CS 160 Compilers

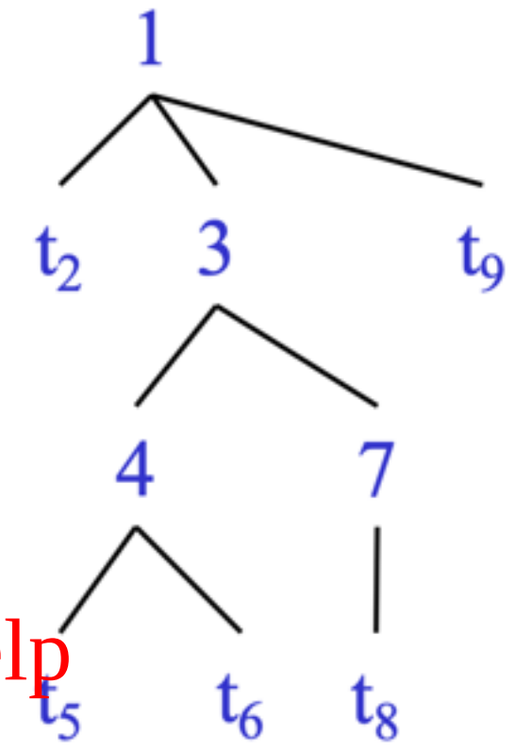
Lecture 11: More about Parsing

Yu Feng

Spring 2023

Parsing as a Search

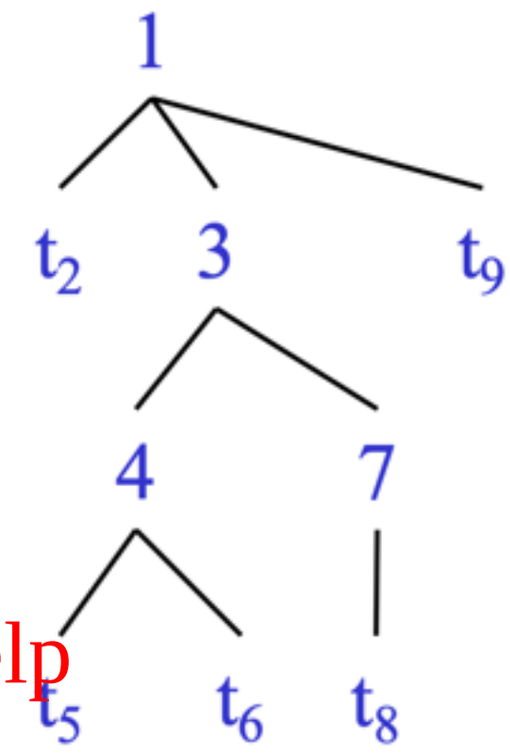
- Idea: treat parsing as a search
- Each node is a string of terminals and nonterminal from the start symbol.
- There is an edge from node α to node β iff $\alpha \Rightarrow \beta$.



Top-Down parsing: the idea

- The parse tree is c[illegible]

  - 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

- A Consider the gramm

→ T | T + E

T → int | int * T | (E)

- Token stream is: (int_5)

- Start with top-level non-terminal E

- Try the rules for E in order

Recursive descent parsing

$E \rightarrow T \mid T + E$

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

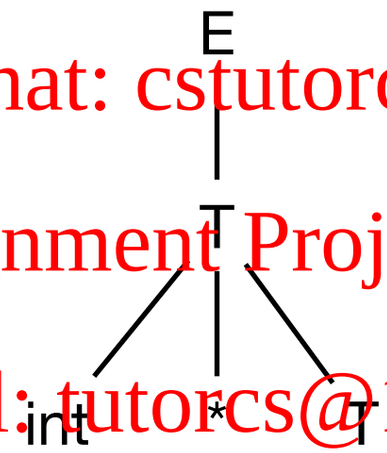
Mismatch: int is not (!
Backtrack …

(int_5)

Recursive descent parsing

$E \rightarrow T \mid T + E$

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



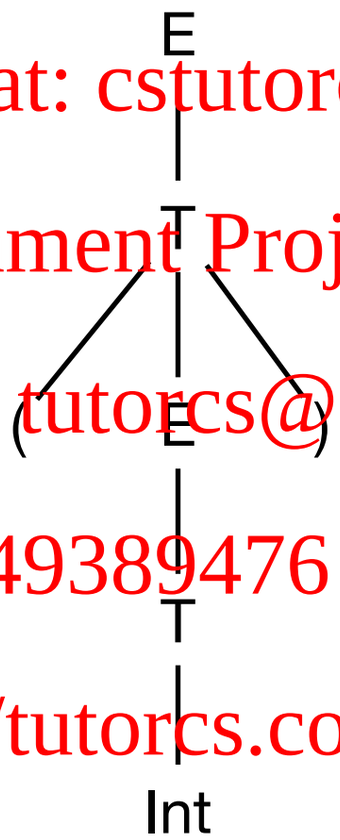
Mismatch: int is not (!
Backtrack …

(int_5)

Recursive descent parsing

$E \rightarrow T \mid T + E$

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



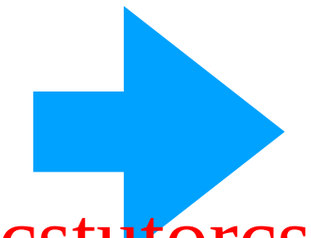
Match: advance input

Accept: end of input

(int_5)

Problems in Top Down Parsing

A → Aa



A

Aa

Aaa

Aaaa

Aaaaa

...

Eliminate Left Recursion

- A nonterminal A is left recursive iff A ⇒* Ac for some string c

- Leftmost DFS may fail on left-recursive grammars

- Eliminate left recursion via rewriting the rules

$$A \rightarrow Aa \mid c$$

$$A \rightarrow cA'$$
$$A' \rightarrow aA' \mid \varepsilon$$

Challenges in Top-Down Parsing

- Top-down parsing b[illegible] 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 vs. 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

$$
\begin{aligned}
E &\rightarrow T \\
E &\rightarrow E + T \\
T &\rightarrow \texttt{int} \\
T &\rightarrow (E)
\end{aligned}
$$

$$
\begin{aligned}
& \texttt{int} + (\texttt{int} + \texttt{int} + \texttt{int}) \\
\Rightarrow\ & \mathbf{T} + (\texttt{int} + \texttt{int} + \texttt{int}) \\
\Rightarrow\ & \mathbf{E} + (\texttt{int} + \texttt{int} + \texttt{int}) \\
\Rightarrow\ & E + (\mathbf{T} + \texttt{int} + \texttt{int}) \\
\Rightarrow\ & E + (\mathbf{E} + \texttt{int} + \texttt{int}) \\
\Rightarrow\ & E + (E + \mathbf{T} + \texttt{int}) \\
\Rightarrow\ & E + (\mathbf{E} + \texttt{int}) \\
\Rightarrow\ & E + (E + \mathbf{T}) \\
\Rightarrow\ & E + (\mathbf{E}) \\
\Rightarrow\ & E + \mathbf{T} \\
\Rightarrow\ & \mathbf{E}
\end{aligned}
$$

Predictive Parsing

- The leftmost DFS/E[illegible]hms are backtracking algorithms.
- Guess which produ[illegible]e, 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 there is 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

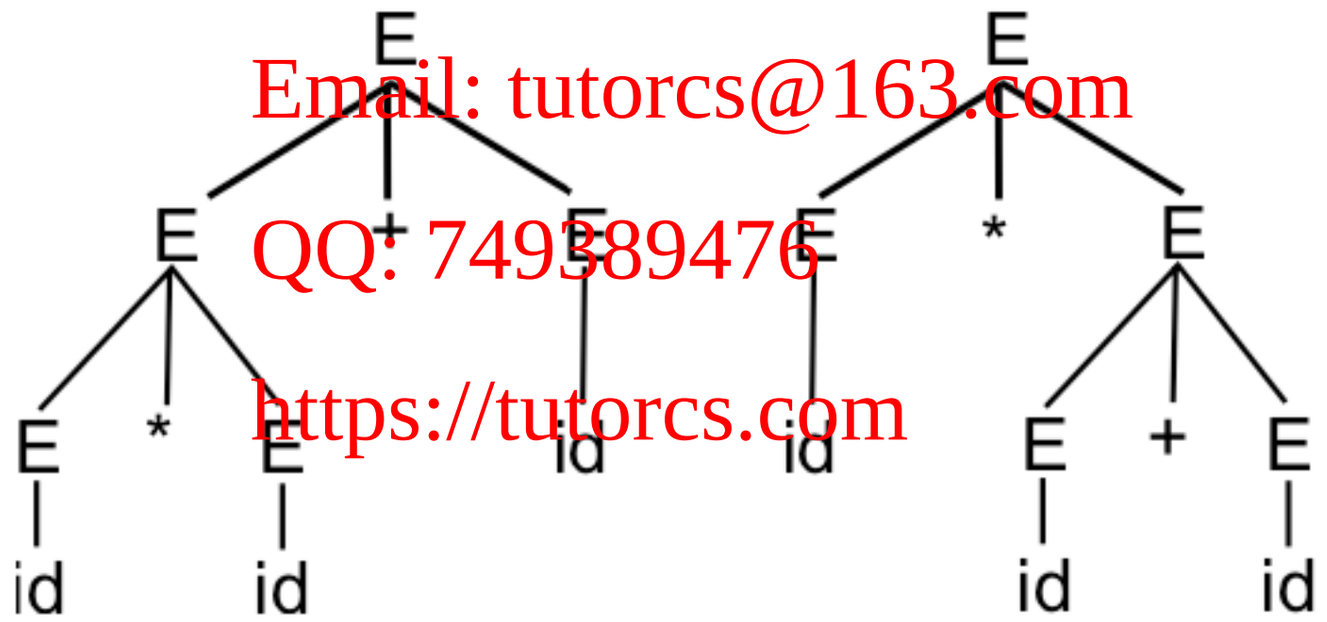
- Consider this gran

$EXPR \rightarrow E * E$

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

$| \; id$

- 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 right (or left)

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


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



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