2.41 *LL(1) conditions for top-down graphs*

An alternative chain is LL(1) if an arbitrary input symbol matches at most one of its nodes. A top-down graph is LL(1) if all of its alternative chains are LL(1).

The top-down graph of Fig. 2.9 is therefore LL(1) if *T* cannot start with + or - and if *E* cannot be followed by + or - (these symbols would match the ε - node).

Since each EBNF production corresponds to a top-down graph, the LL(1) conditions for top-down graphs are also the LL(1) conditions for EBNF grammars. In order to check if an EBNF grammar is LL(1), it is easiest to generate its top-down graph and check if it meets the LL(1) conditions. The LL(1) conditions for EBNF grammars can also be derived from the definition of the EBNF grammar alone without constructing a top-down graph. However, this is cumbersome and results in no new insights. We therefore omit the description and leave the task to the interested reader.

LL(1) Top-down graphs and grammars of programming languages

If top-down graphs are to have practical value, one must be able to represent the grammars of programming languages as LL(1) top-down graphs, and therefore as LL(1) EBNF grammars. We may ask, therefore, if they do this without exception, or if there are constructs that resist an LL(1) representation, and if so, what can be done about it. First of all, LL(1) violations by left-recursive productions and by the start of several alternatives with the same symbol can easily be avoided in top- down graphs and in EBNF notation. Remaining LL(1) violations can usually be removed with various tricks that are determined with insight into the particular situation. As an aid for the 'grammar designer', we will treat several typical cases and distinguish between the following five methods:

1. substitution and factorization;
2. alphabet extension;
3. syntactic extension;
4. acceptance of non-LL(1) constructs;
5. misceflaneous transformations.

Substitution and factorization. Consider a production with two alternatives that start with different nonterminals *X* and *Y*, where *X* and *Y* can start with the same symbol (terminal or nonterminal). Then it is often possible to

replace the symbols *X* and *Y* by the right-hand side of their productions, and to extract their common starting string by left-factorization.

This can be simple and obvious as in the various **DO** instructions PLM/80 (and similarly in PL/1):

```
statement =
    ...
    | dostatement
    | whilestatement
    | forstaternent
    | casestatement
    | ...

    dostatement        = " DO" ";" block.
    whilestatement     = "DO" "WHILE" expr ";" [statement} ending.
    forstatement       = "DO" ident "=" expr "TO" expr ["BY" expr] ";"
                         {statement} ending.
    casestatement      = "DO" "CASE" expr ";" {statement} ending.
```

By substitution and factorization this results in

```
    statement =
    ...
    | "DO"
        (";" block
            | ("CASE" expr ";"
            | "WHILE" expr ";"
            | ident "=" expr "TO" expr ["BY" expr] ";"
            ) {statement} ending
        )
    | ...
```

However, it can also be difficult. In grammars such as Modula-2 a *factor* can be a *set* or a *designator*, and both can begin with an identifier:

```
    factor     = ... | designator [actpars] | set | ...
    designator = qualident {"." ident | "[" exprlist "]" | "↑" }.
    set        = [qualident] "{" [elementlist] "}".
    qualident  = ident {"." ident}.
```

Note that even the production for *designator* taken alone is not LL(1). For instance, *ident.ident* may be simply a *qualident* or a *qualident* followed by *ident*

The removal of the LL(1) conflict consists of combining *designator* and *set* into a new symbol *deset*, and then splitting *designator* into *ident* and remainder *desigrest*. After several substitutions and factorizations, the following LL(1) constructs result:

```
    factor = ... | deset | ...

    deset =
        "{" [elementlist] "}"
```

```
| ident {"." ident}
    ( ( "^" | "[" exprlist "]" ) desigrest [actpars]
    | "{" [elementlist] "}"
    | [actpars]
    ).

desigrest  = {"." ident | "[" exprlist "]" | "↑" }.
```

The equivalence of the old and new constructs can no longer be easily seen.

Alphabet extension. In selecting an alternative, it is fairly common for two lookahead symbols to be necessary to find the right one. The main example of this is when labels appear in front of statements:

```
statement = [ident ":"] (ident ":=" expr | ifstatement | ...).
```

An *ident* at the beginning of a statement may be a label or the left part of an assignment. This can only be determined by the symbol following *ident*. This conflict can often be resolved by extending the terminal alphabet. In the preceding case, the word *label* can be added to the alphabet, and the lexical analyzer can be required to supply a *label* instead of an *ident* if *ident* is followed by a ':'. In this case, the lexical analyzer is used to resolve the LL(1) conflict.

This method leads to complications if the lexical analyzer is required to carry out a wider inspection of context to determine whether or not to substitute two terminals by another. For example, in Algol 60, '*ident :*' does not always mean the label of an instruction. An identifier may also appear in a declaration, as in *ARRAY(n : m)*. In such cases, the lexical analyzer is no longer independent of the syntax analyzer since it must consider the context.

Syntactic extension. In Algol 60 there exist multiple assignments, such as

```
assignment = designator ":=" {designator ":="} expr.
```

where *expr* can start with *designator*. This LL(1) violation is very nasty. It can be removed by 'substitution and factorization', but this is very cumbersome (the reader should try it). It is easier to 'expand' the designator inside the curly brackets to *expr*. This requires the introduction of an additional production for *assignrest*:

```
assignment = designator ":=" assignrest.
assignrest = expr [":=" assignrest]
```

The syntactic extension must be compensated by a semantic restriction. If in the production for *assignrest* the right-recursive part is present, *expr* must be restricted to be a *designator*. This can be achieved by the introduction of a boolean attribute *isdesignator*. Anticipating knowledge from Chapter 3, this

may be written as an attributed grammar as follows:

```
assignrest =
    expr ↑ isdesignator
        [":="                where (isdesignator) [1]
         assignrest].
```

This means: by syntactic extensions, portions of the language definition are moved from syntax to static semantics.

Acceptance of non-LL(1) constructs. If it is known that the parser tries to match the alternatives in the order they are written, some LL(1) violations can be left alone. The best known case is the dangling else:

```
ifstatement = "IF" expr "THEN" statement ["ELSE" statement].
```

Although this construct is not LL(1), and is even ambiguous (see Example 2.22), it can be left alone if one can be sure that the paeser, having recognized the statement following *THEN*, first tries to detect the optional *ELSE*, and only regards the entire *if* statement as complete if there is no *ELSE*.

Other transformations. Sometimes, a grammar that is not LL(1) can be transformed into an equivalent LL(1) grammar by simple transformations that do not fall into any of the four categories above. For example, in Algol 60, a block is defined as

```
block = head ";" body.
head  = "begin" declaration {";" declaration}.
```

This construct is not LL(1) since the semicolon is used in a dual role. It separates adjacent declarations and it separates body from head. The solution is simple: The grammar can be transformed so that the semicolon becomes a terminator instead of a separator:

```
block = head body
head  = "begin" declaration ";" {declaration ";"}
```

The necessity of such transformations, their difficulty, and the uncertainty of executing these transformations correctly is a weakness of the LL-method, and often a cause for criticism. In bottom-up analyzable LR(1) gramnar, no transformations, or only a few, are needed, so research has been focused on the LR-method. However, syntax is but one aspect. What is gained with the LR-method must be paid back by the connection of semantics to syntax: it is much more inflexible in the LR-method than in the LL-method, often leads to violations of the LR-property, and then also requires transformations. In addition, the LL(1)-method is much easier to understand than the LR-method. This results in easier transformations and more understandable error messages.

---

[1] Note the `where (isdesignator)` is called a *Context Condition*; in Coco/R a context condition is represented as a semantic action, exampe:

```
(. if not isdesignator then SemError(100, ''); .)
```