CS 320: Principles of Programming Languages

Katie Casamento, Portland State University

Winter 2018
Week 4: Parsing

Parsing in the real world

Parsers are a common component in a wide variety of programs:

- PL tools
    - Compilers/interpreters
    - Documentation generators
    - Code formatters
    - …
- Configuration files
    - XML, JSON, …
- Data file formats
    - DOCX, PDF, …

What's still missing?

We have theories of lexical grammars and parsing grammars, but how do we apply them to write code that turns input streams into ASTs?

- Automating tokenizing
  - Generating a token stream from a character stream
- Automating the CFG derivation process
  - Generating a parse tree from a token stream
- Converting concrete syntax trees (parse trees) into abstract syntax trees
  - Removing unnecessary detail
- Efficiency concerns
  - Computational complexity vs. linguistic expressivity

Parsing in practice

Several different approaches:

- A *parser generator* (or *compiler compiler*) reads a file describing the syntax of a language and generates code for parsing the syntax in some host language
    - Yacc/Bison (C/C++), Javacc (Java), Happy (Haskell), …
    - Usually also includes lexer generator functionality
    - Efficient but often inflexible and hard to debug
- A *parser combinator* library contains a set of primitive parsers and functions to combine them into larger parsers
    - Parsec (Haskell) & ports (Java/C#/JS/…)
    - Lexers are a special case of parsers
    - Flexible and maintainable but often less efficient
- A *recursive descent parser* can often be written straightforwardly by hand

Tokenizing

Token attributes

Token types might be associated with a variety of types of data (*attributes*) in the host language:

- Identifier tokens tagged with a string representation of the identifier
- Integer literal tokens tagged with an int representation of the literal
- Boolean literal tokens tagged with a bool representation of the literal
- …

A tokenizer library/generator will often offer a way to annotate lexical rules with a function to extract attributes from the matched string.

In Haskell (Lex.lhs)

```
type LexGrammar t = [(Regex, String -> Maybe t)]

tokenize :: LexGrammar t -> String -> [t]
```

- Each rule in a lexical grammar is a regular expression paired with a function to turn the matched string into a token of type `t`
- Returning `Nothing` omits the lexeme from the output token stream (for whitespace/comments)
- The `tokenize` function tries each rule, returning the result from the first one that matches and moving on to tokenize the rest of the string (or failing if none match)

In Haskell (PropLex.lhs)

```
type LexGrammar t = [(Regex, String -> Maybe t)]

    propLexGrammar :: LexGrammar PropTok
    propLexGrammar =
      [ (Sing '(',              \cs -> Just T_LPAREN),
        (Sing ')',              \cs -> Just T_RPAREN),
        (Sing '&',              \cs -> Just T_AND),
         ...
        (Plus (range 'A' 'Z'), \cs -> Just (T_IN cs)),
        (whitespace,            \cs -> Nothing) ]
```

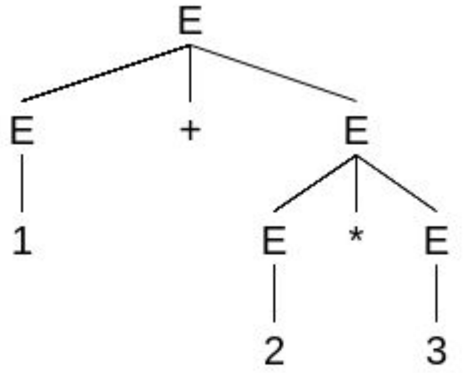
Parsing

Brute force

- A CFG gives rules for generating parse trees
- Simple idea: iterate through all possible parse trees until we find a match

How do we know if a parse tree matches an input token stream?

`"1 + 2 * 3"` —lexes to→ `[1, +, 2, *, 3]` —parses to (?)→



Just read the leaves of the tree left to right!

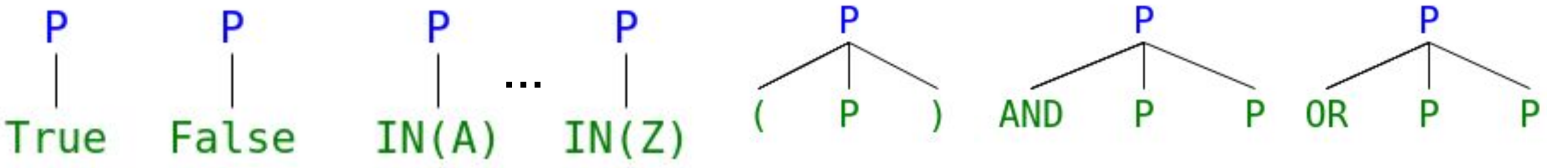
Brute force algorithm

```
Prop =
    P → TRUE
    P → FALSE
    P → (P)
    P → AND P P
    P → OR P P
    P → NOT P
    P → IN c
```

Begin with the start symbol as the root of a parse tree:

P

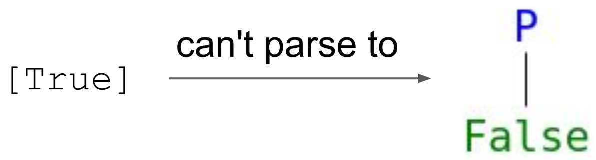
Add each possible expansion to a queue:



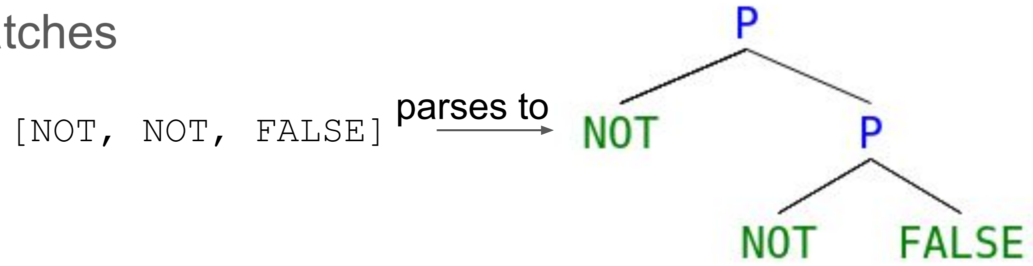
Brute force algorithm

For each loop iteration: if the queue is empty, report failure; otherwise, pop a tree off the queue.

If the tree is fully expanded and it doesn't match the input stream, continue to the next loop iteration.

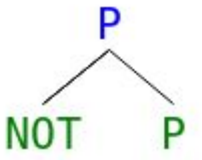


If the tree is fully expanded and it matches the input stream, return it (success).

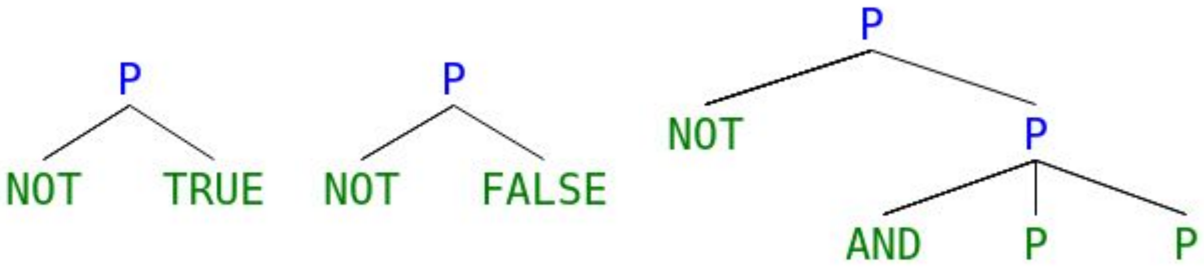


Brute force algorithm

If the tree isn't fully expanded, push onto the work queue every tree generated by expanding the bottom-leftmost nonterminal.



expands to



etc.

Termination

Does it terminate?

Potential problems:

- A CFG usually describes an infinite language
  - All finite languages are regular
  - Infinite set of possible parse trees
- The set of terminals might be infinite
  - In Arithmetic: `T = {'(', ')', '+', '*'}` ∪ $\mathbb{N}$
  - In Prop: T contains all valid input names of arbitrary length
- There might be a *cycle* in the grammar
  - Trivial example: `P = {A → a, A → A}` generates only the string "a" but can take any arbitrary number of steps to get there

Termination metric

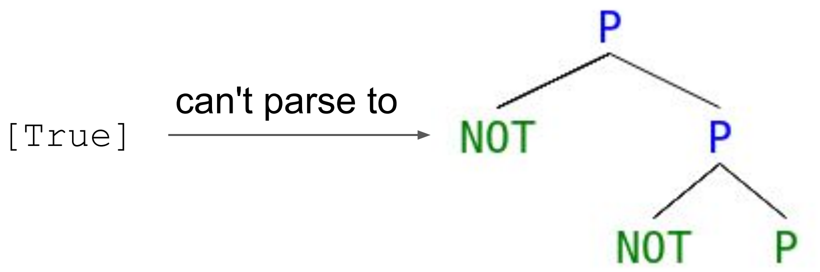
So how do we know when to stop generating trees?

- Restrict the set of terminals to a reasonable finite subset
  - In Arithmetic, restrict numbers to be between 0 and MAXINT
  - In Prop, restrict variable names to be at most N characters long
  - With a finite set of terminals, there are a **finite number of possible next steps** from any point in a derivation
- Remove cycles
  - Every CFG has an equivalent cycle-free Chomsky normal form
  - There exist algorithms to transform CFGs into Chomsky normal form
  - Without cycles, every production will either **terminate or generate more terminals after some finite number of derivation steps**
- Now there's a **finite set of derivations with up to N terminals**

Termination metric

Add another case to the algorithm loop:

If there are more terminals in the leaves than in the input stream, continue to the next loop iteration.



CST to AST

Now we've got a working way to turn a stream of characters into a parse tree/CST:

```
tokenize :: LexGrammar t -> String -> [t]
parse    :: CFG t          -> [t]    -> CST t
```

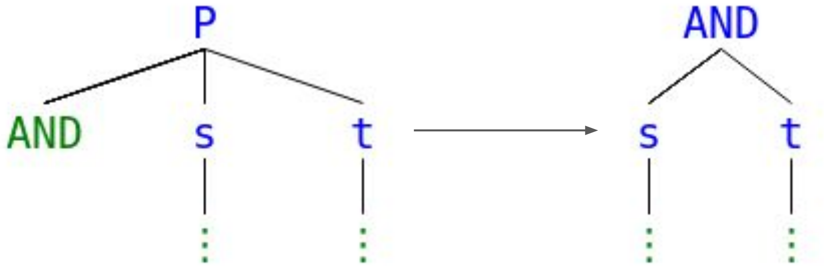
(ignoring some implementation details)

One step remains: we need to turn a CST into an AST by removing irrelevant information (like parentheses).

CST to AST

This part is straightforward pattern matching:

```
cstToProp :: CST PropTok -> Prop
```



(where s/t are arbitrary subtrees)

Efficiency

```
Prop =
    P → TRUE
    P → FALSE
    P → (P)
    P → AND P P
    P → OR P P
    P → NOT P
    P → IN c
```

How many derivations are there with up to N terminals?

(input names restricted to single upper-case letters)

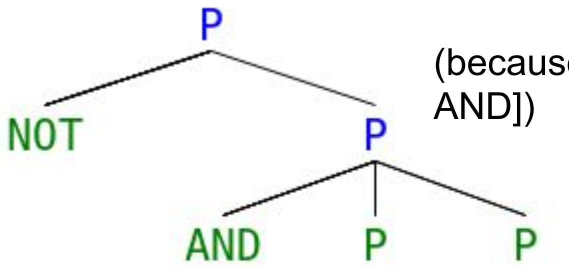
1. 28 (TRUE/FALSE/IN)
2. 56 (length 1 derivations + NOT)
3. 1680 (length 1&2 derivations + AND/OR/parens)
4. 4900
5. 102256
6. 386288
7. … (very many)

Efficiency

We can prune the work queue more aggressively:

If the N leftmost terminal leaves of the tree (starting at the bottom-leftmost node) don't match the first N tokens of the input stream, continue to the next loop iteration.

`[NOT, OR, TRUE, FALSE]` $\xrightarrow{\text{can't match}}$



(because [NOT, OR] ≠ [NOT, AND])

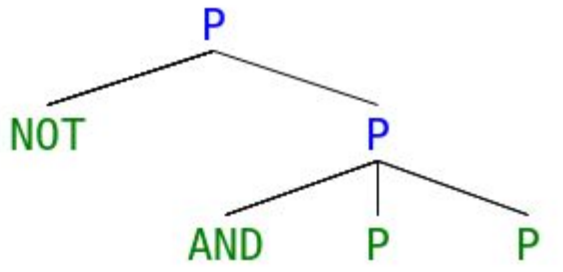
This works reasonably well because our parser is generating *left-most derivations*.

Efficiency

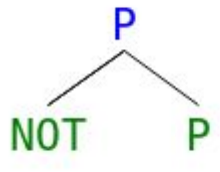
Can we remove the work queue altogether?

We use it for *backtracking*: if we expand a tree in what turns out to be the wrong way, we need a way to go back to a previous state and try the next possibility.

`[NOT, OR, TRUE, FALSE]` $\xrightarrow{\text{can't match}}$



so we backtrack to try the next expansion of

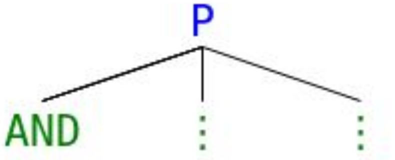


Efficiency

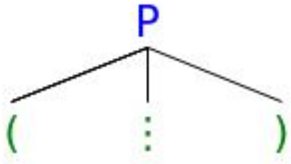
Do we need backtracking?

We know the form a Prop parse tree for a token stream must have after looking at the first token of the stream:

- `[AND, …]` must look like



- `[L_PAREN, …]` must look like



This is called *lookahead* - we can parse this Prop grammar with a lookahead of 1.

LL grammars

- An *LL(k)* grammar is one that can be parsed with a lookahead of at most k
    - i.e. the form of all valid parse trees for a token stream can be determined by looking at only the first k tokens of the stream
- The set of languages that can be parsed by an LL(k) parser for some finite k is a subset of the context-free languages
- An LL(k) language can be parsed in worst-case linear time
    - A *recursive descent parser* is a set of mutually recursive routines which each parse one or more of the productions in the CFG
    - A *parser generator* generates code to execute a finite-state automaton equivalent to the CFG
- Some subsets of the LL(k) class of grammars can be parsed more efficiently
    - LR(0), SLR, LALR(1), LR(1), …

Recursive descent parsing

Prop recursive descent pseudocode

```
Prop =
    P → TRUE
    P → FALSE
    P → (P)
    P → AND P P
    P → OR P P
    P → NOT P
    P → IN c
```

```
i ← 0

parseProp =
    if input[i] == TRUE:
        return TRUE
    else if input[i] == FALSE:
        return FALSE
    ...
```

Prop recursive descent pseudocode

```
Prop =
    P → TRUE
    P → FALSE
    P → (P)
    P → AND P P
    P → OR P P
    P → NOT P
    P → IN c
```

```
eat tok =
    if input[i] != tok: throw error
    i++

parseProp =
    ...
    else if input[i] == L_PAREN:
        i++              -- move past left paren
        p ← parseProp   -- recurse (which will modify i)
        eat R_PAREN
        return p
    ...
```

Prop recursive descent pseudocode

```
Prop =
    P → TRUE
    P → FALSE
    P → (P)
    P → AND P P
    P → OR P P
    P → NOT P
    P → IN c
```

```
...
else if input[i] == AND:
    i++
    p ← parseProp
    q ← parseProp
    return (AND p q)
else if input[i] == OR:
    i++
    prop1 ← parseProp
    prop2 ← parseProp
    return (OR p q)
...
```

Prop recursive descent pseudocode

```
Prop =
    P → TRUE
    P → FALSE
    P → (P)
    P → AND P P
    P → OR P P
    P → NOT P
    P → IN c
```

```
...
else if input[i] == NOT:
    i++
    p ← parseProp
    return (NOT p)
else if input[i] == IN:
    i++
    ident ← parseIdent  -- assumed to be defined already
    return (IN ident)
```

Recursive descent complexity

This is O(n) for the length of the input stream:

- The value of `i` never decreases, so each token is processed exactly once
- A constant amount of work is done for each token

Parser combinators

This parser is one monolithic function - can we split it up into one function for each production?

```
parseTrue =
    if input[i] == TRUE:
        return TRUE
    else fail
```

```
parseFalse =
    if input[i] == FALSE:
        return FALSE
    else fail
```

```
parseProp =
    parseTrue <|> parseFalse <|> parseAnd <|> ...
```

Parser combinators

(<|>) is a *parser combinator*: it takes two parsers (functions) and combines them into a parser that succeeds if either of the given parsers succeeds, trying the first and then the second.

In pseudo-Haskell:

```
(p <|> q) =
    try p:
        on success: return p's result
        on failure: try q, return q's result or failure
```

This should look familiar from regular expressions - it's like the | operator.

Parser combinators

A parser combinator library offers primitive parsers along with combinators to build larger parsers out of them.

```
type Parser t a = ... -- parser over tokens of type t
                      -- returning a value of type a
string :: String -> Parser String
```

match exact string

```
(>>) :: Parser a -> Parser b -> Parser b
```

sequencing/concatenation (regex •), keeping the result of the second parser

```
many :: Parser a -> Parser [a]
```

zero or more (regex *)

```
sepBy :: Parser a -> Parser b -> Parser [a]
```

zero or more with a separator (e.g. comma)

```
...
```

```
Arithmetic =

E → E + E
E → (E)
E → n
```

Termination

Do recursive descent parsers always terminate?

```
parseExpr =
    parsePlus <|> parseNum <|> parseParens

parsePlus =
    p ← parseExpr
    eat('+')
    q ← parseExpr
    return (p + q)
```

Infinite recursion:

```
parseExpr => parsePlus =>
parseExpr => parsePlus =>
parseExpr => parsePlus =>
parseExpr => ...
```

Left-factoring

- The problem is with *left-recursive* CFG rules, where the production on the right-hand side of the rule starts with the nonterminal on the left-hand side (directly or indirectly)
- We can *left-factor* the grammar to remove this kind of rule
- Left-factoring may cause a combinatorial blowup in the number of CFG rules
- Some tools for creating CFG parsers can automatically left-factor grammars

```
Arithmetic =

    E → E + E
    E → (E)
    E → n
```

```
Arithmetic =

    E → (E)
    E → n
    E → (E) + E
    E → n + E
```

Parsing without state

Since we don't have mutable state in Haskell:

```
type Parser t a = [t] -> Maybe (a, [t])
```

A parser is a function from an input stream to an output value and some suffix of the input (similar to the regex `check` function from HW2).

Parsing without state

So in real Haskell (for example):

```
type Parser t a = [t] -> Maybe (a, [t])


many :: Parser t a -> Parser t [a]
many p cs =
    case p cs of
        Just (x, cs') ->
            case many p cs' of
                Just (xs, cs'') -> Just (x:xs, cs'')
                Nothing -> Just (x, cs')
        Nothing -> Just ([], cs)
```

Parser generation

LL(1) table parsing

An LL(1) parser can be implemented as a finite state automaton, containing the following components:

- A LL(1) grammar (CFG)
- An input stream of tokens (terminals)
- A stack of symbols (nonterminals + terminals)
- A table indicating which rule to apply at each step

LL(1) table parsing

- Start with only the start symbol and $ (end of stream) on the stack
- Loop until the stack is empty:
    - Pop off the the top stack symbol
        - If the stack symbol is a terminal:
            - If it matches the leftmost input token, discard the input token and continue to the next loop iteration
            - If it doesn't match the leftmost input token, return failure
        - If the stack symbol is a nonterminal, look up the corresponding table cell
            - If the cell is empty, return failure
            - Otherwise, record the rule number in the cell, push the corresponding production onto the stack, and continue to the next loop iteration
- On successful completion, the list of recorded rule numbers describes a leftmost derivation of the input string (used to reconstruct the parse tree)

LL(1) table parsing example

1. `S → F`
2. `S → (S + F)`
3. `F → a`

	(	)	a	+	$
S	2		1		
F			3		

- Leftmost input symbol: `'('`
- Top stack symbol: `S`
- Rule number: 2
- Rule: `S → (S + F)`

Input:

`"(a+a)$"`

Old stack:

`S, $`

New stack:

`'(', S, '+', F, ')', $`

Rule record:

`[2]`

(example from Wikipedia page for "LL parser")

LL(1) table parsing example

- Leftmost input symbol: `'('`
- Top stack symbol: `'('`

```
1.  S → F
2.  S → (S + F)
3.  F → a
```

	(	)	a	+	$
S	2		1		
F			3		

Input:

`"(a+a)$"`

Old stack:

`'(', S, '+', F, ')', $`

New stack:

`S, '+', F, ')', $`

Rule record:

`[2]`

(example from Wikipedia page for "LL parser")

LL(1) table parsing example

- Leftmost input symbol: `'a'`
- Top stack symbol: `S`
- Rule number: 1
- Rule: `S → F`

1. S → F
2. S → (S + F)
3. F → a

	(	)	a	+	$
S	2		1		
F			3		

Input:

`"a+a)$"`

Old stack:

`S, '+', F, ')', $`

New stack:

`F, '+', F, ')', $`

Rule record:

`[2, 1]`

(example from Wikipedia page for "LL parser")

LL(1) table parsing example

1. `S → F`
2. `S → (S + F)`
3. `F → a`

- Leftmost input symbol: `'a'`
- Top stack symbol: `F`
- Rule number: 3
- Rule: `F → a`

	(	)	a	+	$
S	2		1		
F			3		

Input:

`"a+a)$"`

Old stack:

`F, '+', F, ')', $`

New stack:

`a, '+', F, ')', $`

Rule record:

`[2, 1, 3]`

(example from Wikipedia page for "LL parser")

LL(1) table parsing example

- Leftmost input symbol: `'a'`
- Top stack symbol: `'a'`

(example from Wikipedia page for "LL parser")

```
1.  S → F
2.  S → (S + F)
3.  F → a
```

	(	)	a	+	$
S	2		1		
F			3		

Input:

`"a+a)$"`

Old stack:

`a, '+', F, ')', $`

New stack:

`'+', F, ')', $`

Rule record:

`[2, 1, 3]`

LL(1) table parsing example

- Leftmost input symbol: `'+'`
- Top stack symbol: `'+'`

```
1.  S → F
2.  S → (S + F)
3.  F → a
```

	(	)	a	+	$
S	2		1		
F			3		

Input:

`"+a)$"`

Old stack:

`'+', F, ')', $`

New stack:

`F, ')', $`

Rule record:

`[2, 1, 3]`

(example from Wikipedia page for "LL parser")

LL(1) table parsing example

- Leftmost input symbol: `'a'`
- Top stack symbol: `F`
- Rule number: 3
- Rule: `F → a`

```
1.  S → F
2.  S → (S + F)
3.  F → a
```

	(	)	a	+	$
S	2		1		
F			3		

Input:

    "a)$"

Old stack:

    F, ')', $

New stack:

    a, ')', $

Rule record:

    [2, 1, 3, 3]

(example from Wikipedia page for "LL parser")

LL(1) table parsing example

- Leftmost input symbol: `'a'`
- Top stack symbol: `'a'`

```
1.  S → F
2.  S → (S + F)
3.  F → a
```

	(	)	a	+	$
S	2		1		
F			3		

Input:

`"a)$"`

Old stack:

`a, ')', $`

New stack:

`')', $`

Rule record:

`[2, 1, 3, 3]`

(example from Wikipedia page for "LL parser")

LL(1) table parsing example

- Leftmost input symbol: `')'`
- Top stack symbol: `')'`

```
1.  S → F
2.  S → (S + F)
3.  F → a
```

	(	)	a	+	$
S	2		1		
F			3		

Input:

`")$"`

Old stack:

`')', $`

New stack:

`$`

Rule record:

`[2, 1, 3, 3]`

(example from Wikipedia page for "LL parser")

LL(1) table parsing example

- Leftmost input symbol: `'$'`
- Top stack symbol: `'$'`

1. S → F
2. S → (S + F)
3. F → a

	(	)	a	+	$
S	2		1		
F			3		

Input:

"$"

Old stack:

$

New stack:

Rule record:

[2, 1, 3, 3]

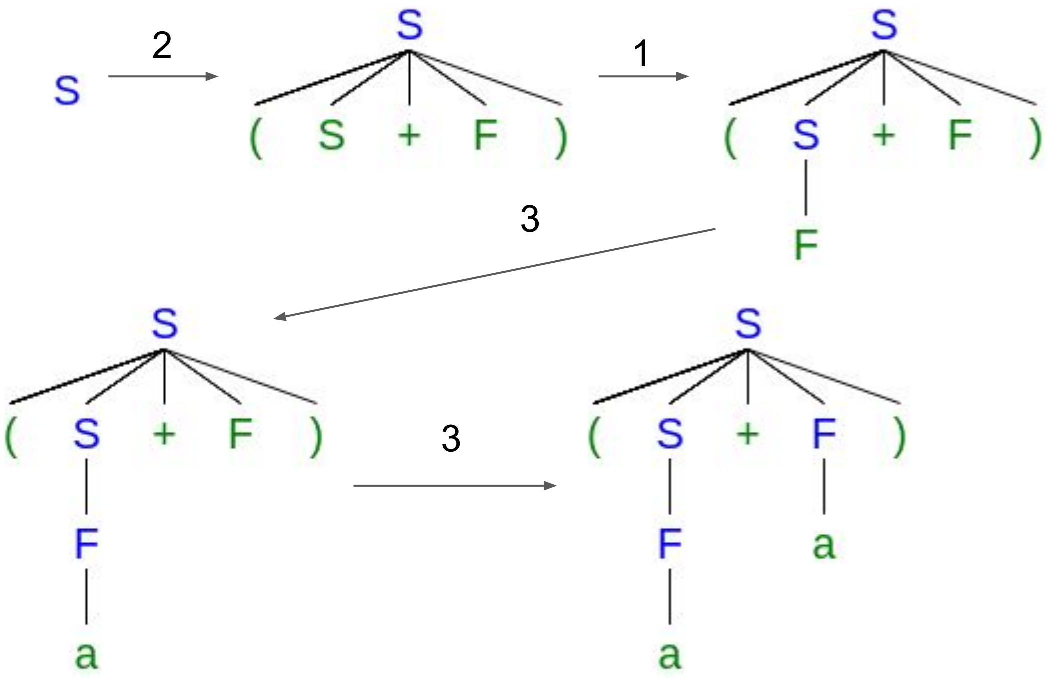
(example from Wikipedia page for "LL parser")

LL(1) table parsing example

1. S → F
2. S → (S + F)
3. F → a

	(	)	a	+	$
S	2		1		
F			3		

What parse tree did we get?



Input:

""

Old stack:

New stack:

Rule record:

[2, 1, 3, 3]

(example from Wikipedia page for "LL parser")

LL(1) table parsing example

What if it fails?

- No entry in the table!

(example from Wikipedia page for "LL parser")

```
1.  S → F
2.  S → (S + F)
3.  F → a
```

	(	)	a	+	$
S	2		1		
F			3		

Input:

```
")a+a($"
```

Old stack:

```
S, $
```

New stack:

Rule record:

```
[]
```

Parser table generation

- There exists an algorithm
  - (beyond the scope of this lecture)
- Tables can get very big very fast
  - Most cells are usually empty: use a sparse array representation
- LL(k) parsing is *top-down* (starts with the root) and produces a *leftmost derivation*
- LR(k) parsing is *bottom-up* (starts with the leaves) and produces a *rightmost derivation*
  - Automata for LR(k) parsing are similar to the LL(k) automaton
  - Tables can be more compressed (SLR, LALR)
  - Still linear time, but often better constant factors

Parsing summary

- Parsers show up in many parts of software development
- Common approaches give methods to automatically construct parsers from grammar definitions
  - Sometimes grammars need to be tweaked by hand
- Subsets of the class of context-free languages describe the sets of languages that can be parsed by specific classes of parsers
  - LL(k), LR(k), LALR, SLR, …
  - Linear worst-case performance in general
  - Can be parsed by automata or hand-constructed parsers
- Parser combinator libraries offer a flexible method of constructing parsers by hand within some host language