### 4.2.3 Grammar transformations

EBNF is a much more flexible notation than BNF. In particular, grouping of alternatives '(...|...|...)' and iteration '*' make it easy to perform useful transformations on a grammar expressed in EBNF. Here we introduce and illustrate some possible transformations. Later, in Section 4.3.4, we shall see how they are used in practice.

#### Left factorization

Suppose that we have alternatives of the form:

$X\ Y \mid X\ Z$

where $X$, $Y$, and $Z$ are arbitrary (extended) REs. We can replace these alternatives by the equivalent extended RE:

$X\ (Y \mid Z)$

The REs $X\ Y \mid X\ Z$ and $X\ (Y \mid Z)$ are equivalent in the sense that they generate exactly the same languages. This fact was illustrated by the first two REs in Example 4.3.

*Example 4.5 Left factorization*

Many programming languages have alternative forms of if-command:

| | | |
|---|---|---|
| single-Command | ::= | V-name **:=** Expression |
| | \| | **if** Expression **then** single-Command |
| | \| | **if** Expression **then** single-Command **else** single-Command |

This production rule can be left-factorized as follows:

| | | |
|---|---|---|
| single-Command | ::= | V-name **:=** Expression |
| | \| | **if** Expression **then** single-Command (ε \| **else** single-Command) |

□

Right factorization is the mirror-image of left factorization, but is less useful in practice.

#### Elimination of left recursion

Suppose that we have a production rule of the form:

$N ::= X \mid N\ Y$

where $N$ is a nonterminal symbol, and $X$ and $Y$ are arbitrary extended REs. This production rule is ***left-recursive***. We can replace it by the equivalent EBNF production rule:

$N ::= X\ (Y)^*$



These production rules are equivalent in the sense that they generate exactly the same languages. The production rule $N ::= X \mid N\ Y$ states that an $N$-phrase may consist either of an $X$-phrase or of an $N$-phrase followed by a $Y$-phrase. This is just a roundabout way of stating that an $N$-phrase consists of an $X$-phrase followed by any number of $Y$-phrases. The production rule $N ::= X\ (Y)^*$ states the same thing more concisely.

*Example 4.6 Elimination of left recursion*

The syntax of Triangle identifiers is expressed in BNF as follows:

| | | |
|---|---|---|
| Identifier | ::= | Letter |
| | \| | Identifier Letter |
| | \| | Identifier Digit |

This production rule is a little more complicated than the form shown above, but we can left-factorize it:

| | | |
|---|---|---|
| Identifier | ::= | Letter |
| | \| | Identifier (Letter \| Digit) |

and now eliminate the left recursion:

| | | |
|---|---|---|
| Identifier | ::= | Letter (Letter \| Digit)* |

□

As illustrated by Example 4.6, it is possible for a more complicated production rule to be left-recursive:

$N ::= X_1 \mid \ldots \mid X_m \mid N\ Y_1 \mid \ldots \mid N\ Y_n$

However, left factorization gives us:

$N ::= (X_1 \mid \ldots \mid X_m) \mid N\ (Y_1 \mid \ldots \mid Y_n)$

and now we can apply our elimination rule:

$N ::= (X_1 \mid \ldots \mid X_m)\ (Y_1 \mid \ldots \mid Y_n)^*$

#### Substitution of nonterminal symbols

Given an EBNF production rule $N ::= X$, we may substitute $X$ for any occurrence of $N$ on the right-hand side of another production rule.

If we substitute $X$ for *every* occurrence of $N$, then we may eliminate the nonterminal $N$ and the production rule $N ::= X$ altogether. (This is possible, however, only if $N ::= X$ is nonrecursive and is the only production rule for $N$.)

Whether we actually choose to make such substitutions is a matter of convenience. If $N$ occurs in only a few places, and if $X$ is uncomplicated, then elimination of $N ::= X$ might well simplify the grammar as a whole.

*Example 4.7 Substitution*

Consider the following production rules, taken from a BNF grammar of Pascal:

```
single-Command   ::=  for Control-Variable := Expression To-or-Downto
                         Expression do single-Command
                   |  ...
Control-Variable ::=  Identifier
To-or-Downto     ::=  to
                   |  downto
```

It makes sense to eliminate Control-Variable and To-or-Downto by substitution:

```
single-Command   ::=  for Identifier := Expression (to | downto)
                         Expression do single-Command
                   |  ...
```

The nonterminal To-or-Downto was present in the first place only because grouping of alternatives '(...|...)' is not possible in BNF. The nonterminal Control-Variable was present only to act as a 'semantic clue' – to emphasize the role this particular identifier plays in the for-command – and not for any grammatical reason. Eliminating such nonterminals simplifies the grammar.

□

### 4.2.4 Starter sets

The ***starter set*** of an RE $X$, written *starters*⟦$X$⟧, is the set of terminal symbols that can start a string generated by $X$. For example:

*starters*⟦**h i s** | **h e r** | **i t s**⟧ = {**h**, **i**}
*starters*⟦(**r e**)* **s e t**⟧ = {**r**, **s**}

since '(**r e**)* **s e t**' generates the set of strings {**set**, **reset**, **rereset**, ...}.

The following is a precise and complete definition of *starters:*

| | | |
|---|---|---|
| *starters*⟦ε⟧ | = { } | |
| *starters*⟦$t$⟧ | = {$t$} | where $t$ is a terminal symbol |
| *starters*⟦$X$ $Y$⟧ | = *starters*⟦$X$⟧ ∪ *starters*⟦$Y$⟧ | if $X$ generates ε |
| *starters*⟦$X$ $Y$⟧ | = *starters*⟦$X$⟧ | if $X$ does not generate ε |
| *starters*⟦$X$ \| $Y$⟧ | = *starters*⟦$X$⟧ ∪ *starters*⟦$Y$⟧ | |
| *starters*⟦$X$*⟧ | = *starters*⟦$X$⟧ | |

(where $X$ and $Y$ stand for arbitrary REs).



We can easily generalize this to define the starter set of an extended RE. There is only one case to add:

| | | |
|---|---|---|
| *starters*⟦$N$⟧ | = *starters*⟦$X$⟧ | where $N$ is a nonterminal symbol defined by production rule $N$ ::= $X$ |

In Example 4.4:

*starters*⟦Expression⟧ = *starters*⟦primary-Expression (Operator primary-Expression)*⟧
= *starters*⟦primary-Expression⟧
= *starters*⟦Identifier⟧ ∪ *starters*⟦ **(** Expression **)** ⟧
= *starters*⟦**a** | **b** | **c** | **d** | **e**⟧ ∪ { **(** }
= {**a**, **b**, **c**, **d**, **e**, **(** }

## 4.3 Parsing

In this section we are concerned with analyzing sentences in some grammar. Given an input string of terminal symbols, our task is to determine whether the input string is a sentence of the grammar, and if so to discover its phrase structure. The following definitions capture the essence of this.

With respect to a particular context-free grammar $G$:

- ***Recognition*** of an input string is deciding whether or not the input string is a sentence of $G$.
- ***Parsing*** of an input string is recognition of the input string plus determination of its phrase structure. The phrase structure can be represented by a syntax tree, or otherwise.

We assume that $G$ is ***unambiguous***, i.e., that every sentence of $G$ has exactly one syntax tree. The possibility of an input string having several syntax trees is a complication we prefer to avoid.

Parsing is a task that humans perform extremely well. As we read a document, or listen to a speaker, we are continuously parsing the sentences to determine their phrase structure (and then determine their meaning). Parsing is subconscious most of the time, but occasionally it surfaces in our consciousness: when we notice a grammatical error, or realize that a sentence is ambiguous. Young children can be taught consciously to parse simple sentences on paper.

In this section we are interested in *parsing algorithms*, which we can use in syntactic analysis. Many parsing algorithms have been developed, but there are only two basic parsing strategies: *bottom-up parsing* and *top-down parsing*. These strategies are characterized by the order in which the input string's syntax tree is reconstructed. (In