CSE443
Compilers

Dr. Carl Alphonce
alphonce@buffalo.edu
343 Davis Hall
Phases of a compiler

Figure 1.6, page 5 of text
Sample grammars


http://blackbox.userweb.mwn.de/Pascal-EBNF.html

https://cs.wmich.edu/~gupta/teaching/cs4850/sumII06/The%20syntax%20of%20C%20in%20Backus-Naur%20form.htm
Ambiguity
A grammar is ambiguous if and only if it generates a sentential form that has two or more distinct parse trees.

Operator precedence and operator associativity are two examples of ways in which a grammar can provide unambiguous interpretation.
Operator precedence ambiguity

The following grammar is ambiguous:

\[ \text{<expr>} \rightarrow \text{<expr>} \text{<op>} \text{<expr>} \mid \text{const} \]
\[ \text{<op>} \rightarrow - \mid / \]

The grammar treats the two operators, '-' and '/', equivalently.
An ambiguous grammar for arithmetic expressions

\[ \text{<expr> } \rightarrow \text{ <expr> <op> <expr> } \mid \text{ const} \]

\[ \text{<op> } \rightarrow \text{ / } \mid \text{ -} \]

```
const - const / const
```

```
const - const / const
```
Disambiguating the grammar

This grammar (fragment) is unambiguous:

```
<expr>  ->  <expr> - <term> | <term>
<term>  ->  <term> / const | const
```

The grammar treats the two operators, '-' and '/', differently.

In this grammar, '/' has higher precedence than '-'. Within a given subtree, deeper nodes are evaluated before shallower notes.
Disambiguating the grammar

- If we use the parse tree to indicate precedence levels of the operators, we can remove the ambiguity.
- The following rules give `/` a higher precedence than `-`

\[
\begin{align*}
<expr> & \rightarrow <expr> - <term> \mid <term> \\
<term> & \rightarrow <term> / \text{const} \mid \text{const}
\end{align*}
\]
Derivation of $2 + 5 \times 3$

using C grammar
Recursion and parentheses

- To generate $2+3*4$ or $3*4+2$, the parse tree is built so that $+$ is higher in the tree than $\times$.
- To force an addition to be done prior to a multiplication we must use parentheses, as in $(2+3)*4$.
- Grammar captures this in the recursive case of an expression, as in the following grammar fragment:

\[
\begin{align*}
\langle expr \rangle & \rightarrow \langle expr \rangle + \langle term \rangle \mid \langle term \rangle \\
\langle term \rangle & \rightarrow \langle term \rangle \ast \langle factor \rangle \mid \langle factor \rangle \\
\langle factor \rangle & \rightarrow \langle variable \rangle \mid \langle constant \rangle \mid "(\)\langle expr\)""
\end{align*}
\]
Given a regular language \( L \) we can always construct a context free grammar \( G \) such that \( L = \mathcal{L}(G) \).

For every regular language \( L \) there is an NFA \( M = (S, \Sigma, \delta, F, s_0) \) such that \( L = \mathcal{L}(M) \).

Build \( G = (N, T, P, S_0) \) as follows:

- \( N = \{ N_s \mid s \in S \} \)
- \( T = \{ t \mid t \in \Sigma \} \)
- If \( \delta(i, a) = j \), then add \( N_i \to a N_j \) to \( P \)
- If \( i \in F \), then add \( N_i \to \varepsilon \) to \( P \)
- \( S_0 = N_{s_0} \)
\((a|b)^*abb\)

\[ G = ( \{ A_0, A_1, A_2, A_3 \}, \{ a, b \}, \{ A_0 \rightarrow a A_0, A_0 \rightarrow b A_0, A_0 \rightarrow a A_1, A_1 \rightarrow b A_2, A_2 \rightarrow b A_3, A_3 \rightarrow \epsilon \}, A_0 ) \]
Show that not all CF languages are regular.

To do this we only need to demonstrate that there exists a CFL that is not regular.

Consider $L = \{ a^n b^n \mid n \geq 1 \}$

Claim: $L \in \text{CFL}, \ L \notin \text{RL}$
Proof (sketch):

$L \in \text{CFL}: S \rightarrow aSb | ab$

$L \not\in \text{RL}$ (by contradiction):

Assume $L$ is regular. In this case there exists a DFA $D=(S, \Sigma, \delta, F, s_0)$ such that $L(D) = L$.

Let $k = |S|$. Consider $a^ib^i$, where $i > k$.

Suppose $\delta(s_0, a^i) = s_r$. Since $i > k$, not all of the states between $s_0$ and $s_r$ are distinct. Hence, there are $v$ and $w$, $0 \leq v < w \leq k$ such that $s_v = s_w$. In other words, there is a loop.

This DFA can certainly recognize $a^ib^i$ but it can also recognize $a^jb^i$, where $i \neq j$, by following the loop.

"REGULAR GRAMMARS CANNOT COUNT"
public class Foo {
    public static void main(String[] args) {
        for (int i=0; i<args.length; i++) {
            if (args[i].length() < 3) {
                ... }
            else {
                ... }
        }
    }
}
Context Free Grammars and parsing

- $O(n^3)$ algorithms to parse any CFG exist
- Programming language constructs can generally be parsed in $O(n)$
Top-down & bottom-up

- A top-down parser builds a parse tree from root to the leaves
  - easier to construct by hand
- A bottom-up parser builds a parse tree from leaves to root
  - Handles a larger class of grammars
  - tools (yacc/bison) build bottom-up parsers
Our presentation
First top-down, then bottom-up

- Present top-down parsing first.
- Introduce necessary vocabulary and data structures.
- Move on to bottom-up parsing second.
**vocab: look-ahead**

- The current symbol being scanned in the input is called the **lookahead symbol**.
Top-down parsing

- Start from grammar’s start symbol
- Build parse tree so its yield matches input
- Predictive parsing: a simple form of recursive descent parsing
Basic idea: try to build a derivation 
\[ S \Rightarrow^* \text{ input} \]

\[ S \Rightarrow^* \alpha \]

\[ \ldots?\ldots \]

\[ \Rightarrow^* \text{ input} \]
FIRST(\(\alpha\))

- If \(\alpha \in \text{(NUT)}^*\) then FIRST(\(\alpha\)) is "the set of terminals that appear as the first symbols of one or more strings of terminals generated from \(\alpha\)." [p. 64]

- Ex: If \(A \rightarrow a \beta\) then FIRST(A) = \{a\}

- Ex. If \(A \rightarrow a \beta | B\) then FIRST(A) = \{a\} \cup \text{FIRST(B)}
First sets are considered when there are two (or more) productions to expand $A \in N: A \rightarrow \alpha | \beta$

Predictive parsing requires that $\text{FIRST}(\alpha) \cap \text{FIRST}(\beta) = \emptyset$
If lookahead symbol does not match first set, use $\epsilon$ production not to advance lookahead symbol but instead "discard" non-terminal:

- $\text{optexpt} \rightarrow \text{expr} \mid \epsilon$

"While parsing optexpr, if the lookahead symbol is not in $\text{FIRST}(\text{expr})$, then the $\epsilon$ production is used" [p. 66]
Left recursion

- Grammars with left recursion are problematic for top-down parsers, as they lead to infinite regress.
Left recursion example

Grammar:

expr -> expr + term | term
term -> id

FIRST sets for rule alternatives are not disjoint:

FIRST(expr) = id
FIRST(term) = id
Grammar:

\[ A \rightarrow \alpha \beta \]

\[ \text{expr} \rightarrow \text{expr} + \text{term} \mid \text{term} \]

\[ \text{term} \rightarrow \text{id} \]

FIRST sets for rule alternatives are not disjoint:

- \( \text{FIRST}(\text{expr}) = \text{id} \)
- \( \text{FIRST}(\text{term}) = \text{id} \)
Rewriting grammar to remove left recursion

- Expr rule is of form $A \rightarrow A \alpha | \beta$
- Rewrite as two rules
  - $A \rightarrow \beta R$
  - $R \rightarrow \alpha R | \epsilon$
Grammar is re-written as:

- expr → term R
- R → + term R | ε