Context-Free Grammars

- The notation for context-free grammars (CFGs) is also called Backus-Naur Form (BNF).
- A CFG consists of
  - A set of terminals: t
  - A set of non-terminals: N
  - A start symbol: S
  - A set of productions: X → Y
- CFGs are a natural notation for the recursive structure.
- CFG is a generator for a context-free language.

Context-Free Grammars

- Expression grammar with precedence and associativity

$$
\begin{aligned}
\textit{expr} &\longrightarrow \texttt{id} \mid \texttt{number} \mid - \textit{expr} \mid (\textit{expr}) \\
&\quad\mid \textit{expr}\ \textit{op}\ \textit{expr} \\
\textit{op} &\longrightarrow + \mid - \mid * \mid /
\end{aligned}
$$

id → letter (letter | digit)*

Derivation

$$
\begin{aligned}
\textit{expr} &\longrightarrow \texttt{id} \mid \texttt{number} \mid - \textit{expr} \mid (\textit{expr}) \\
&\quad\mid \textit{expr}\ \textit{op}\ \textit{expr} \\
\textit{op} &\longrightarrow + \mid - \mid * \mid /
\end{aligned}
$$

- In this grammar, generate the string
  "slope * x + intercept"

expr → expr op expr → id (slope) op expr

→ id (slope) * expr → id (slope) * expr op expr

→ id(slope) * id (x) + id (intercept)

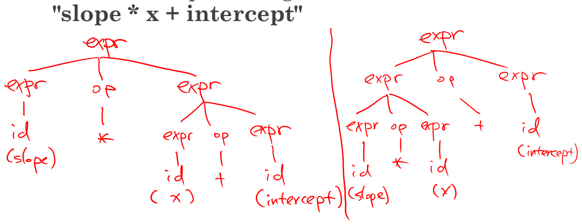
Derivation and Parse Trees

- A derivation is a sequence of productions
  - S → … → … → …
- A derivation can be drawn as a tree
  - Start symbol is the tree's root
  - For a production $X \rightarrow Y_1 \cdots Y_n$, $Y_1 \cdots Y_n$ are children of node X

Parse Tree

$$
\begin{aligned}
\textit{expr} &\longrightarrow \texttt{id} \mid \texttt{number} \mid - \textit{expr} \mid (\textit{expr}) \\
&\quad\mid \textit{expr}\ \textit{op}\ \textit{expr} \\
\textit{op} &\longrightarrow + \mid - \mid * \mid /
\end{aligned}
$$

- Parse tree for expression grammar for
  "slope * x + intercept"



Note on Derivations

- A parse tree has
  - Terminals at the leave.
  - Non-terminals at the internal-nodes.
- Left-most & Right-most derivations
  - Left-most: at each step, replace the left-most non-terminal
  - Right-most: at each step, replace the right-most non-terminal

Ambiguity

- A grammar is ambiguous if it has more than one tree for some string.
- Ambiguity is unacceptable.
  - How can we deal with ambiguity?
    - rewrite the grammar

Unambiguous Grammar

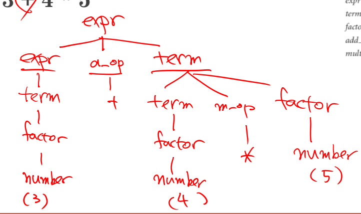
- A better version because it is
  - unambiguous
  - enforces precedence of * over +

$$
\begin{aligned}
expr &\longrightarrow term \mid expr\ add\_op\ term \\
term &\longrightarrow factor \mid term\ mult\_op\ factor \\
factor &\longrightarrow \texttt{id} \mid \texttt{number} \mid \texttt{-}\ factor \mid \texttt{(}\ expr\ \texttt{)} \\
add\_op &\longrightarrow \texttt{+} \mid \texttt{-} \\
mult\_op &\longrightarrow \texttt{*} \mid \texttt{/}
\end{aligned}
$$

Unambiguous Grammar

- Parse tree for expression grammar for

3 + 4 * 5



$$
\begin{aligned}
expr &\longrightarrow term \mid expr\ add\_op\ term \\
term &\longrightarrow factor \mid term\ mult\_op\ factor \\
factor &\longrightarrow \texttt{id} \mid \texttt{number} \mid \texttt{-}\ factor \mid \texttt{(}\ expr\ \texttt{)} \\
add\_op &\longrightarrow \texttt{+} \mid \texttt{-} \\
mult\_op &\longrightarrow \texttt{*} \mid \texttt{/}
\end{aligned}
$$

Ambiguity

- Is there general techniques for handling ambiguity?
  - No.
- Is this possible to convert automatically an ambiguous grammar to an unambiguous one?
  - No.

Recursive Descent Parsing

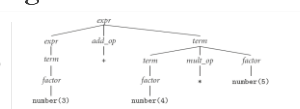
- Parse tree for expression grammar

```
function expr() {
   if (token == term) …
   else if (token == expr) …
   else error()
}
```

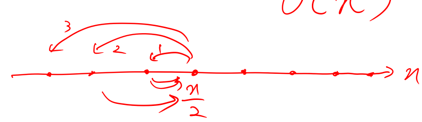
$$
\begin{aligned}
expr &\longrightarrow term \mid expr\ add\_op\ term \\
term &\longrightarrow factor \mid term\ mult\_op\ factor \\
factor &\longrightarrow \texttt{id} \mid \texttt{number} \mid \texttt{-}\ factor \mid \texttt{(}\ expr\ \texttt{)} \\
add\_op &\longrightarrow \texttt{+} \mid \texttt{-} \\
mult\_op &\longrightarrow \texttt{*} \mid \texttt{/}
\end{aligned}
$$

Complexity of Parsing

$$
\begin{aligned}
expr &\longrightarrow term \mid expr\ add\_op\ term \\
term &\longrightarrow factor \mid term\ mult\_op\ factor \\
factor &\longrightarrow \texttt{id} \mid \texttt{number} \mid \texttt{-}\ factor \mid \texttt{(}\ expr\ \texttt{)} \\
add\_op &\longrightarrow \texttt{+} \mid \texttt{-} \\
mult\_op &\longrightarrow \texttt{*} \mid \texttt{/}
\end{aligned}
$$



$O(n^3)$



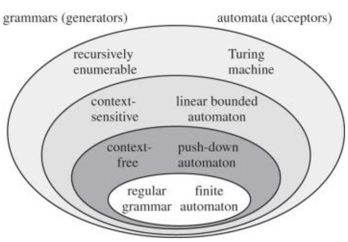
Predictive Parsers

- Recursive descent parsers are inefficient because of backtracking
- Predictive parsers can predict which production to use, how?
  - By looking at the next few tokens

Predictive Parsers

- Predictive parsers accept LL(k) or LR(k) grammars
  - The first L means "Left – right"
  - The second L(R) means "Left(Right) derivation"
  - k means "predict based on k– tokens of lookahead"

Chomsky hierarchy



Natural Language Processing