Programming Languages
2nd edition
Tucker and Noonan

Chapter 2
Syntax

A language that is simple to parse for the compiler is also simple to parse for the human programmer.

N. Wirth
Contents

2.1 Grammars
   2.1.1 Backus-Naur Form
   2.1.2 Derivations
   2.1.3 Parse Trees
   2.1.4 Associativity and Precedence
   2.1.5 Ambiguous Grammars

2.2 Extended BNF

2.3 Syntax of a Small Language: Clite
   2.3.1 Lexical Syntax
   2.3.2 Concrete Syntax

2.4 Compilers and Interpreters

2.5 Linking Syntax and Semantics
   2.5.1 Abstract Syntax
   2.5.2 Abstract Syntax Trees
   2.5.3 Abstract Syntax of Clite
2.1.4 Associativity and Precedence

A grammar can be used to define associativity and precedence among the operators in an expression.

*E.g., + and - are left-associative operators in mathematics;*  
* * and / have higher precedence than + and -.*

Consider the more interesting grammar $G_1$:

```
Expr -> Expr + Term | Expr – Term | Term  
Term -> Term * Factor | Term / Factor |  
   Term % Factor | Factor  
Factor -> Primary ** Factor | Primary  
Primary -> 0 | ... | 9 | ( Expr )
```
Parse of $4^{**}2^{**}3+5*6+7$ for Grammar $G_1$

Figure 2.3
### Associativity and Precedence for Grammar $G_1$

**Table 2.1**

<table>
<thead>
<tr>
<th>Precedence</th>
<th>Associativity</th>
<th>Operators</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>right</td>
<td>**</td>
</tr>
<tr>
<td>2</td>
<td>left</td>
<td>* / %</td>
</tr>
<tr>
<td>1</td>
<td>left</td>
<td>+ -</td>
</tr>
</tbody>
</table>

**Note:** These relationships are shown by the structure of the parse tree: highest precedence at the bottom, and left-associativity on the left at each level.
2.1.5 Ambiguous Grammars

A grammar is *ambiguous* if (even just) one of its strings has two or more different parse trees.

*E.g.*, Grammar $G_1$ above is unambiguous.

C, C++, and Java have a large number of

– *operators and*

– *precedence levels*

Instead of using a large grammar, we can:

– *Write a smaller ambiguous grammar, and*

– *Give separate precedence and associativity (e.g., Table 2.1)*
An Ambiguous Expression Grammar $G_2$

$Expr \rightarrow Expr \ Op \ Expr \mid (\ Expr\ ) \mid Integer$

$Op \rightarrow + \mid - \mid \ast \mid / \mid \% \mid \^{**}$

Notes:

– $G_2$ is equivalent to $G_1$, that is, its language is the same.
– $G_2$ has fewer productions and nonterminals than $G_1$.
– However, $G_2$ is ambiguous.
Ambiguous Parse of 5-4+3 Using Grammar $G_2$  
Figure 2.4
The Dangling Else

\[\text{IfStatement} \rightarrow \text{if ( Expression ) Statement} \mid \text{if ( Expression ) Statement else Statement}\]

\[\text{Statement} \rightarrow \text{Assignment} \mid \text{IfStatement} \mid \text{Block}\]

\[\text{Block} \rightarrow \{ \text{Statements} \}\]

\[\text{Statements} \rightarrow \text{Statements Statement} \mid \text{Statement}\]
Example

With which ‘if’ does the following ‘else’ associate? (assuming indentation doesn’t count)

```java
if (x < 0)
    if (y < 0) y = y - 1;
else y = 0;
```

Answer: either one!
The *Dangling Else* Ambiguity

Figure 2.5

```
if (Expression) Statement
  |        |     |
  |        |     | IfStatement
  |        |     |   x<0
  |        |     |   IfStatement
  |        |     |     | y<0
  |        |     |     | y = y-1;
  |        |     | else
  |        |     |     | y = 0;
if (Expression) Statement else Statement
  |        |     | IfStatement
  |        |     |   x<0
  |        |     |   IfStatement
  |        |     |     | y<0
  |        |     |     | y = y-1;
```
Solving the dangling else ambiguity

1. Algol 60, C, C++: associate each else with closest if; use {} or begin...end to override.
2. Algol 68, Modula, Ada: use explicit delimiter to end every conditional (e.g., if...fi)
3. Java: rewrite the grammar to limit what can appear in a conditional:

   \[
   \text{IfThenStatement} \rightarrow \text{if ( Expression ) Statement} \\
   \text{IfThenElseStatement} \rightarrow \text{if ( Expression ) StatementNoShortIf} \\
   \text{else Statement} \\
   \]

   The category \text{StatementNoShortIf} includes all except \text{IfThenStatement}.
2.2 Extended BNF (EBNF)

BNF:

- *recursion for iteration*
- *nonterminals for grouping*

EBNF: additional metacharacters

- `{ }` for a series of zero or more
- `( )` for a list, must pick one
- `[ ]` for an optional list; pick none or one
EBNF Examples

*Expression* is a list of one or more *Terms* separated by operators + and -

\[
Expression \rightarrow Term \{ ( + | - ) Term \}
\]

*IfStatement* -> if (Expression) Statement [else Statement]

*C-style EBNF* lists alternatives vertically and uses \(\text{opt}\) to signify optional parts. E.g.,

*IfStatement*:

\[
\text{if (Expression) Statement ElsePart}_{\text{opt}}
\]

*ElsePart*:

\[
\text{else Statement}
\]
EBNF to BNF

We can always rewrite an EBNF grammar as a BNF grammar. E.g.,

\[ A \rightarrow x \{ y \} z \]

can be rewritten:

\[ A \rightarrow x \ A' \ z \]

\[ A' \rightarrow | \ y \ A' \]

(Rewriting EBNF rules with ( ), [ ] is left as an exercise.)

While EBNF is no more powerful than BNF, its rules are often simpler and clearer.
Syntax Diagram for *Expressions with Addition*

Figure 2.6