CSE443
Compilers

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

Figure 1.6, page 5 of text
Recap

Lexical analysis:
LEX/FLEX (regex \(\rightarrow\) lexer)

Syntactic analysis:
YACC/BISON (grammar \(\rightarrow\) parser)
Continuing from Friday

With precedence rule forcing an expression like 2+3*4 to be interpreted as 2+(3*4), how can we modify the grammar to allow (2+3)*4 as a valid expression?

<expr> -> <expr> + <term> | <term>
<term> -> <term> * <factor> | <factor>
(factor) -> <variable> | <constant> | '(' <expr> ')'
Lecture discussion

There are many reasons to study the syntax of programming languages.

When learning a new language you need to be able to read a syntax description to be able to write well-formed programs in the language.

Understanding at least a little of what a compiler does in translating a program from high-level to low-level forms deepens your understanding of why programming languages are designed the way they are, and equips you to better diagnose subtle bugs in programs.

The next slide shows the “evaluation order” remark in the C++ language reference, which alludes to the order being left unspecified to allow a compiler to optimize the code during translation.
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:

```cpp
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.
Returning to an earlier question

A few lectures back the question was asked whether there are context free languages which are not regular.
The traditional Chomsky hierarchy

grammars (generators) and languages
automata (acceptors)

recursively enumerable language

context-sensitive language

context-free language

regular finite-state language

the traditional Chomsky hierarchy

turing machine

linear-bound automaton

push-down automaton

Syntactic structure

Lexical structure
**RL ⊆ CFL**

**proof sketch**

- 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 \( P \)
  - \( S_0 = N_{s_0} \)
\[(a|b)\star abb\]

\[G = ( \{ A_0, A_1, A_2, A_3 \}, \{ a, b \}, \{ A_0 \to a \ A_0, A_0 \to b \ A_0, A_0 \to a \ A_1, A_1 \to b \ A_2, A_2 \to b \ A_3, A_3 \to \varepsilon \}, A_0 ) \]
RL $\subseteq$ CFL

proof sketch

- 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: $L \in \text{CFL}, L \notin \text{RL}$
RL ⊆ CFL

proof sketch

$L \in$ CFL: $S \rightarrow aSb \mid ab$

$L \notin$ 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 $a^j b^i$, 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.
vocab: look-ahead

The current symbol being scanned in the input is called the lookahead symbol.
Top-down parsing
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
If \( \alpha \in (NUT)^* \) then \( \text{FIRST}(\alpha) \) 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 \( A \to a \beta \) then \( \text{FIRST}(A) = \{a\} \)

Ex: If \( A \to a \beta \mid B \) then \( \text{FIRST}(A) = \{a\} \cup \text{FIRST}(B) \)
FIRST(\(\alpha\))

- First sets are considered when there are two (or more) productions to expand \(A \in N: A \rightarrow \alpha \mid \beta\)

- Predictive parsing requires that \(\text{FIRST}(\alpha) \cap \text{FIRST}(\beta) = \emptyset\)
$\varepsilon$ productions

- If lookahead symbol does not match first set, use $\varepsilon$ production not to advance lookahead symbol but instead "discard" non-terminal:

\[
\text{optexpt} \rightarrow \text{expr} \mid \varepsilon
\]

- "While parsing optexpr, if the lookahead symbol is not in FIRST(expr), then the $\varepsilon$ 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`
Left recursion example

- Grammar:
  - expr -> expr + term | term
  - term -> id

- FIRST sets for rule alternatives are not disjoint:
  - FIRST(expr) = id
  - FIRST(term) = id
Rewriting grammar to remove left recursion

- expr rule is of form $A \rightarrow A\ \alpha \mid \beta$
- Rewrite as two rules
  - $A \rightarrow \beta \ R$
  - $R \rightarrow \alpha \ R \mid \epsilon$
Grammar is rewritten as

- $\text{expr} \rightarrow \text{term} \ R$
- $R \rightarrow + \text{term} \ R \ | \ \varepsilon$