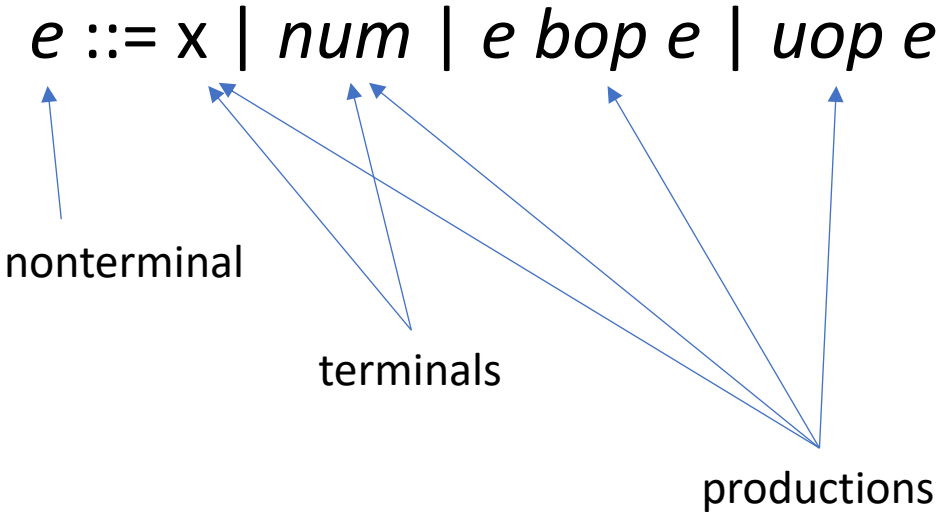
CS443: Compiler Construction

Lecture 3: Grammars, LL parsing

BNF grammars are “context-free”

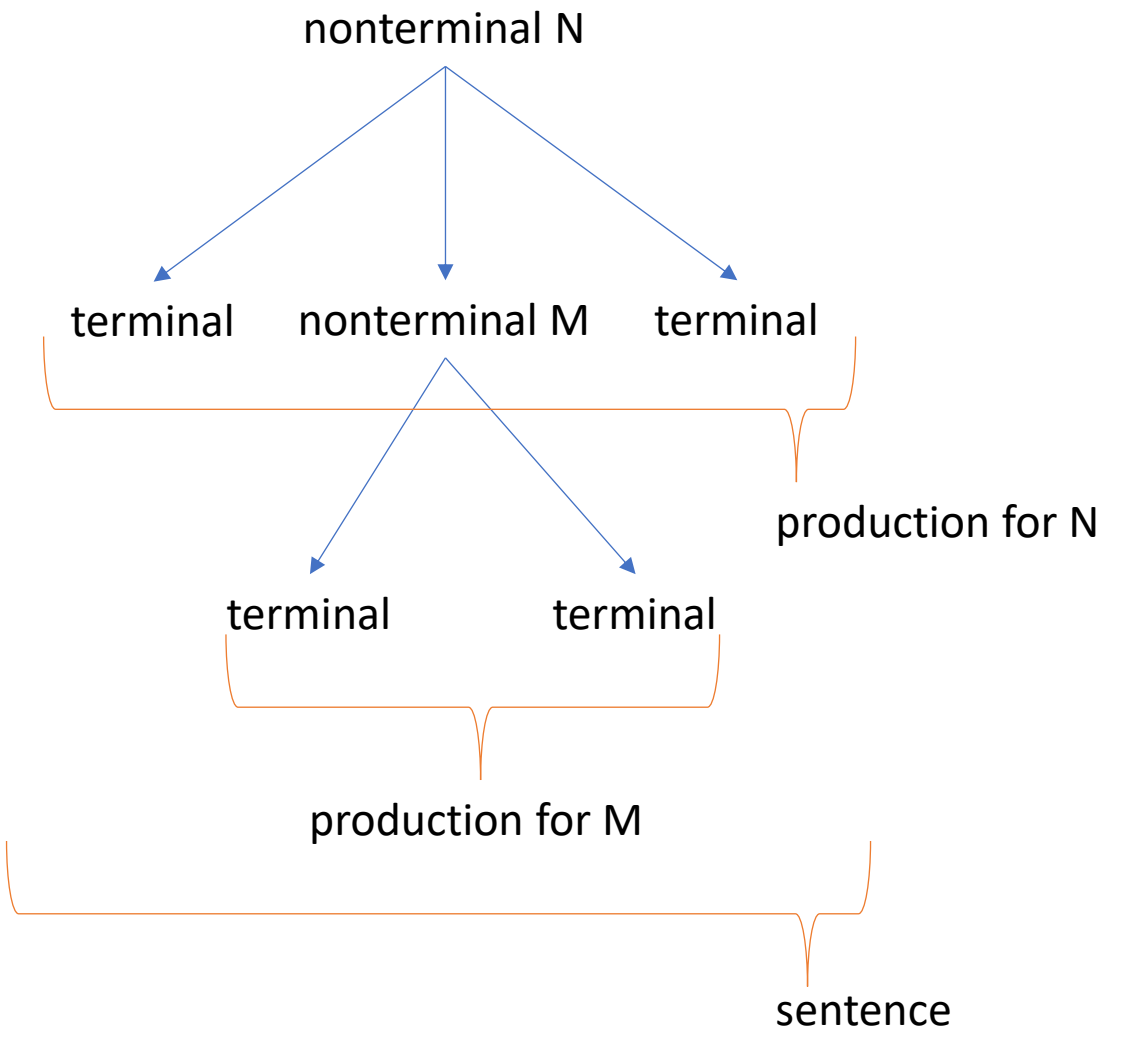


Derivation: Expand one nonterminal at a time using a production

e ::= n | e + e | (e) Input: 1 + (2 + 3)

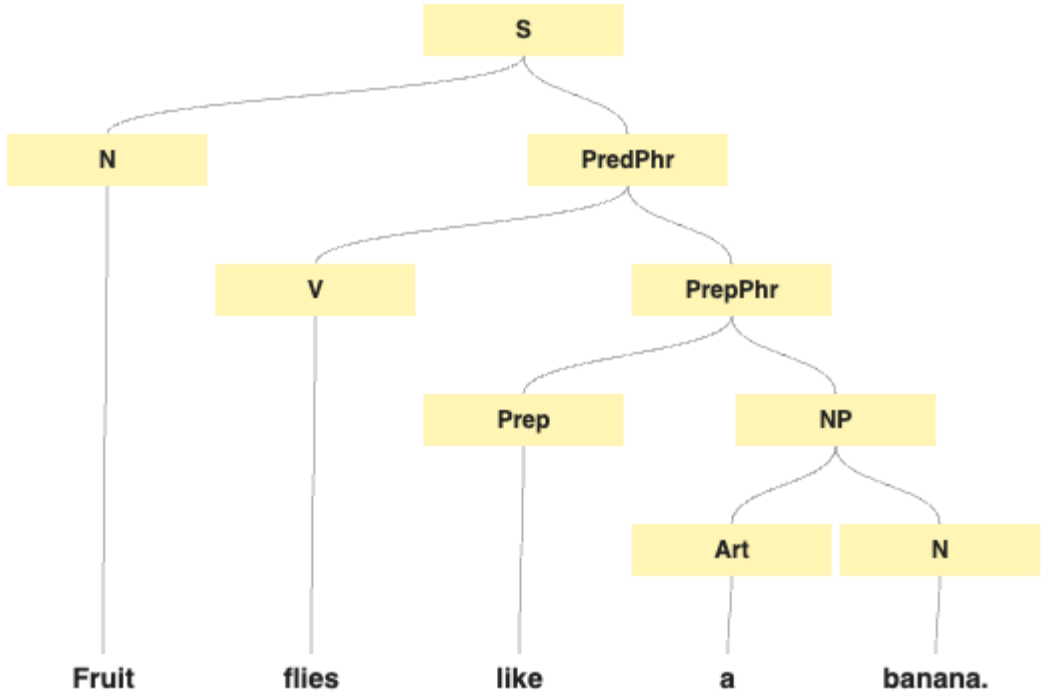
Leftmost Derivation	Rightmost Derivation
e	e
e + e	e + e
1 + e	e + (e)
1 + (e)	e + (e + e)
1 + (e + e)	e + (e + 3)
1 + (2 + e)	e + (2 + 3)
1 + (2 + 3)	1 + (2 + 3)

Parsing: Produce a *parse tree* from a stream of tokens

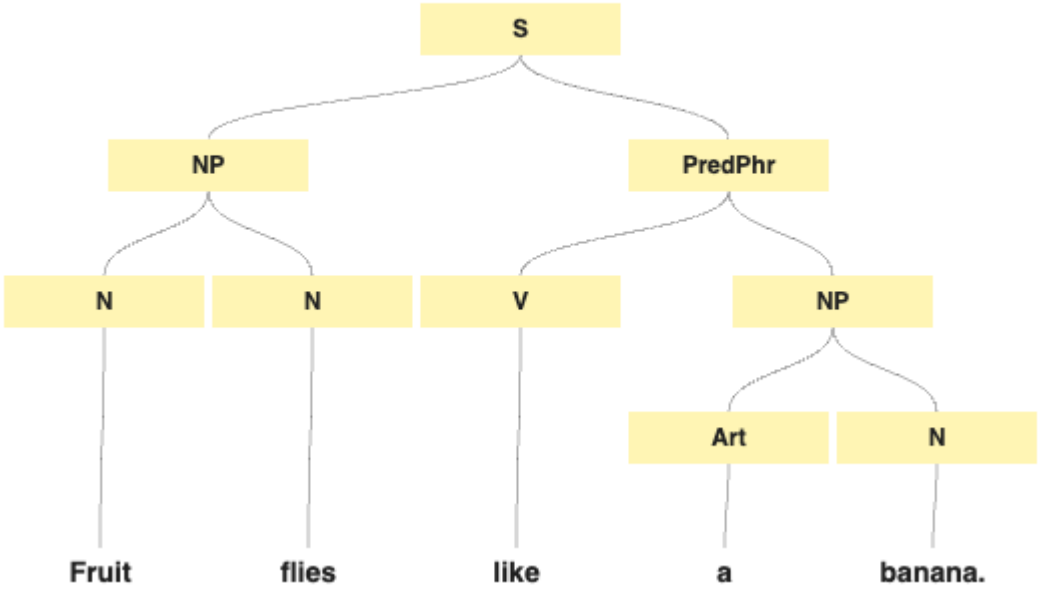


Ambiguous grammars allow multiple correct parse trees

- “Fruit flies like a banana”



(Diagrams courtesy Prof. Hannah Ringler)



Corollary: In an unambiguous grammar, the leftmost derivation and rightmost derivation (and all other derivations) have the same parse tree

Associativity is one source of ambiguity

- *e ::=* num | *e* + *e* | *e* – *e*
- 1 + 2 + 3 + 4 → ((1 + 2) + 3) + 4, (1 + 2) + (3 + 4), …
- Solution:
  e ::= num | e + num | e – num | *e* + (*e*) | *e* – (*e*)

Precedence is one possible source of ambiguity

- Abstract syntax: *e ::= e* + *e* | *e* – *e* | *e* * *e* | *e* / *e*
- 1 + 2 * 3 – 4 ????

- Solution: Factoring out productions

- *f ::=* num | (*e*)
  t ::= f | *t* * *f* | *t* / *f*
  e ::= t | *e* + *t* | *e* - *t*

Classic example: “dangling else”

- *s* ::= if *e s* | if *e s* else *s*

- if e_1 if e_2 s_1 else s_2
  - By convention: if e_1 (if e_2 s_1 else s_2)

- Solution:
  closedstmt ::= if *e closedstmt* else *closedstmt*
  | … (non-if stmts not ending with an *openstmt*)
  openstmt ::= if *e closedstmt* else *openstmt* | if *e stmt*
  | … (non-if stmts ending with an *openstmt*)
  stmt ::= *openstmt* | *closedstmt*

Recursive descent (or “predictive”) parsing – simple algorithm for simple grammars

```
let rec parse_e (l: token list) =
  match l with
  | IF::l’ ->
      (match parse_e l’ with
       | THEN::l’’ ->
         (match parse_e l’’ with
          | ELSE::l’’’ -> parse_e l’’’
          | [] -> error ())
      | [] -> error ())
  | (VAR x)::l’ -> l’
  | [] -> error ()
```

e ::= if e then e else e
| x

Recursive descent (or “predictive”) parsing – simple algorithm for simple grammars

```
let rec parse_e (l: token list) =
  match l with
  | IF::l’ ->
      (match parse_e l’ with
       | THEN::l’’ ->
         (match parse_e l’’ with
          | ELSE::l’’’ -> parse_e l’’’
          | [] -> error ())
      | [] -> error ())
  | (VAR x)::l’ -> l’
  | _ -> (match parse_e l with …
```

e ::= if e then e else e
| x
| e; e

Infinite loop (“left recursion”)

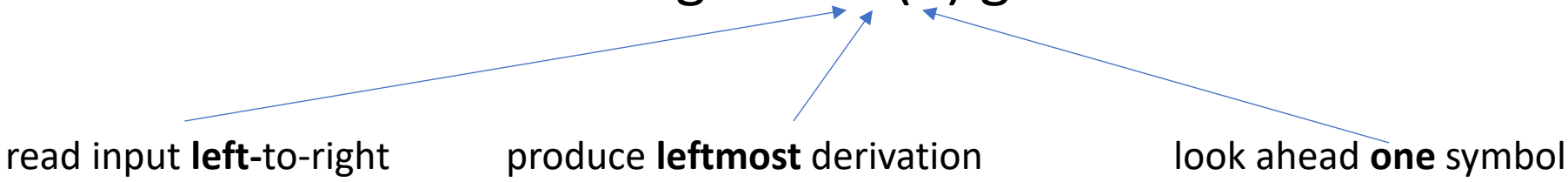
Recursive descent (or “predictive”) parsing – simple algorithm for simple grammars

```
let rec parse_e (l: token list) =
  match l with
  | IF::l’ ->
      (match parse_e l’ with
       | THEN::l’’ ->
         (match parse_e l’’ with
          | ELSE::l’’’ -> parse_e l’’’
          | [] -> error ())
      | [] -> error ())
  | IF::l’ -> …
```

e ::= if e then e else e
| if e then e

Recursive descent (or “predictive”) parsing – simple algorithm for simple grammars

- Works without backtracking for “LL(1) grammars”



See Appel for algorithms for:

- Getting rid of left recursion
- Determining if a grammar is LL(1)
- Building a predictive parser for LL(1) grammars

English is not LL(k) (or LR(k)) for *any* k!

(Contrast: “Where music rebels and creativity thrive”)