Chapter 6
Foundations of Textual Concrete Syntax

Avram Noam Chomsky.[1]

Abstract In this chapter, we consider the notion of *concrete syntax* of software languages thereby complementing the earlier discussion of *abstract syntax* (Chapters 3and 4). Concrete syntax is tailored towards processing (reading, writing, editing) by humans who are language users, while abstract syntax is tailored towards processing by programs that are authored by language implementers. In this chapter, we focus on the concrete syntax of *string languages* as defined by *context-free grammars* (CFGs). In fact, we cover only textual concrete syntax; we do not cover visual concrete syntax. We introduce the algorithmic notion of *acceptance* for a membership test for a language. We also introduce the algorithmic notion of parsing for recovering the grammar-based structure of input. We defer the implementation aspects of concrete syntax, including actual parsing approaches, to the next chapter.

[1] There is clearly nothing wrong with the notion of a Turing machine – after all it is Turing-complete, but the way it is described and discussed is clearly very reminiscent of how we think of actual (early) computing machines working operationally, if not mechanically. Personally, I have always felt more attracted to the lambda calculus, with its high level of abstraction, much more focused on computation than on operation. Likewise, I admire the Chomsky hierarchy [4], as it defines grammars in a fundamental manner, including a semantics that makes no operational concessions. There is a need for well-engineered grammar forms, such as parsing expression grammars [5], but all such work stands on the shoulders of Chomsky.

© Springer International Publishing AG, part of Springer Nature 2018
R. Lämmel, *Software Languages*,
https://doi.org/10.1007/978-3-319-90800-7_6

6.1 Textual Concrete Syntax

A *grammar* is a collection of rules defining the syntax of a language's syntactic categories such as statements and expressions. We introduce a basic grammar notation and a convenient extension here. We also show that a grammar can be understood in a "generative" sense, i.e., a grammar derives ("generates") language elements as strings.

6.1.1 A Basic Grammar Notation

Let us study the concrete syntax of *BNL* (*B*inary *N*umber *L*anguage). This is the language of unsigned binary numbers, possibly with decimal places, for example, "10" (2 as a decimal number) and "101.01" (5.25 as a decimal number). Let us define the concrete syntax of BNL. To this end, we use "fabricated" grammar notation: *BGL* (*B*asic *G*rammar *L*anguage).

Illustration 6.1 (Concrete syntax of BNL)

BGL resource languages/BNL/cs.bgl

```
[number] number : bits rest ; // A binary number
[single] bits : bit ; // A single bit
[many] bits : bit bits ; // More than one bit
[zero] bit : '0' ; // The zero bit
[one] bit : '1' ; // The nonzero bit
[integer] rest : ; // An integer number
[rational] rest : '.' bits ; // A rational number
```

BGL is really just a notational variation on the classic *Backus-Naur form* (BNF) [1]. A grammar is a collection of rules (say, productions). Each rule consists of a label such as *[number]* for better reference, a left-hand side which is a grammar symbol such as *number* in the first rule, and a right-hand side which is a sequence of grammar symbols. There are two kinds of grammar symbols:

Terminals These are quoted symbols such as "0" and "1"; they must not appear on the left-hand side of context-free rules. The terminals constitute the "alphabet" from which to build strings.

Nonterminals These are alphabetic symbols such as *number*, *bits*, and *rest*; they may appear on both the left- and the right-hand side of rules. In fact, the left-hand side of a context-free rule is a single nonterminal. Nonterminals correspond to syntactic categories.

6.1.2 Derivation of Strings

The intended meaning of a grammar is that rules can be applied from left to right to derive (say, "generate") strings composed of terminals such that nonterminals are replaced by right-hand sides of rules and terminals remain. We often assume that a grammar identifies a distinguished nonterminal – the *start symbol* – from which to start derivation. We may also just assume that the left-hand side of the first production is simply the start symbol. Derivation is illustrated below for a binary number.

Illustration 6.2 (Derivation of a string)
The following sequence of steps derives the terminal string "10" from the nonterminal number:

• *number*	*Apply rule [number]*
• *bits rest*	*Apply rule [integer] to rest*
• *bits*	*Apply rule [many] to bits*
• *bit bits*	*Apply rule [zero] to bit*
• *'1' bits*	*Apply rule [single] to bits*
• *'1' bit*	*Apply rule [zero] to bit*
• *'1' '0'*	

We assume that a "well-formed" grammar must permit derivation of terminal sequences for each of its nonterminals and that each nonterminal should be exercised by some of the derivations, starting from the start symbol. Such well-formedness is meant to rule out "nonsensical" grammars.

Exercise 6.1 (An alternative derivation) [Basic level]
There is actually more than one way to derive the terminal sequence in Illustration 6.2. Identify an alternative derivation.

Exercise 6.2 (Derivation of a string) [Basic level]
Present the derivation sequence for "101.01" in the style of Illustration 6.2.

Exercise 6.3 (BNL with signed numbers) [Basic level]
Extend the grammar in Illustration 6.1 to enable signed binary numbers.

6.1.3 An Extended Grammar Notation

Consider again the grammar in Illustration 6.1. Optionality of the fractional part is encoded by the rules *[integer]* and *[rational]*, subject to an "auxiliary" nonterminal *rest*. Sequences of bits are encoded by the rules *[single]* and *[many]*, subject to an "auxiliary" nonterminal *bits*. These are recurring idioms which can be expressed more concisely in the *extended Backus-Naur form* [7] (EBNF). We propose a related grammar notation here: *EGL* (*E*xtended *G*rammar *L*anguage). Let us illustrate EGL here with a concise syntax definition for BNL.

Illustration 6.3 (EBNF-like concrete syntax of BNL)

EGL resource languages/BNL/EGL/cs.egl

```
[number] number : { bit }+ { '.' { bit }+ }? ;
[zero] bit : '0' ;
[one] bit : '1' ;
```

Optionality of a phrase is expressed by the form "$\{\ldots\}?$". Repetition zero, one, or more times is expressed by the form "$\{\ldots\}*$". Repetition one or more times is expressed by the form "$\{\ldots\}+$". Rule labels are optional in EGL. In particular, we tend to leave out labels for nonterminals with only one alternative.

The extended notation (EGL) can be easily reduced ("desugared") to the basic notation (BGL) by modeling the EGL-specific phrases through additional rules, also subject to extra (fresh) nonterminals. There are these cases:

- Given one or more occurrences of a phrase $\{s_1 \ \cdots \ s_n\}?$ with grammar symbols $s_1, \ldots, s_n$ and a fresh nonterminal x, each occurrence is replaced by x and two rules are added:
  - x : ;
  - x : $s_1 \cdots s_n$;
- Given one or more occurrences of a phrase $\{s_1 \ \cdots \ s_n\}*$ and a fresh nonterminal x, each such occurrence is replaced by x and two rules are added:
  - x : ;
  - x : $s_1 \cdots s_n \, x$;
- Given one or more occurrences of a phrase $\{s_1 \ \cdots \ s_n\}+$ and a fresh nonterminal x, each such occurrence is replaced by x and two rules are added:
  - x : $s_1 \cdots s_n$;
  - x : $s_1 \cdots s_n \, x$;

Exercise 6.4 (Grammar notation translation) [Intermediate level]
The full EBNF notation [7] supports nested groups of alternatives. If such grouping was expressible in (an extended) EGL, then we could use grammar rules such as "s : a { b | c }? d ;" where the group of alternatives is "b | c". Reduce ("desugar") this group form to the basic notation.

6.1.4 Illustrative Examples of Grammars

We define the concrete syntax of a few more languages here. We revisit ("fabricated") languages for which we already defined the abstract syntax in Chapter 3.

6.1.4.1 Syntax of Simple Expressions

Let us define the concrete syntax of the expression language BTL.

Illustration 6.4 (Concrete syntax of BTL)

BGL resource languages/BTL/cs.bgl

```
[true] expr : "true" ;
[false] expr : "false" ;
[zero] expr : "zero" ;
[succ] expr : "succ" expr ;
[pred] expr : "pred" expr ;
[iszero] expr : "iszero" expr ;
[if] expr : "if" expr "then" expr "else" expr ;
```

That is, we assume "curried" notation (juxtaposition) for function application, i.e., for applying the operators *'pred'*, *'succ'*, and *'iszero'*. That is, we write succ zero instead of succ(zero). Curried notation is also used, for example, in the functional programming language Haskell.

6.1.4.2 Syntax of Simple Imperative Programs

Let us define the concrete syntax of the imperative programming language BIPL.

Illustration 6.5 (Concrete syntax of BIPL)

EGL resource languages/BIPL/cs.egl

```
// Statements
[skip] stmt : ';' ;
[assign] stmt : name '=' expr ';' ;
[block] stmt : '{' { stmt }* '}' ;
[if] stmt : 'if' '(' expr ')' stmt { 'else' stmt }? ;
[while] stmt : 'while' '(' expr ')' stmt ;

// Expressions
[or] expr : bexpr { '||' expr }? ;
[and] bexpr : cexpr { '&&' bexpr }? ;
[lt] cexpr : aexpr { '<' aexpr }? ;
[leq] cexpr : aexpr { '<=' aexpr }? ;
[eq] cexpr : aexpr { '==' aexpr }? ;
[geq] cexpr : aexpr { '>=' aexpr }? ;
[gt] cexpr : aexpr { '>' aexpr }? ;
[add] aexpr : term { '+' aexpr }? ;
[sub] aexpr : term { '−' aexpr }? ;
[mul] term : factor { '*' term }? ;
[negate] factor : '−' factor ;
[not] factor : '!' factor ;
[intconst] factor : integer ;
[var] factor : name ;
[brackets] factor : '(' expr ')' ;
```

There are several different statement and expression forms. For instance, the first rule ([skip]) defines the syntax of an empty statement; the second rule ([assign]) defines the syntax of assignment with a variable to the left of "=" and an expression to the right of "=". The rule for if-statements makes the 'else' branch optional, as in the C and Java languages.

The rules for expression forms are layered with extra nonterminals *bexpr* (for "Boolean expressions"), *cexpr* (for "comparison expressions"), etc. to model operator priorities such as that "*" to bind more strongly than "+". We note that the syntax of names and integers is left unspecified here.

Exercise 6.5 (Priorities of alternatives) [Intermediate level]
Practical grammar notations and the corresponding parsing approaches support a more concise approach to the modeling of priorities, not just of operators but possibly of alternatives (rules) in general. Study some grammar notation, for example, YACC [8], SDF [12], or ANTLR [11], with regard to priorities and sketch a possible extension of EGL, illustrated with a revision of the BIPL grammar.

6.1.4.3 Syntax of Simple Functional Programs

Let us define the concrete syntax of the functional programming language BFPL.

Illustration 6.6 (Concrete syntax of BFPL)

EGL resource languages/BFPL/cs.egl

```
// Program = functions + main expression
program : { function }* main ;
function : funsig fundef ;
funsig : name '::' funtype ;
fundef : name { name }* '=' expr ;
funtype : simpletype { '−>' simpletype }* ;
main : 'main' '=' 'print' '$' expr ;

// Simple types
[inttype] simpletype : 'Int' ;
[booltype] simpletype : 'Bool' ;

// Expressions
[unary] expr : uop subexpr ;
[binary] expr : '(' bop ')' subexpr subexpr ;
[subexpr] expr : subexpr ;
[apply] expr : name { subexpr }+ ;
[intconst] subexpr : integer ;
[brackets] subexpr : '(' expr ')' ;
[if] subexpr : 'if' expr 'then' expr 'else' expr ;
[arg] subexpr : name ;

// Unary and binary operators
[negate] uop : '−' ;
[not] uop : 'not' ;
[add] bop : '+' ;
...
```

The syntax of BFPL is focused on expression forms. There are further syntactic categories for programs (as lists of functions combined with a “main” expression) and function signatures. The central expression form is that of function application. Curried notation is assumed. Operators are applied in (curried) prefix notation too. Thus, operator priorities are not modeled. We note that the syntax of names and integers is left unspecified here.

6.1.4.4 Syntax of Finite State Machines

Let us define the concrete syntax of the DSML FSML.

Illustration 6.7 (Concrete syntax of FSML)

EGL resource languages/FSML/cs.egl

```
fsm : {state}* ;
state : {'initial'}? 'state' stateid '{' {transition}* '}' ;
transition : event {'/' action}? {'->' stateid}? ';' ;
stateid : name ;
event : name ;
action : name ;
```

That is, an FSM is a collection of *state* declarations, each of which groups state transitions together. Each transition identifies an *event* (say, an input symbol), an optional *action* (say, an output symbol), and an optional target *stateid*. An omitted target state is taken to mean that the target state equals the source state. We note that the syntax of names is left unspecified here.

Exercise 6.6 (EGL to BGL reduction) [Basic level]
Apply the EGL-to-BGL reduction to the definition of the syntax of FSML in Illustration 6.7.

6.2 Concrete versus Abstract Syntax

The definitions of concrete and abstract syntax differ in that they model text-based versus tree- or graph-based languages. In addition, concrete and abstract syntax also differ in terms of intention – they are targeted towards the language user and the language implementer, respectively. This difference in intention affects the level of abstraction in the definitions. Abstraction potential arises from constructs that have a rich concrete syntax, but where fewer details or variations are sufficient to ultimately assign meaning to the constructs. We look at such differences in the sequel.

At the most basic level, concrete and abstract syntax differ just in terms of representation or notation. Here are some definition fragments of the expression language BTL:

```
-- Concrete syntax of BTL
[zero] expr : "zero" ;
[succ] expr : "succ" expr ;
```

```
-- Abstract syntax of BTL
symbol zero : → expr ;
symbol succ : expr → expr ;
```

In the concrete syntax, "succ" is modeled as a prefix symbol because it precedes its operand in the grammar rule. In the abstract syntax, succ is a prefix symbol simply because *all* symbols are prefix symbols in such a basic, signature-based abstract syntax. In the concrete syntax, we have full control over the notation. For instance, the rule [succ] favors curried notation for function application, i.e., using juxtaposition instead of parentheses and commas. Again, however, in the abstract syntax, uncurried notation is cemented into the formalism as the assumed representation of terms (trees).

Exercise 6.7 (Uncurried notation for BTL expressions) [Basic level]
Revise the concrete syntax of BTL to use uncurried notation instead.

Let us also consider the differences between concrete and abstract syntax for the imperative language BIPL. We have already pointed out earlier (Section 6.1.4.2) that the grammar of BIPL models expression forms with a dedicated nonterminal for each operator priority. Such layering makes no sense in the tree-based abstract syntax and it is indeed missing from the earlier signature (Section 3.1.5.2). Another difference concerns the if-statement:

```
-- Concrete syntax of BIPL
[if] stmt : 'if' '(' expr ')' stmt { 'else' stmt }? ;
```

```
-- Abstract syntax of BIPL
symbol if : expr × stmt × stmt → stmt ;
```

That is, the else-part is optional in the concrete syntax, whereas it is mandatory in the abstract syntax. An optional else-part is convenient for the language user because no empty statement (“skip”) needs to be filled in to express the absence of an else-part. A mandatory else-part is convenient for the language implementer because only one pattern of the if-statement needs to be handled.

There is another major difference we should point out:

```
-- Concrete syntax of BIPL
[block] stmt : '{' { stmt }* '}' ;
```

```
-- Abstract syntax of BIPL
symbol seq : stmt × stmt → stmt ;
```

That is, in the concrete syntax, sequences of statements are formed as statement blocks with enclosing braces. This notation was chosen to resemble the syntax of C and Java. In the abstract syntax, there is a binary combinator for sequential composition. This simple model is convenient for the language implementer. For this correspondence between concrete and abstract syntax to be sound, we must assume that statement blocks have here no meaning other than sequential composition. By contrast, in C and Java, statement blocks actually define scopes with regard to local variables.

Exercise 6.8 (Abstraction for FSML) [Basic level]
Identify the differences between the concrete and the abstract syntax of FSML.

Exercise 6.9 (Abstraction for BFPL) [Basic level]
Identify the differences between the concrete and the abstract syntax of BFPL.

We mention in passing that some metaprogramming systems, for example, Rascal [10, 9] and Stratego XT [13, 3], advertise the view that, under certain conditions, there may not even be an abstract syntax for a language; all language processing is implemented on top of the concrete syntax, subject to suitable support for concrete object syntax, as we will discuss later (Section 7.5). The assumption is here that a metaprogrammer may prefer using the familiar, concrete syntactical patterns of the object language as opposed to the more artificial patterns according to an abstract syntax definition.

6.3 Languages as Sets of Strings

Let us complement the informal explanations of concrete syntax definitions given so far with formal definitions drawn from formal language theory [6]. In particular, we will define the meaning of grammars in a set-theoretic sense, i.e., a grammar "generates" a language as a set of strings.

6.3.1 Context-Free Grammars

BGL (or BNF) and EGL (or EBNF) are grammar notations for the fundamental formalism of context-free grammars (CFGs).

Definition 6.1 (Context-free grammar) *A CFG G is a quadruple $\langle N, T, P, s\rangle$ where N is a finite set of nonterminals, T is a finite set of terminals, with $N \cap T = \emptyset$, P is a finite set of rules (or productions) as a subset of $N \times (N \cup T)^*$, and $s \in N$ is referred to as the start symbol.*

As noted before, in the BGL and EGL grammar notations, we use the convention that the left-hand side of a grammar's first rule is considered the start symbol. Also, we note that BGL and EGL rules may be labeled whereas no labels are mentioned in the formal definition. Labels are simply to identify rules concisely.

6.3.2 The Language Generated by a Grammar

Rules can be applied in a "generative" sense: replace a nonterminal by a corresponding right-hand side. By many such replacements, one may eventually derive terminal strings. This is the foundation for interpreting a grammar as the definition of a language, namely the set of all terminal strings that are derivable from the grammar's start symbol.

Definition 6.2 (Context-free derivation) *Given a CFG $G = \langle N, T, P, s\rangle$ and a sequence $p\ n\ q$ with $n \in N$, $p, q \in (N \cup T)^*$, the sequence $p\ r\ q$ with $r \in (N \cup T)^*$ is called a derivation, as denoted by $p\ n\ q \Rightarrow p\ r\ q$, if there is a production $\langle n, r\rangle \in P$.*

The transitive closure of "$\Rightarrow$" is denoted by "$\Rightarrow^+$". The reflexive closure of "$\Rightarrow^+$" is denoted by "$\Rightarrow^*$".

Definition 6.3 (Language generated by a CFG) *Given a CFG $G = \langle N, T, P, s\rangle$, the language $L(G)$ generated by G is defined as the set of all the terminal sequences that are derivable from s. That is:*

$$L(G) = \{\ w \in T^* \mid s \Rightarrow^+ w\ \}$$

6.3.3 Well-Formed Grammars

Well-formedness constraints on grammars for ruling out nonsensical grammars are defined formally as follows.

Definition 6.4 (Well-formed CFG)
A CFG $G = \langle N, T, P, s\rangle$ is called well-formed if the following two conditions hold for each $n \in N$:

Productivity *There exists $w \in T^*$ such that $n \Rightarrow^+ w$.*
Reachability *There exist $p, q \in (N \cup T)^*$ such that $s \Rightarrow^* p\ n\ q$.*

Exercise 6.10 (Productivity of CFG) [Basic level]
Give a simple grammar that violates productivity defined in Definition 6.4.

Exercise 6.11 (Reachability of CFG) [Basic level]
Give a simple grammar that violates reachability defined in Definition 6.4.

Exercise 6.12 (Well-formed signature) [Intermediate level]
Consider again Definition 6.4 for well-formed CFGs. Transpose this definition, with its components for productivity and reachability, to signatures as used in abstract syntax definition (Chapter 3).

6.3.4 The Notion of Acceptance

Suppose we want to decide whether a given terminal string is an element of $L(G)$. We cannot perform a direct membership test because the set $L(G)$ is infinite for any nontrivial syntax definition. We need a computable kind of membership test instead. To this end, we introduce the algorithmic notion of acceptance. The term "recognition" is also used instead. Further, we may speak of an "acceptor" or "recognizer" instead, when we want to refer to the actual functionality for acceptance.

Definition 6.5 (Acceptor) *Given a CFG $G = \langle N, T, P, s \rangle$, an acceptor for G is a computable predicate a_G on T^* such that for all $w \in T^*$, $a_G(w)$ holds iff $s \Rightarrow^+ w$.*

The process of applying an acceptor is referred to as acceptance. In practice, we are interested in "descriptions" of such predicates. For instance, the grammar itself may serve as a description and the predicate may be obtained by "interpreting" the grammar. It is known from formal language theory that the membership problem for CFGs is decidable and, thus, a computable predicate such as the one in the definition can be assumed to exist. We will discuss some options later (Section 7.2).

6.4 Languages as Sets of Trees

An acceptor only answers the question whether a given string w is an element of the language generated by some grammar G. A parser, in addition, reports on the structure of w based on the rules of G. The structure is represented as a concrete syntax tree (CST). We may also say "parse tree" instead of "CST". Success of parsing means that at least one CST is returned. Failure of parsing means that no CST is returned. In this manner, we assign meaning to grammars in a second manner.

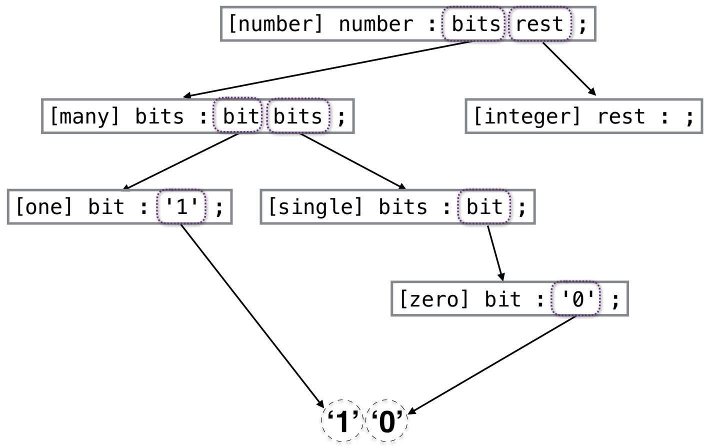

Fig. 6.1 CST for the binary number “10”.

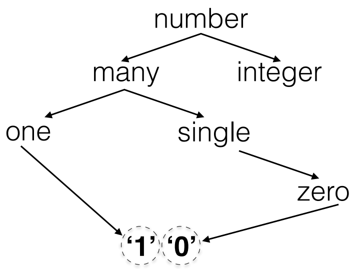

Fig. 6.2 Alternative CST representation.

6.4.1 Concrete Syntax Trees

A CST for a terminal string *w* contains the terminals of *w* as leaf nodes in the same order. Each CST node with its subtrees represents the application of a grammar rule except for some leaf nodes that simply represent terminals. The root node corresponds to a rule application for the start symbol. Before formalizing this intuition, let us look at some examples.

CSTs can be represented or visualized in different ways. The representation in Fig. 6.1 uses rules as node infos. In the figure, we circled right-hand side grammar symbols to better emphasize the correspondence between them and the subtrees. The visualization in Fig. 6.2 is more concise. BGL’s rule labels are used as node infos here.

We are ready to define the CST notion formally.

Definition 6.6 (Concrete syntax tree) *Given a CFG $G = \langle N, T, P, s \rangle$ and a string $w \in T^*$, a CST for w according to G is a tree as follows:*

- *Nodes hold a rule or a terminal as info.*
- *The root holds a rule with s on the left-hand side as info.*
- *If a node holds a terminal as info, then it is a leaf.*
- *If a node holds rule $n \to v_1 \cdots v_m$ with $n \in N$, $v_1, \ldots, v_m \in N \cup T$ as info, then the node has m branches with subtrees t_i for $i = 1, \ldots, m$ as follows:*
  - *If v_i is a terminal, then t_i is a leaf with terminal v_i as info.*
  - *If v_i is a nonterminal, then t_i is a tree with a rule as info such that v_i is the left-hand side of the rule.*
- *The concatenated terminals at the leaf nodes equal w.*

6.4.2 *The Notion of Parsing*

Let us make the transition from acceptance to parsing.

Definition 6.7 (Parser) *Given a CFG $G = \langle N, T, P, s \rangle$, a parser for G is a partial function p_G from T^* to CSTs such that for all $w \in L(G)$, $p_G(w)$ returns a CST of w and for all $w \notin L(G)$, $p_G(w)$ is not defined.*

The process of applying a parser is referred to as parsing. A parser returns no CST for a given input exactly in the same cases as when an acceptor fails.

6.4.3 *Ambiguous Grammars*

If a parser has a choice of what CST to return, then this means that the grammar is ambiguous, as formalized by the following definition.

Definition 6.8 (Ambiguous grammar) *A CFG $G = \langle N, T, P, s \rangle$ is called ambiguous, if there exists a terminal string $w \in T^*$ with multiple CSTs.*

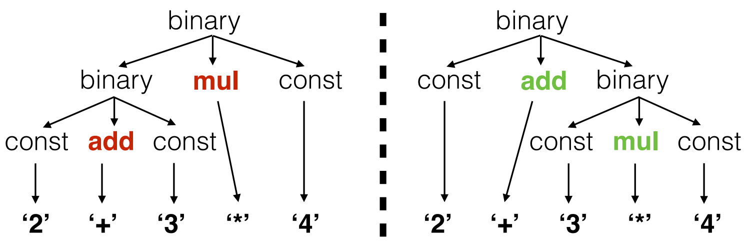

Fig. 6.3 Alternative CSTs for an arithmetic expression.

Let us consider a simple example for ambiguities.

Illustration 6.8 (Ambiguous grammar for arithmetic expressions)

EGL resource languages/EGL/samples/ambiguity.egl

```
[binary] expr : expr bop expr ;
[const] expr : integer ;
[add] bop : '+' ;
[mul] bop : '*' ;
```

In the grammar shown above, the syntax of binary expression is defined ambiguously. This is demonstrated in Fig. 6.3 by showing two CSTs for the expression "$2+3*4$". The tree on the right meets our expectation that "*" binds more strongly than "+". The grammar for BIPL (Illustration 6.5) addresses this problem by describing layers of expressions with dedicated nonterminals for the different priorities.

We mention in passing that Definition 6.7 could be revised to make a parser possibly return a collection of CSTs, i.e., a parse-tree forest. This may be useful in practice and may require an extra phase of filtering to identify a preferred tree eventually [2].

Exercise 6.13 (Ambiguous grammar) [Basic level]
Consider the rule for if-statements taken from Illustration 6.5:

```
[if] stmt : 'if' '(' expr ')' stmt { 'else' stmt }? ;
```

Demonstrate that this rule implies an ambiguous grammar.

While both concrete and abstract syntax, as discussed thus far, provide a tree-based definition of a software language, there is an important difference. In the case of concrete syntax, the trees arise as a secondary means: to represent the derivation of language elements, which are strings. In the case of abstract syntax, the trees correspond to the language elements themselves.

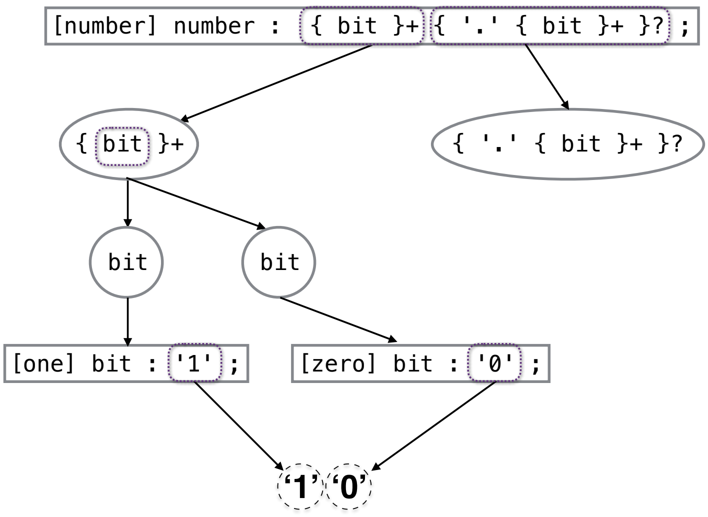

Fig. 6.4 A CST with extra nodes due to EGL expressiveness.

Definition 6.6 (CST) applies to the basic grammar notation of BGL only. CSTs for the extended grammar notation of EGL require extra nodes:

- Each occurrence of "?", "*", and "+" within a rule is subject to an extra node with 0, 1, or more branches.
- Each branch of these extra nodes is rooted in an extra node with the list of symbols that are optional or to be repeated.

These extra nodes are visualized by ellipses in Fig. 6.4. The example in the figure is a variation on Fig. 6.1.

6.5 Lexical Syntax

In the illustrative grammars presented earlier, we left out some details: the syntax of names (FSML, BIPL, and BFPL) and integers (BIPL and BFPL). Such details are usually considered to be part of what is called the *lexical syntax*. That is, the lexical syntax covers the syntactic categories that correspond to syntactical units without any "interesting" tree-based structure. The approach of defining a separate lexical syntax is something of a dogma, but we give in on this dogma for now.

Let us define the lexical syntax of FSML. The earlier grammar contained rules for redirecting several nonterminals for different kinds of names or symbols to the nonterminal name. Thus:

```
stateid : name ;
event : name ;
action : name ;
```

The nonterminal name can be defined by a rule as follows:

```
name : { alpha }+ ;
```

Here, alpha is a predefined nonterminal for uppercase and lowercase letters[2]. Thus, FSML's name is defined as nonempty sequences of letters. Generally, the lexical syntax of a language can be defined by grammar rules too. In practice, different grammar notations of varying expressiveness are used for this purpose.

There is a pragmatic reason for not having included the above rule in the earlier grammar. In one way (by separation, as done here) or another, we need to describe the indivisible lexical units of the language as opposed to divisible syntactical units that may contain white space (space, tab, newline, line feed) or comments. In the case of FSML, we want to admit white space everywhere – except, of course, within names or the terminals such as *'state'* and *'−>'*. Further, we may also want to declare the lexical syntax of white space and comments. To this end, we define a special nonterminal layout, which, by convention, defines the lexical syntax of strings to be skipped anywhere in the input between (but not within) lexical units. Let us provide the complete lexical syntax of FSML.

Illustration 6.9 (Lexical syntax of finite state machines (FSML))

EGL resource languages/FSML/ls.egl

```
name : { alpha }+ ;
layout : { space }+ ;
```

Here, space is a "predefined" nonterminal which subsumes "white space", i.e., space, tab, newline, and line feed. Thus, FSML's layout is defined as a non-empty sequence of such white space characters.

Let us consider another example.

Illustration 6.10 (Lexical syntax of imperative programs (BIPL))

EGL resource languages/BIPL/ls.egl

```
name : { alpha }+ ;
integer : { digit }+ ;
layout : { space }+ ;
layout : '//' { { end_of_line }~ }* end_of_line ;
```

---

[2] In the rules for the lexical syntax, we assume predefined nonterminals for common character classes such as *alpha*, *space*, *digit*, and *end_of_line*.

BIPL's name is defined in the same way as in FSML (line 1). BIPL's integer is defined as nonempty sequence of digits (line 2). There are two rules for layout. The first one models white space in the same way as in FSML (line 3); the second one models C/Java-style line comments (line 4). In the last rule, we use negation "~" to express that no "end-of line" character is admitted in a given position.

Let us consider yet another example.

Illustration 6.11 (Lexical syntax of functional programs (BFPL))

EGL resource languages/BFPL/ls.egl

```
name : lower { alpha }* ;
integer : { digit }+ ;
layout : { space }+ ;
layout : '−−'{ { end_of_line }~ }* end_of_line ;
```

BFPL's name is defined as non-empty sequence of letters starting in lowercase. BFPL's integer is defined in the same way as in BIPL. There are two rules for layout. The first one captures white space in the same way as before; the second one models Haskell-style line comments.

Exercise 6.14 (Primitive types in syntax definitions) [Intermediate level]
Provide a convincing hypothesis that explains why the extended signature notation (ESL) features primitive types such as string *and* integer *whereas EGL does not.*

6.6 The Metametalevel

The grammar notations BGL and EGL correspond to proper software languages in themselves. In this section, the concrete and abstract syntaxes of these syntax definition languages are defined. Accordingly, we operate at the metametalevel. This development enables, for example, a systematic treatment of acceptance and parsing. We also revisit the abstract syntax definition languages BSL, ESL, and MML and define their concrete syntaxes, as we have only defined their abstract syntaxes previously (Section 3.4).

6.6.1 The Signature of Grammars

Let us define the abstract syntax of concrete syntaxes. In this manner, concrete syntaxes can be processed programmatically, for example, when implementing ("generating") parsers. To make the presentation more approachable, the basic grammar notation (BGL) is covered first.

Specification 6.1 (The ESL signature of BGL grammars)

ESL resource languages/BGL/as.esl

```
type grammar = rule⋆ ;
type rule = label × nonterminal × gsymbols ;
type gsymbols = gsymbol⋆ ;
symbol t : terminal → gsymbol ;
symbol n : nonterminal → gsymbol ;
type label = string ;
type terminal = string ;
type nonterminal = string ;
```

The abstract syntactical representation of grammars can be illustrated as follows.

Illustration 6.12 (BNL's grammar in abstract syntax)

Term resource languages/BNL/cs.term

```
[
  (number,number,[n(bits),n(rest)]),
  (single,bits,[n(bit)]),
  (many,bits,[n(bit),n(bits)]),
  (zero,bit,[t('0')]),
  (one,bit,[t('1')]),
  (integer,rest,[]),
  (rational,rest,[t('.'),n(bits)])
].
```

Let us now provide the signature for the extended grammar notation.

Specification 6.2 (The ESL signature of EGL grammars)

ESL resource languages/EGL/as.esl

```
type grammar = rule⋆ ;
type rule = label? × nonterminal × symbols ;
type symbols = symbol⋆ ;
symbol t : terminal → symbol ;
symbol n : nonterminal → symbol ;
symbol star : symbols → symbol ;
symbol plus : symbols → symbol ;
symbol option : symbols → symbol ;
symbol not : symbols → symbol ;
type label = string ;
type terminal = string ;
type nonterminal = string ;
```

We should impose constraints on the language of grammars, such as that the rule labels are distinct and that the conditions of productivity and reachability (Definition 6.4) are met, but we omit a discussion of these routine details.

6.6.2 The Signature of Concrete Syntax Trees

On top of the signature of grammars, we can also define a signature for CSTs, which is useful, for example, as a fundamental representation format for parsing results. We cover only the basic grammar notation (BGL) here. We introduce a corresponding language: *BCL* (*B*GL *C*ST *L*anguage).

Specification 6.3 (Signature of BGL-based CSTs)

ESL resource languages/BCL/as.esl

```
symbol leaf : terminal → ptree ;
symbol fork : rule × ptree* → ptree ;
// Rules as in BGL
...
```

Thus, there is a *leaf* symbol for a terminal, and there is a *fork* symbol which combines a rule and a list of subtrees for the nonterminals on the right-hand side of the rule. An actual CST, which conforms to the signature, is shown below.

Illustration 6.13 (CST for the binary number “10”)

Term resource languages/BNL/samples/10.tree

```
fork(
  (number,number,[n(bits),n(rest)]), % rule
  [ % list of branches
    fork( % 1st branch
      (many,bits,[n(bit),n(bits)]), % rule
      [ % list of branches
        fork( % 1st branch
    (one,bit,[t('1')]), % rule
    [leaf('1')]), % leaf
  fork( % 2nd branch
    (single,bits,[n(bit)]), % rule
    [ % list of branches
      fork( % 1st branch % rule
        (zero,bit,[t('0')]),
        [leaf('0')])])]), % leaf
    fork( % 2nd branch
      (integer,rest,[]), % rule
      [])]). % empty list of branches
```

Exercise 6.15 (CSTs for EGL) [Basic level]
Devise a signature for CSTs for the extended grammar notation EGL.

The signature as stated above is underspecified. For a CST to be well-formed, it must use only rules from the underlying grammar and it must combine them in a correct manner, as constrained by Definition 6.6.

6.6.3 The Grammar of Grammars

We can also devise a grammar of grammars, which is useful, for example, for parsing grammars. To make the presentation more approachable, the basic grammar notation (BGL) is covered first.

Specification 6.4 (The EGL grammar of BGL grammars)

EGL resource languages/BGL/cs.egl

```
grammar : {rule}* ;
rule : '[' label ']' nonterminal ':' gsymbols ';' ;
gsymbols : {gsymbol}* ;
[t] gsymbol : terminal ;
[n] gsymbol : nonterminal ;
label : name ;
terminal : qstring ;
nonterminal : name ;
```

Let us now provide the grammar for the extended grammar notation.

Specification 6.5 (The EGL grammar of EGL grammars)

EGL resource languages/EGL/cs.egl

```
grammar : {rule}* ;
rule : {'[' label ']'}? nonterminal ':' gsymbols ';' ;
gsymbols : {gsymbol}* ;
[t] gsymbol : terminal ;
[n] gsymbol : nonterminal ;
[star] gsymbol : '{' gsymbols '}' '*' ;
[plus] gsymbol : '{' gsymbols '}' '+' ;
[option] gsymbol : '{' gsymbols '}' '?' ;
[not] gsymbol : '{' gsymbols '}' '~' ;
label : name ;
terminal : qstring ;
nonterminal : name ;
```

We also provide a separate grammar for the lexical syntax.

Illustration 6.14 (Lexical syntax of EGL)

EGL resource languages/EGL/ls.egl

```
qstring : quote { { quote }~ }+ quote ;
name : { csymf }+ ;
layout : { space }+ ;
layout : '//' { { end_of_line }~ }* end_of_line ;
```

6.6.4 *The Grammar of Signatures*

Let us also define the concrete syntax of tree-based abstract syntaxes, as useful, for example, for parsing signatures. The basic signature notation (BSL) is covered first.

Specification 6.6 (The EGL grammar of BSL signatures)

EGL resource languages/BSL/cs.egl

```
signature : { symbol ';' }* ;
symbol : 'symbol' name ':' args '->' name ;
args : { name { '#' name }* }? ;
```

Let us now provide the grammar for the extended signature notation.

Specification 6.7 (The EGL grammar of ESL signatures)

EGL resource languages/ESL/cs.egl

```
signature : { decl ';' }* ;
[type] decl : 'type' name '=' typeexprs ;
[symbol] decl : 'symbol' name ':' { typeexprs }? '->' name ;
typeexprs : typeexpr { '#' typeexpr }* ;
typeexpr : factor cardinality ;
[boolean] factor : 'boolean' ;
[integer] factor : 'integer' ;
[float] factor : 'float' ;
[string] factor : 'string' ;
[term] factor : 'term' ;
[tuple] factor : '(' typeexpr { '#' typeexpr }+ ')' ;
[sort] factor : name ;
[star] cardinality : '*' cardinality ;
[plus] cardinality : '+' cardinality ;
[option] cardinality : '?' cardinality ;
[none] cardinality : ;
```

We also provide a separate grammar for the lexical syntax.

Illustration 6.15 (Lexical syntax of ESL)

EGL resource languages/ESL/ls.egl

```
name : { csymf }+ ;
layout : { space }+ ;
layout : '//' { { end_of_line }~ }* end_of_line ;
```

6.6.5 The Grammar of Metamodels

It remains to provide the grammar of metamodels (MML) which is useful, for example, for parsing metamodels.

Specification 6.8 (The EGL grammar of MML metamodels)

EGL resource languages/MML/cs.egl

```
metamodel : { classifier }* ;
[datatype] classifier : 'datatype' name ';' ;
[enum] classifier : 'enum' name '{' name {',' name}* '}' ;
[class] classifier : abstract 'class' name super members ;
super : { 'extends' name }? ;
[abstract] abstract : 'abstract' ;
[concrete] abstract : ;
members : '{' { member }* '}' ;
member : kind name ':' type ';' ;
[value] kind : 'value' ;
[part] kind : 'part' ;
[reference] kind : 'reference' ;
type : name cardinality ;
[plus] cardinality : '+' ;
[star] cardinality : '*' ;
[option] cardinality : '?' ;
[one] cardinality : ;
```

We also provide a separate grammar for the lexical syntax.

Illustration 6.16 (Lexical syntax of MML)

EGL resource languages/MML/ls.egl

```
name : { csymf }+ ;
layout : { space }+ ;
layout : '//' { { end_of_line }~ }* end_of_line ;
```

Summary and Outline

We have explained how (context-free) grammars (or different notations for them) may serve for modeling string-based concrete syntax. We have defined two different semantics of grammars: (i) a set-theoretic semantics, defining a software language as a set of strings; and (ii) a tree-oriented semantics, defining the structure of language elements in terms of the productions of a grammar. Further, we have defined the fundamental notions of acceptance and parsing, which ultimately have to be complemented by algorithms for parsing.

In the next chapter, we will discuss the implementation of concrete syntax, including basic parsing algorithms and practical approaches to parsing, formatting, and mapping between concrete and abstract syntax, as well as the use of concrete syntax in metaprograms.

References

1. Backus, J.W., Bauer, F.L., Green, J., Katz, C., McCarthy, J., Perlis, A.J., Rutishauser, H., Samelson, K., Vauquois, B., Wegstein, J.H., van Wijngaarden, A., Woodger, M.: Revised report on the Algorithm Language ALGOL 60. Commun. ACM **6**(1), 1–17 (1963)
2. van den Brand, M., Scheerder, J., Vinju, J.J., Visser, E.: Disambiguation filters for scannerless generalized LR parsers. In: Proc. CC 2002, *LNCS*, vol. 2304, pp. 143–158. Springer (2002)
3. Bravenboer, M., Kalleberg, K.T., Vermaas, R., Visser, E.: Stratego/XT 0.17. A language and toolset for program transformation. Sci. Comput. Program. **72**(1-2), 52–70 (2008)
4. Chomsky, N.: Three models for the description of language. IRE Transactions on Information Theory **2**(3), 113–124 (1956)
5. Ford, B.: Parsing expression grammars: A recognition-based syntactic foundation. In: Proc. POPL, pp. 111–122. ACM (2004)
6. Hopcroft, J., Motwani, R., Ullman, J.: Introduction to Automata Theory, Languages, and Computation. Pearson (2013). 3rd edition
7. ISO/IEC: ISO/IEC 14977:1996(E). Information Technology. Syntactic Metalanguage. Extended BNF. (1996). Available at http://www.cl.cam.ac.uk/~mgk25/iso-14977.pdf
8. Johnson, S.C.: YACC—Yet Another Compiler Compiler. Computer Science Technical Report 32, AT&T Bell Laboratories (1975)
9. Klint, P., van der Storm, T., Vinju, J.J.: RASCAL: A domain specific language for source code analysis and manipulation. In: Proc. SCAM, pp. 168–177. IEEE (2009)
10. Klint, P., van der Storm, T., Vinju, J.J.: EASY meta-programming with Rascal. In: GTTSE 2009, Revised Papers, *LNCS*, vol. 6491, pp. 222–289. Springer (2011)
11. Parr, T.: The Definitive ANTLR 4 Reference. Pragmatic Bookshelf (2013). 2nd edition
12. Visser, E.: Syntax definition for language prototyping. Ph.D. thesis, University of Amsterdam (1997)
13. Visser, E.: Stratego: A language for program transformation based on rewriting strategies. In: Proc. RTA, *LNCS*, vol. 2051, pp. 357–362. Springer (2001)