LL Parsing

Dr. Mattox Beckman

UNIVERSITY OF ILLINOIS AT URBANA-CHAMPAIGN
DEPARTMENT OF COMPUTER SCIENCE

Objectives

The topic for this lecture is a kind of grammar that works well with recursive-descent parsing.

- Classify a grammar as being LL or not LL.
- Use recursive-descent parsing to implement an LL parser.
- Explain how left-recursion and common prefixes defeat LL parsers.

What Is LL(n) Parsing?

- An LL parse uses a **L**eft-to-right scan and produces a **L**eftmost derivation, using **n** tokens of lookahead.
- A.k.a. top-down parsing

Example Grammar:

$$S \rightarrow + \, E \, E$$
$$E \rightarrow \texttt{int}$$
$$E \rightarrow * \, E \, E$$

Syntax Tree:

S

Example Input:

`+ 2 * 3 4`

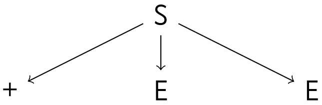
What Is LL(n) Parsing?

- An LL parse uses a **L**eft-to-right scan and produces a **L**eftmost derivation, using **n** tokens of lookahead.
- A.k.a. top-down parsing

Example Grammar:

$$S \rightarrow + \; E \; E$$
$$E \rightarrow \texttt{int}$$
$$E \rightarrow * \; E \; E$$

Syntax Tree:



Example Input:

`+ 2 * 3 4`

What Is LL(n) Parsing?

- An LL parse uses a **L**eft-to-right scan and produces a **L**eftmost derivation, using **n** tokens of lookahead.
- A.k.a. top-down parsing

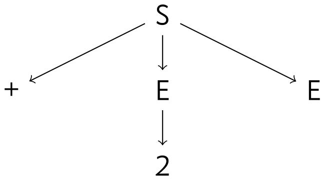
Example Grammar:

$$S \rightarrow + \, E \, E$$
$$E \rightarrow \texttt{int}$$
$$E \rightarrow * \, E \, E$$

Example Input:

`+ 2 * 3 4`

Syntax Tree:



What Is LL(n) Parsing?

- An LL parse uses a **L**eft-to-right scan and produces a **L**eftmost derivation, using **n** tokens of lookahead.
- A.k.a. top-down parsing

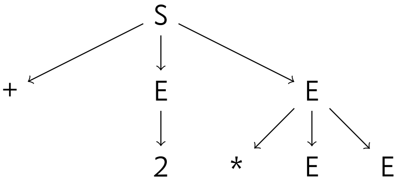
Example Grammar:

$$S \to + \, E \, E$$
$$E \to \texttt{int}$$
$$E \to * \, E \, E$$

Example Input:

`+ 2 * 3 4`

Syntax Tree:



What Is LL(n) Parsing?

- An LL parse uses a **L**eft-to-right scan and produces a **L**eftmost derivation, using **n** tokens of lookahead.
- A.k.a. top-down parsing

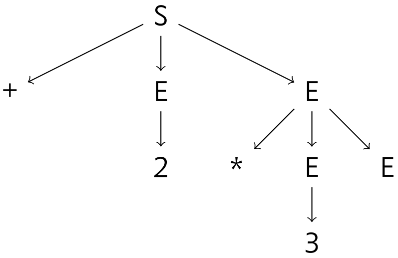
Example Grammar:

$$S \to + \, E \, E$$
$$E \to \texttt{int}$$
$$E \to * \, E \, E$$

Example Input:

`+ 2 * 3 4`

Syntax Tree:



What Is LL(n) Parsing?

- An LL parse uses a **L**eft-to-right scan and produces a **L**eftmost derivation, using **n** tokens of lookahead.
- A.k.a. top-down parsing

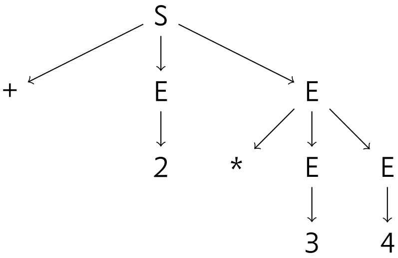
Example Grammar:

$$S \rightarrow + \; E \; E$$
$$E \rightarrow \texttt{int}$$
$$E \rightarrow * \; E \; E$$

Example Input:

`+ 2 * 3 4`

Syntax Tree:



How to Implement It

Interpreting a Production

- Think of a production as a function definition.
- The LHS is the function being defined.
- Terminals on RHS are commands to consume input.
- Nonterminals on RHS are subroutine calls.

- For each production, make a function of type `[String] -> (Tree,[String])`.
  - Input is a list of tokens.
  - Output is a syntax tree and remaining tokens.
- Of course, you need to create a type to represent your tree.

Things to Notice

Key Point – Prediction

- Each function immediately checks the first token of the input string to see what to do next.

```
getE [] = undefined
getE ('*':xs) =
   let e1,r1 = getE xs
       e2,r2 = getE r1
    in (ETimes e1 e2, r2)
getE .... -- other code follows
```

Left Recursion

Left Recursion Is Bad

- A rule like $E \to E + E$ would cause an infinite loop.

```
getE xx =
  let e1,r1 = getE xx
      ('+':r2) = r1
      e2,r3 = getE r2
   in (EPlus e1 e2, r3)
```

Rules with Common Prefixes

Common Prefixes Are Bad

- A pair of rules rule like $\begin{array}{rl} E \rightarrow & - E \\ \mid & - E\ E \end{array}$ would confuse the function.
  Which version of the rule should be used?

```
getE ('-':xs) = ... -- unary rule
getE ('-':xs) = ... -- binary rule
```

- NB: Common prefixes must be for the *same* nonterminal. E.g., $E \rightarrow x\ A$ and $S \rightarrow x\ B$ do not count as common prefixes.