CSE443
Compilers

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

Figure 1.6, page 5 of text
Review
Programming Language Grammar Fragment

<program> -> <stmt-list>
<stmt-list> -> <stmt> | <stmt> ; <stmt-list>
<stmt> -> <var> = <expr>
<var> -> a | b | c | d
<expr> -> <term> + <term> | <term> - <term>
<term> -> <var> | const

Notes:
<var> is defined in the grammar
const is not defined in the grammar
A leftmost derivation of

\[ a = b + \text{const} \]

\[
<\text{program}> \Rightarrow <\text{stmt-list}>
  \Rightarrow <\text{stmt}>
  \Rightarrow <\text{var}> = <\text{expr}>
  \Rightarrow a = <\text{expr}>
  \Rightarrow a = <\text{term} > + <\text{term}>
  \Rightarrow a = <\text{var} > + <\text{term}>
  \Rightarrow a = b + <\text{term}>
  \Rightarrow a = b + \text{const}
\]
Parse tree

\[
\begin{align*}
\text{<program> } & \\
\text{<stmt-list> } & \\
\text{<stmt> } & \\
\text{<var> } = & \text{<expr> } \\
\text{a} & \text{<term> + <term> } \\
\text{<var> } & \text{const } \\
\text{b} & 
\end{align*}
\]
Parse trees and compilation

- A compiler builds a parse tree for a program (or for different parts of a program).
- If the compiler cannot build a well-formed parse tree from a given input, it reports a compilation error.
- The parse tree serves as the basis for semantic interpretation/translation of the program.
BNF and EBNF
Extended BNF

• Optional parts are placed in brackets [ ]
  \[ \text{proc\_call} \rightarrow \text{ident} \ (\langle \text{expr\_list} \rangle) \]

• Alternative parts of RHSs are placed inside parentheses and separated via vertical bars
  \[ \text{term} \rightarrow \text{term} \ (+|\ -) \ \text{const} \]

• Repetitions (0 or more) are placed inside braces
  \[ \{} \]
  \[ \text{ident} \rightarrow \text{letter} \ \{ \text{letter} | \text{digit} \} \]
Comparison of BNF and EBNF

• sample grammar fragment expressed in BNF
  \[
  \text{<expr>} \rightarrow \text{<expr>} \ + \ \text{<term>}
  \ | \ \text{<expr>} \ - \ \text{<term>}
  \ | \ \text{<term>}
  \text{<term>} \rightarrow \text{<term>} \ * \ \text{<factor>}
  \ | \ \text{<term>} \ / \ \text{<factor>}
  \ | \ \text{<factor>}
  \]

• same grammar fragment expressed in EBNF
  \[
  \text{<expr>} \rightarrow \text{<term>} \ {\{ (+ \ | \ -) \ <term> \}}
  \text{<term>} \rightarrow \text{<factor>} \ {\{ (* \ | \ /) \ <factor> \}}
  \]
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:

<expr>  -->  <expr> <op> <expr>  |  const
<op>    -->  -  |  /

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

<expr> -> <expr> <op> <expr> | const
<op> -> / | -

```
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 `-`

\[
\begin{align*}
<expr> & \rightarrow <expr> \ - \ <term> \ \mid \ <term> \\
<term> & \rightarrow <term> \ / \ const \ \mid \ const
\end{align*}
\]
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 *.
- 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:

  \[
  \begin{align*}
  \text{<expr>} & \to \text{<expr>} + \text{<term>} | \text{<term>} \\
  \text{<term>} & \to \text{<term>} * \text{<factor>} | \text{<factor>} \\
  \text{<factor>} & \to \text{<variable>} | \text{<constant>} | "(" \text{<expr>} ")"
  \end{align*}
  \]
Sample grammars
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
Derivation of $2 + 5 * 3$

Using C grammar
Language Semantics

What's in language specification?

What's left up to the language implementor?
6.2.2 Evaluation Order

The order of evaluation of subexpressions within an expression is undefined. In particular, you cannot assume that the expression is evaluated left to right. For example:

```
int x = f(2) + g(3); // undefined whether f() or g() is called first
```

C++ Programming Language, 3rd edition.
Bjarne Stroustrup. (c) 1997. Page 122.
A compiler translates high level language statements into a much larger number of low-level statements, and then applies optimizations. The entire translation process, including optimizations, must preserve the semantics of the original high-level program.

By not specifying the order in which subexpressions are evaluated (left-to-right or right-to-left) a C++ compiler can potentially re-order the resulting low-level instructions to give a “better” result.
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 $P$
- $S_0 = N_{s_0}$
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 }
RL ⊊ CFL

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):

\[ L \in \text{CFL}: S \rightarrow aSb \mid ab \]

\[ L \notin \text{RL} \text{ (by contradiction):} \]

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 \( a^i b^i \), 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 \( \omega \), \( 0 \leq v < \omega \leq k \) such that \( s_v = s_\omega \). In other words, there is a loop.

This DFA can certainly recognize \( a^i b^i \) but it can also recognize \( a^j b^i \), 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.
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
\( S \Rightarrow \star \) input

\[ S \Rightarrow \star \alpha \]

...??...

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

- If \(\alpha \in (NUT)^*\) 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 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 $\text{FIRST}(\alpha) \cap \text{FIRST}(\beta) = \emptyset$