Phases of a compiler

Figure 1.6, page 5 of text
Example

$L = \{ \ 0, 1, 00, 11, 000, 111, 0000, 1111, \ldots \ \}$

$G = ( \{ 0,1 \}, \{ S, ZeroList, OneList \},$
   $\{ S \to ZeroList \mid OneList,$
   $ZeroList \to 0 \mid 0 \ ZeroList,$
   $OneList \to 1 \mid 1 \ OneList \},$
   $S)$
Derivations from G

Derivation of 0 0 0 0
S $\rightarrow$ ZeroList
  $\rightarrow$ 0 ZeroList
  $\rightarrow$ 0 0 ZeroList
  $\rightarrow$ 0 0 0 ZeroList
  $\rightarrow$ 0 0 0 0

Derivation of 1 1 1
S $\rightarrow$ OneList
  $\rightarrow$ 1 OneList
  $\rightarrow$ 1 1 OneList
  $\rightarrow$ 1 1 1
Observations

- Every string of symbols in a derivation is a sentential form.
- A sentence is a sentential form that has only terminal symbols.
- A leftmost derivation is one in which the leftmost nonterminal in each sentential form is the one that is expanded.
- A derivation can be leftmost, rightmost, or neither.
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

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

\[
\begin{align*}
\langle \text{program} \rangle & \Rightarrow \langle \text{stmt-list} \rangle \\
& \Rightarrow \langle \text{stmt} \rangle \\
& \Rightarrow \langle \text{var} \rangle = \langle \text{expr} \rangle \\
& \Rightarrow a = \langle \text{expr} \rangle \\
& \Rightarrow a = \langle \text{term} \rangle + \langle \text{term} \rangle \\
& \Rightarrow a = \langle \text{var} \rangle + \langle \text{term} \rangle \\
& \Rightarrow a = b + \langle \text{term} \rangle \\
& \Rightarrow a = b + \text{const}
\end{align*}
\]
Parse tree

```plaintext
<program>
  |
<stmt-list>
  |
<stmt>
  |
<var> = <expr>
  |
  a <term> + <term>
    |
    <var> const
      |
      b
```
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.
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
  \[ \text{<term>} \rightarrow \text{<term>} (\text{+|=} \text{const}) \]

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

• sample grammar fragment expressed in BNF

  \[
  \begin{align*}
  \langle expr \rangle & \rightarrow \langle expr \rangle + \langle term \rangle \\
  & \quad \mid \langle expr \rangle - \langle term \rangle \\
  & \quad \mid \langle term \rangle \\
  \langle term \rangle & \rightarrow \langle term \rangle \star \langle factor \rangle \\
  & \quad \mid \langle term \rangle / \langle factor \rangle \\
  & \quad \mid \langle factor \rangle
  \end{align*}
  \]

• same grammar fragment expressed in EBNF

  \[
  \begin{align*}
  \langle expr \rangle & \rightarrow \langle term \rangle \{(+ | -) \langle term \rangle\} \\
  \langle term \rangle & \rightarrow \langle factor \rangle \{(* | /) \langle factor \rangle\}
  \end{align*}
  \]
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:

\[
\begin{align*}
\text{<expr>} & \rightarrow \text{<expr>} \text{<op>} \text{<expr>} \mid \text{const} \\
\text{<op>} & \rightarrow - \mid / 
\end{align*}
\]

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

\[
\begin{align*}
\langle expr \rangle & \rightarrow \langle expr \rangle \ \langle op \rangle \ \langle expr \rangle \ | \ \text{const} \\
\langle op \rangle & \rightarrow / \ | \ - \\
\end{align*}
\]
Disambiguating the grammar

This grammar (fragment) is unambiguous:

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

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

In this grammar, ‘/’ has higher precedence than ‘-’. 
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 -

\[
<\text{expr}> \rightarrow <\text{expr}> - <\text{term}> \mid <\text{term}>
\]
\[
<\text{term}> \rightarrow <\text{term}> / \text{const} \mid \text{const}
\]
Sample grammars


https://docs.oracle.com/javase/specs/

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

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 *.
• 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:

<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:

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

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.
Figure 1.6: Phases of a compiler
Given a regular language $L$ we can always construct a context free grammar $G$ such that $L = L(G)$.

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 $N_i \rightarrow a N_j$ to $P$
- If $i \in F$, then add $N_i \rightarrow \epsilon$ to $P$
- $S_0 = N_{s_0}$
$G = ( \{ A_0, A_1, A_2, A_3 \}, \{ a, b \}, \{ A_0 \rightarrow a \ A_0, A_0 \rightarrow b \ A_0, A_0 \rightarrow a \ A_1, A_1 \rightarrow b \ A_2, A_2 \rightarrow b \ A_3, A_3 \rightarrow \epsilon \} , A_0 \}$
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 $w$, $0 \leq v < w \leq k$ such that $s_v = s_w$. In other words, there is a loop.

This DFA can certainly recognize $a^i b^i$ 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 { ... }
    }
}