Predictive Parsing, LL(k) Parsers

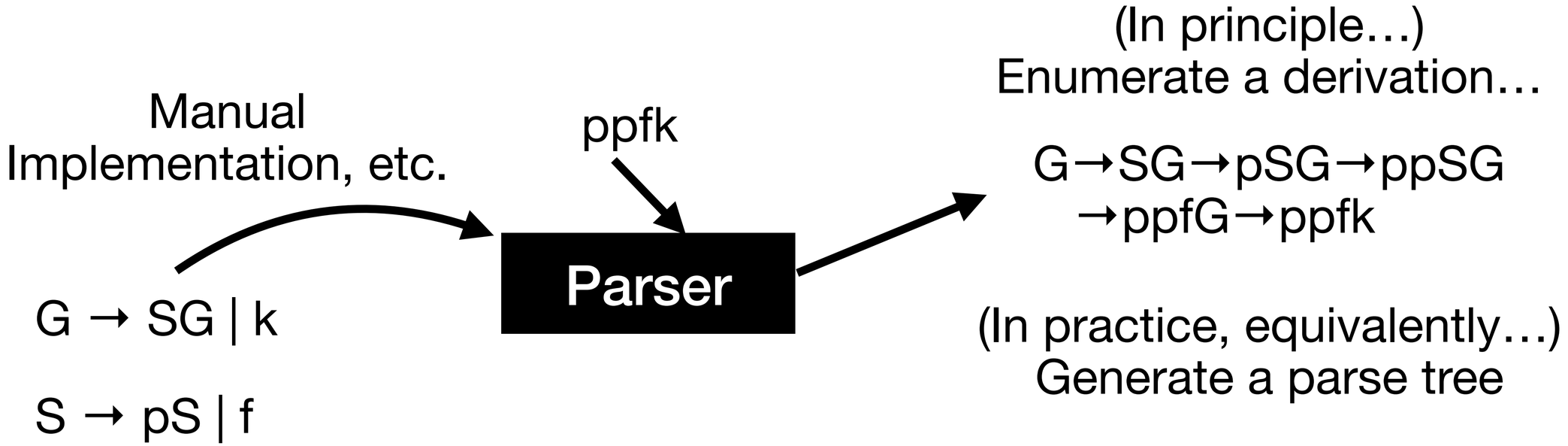
CIS531 — Fall 2025, Syracuse

Kristopher Micinski

Syracuse University
College of Engineering
& Computer Science

How do we *parse* a given grammar

- Last week: talked about **grammars**, talked about **derivations**
  - A grammar is defined via terminals / nonterminals, start symbol, and productions (rules)
  - Derivations start with the start with the start symbol, follow productions
- This week: given a grammar, how to implement a parser that recognizes it



If we have a grammar, how do we recognize it…?

- If we start with a given grammar:

$$S \rightarrow \text{"a"}\ S \mid \text{"a"}\ X$$
$$X \rightarrow \text{"b"} \mid \text{"c"}$$

- The question is how to *recognize* it. We can view this as a search problem (Earley algorithm):
  - Start with the start symbol (S, in this case)
  - Now, consider all possible (branching) derivations, starting from G by running *any* rule
  - This yields a large branching space of possible parse trees
    - When two rules apply—**branch, try both**, generate all possible derivations in parallel
  - 👍 Intuitive, obviously correct
  - 👎 To recognize “aaaaaaaaaaaaa…ab” we explore O(2^n) branches before seeing the final b!

Derivations: Leftmost, Rightmost…

- **Definition**: A given grammar may have any number of derivations for a given string. Among these derivations, we will label two important classes:
  - A **leftmost** derivation is a derivation such that the *expanded* nonterminal is always in the leftmost position, lexically
  - A **rightmost** derivation is a derivation such that the expanded nonterminal is always in the rightmost position, lexically

S → A B
A → "x" A | "y"
B → "z" B | "w"

S → A B
→ "x" A B
→ "x" "x" A B
→ "x" "x" “y” B
→ "x" "x" "y" "z" B
→ "x" "x" "y" "z" "w"

S → A B
→ A “z” B
→ A "z" "w"
→ "x" A "z" "w"
→ "x" "x" A "z" "w"
→ "x" "x" "y" "z" "w"

Definition: When a grammar is *ambiguous*

- A grammar is **ambiguous** iff there exists some string s in the grammar such that s has two *distinct* leftmost derivations (equivalently, two distinct rightmost derivations)
- Below are two grammars—one is ambiguous (this definition), one is not
  - Explain which one is ambiguous—you **must** show multiple leftmost derivations

S → A B
A → "x"
B → "y"

G → G G | "a"

Definition: When a grammar is *ambiguous*

- A grammar is **ambiguous** iff there exists some string s in the grammar such that s has two *distinct* leftmost derivations (equivalently, two distinct rightmost derivations)
- Below are two grammars—one is ambiguous (this definition), one is not
  - Explain which one is ambiguous—you **must** show multiple leftmost derivations

S → A B
A → "x"
B → "y"

G → G G | "a"

Answer: this one, because

G → G G → “a” G → “a” G G → “a” “a” G → “a” “a” “a”

G → G G → G G G → “a” G G → “a” “a” G → “a” “a” “a”

More Practice: Ambiguous Grammars...

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

Draw a leftmost derivation for...

```
1 / 2 / 3
```

Now draw *another* leftmost derivation

Why is ambiguity important?

- If our grammar is ambiguous there is **no hope to writing a parser** other than one which tries to find a *set* of derivations, we want derivations to be **unique**
  - Put differently: we don't want to write in a programming language where one string has multiple parse trees—it means that there has to be some policy to disambiguate!
    - Very confusing—in practice, we **disallow** ambiguous grammars for programming
    - However, ambiguous grammars are *common* in natural language
- We will focus on unambiguous grammars

LL(1), Predictive Parsing

- An LL parser works (a) L—*left-to-right* and produces the (b) L—*leftmost derivation*
- An LL(k) parser is an LL parser that uses k tokens of *lookahead*
- We will expand on these definitions in a few slides, but for now, here is an example of an LL(1) grammar:

$$S \rightarrow A$$
$$S \rightarrow BA$$
$$A \rightarrow \text{“x”}$$
$$B \rightarrow \text{“y”}$$

- Intuitively, the grammar has the property that we can look at one token of lookahead (e.g., “x” or “y”) and pick which production has to apply
  - If we’re trying to parse S and we see “x”, we know we want $S \rightarrow A$
  - If we’re trying to parse S and we see “y”, we know we want $S \rightarrow BA$

Motivation: FIRST and FOLLOW sets…

- When a grammar is LL(1), then I can look at a single character and *predict* which rule I **must** apply if a parse tree exists—it could fail, but structure of the grammar **commits** us to a choice

S → A
S → BA
A → “x”
B → “y”

- Consider any leftmost derivation…
  - Begins with… S →
  - If I see “x”, then then choosing S → BA will **never** work, why?
    - Answer: because B requires me to first match “y”
  - If I see “y”, then choosing S → A will never work, why?
    - Answer: because A requires me to first match “x”

Definition: FIRST set

- We now present the definitions of the FIRST and FOLLOW sets of a grammar
- Given a grammar G, consisting of terminals, nonterminals, rules, and a symbol…
- FIRST(X) is the set of all terminals that can appear as the first symbol of some string derived from X, where X is either a nonterminal, a terminal, or a sequence of grammar symbols.
  - If X is a terminal, then FIRST(X) = {X}.
  - If X can derive ε (the empty string), then ε ∈ FIRST(X).
  - If X is a sequence (e.g., YZ…), then FIRST(X) ⊇ FIRST(Y) (excluding ε), and if Y ⇒* ε, then also FIRST(Z), and so on.
- For example, for the grammar:

$$S \rightarrow AB$$
$$A \rightarrow \varepsilon \mid a$$
$$B \rightarrow b$$

FIRST(A) = {ε, a}
FIRST(B) = {b}
FIRST(S) = {a,b}

Definition: FOLLOW set

- For a nonterminal A, FOLLOW(A) is the set of terminals that can appear *immediately to the right* of A
  - If S is the start symbol then $ ∈ FOLLOW(A), where $ is the end-of-stream symbol
  - For any production that looks like B → αAaβ, then a ∈ FOLLOW(A)
  - For any production that looks like B → αAXβ, where X is a nonterminal…
    - (FIRST(X) \ {ε}) ⊆ FOLLOW(A)
  - If there is a production B → αAβ and β is nullable (β can derive ε), then…
    - FOLLOW(B) ⊆ FOLLOW(A)
  - We apply these rules iteratively until we learn no more information…

Example: Calculating a FOLLOW set

- Let's calculate FOLLOW for our tiny grammar…
  S → A B
  A → ε | a
  B → b
- FOLLOW(S): start symbol, {$}
- FOLLOW(A): in S → A B, we can see that B follows A. So we add FIRST(B), i.e., {b}
- FOLLOW(B): because we have S → A B. Now now that FOLLOW(B) includes FOLLOW(S), so we conclude that FOLLOW(B) = {$}
- Thus…
  FOLLOW(S) = {$}
  FOLLOW(A) = {b}
  FOLLOW(B) = {$}

Definition: LL(1) Condition

- A grammar is LL(1) iff, for each nonterminal in the grammar A:
  - For each pair of productions $A \to \alpha$ and $A \to \beta$ with $\alpha \neq \beta$...
    - $\text{FIRST}(\alpha) \cap \text{FIRST}(\beta) = \varnothing$
      - Intuitively: “The FIRST sets of each production of A are disjoint”
      - Consequence: the first set *uniquely determines* the production
    - If $\varepsilon \in \text{FIRST}(\alpha)$, then...
      - $(\text{FIRST}(\alpha) \setminus \{\varepsilon\}) \cap \text{FOLLOW}(A) = \varnothing$
      - Intuitively: No conflict between choosing $A \to \varepsilon$ and some other production

More Practice: FIRST/FOLLOW sets

- Another grammar to try:
- S → A B
  A → x | y
  B → z | t
- FIRST(A) = {x,y}, FIRST(B) = {z,t}, FIRST(S) = FIRST(A) = {x,y}
- FOLLOW(S) = {$}, FOLLOW(A)={z,t}, FOLLOW(b)={$}
- Does the grammar have the LL1 property?
  - Yes, the first sets for A,B's alternative branches are disjoint!

Writing the code: Recursive Descent Parsers

- If a grammar is LL(1) we can write a **very simple** parser called a “recursive descent” parser
- Idea: use a function peek() to get the *lookahead*
  - Because rules are disjoint, decide which production to take based on lookahead!
  - Each nonterminal A turns into a recursive function, parse_A which:
    - Branches on the lookahead using peek():
      - Decides which production to apply
      - ***Always possible!*** LL(1) conditions force productions to have *disjoint* START sets
    - When we expect a terminal, call a function consume(‘a’) which expects the next character to be ‘a’ and advances the token stream (error if no match for ‘a’)

An example: LL(1) parser

```
S → A B
A → ε | 'a'
B → 'b'
```

```
def parse_S():
  parse_A()
  parse_B()

def parse_A():
  if peek() == ‘a’:
    consume(‘a’)
    return
  else:
    return
```

```
def parse_B():
  consume(‘b’)
```

Example 2: LL(1) parser….

```
S → ‘x’ S | ‘y'
```

```
def parse_S():
  if peek() == ‘x’:
    consume(‘x’)
    # Notice the recursion…
    parse_S()
  elif  peek() == ‘y’:
    consume(‘y’)
  else:
    error(“parse error, …”)
```

Example 3

```
S    → id Rest
Rest → ‘(‘ Rest ‘)’ | id
```

```
def parse_id():
  # Match current token again
  # a class of identifiers…

def parse_S():
  parse_id()
  parse_Rest()

parse_Rest():
  if peek() == ‘(‘:
    consume(‘(‘)
    parse_Rest()
    consume(‘)’)
  else:
    parse_id()
```

Writing recursive descent parsers in *Racket*

- In pseudo-code we used peek() and consume()… but they are very mutable
  - Advance some globalized notion of the “current token”
- In Racket, we want to write ***purely functional*** code, so no possibility of doing this
  - Also, instead of just *matching* as we did in pseudocode, we want to return a tree
- Key idea: each nonterminal A turns into a function which accepts the input stream (list)
  - Returns ***two*** values:
    - (a) the syntax tree (result), along with…
    - (b) the *rest* of the unconsumed token stream

Issue: Left Recursion

- Recursive descent parsers are simple, intuitive, and generally easy to write
  - 👎 Unfortunately, *not all grammars are LL(1)*
- One clear issue: LL parsers can **not** handle left recursion:

```
A → A - P | P
P → P / I | I
I → number
```

- This grammar *does* have left recursion, and it is <u>helpful</u> in the following way:
- 1 - 2 - 3 should be parsed as (1 - 2) - 3:
  - A → A - P → A - P - P → P - P - P → 1 - P - P → 1 - 2 - P → 1 - 2 - 3
  - If we draw the parse tree, we see that we get the intended associativity for -

LL parsers cannot handle left recursion…

- Grammars with left recursion are *never* LL(k) for any k…

```
A → A - P | P
P → P / I | I
I → number
```

- To write parse_A we would **immediately recur on A**
  - Yields infinite recursion! Violates LL principles: bounded lookahead predicts production
- So no way to write the above grammar using a recursive descent parser…

```
def parse_A():
  parse_A() # infinite recursion…
  consume('-') # never get here…
  parse_A()
```

Grammar Transformations: Left Factoring

- In some cases, we can **rewrite** a grammar to be LL(k), one example is *left factoring*
- For example, if we have a rule $A \rightarrow \alpha\beta_1 \mid \alpha\beta_2$,
- Any grammar including this rule not LL(1): the FIRST sets of both productions include α
- We apply left factoring to split the rule into two rules:
  $A \rightarrow \alpha A'$
  $A' \rightarrow \beta_1 \mid \beta_2$
- Common transformation—allows us to make LL(k) parsers LL(1)
  - But in practice: much more natural to write the rules as LL(k)
- Some grammars ***cannot*** be made LL(k)
  - For these we use hacks (left association) or (more common) use LR parsing
    - LALR, SLR, etc.
    - We will not cover these—but I will demo Yacc a bit

LR (shift/reduce) parsing

- We did not talk much about the other large class of parsing algorithms, LR parsers
- LR(k) parsers construct the ***rightmost*** derivation, working left-to-right
  - Nice advantage—no issue with left recursion in grammars!
  - (Handle associativity properly, no factoring/tricks)
- **Key idea:** maintain a stack of symbols (terminals / nonterminals)
  - At every (next) input, you can either shift onto the stack, or reduce the stack by applying a transformation via two tables:
  - *Action table*: shift, reduce, accept, error
  - *Goto* table: jump post-reduction
- 👍 — works for most languages you'd want to write, fast to implement
- 👎 — requires a parser generator (tables are too tedious to do by hand for any nontrivial language), shift/reduce, reduce/reduce conflicts are hard to debug!

Parsing: Fin

- My goal was to give you the basics of grammars, along with their key properties and transformations. Can you define: grammar, LL(k), LR, recursive descent?
- What to know / practice: could you write a simple recursive-descent parser?
- One exam problem (making clear now): given some relatively simple grammar, can you write a recursive descent parser?
  - You can use any language—if you want to use pseudocode, fine, as long as I can get the idea