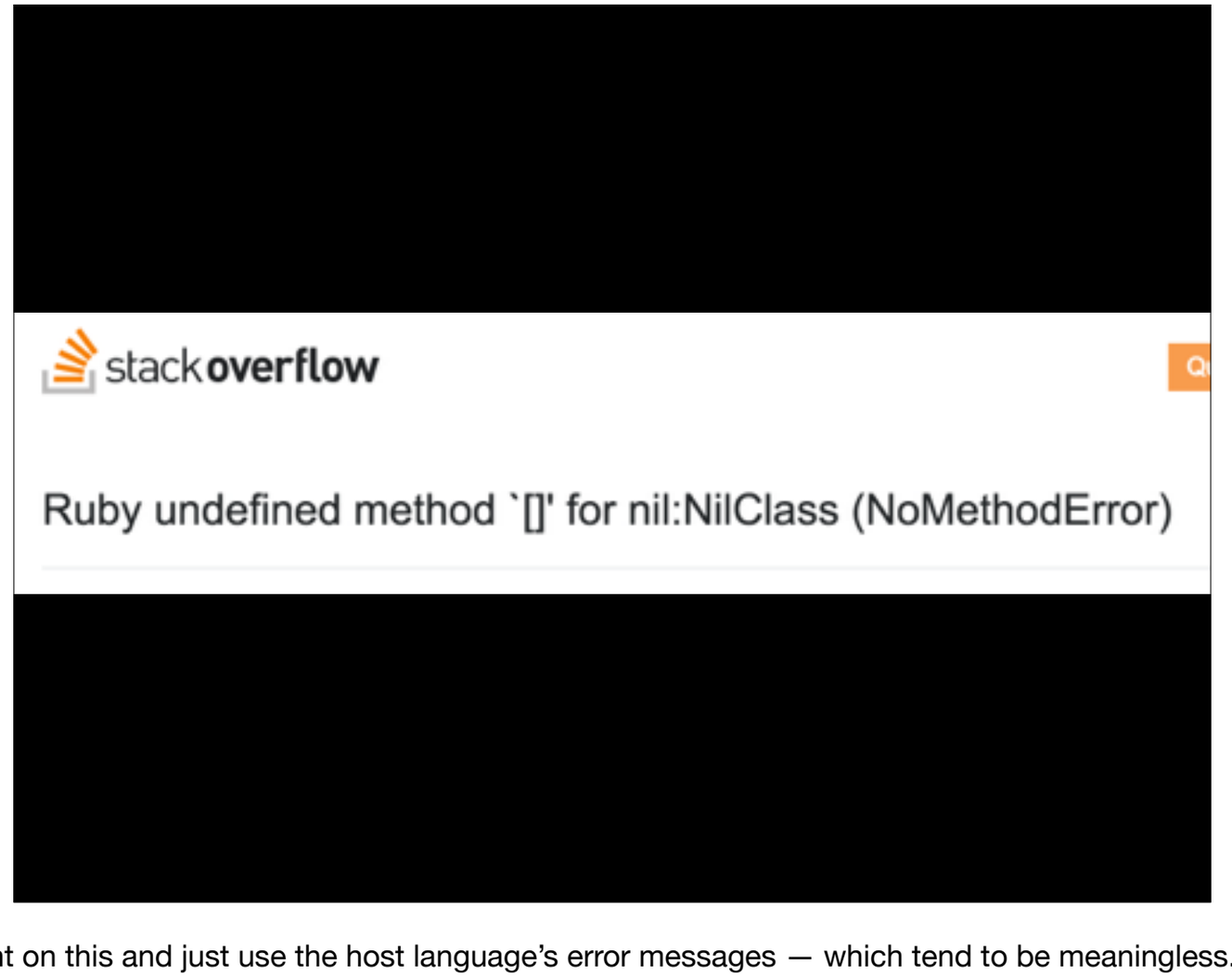
That’s pretty much it, really.

You can either convert the grammar to code like that, or write table-driven code that uses the grammar definition, or generate code. There should be a direct correspondence between the grammar and the parser code.

Error messages

An important benefit of RDPs is that the parser's current state is usually pretty meaningful to humans - it's straightforward to turn the current context into good error messages.

That isn't as easy for many kinds of parsers, and adding error messages after the fact can be really difficult.



Even worse, embedded DSLs often punt on this and just use the host language's error messages — which tend to be meaningless.

Parsers make it easier to guide people when they make mistakes. Good error messages are important!

Grammar Requirements

Unfortunately, an RDP isn’t capable of parsing all languages.

Grammars it can parse need to meet two requirements.

Next token decides

For any point with multiple productions, we need to be able to pick the right one based on just the next token. This token is called the “lookahead”.

If there are multiple productions that take the next token, we’re stuck: we can’t branch both ways, at least not with a basic RDP.

(I’ll talk about a parser that can do this later.)

(Exceptions & Flexibility)

Since our parser is a general programming language, we can work around cases that are locally ambiguous, without the whole thing falling apart.

Sometimes we can look ahead two or three tokens before we decide, but keep it in the scope of a single nonterminal, so it doesn't impact the rest of the parser.

example: "for (x in things) …" vs "for (i = 0; i < 10; i++) …" both start with "for", but might peek ahead two tokens to check for "in" or "=" to tell which kind of for loop they are.

Avoid Infinite Loops

Also, nonterminals need to consume at least one token before referring to themselves.

Otherwise, the parser would get stuck in an infinite loop.

```
Expr: Expr op num
```

This won’t work, because Expr will try to build an Expr by calling Expr to build an Expr, …

A self-reference at the left of a nonterminal like this is called “left recursion”.

LL(1)

Informally, these requirements are what people mean when they say a grammar is “LL(1)”.

That stands for Leftmost-derivation, Lookahead of 1.

The parse structure can be decided from the tokens on the left (i.e., the start), and it needs at most 1 token of lookahead to decide.

See also: LR(1)

You may see other similar terms, like “LR(1)”. This is a *right*most derivation, with a lookahead of 1. This is the general model that tools like yacc/bison use. Since the context at the right / end of a parse doesn’t fit function calls the way the left does, LR(1) parser generators usually work by building a lookup table.

Shift/reduce

This uses the current state and the input token to pick a new state, and whether to shift or reduce.

Shift: Push the input onto the stack.
Reduce: Pop things off the stack and replace with a nonterminal.

Shift/reduce errors

A shift/reduce error means it can’t decide whether to shift or reduce in the table somewhere.

“Look both ways when crossing”
Photo by Garrett, CC-BY, scaled.

But, what if we want to look both ways?

Earley Parsers

There's another kind of parser that doesn't have these limitations.
It handles ambiguity, because it can try all possible parses, in parallel.

```
S: S S S
 | S S
 | a
 | (empty)
```

This grammar has left recursion. And right recursion! This is a combinatorial explosion.

An Earley parser can handle this, in linear time even.

Chart

Earley parsers use a table, called a “chart”, to track all the potential parses in progress.

Anything already in the table is re-used, and the chart starts trying to match the start symbol.

These parses in progress are called “items”, which is an awkwardly generic name.

```
Item @ 2: . Thing name Stuff
Item @ 2: Thing . name Stuff
Item @ 2: Thing name . Stuff
Item @ 2: Thing name Stuff .
```

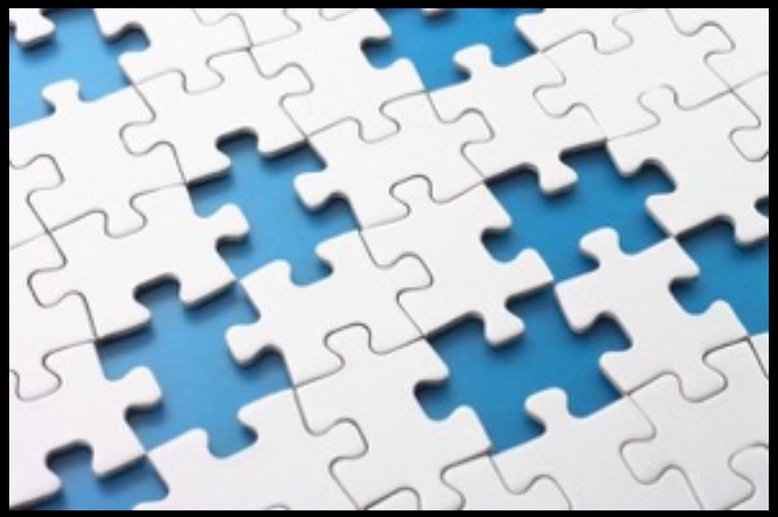
Items have a nonterminal name and starting position, then a production with a dot indicating how much has matched so far.

Three sub-tasks

Earley parsing is usually presented as having three subtasks, each of which repeats until it has no more work to do.

Scanner

The Scanner takes the current input…



“Missing Piece”
Photo by OnePoint Services, CC-BY.

and finds where to plug it in.

```
Stmt @ 2: var  . op Expr

            + op becomes

Stmt @ 2: var op . Expr
```

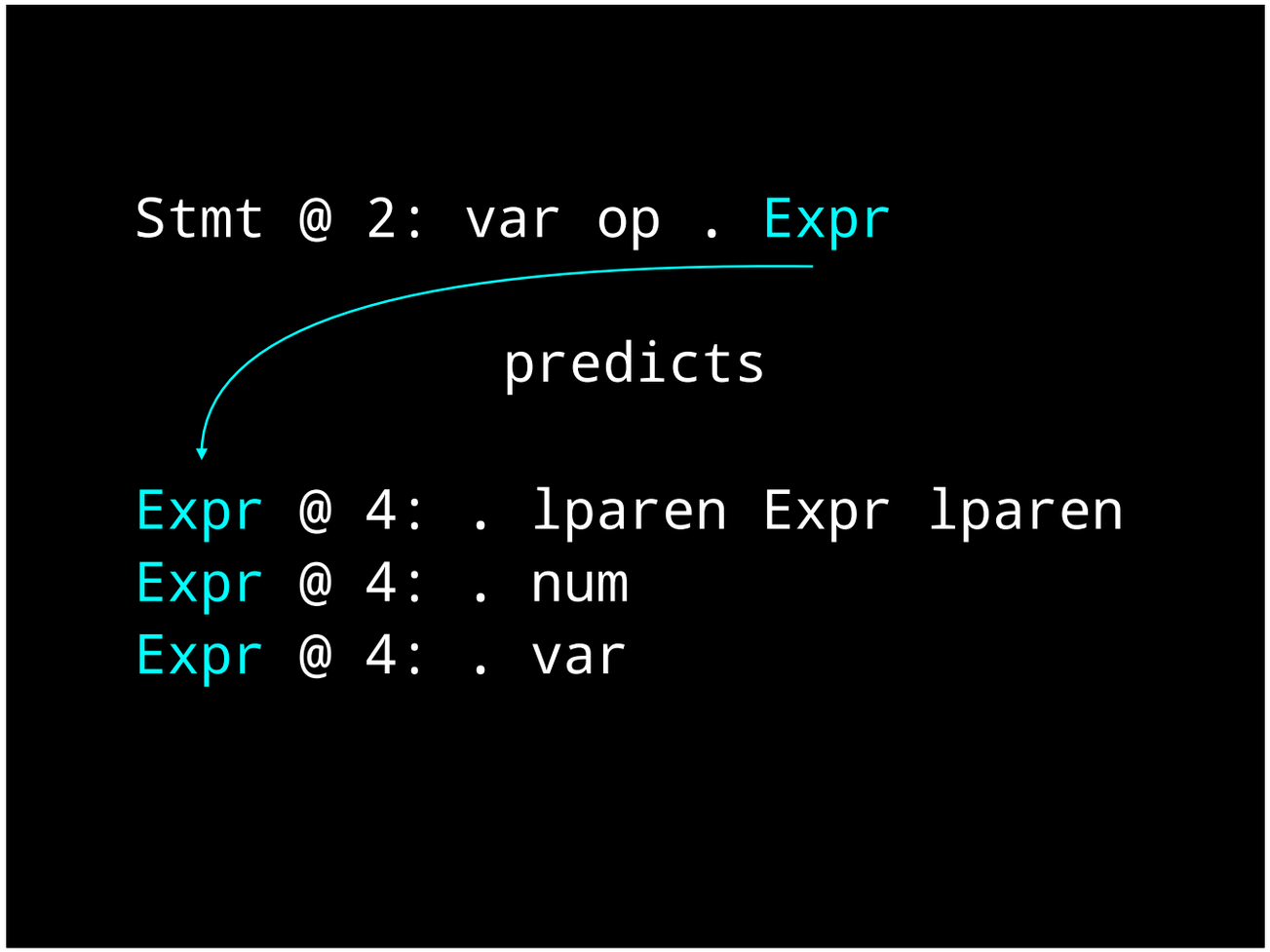
Items waiting on an operator get moved to the next chart column, and their dot steps over “op”.

Predictor

If any partial parses are waiting for a nonterminal, the Predictor…

"Shopping List"
Photo by EvelynGiggles, CC-BY.

…adds it to the shopping list.



For anything with a dot before a nonterminal, add productions for that nonterminal to the current column.

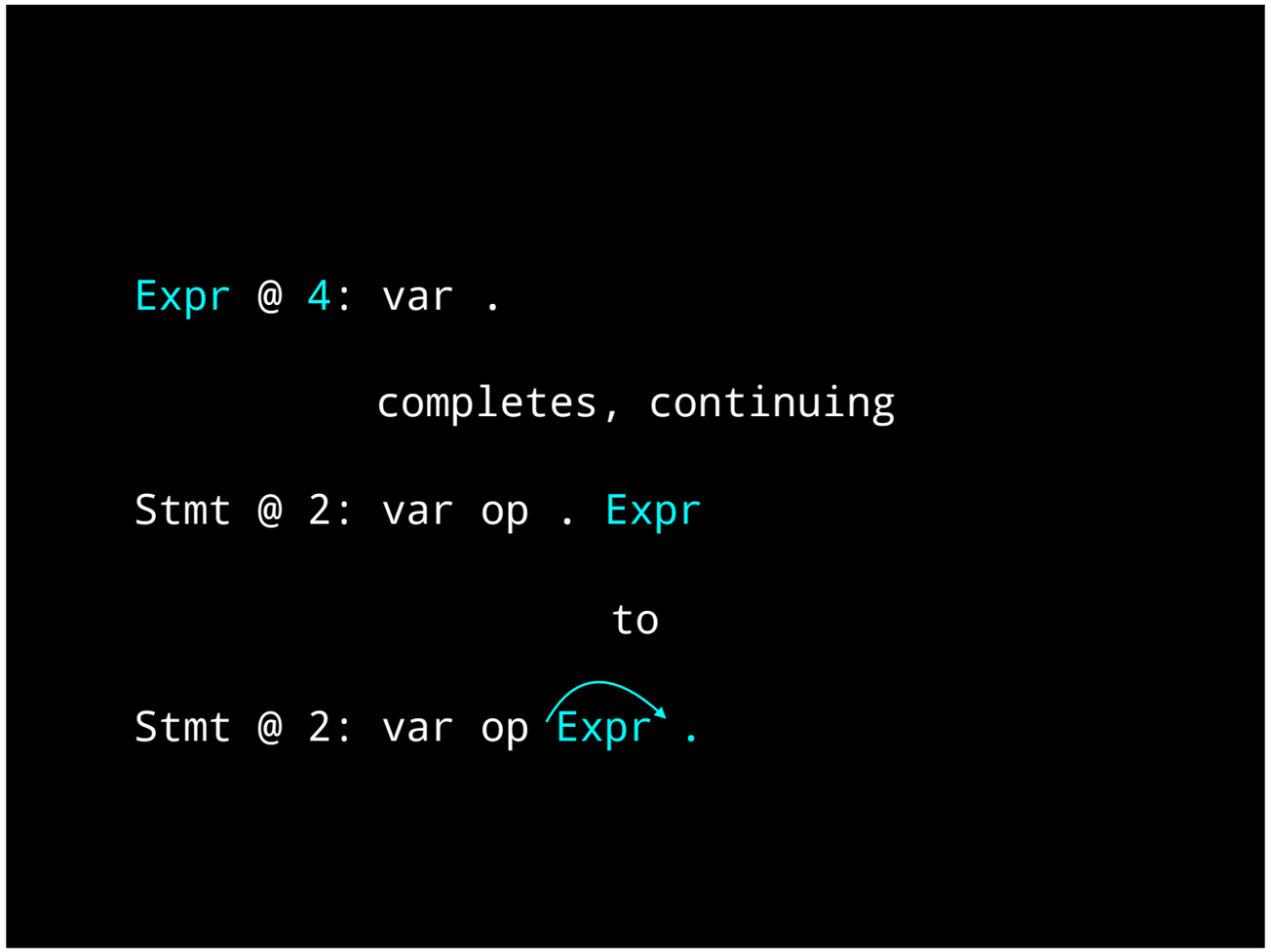
At the beginning of the parse, the Predictor adds all the start symbol's productions to the first column.

Completer

The Completer takes items where the dot made it to the end, find the items that were waiting for them…

“Bricklayers”
Photo by Scott Lewis, CC-BY.

…and plugs in the nonterminal.



For anything with a dot at the end, go back to the chart column where it was initially created, and advance anything with a dot before the completed nonterminal.

Like the scanner, but with completed nonterminals.

Avoiding useless work

Despite their flexibility, Earley parsers avoid a lot of useless work.
It builds nonterminals bottom-up from the input, but only tries nonterminals that could be reached from the start symbol and progress so far.

It also memoizes everything, eliminating redundant processing.

That’s it…right?

That’s pretty much all you need for a basic Earley parser.

At the end of the input, check for a matched start symbol spanning the whole input.

Optimizations

- Handle empty nonterminals
- Filter predictions using lookahead
- Leo's right recursion optimization

The parser could be easily adapted to handle productions with empty bodies: they are predicted with the dot at the end (of nothing), and completing them adds to the current column rather than the next (since no tokens were consumed).

The predictor confidently predicts things which are immediately proven wrong. That could be filtered the same way the RDP used to choose productions.

There's a classic optimization by Joop Leo that keeps right recursion from building up deeper and deeper items along the way.

Wait: what about parse trees?

Unfortunately, getting a parse tree out of an Earley parser is tricky. Since they’re so flexible, there may be several options.

If you only need one, you can arbitrarily commit to the first choice: Look for the start symbol, then find items in the chart that fill it, then look for items that fill those, …

Embracing ambiguity

The real power of Earley parsers is that they can give you all the possible parses and let you decide, though.

There's no need to throw out that info immediately. The chart gives us the big picture of what's happening with the parse, and we can work with that info in many other ways (e.g. suggesting autocompletions).

There’s a data structure called a SPPF, a persistent tree structure combining all parse trees, with structural sharing.

I spent a really long time struggling with papers that explain how to build them, but kept running into implementation issues. If you want to try, look for “SPPF-Style Parsing From Earley Recognizers” by Elizabeth Scott, and E. Scott and A. Johnstone’s related papers.

Another option…

There’s another data structure that excels at representing potentially infinite, self-referential trees.

In fact, it’s been hiding in plain sight all along!

A grammar!

That’s what we started with, right?

Beginning at the start symbol and choosing productions, we can generate all possible strings in a language.

Intersection Grammars

However, we can go one step further: We can take a grammar and input, and get back a version of the grammar constrained to the input. (A curried grammar.)

Rather than producing all strings within the language, it will only produce the original text. If it can produce it in multiple ways, those correspond to alternative parse trees.

```
Start: Expr
Expr: Expr op Expr
    | var

            “a + a + a”
```

For example, let’s take this grammar for recognizing variables being added.

```
“a + a + a”

“(a + a) + a”   left  ?
“a + (a + a)”   ? right
```

This is ambiguous, because it could group tighter on the left, or the right.
And Earley will give us both.

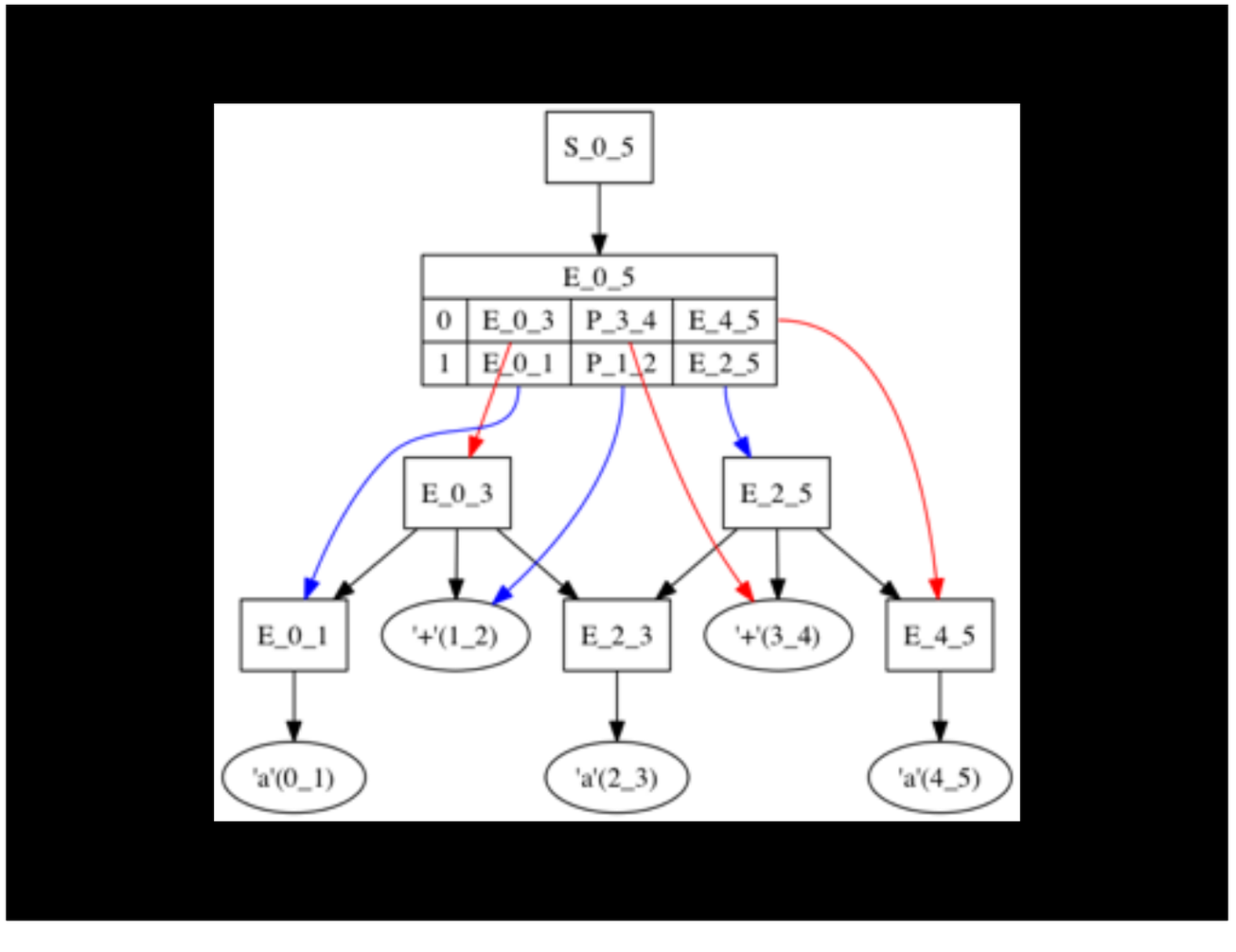
```
Start[0,5]: Expr[0,5]

// Left version
Expr[0,5]: Expr[0,3] op[3,4] Expr[4,5]
// Right version
Expr[0,5]: Expr[0,1] op[1,2] Expr[2,5]

Expr[0,1]: var[0,1]
Expr[2,3]: var[2,3]
Expr[4,5]: var[4,5]
```

This is the same NTs as the original grammar, but now the productions are restricted to specific ranges where they appear in the input. (0 is before first ‘a’, 5 is after last ‘a’.)

We can also see there’s multiple choices for `Expr[0,5]`, so we know it found ambiguity.



If you’re having a hard time visualizing that…
(E: Expr, P: Plus (“op”))

Structural sharing (by name/span)

Because everything is referenced by name & span, different parse trees contained in the grammar that have subsections in common will automatically share.

This keeps memory usage from exploding.

Earley + Intersection Grammar

These completed items with spans are really close to the items we already get from the Earley parser's completer — we just need to save a little more info.

We just need to keep track of beginning and ending spans as things fall into place.

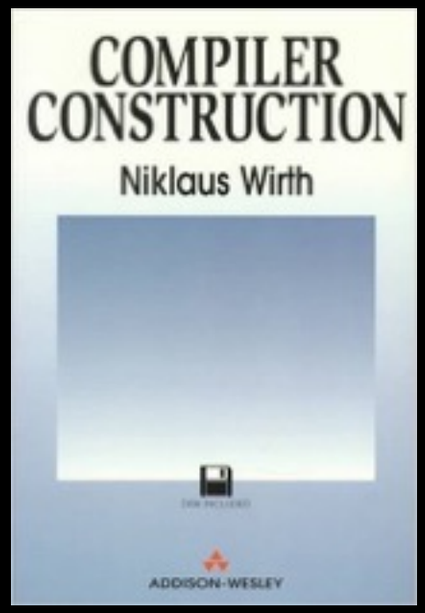
We'll end up with some useless completed items in the end, but they won't be reachable form the completed start symbol. They can be garbage-collected.

Closing

- Parsing terms
- Recursive Descent Parsing
- Earley Parsing
- Intersection Grammars
- Earley + Intersection Parsing

We’ve covered a lot of ground.

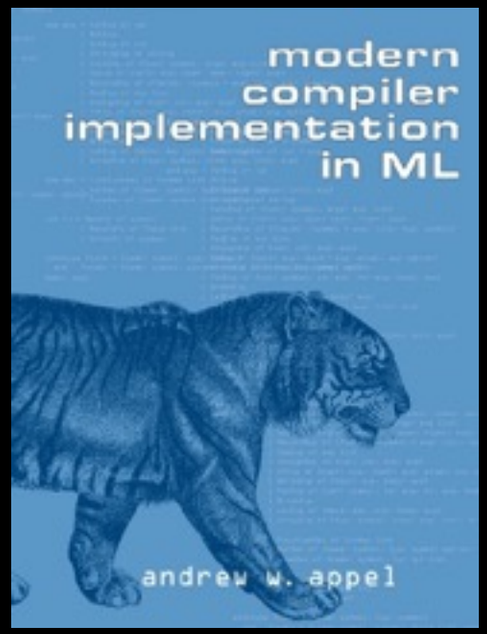
To learn more



Compiler Construction
by Niklaus Wirth

This is a great, short book, building up a bytecode virtual machine and compiler for a dialect of Pascal called Oberon. The PDF is free online.

It has a quick introduction to tokenizing and building recursive descent parsers. Wirth aims to get you from a simple tokenizer and parser to a full compiler, quickly, so you can see the big picture. (Rather than getting lost in the weeds choosing between 10 different kinds or parsers, or whatever.)



Modern Compiler Implementation in ML
by Andrew Appel

This is another great compiler textbook. It goes deeper than the Wirth book, including compiler stuff outside the scope of this talk.

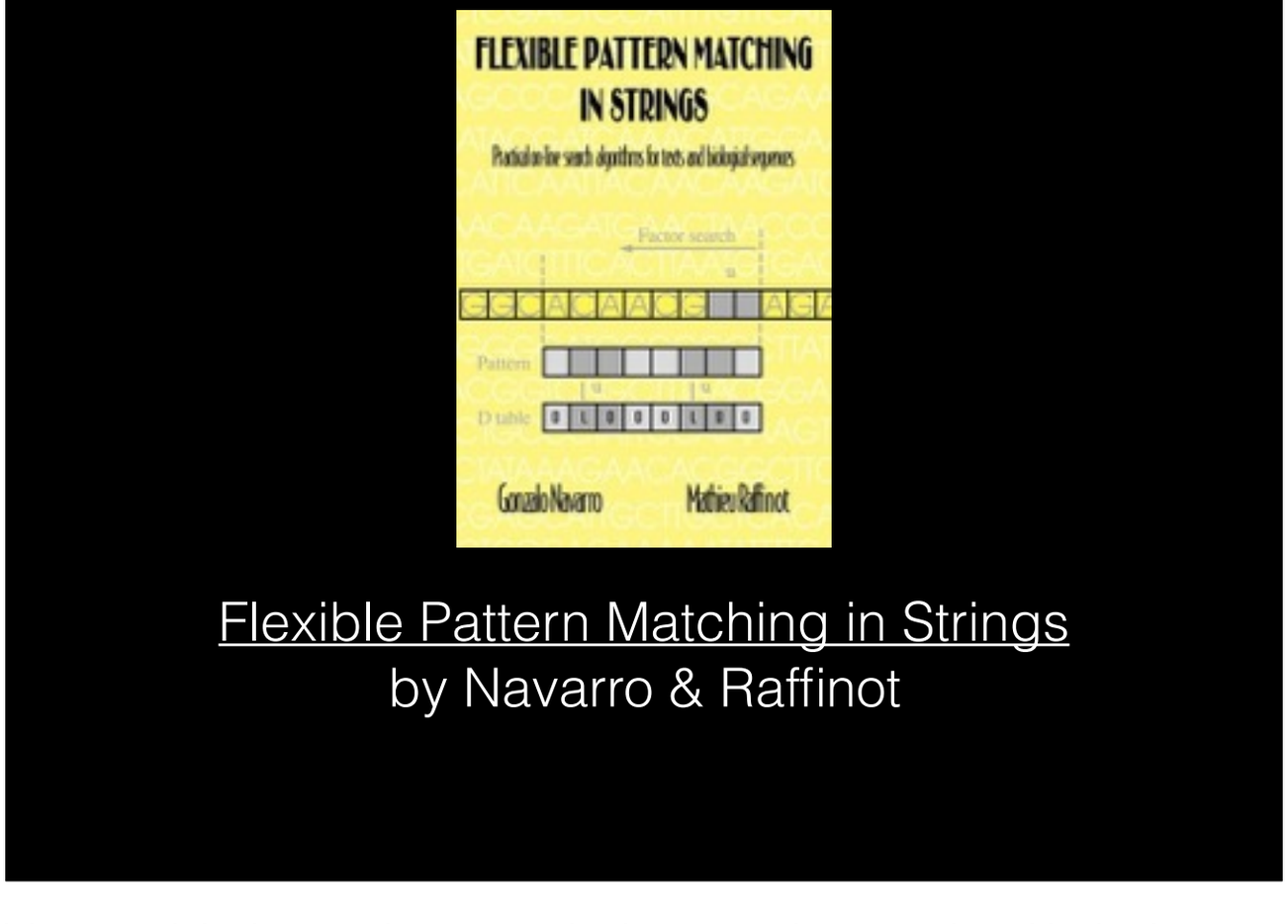
In general, this is what I'd recommend before the dragon book. It comes in ML, Java, and C versions. Standard ML is a much better fit for presenting its ideas, so get that one.

Parsing Techniques: A Practical Guide
by Dick Grune

This is an outstanding book about all things parsing, with an excellent overview of 50+ years of a rich subfield in CS, and an extensive bibliography.

It’s one of my favorite CS references.

The Earley/Intersection parsing stuff is section 13.4.



Flexible Pattern Matching in Strings
by Navarro & Raffinot

While not strictly about parsing, this book has great info about modern ways to implement regular expressions. It’s also focuses on pattern search in genetic sequences, which is a fascinating problem.

Thanks!