CS 160 Compilers

Lecture 11: More about Parsing

Yu Feng

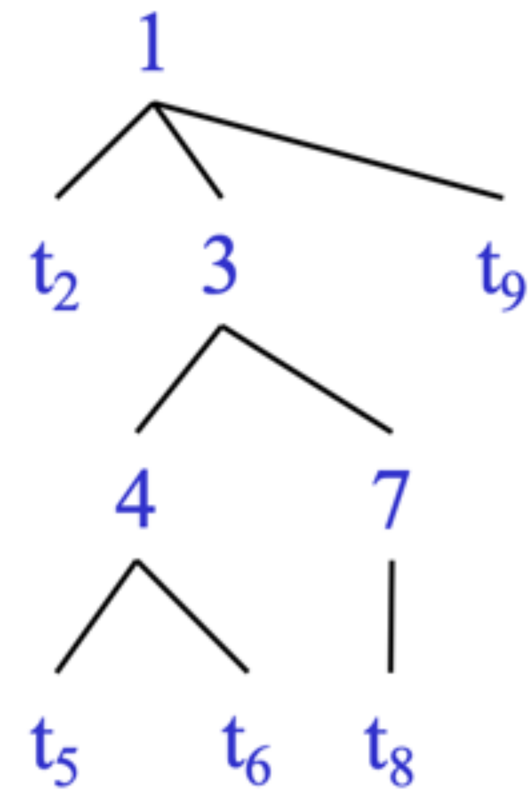
Spring 2023

Parsing as a Search

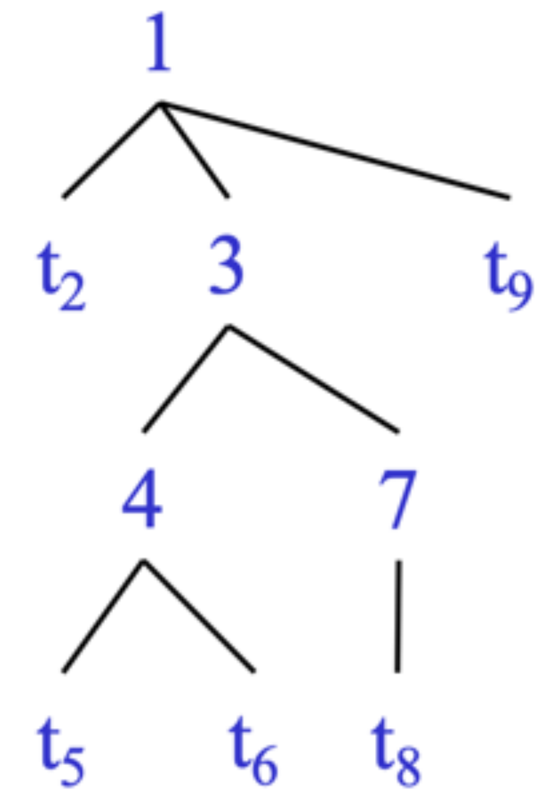
- Idea: treat parsing as a graph search

- Each node is a string of terminals and nonterminal from the start symbol

- There is an edge from node α to node β iff $\alpha \Rightarrow \beta$.



Top-Down parsing: the idea



- The parse tree is constructed
  - From the top
  - From left to right
- Terminals are seen in order of appearance in the token stream:
  - t_2 t_5 t_6 t_8 t_9

Recursive Descent Parsing

Recursive descent parsing

- A Consider the grammar

$$E \rightarrow T \mid T + E$$

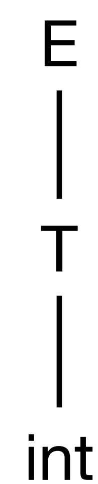
$$T \rightarrow int \mid int * T \mid (E)$$

- Token stream is: (int_5)
- Start with top-level non-terminal E
- Try the rules for E in order

Recursive descent parsing

$E \rightarrow T \mid T + E$

$T \rightarrow int \mid int * T \mid (E)$



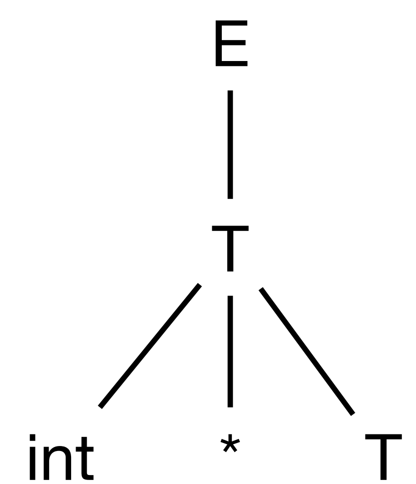
Mismatch: int is not (!
Backtrack …

(int_5)

Recursive descent parsing

$E \rightarrow T \mid T + E$

$T \rightarrow \text{int} \mid \text{int} * T \mid (E)$



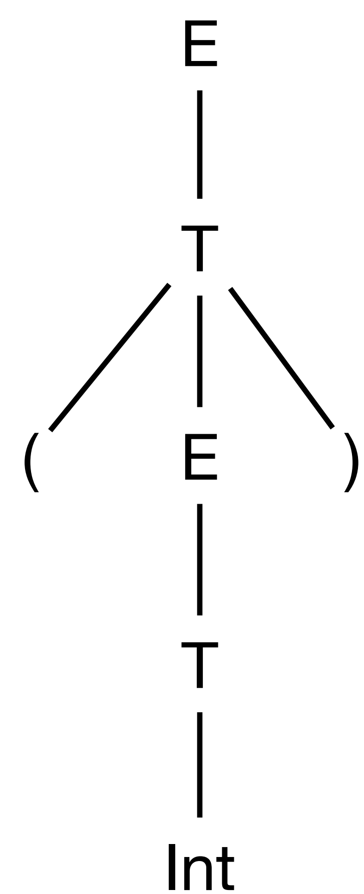
Mismatch: int is not (!
Backtrack …

$(\ \text{int}_5 \ )$

Recursive descent parsing

$E \rightarrow T \mid T + E$

$T \rightarrow \text{int} \mid \text{int} * T \mid (E)$



Match: advance input

Accept: end of input

(int_5)

Problems in Top-Down Parsing

`A → Aa | c`

```
A
Aa
Aaa
Aaaa
Aaaaa
...
```

Eliminate Left Recursion

- A nonterminal A is said to be left recursive iff $A \Rightarrow^* Ac$ for some string c
- Leftmost DFS may fail on left-recursive grammars
- Eliminate left recursion via rewriting the rules

$$A \rightarrow Aa \mid c \quad = \quad \begin{array}{l} A \rightarrow cA' \\ A' \rightarrow aA' \mid \varepsilon \end{array}$$

Challenges in Top-Down Parsing

- Top-down parsing begins with virtually no information
    - Begins with just the start symbol, which matches every program.
- How can we know which productions to apply?
    - In general, we can't.
- There are some grammars for which the best we can do is guess and backtrack if we're wrong.

Top-Down v.s. Bottom-Up

- Top Down Parsing
    - Beginning with the start symbol, try to guess the productions to apply to end up at the user’s program.
- Bottom-Up Parsing
    - Beginning with the user's program, try to apply productions in reverse to convert the program back into the start symbol.

Bottom-up Parsing

$E \rightarrow T$
$E \rightarrow E + T$
$T \rightarrow \texttt{int}$
$T \rightarrow (E)$

```
  int + (int + int + int)
⇒ T + (int + int + int)
⇒ E + (int + int + int)
⇒ E + (T + int + int)
⇒ E + (E + int + int)
⇒ E + (E + T + int)
⇒ E + (E + int)
⇒ E + (E + T)
⇒ E + (E)
⇒ E + T
⇒ E
```

Predictive Parsing

- The leftmost DFS/BFS algorithms are backtracking algorithms.
- Guess which production to use, then back up if it doesn't work.
- Try to match a prefix by sheer dumb luck.
- There is another class of parsing algorithms called predictive algorithms.
- Based on remaining input, predict (without backtracking) which production to use.

Ambiguity

- A grammar is ambiguous if it has more than one parse tree for some string
- Equivalently: There is more than one left-most or right-most derivation for some string
- Ambiguity is bad!
- Leaves meaning of programs ill-defined

Ambiguity

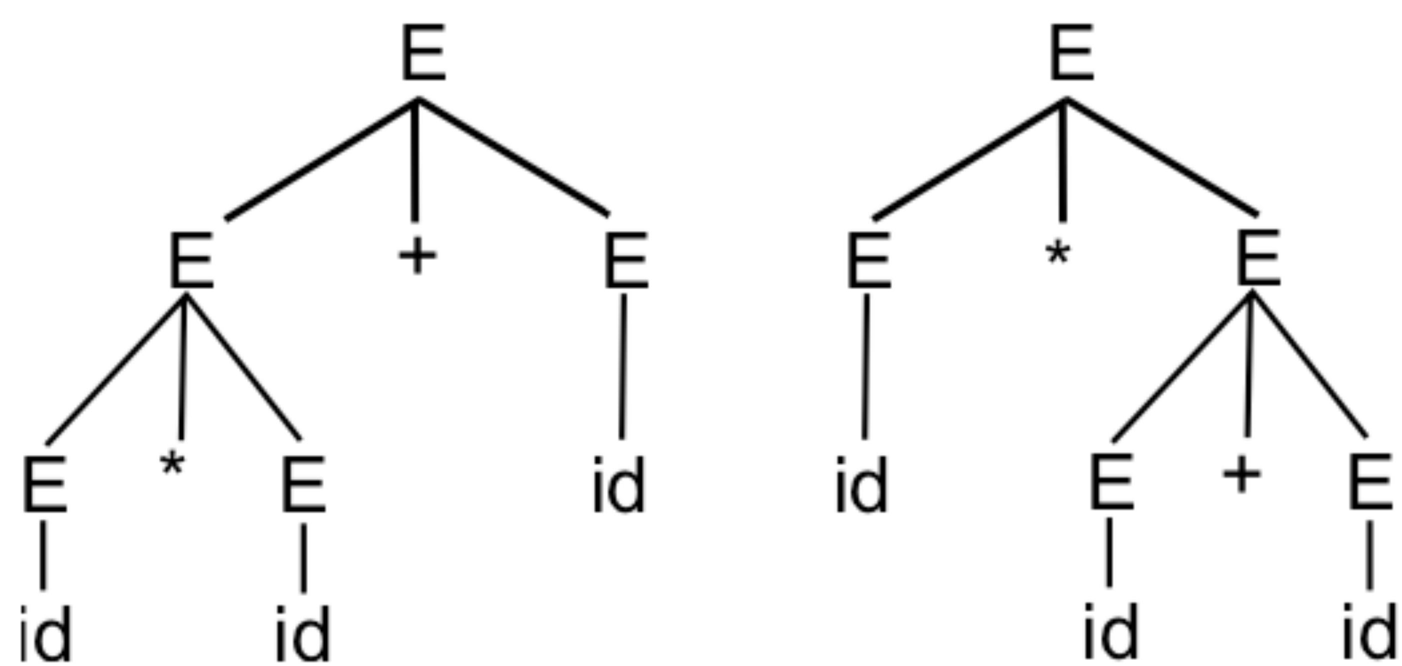
- Consider this grammar:

$EXPR \rightarrow E * E$

$| \; E+E \; | \; (E)$

$| \; id$

- Now, this string *id*id+id* has two parse trees!



Dealing with ambiguity

- Solution: Eliminate ambiguity by adding nonterminals and allowing recursion only on the left (or right)

- Higher-precedence operators go farther from the start symbol.

```
S → S + S | S * S | ( S ) | number
```

$$\Downarrow$$

```
S0 → S0 + S1 | S1
S1 → S2 * S1 | S2
S2 → number | ( S0 )
```