LR Parsing
=
Grammar Transformation + *LL* Parsing

Making *LR* Parsing More Understandable *And* More Efficient

Peter Pepper

No. 99 – 5

April, 1999

Abstract

The paper has three aims. Its primary focus is a derivation method which is — in contrast to many of the classical presentations in the literature — *easy to comprehend* and thus easy to adapt to different needs. Secondly, it presents an improved LR parser which has the *power* of LR parsing, but (almost) the *efficiency* of $LALR$ parsing. Finally, it elucidates the strong conceptual relationships that actually exist between LL and LR parsing.

It is also briefly shown that the flexibility and easy adaptability of our techniques open up the possibility for *new applications*. As an example, we outline how type-based overload resolution (and thus type analysis) for arbitrary mixfix operators can be implemented in the framework of a polymorphic functional language.

Contents

Chapter 1

Introduction

The area of parsing is undoubtedly one of the best researched areas in computer science.[1] One might therefore wonder whether there are any new insights left worth presenting in a paper. To make this clear from the beginning: our paper, too, will not reveal new parsing techniques which have never been heard of before. Rather, our studies remain within the classical (and highly successful) realm of LL and LR parsing. Nonetheless, there are still a number of contributions made in this paper:

1. First and foremost, we present a *new derivation process* for LR-style parsing which has several advantages:
   - It is much *simpler* and thus much *easier to comprehend* (and verify) than the traditional ways of presenting LR parsing.[2] The main reason is that the entire development takes place within a single formalism rather than switching back and forth between several mechanisms (grammars, automata, tables, programs, ...).
   - This simplicity and clarity makes the approach very *flexible* and thus facilitates the construction of *variations* (e.g. incremental or parallel parsers and parser generators) as well as integration into different environments.
   - It is easy to obtain a parser which combines LR-style efficiency with the full backtrack power needed to cope with (those parts of) languages that do not satisfy the necessary prerequisites. This is sometimes referred to as *generalized LR parsing* [36, 31].
2. Equally importantly, our approach *unifies* the paradigms of $LL(k)$ and $LR(k)$ parsing. That is, we have one derivation method that *simultaneously generates an $LL(k)$ and an $LR(k)$ parser.*[3]
3. As a somewhat surprising result, our method generates a parser that combines the *power of LR parsing with the efficiency of LALR parsing.* That is, the parser is $LR(1)$, but the size of its tables is much closer to that of an $LALR(1)$ parser than to that of traditional $LR(1)$ parsers.
4. Our approach trivially includes *sentential-form parsing* and is thus suited for tools such as syntax-directed editors, integrated software development environments, transformation

[1] An extensive bibliography is given by Nijholdt [20].

[2] This claim is confirmed by several years of teaching the method to students; they now find LR parsing no more complex than LL parsing.

[3] As a matter of fact, our protoypical implementation has a command line option that is only checked at a handfull of points in the program in order to output either an $LL(1)$ or an $LR(1)$ parser.

systems and the like.[4]

5. The implementation is easily adaptable to an *incremental parser generation*, which is also an important prerequisite for many applications.

6. We obtain *informative insights* into the relationship between different parsing paradigms, showing, for example, why there is actually not much difference between LL and LR parsing,[5] and why LR parsing is more powerful than LL parsing.

7. The flexibility opens up *new application domains*, such as the use of LR-style parsing for type analysis and overload resolution in polymorphic languages.

As regards the comprehensibility, every mathematician is familiar with the uneasy feeling caused by certain proofs: you can follow each and every step, but at the end you still wonder: How do all these steps lead to the result? What is the *idea* behind the proof? In short, *why did the proof work?* To many people, the LR-parsing techniques presented in articles and textbooks fall into the same category. On a step-by-step basis, the αs, βs, and γs somehow fit together, but the big picture does not emerge.

This effect became particularly noticeable when I tried to derive a *parallel* parser. It was by no means clear where and which changes had to be applied to the standard table-generating procedures in order to achieve the necessary adaptions. A similar problem arose when we tried to allow the user of our functional language OPAL [11, 26] to define arbitrary mixfix operators (in connection with liberal rules for overloading and polymorphism): the analysis of such programs led to the need for incremental parser generation and "fuzzy" nonterminals. I thus decided to reconsider the whole derivation process for LR-style parsing.

The guiding principle of the design is to make the proofs so straightforward and simple that they are even amenable to automatic verification tools. Thus, the derivation is based solely on *equivalence-preserving transformations of grammars*. In my opinion, this is a much more transparent concept than the traditional approach based on "sets of $LR(0)$ items" with their rather odd use of "dotted production rules" and "spontaneous" and "propagated" lookahead symbols.

A note on the literature. The topic of parsing has been the subject of intensive research over several decades, so it would be completely futile to strive for a list of references which comes even close to completeness. On the other hand, the topic is extremely well covered in standard textbooks. Thus, we basically content ourselves with these textbooks and only add individual references where they specifically address issues we cover.[6]

Following the introduction of basic terminology in Section 2, the paper has three main parts:

Part I (Sections 3–4) presents our approach to LR-style parsing, which may be paraphrased as follows. *Traditionally, LR parsers are obtained by applying a complex generation algorithm to the original grammar. We apply a simple generation algorithm to a transformed grammar.*. This leads to our **Main Theorem** 4.0.2, which can be visualized as follows:

$$\begin{array}{ccc} \mathcal{G} & \xrightarrow{\textit{transform}} & \mathcal{G}' \\ \vdots & & \vdots \\ LR(k) & \text{— iff —} & LL(k) \end{array}$$

[4] In their recent paper [38] Wagner and Graham stress the importance of sentential-form parsing. There it can also be seen how much more complex this extension is for the traditional parsing techniques.

[5] This observation also has been made — albeit in another setting — by others [31].

[6] In order not to interrupt the flow of the presentation we defer most of the references to the footnotes. Unless one is specifically interested in the literature, one may therefore skip all footnotes from now on.

That is, the original grammar $\mathcal{G}$ is $LR(k)$ iff the transformed grammar $\mathcal{G}'$ is $LL(k)$. This way of proceeding is beneficial, because the transformation is very simple as well.

Part II (Sections 5–7) considers the actual parsing process induced by our transformed grammars. It is shown that our approach captures the "conceptual essence" of LR parsing in the sense that it makes the same decisions and thus has the same expressive power as the classical approaches. But we use different implementation techniques and therefore should rather speak of an "emulation" of LR parsing by a top-down parser (actually by an $LL(k)$ parser). On this basis we study the relationship between our approach and the traditional way of proceeding in LR and $LALR$ parsers.

Part III (Sections 8–10) presents some extensions and non-standard applications. These include a short discussion of *generalized LR parsing*, that is, LR parsing for ambiguous grammars, and of error handling.

The ideas underlying this paper have been used over a number of years in student courses at the Technical University of Berlin [23]. Our experience shows that the approach appears to be considerably easier to comprehend than traditional methods. I therefore decided to pursue the subject further in order to make the overall principle even easier to understand. The result of this effort is reported in the following.

Chapter 2

Basic Concepts: Grammars and Transformations

We assume that the reader is familiar with the basic notions of formal languages. Therefore, we list only very briefly our notational conventions here.

2.1 Grammars

The most fundamental notions that we need in our discussion are those of "grammars" and their "derivations" and "reductions".

Definition 2.1.1 *(Grammar) A* ***grammar*** *is a four-tuple* $\mathcal{G} = (\mathcal{T}, \mathcal{N}, S, \mathcal{P})$*, where* $\mathcal{T}$ *is the set of* terminal symbols*,* $\mathcal{N}$ *is the set of* nonterminal symbols*,* $S \in \mathcal{N}$ *is the* start symbol*, and* $\mathcal{P}$ *is the set of* production rules*, which are of the form* $A \to u$ *with* $A \in \mathcal{N}$ *and* $u \in (\mathcal{T} \cup \mathcal{N})^*$*. As usual, we require that* $\mathcal{N} \cap \mathcal{T} = \emptyset$*. By* $\mathcal{V} = \mathcal{N} \cup \mathcal{T}$ *we denote the set of all symbols.* □

Definition 2.1.2 *(Derivation, reduction) A* ***derivation*** *of a string* s *is a sequence of derivation steps, beginning with the start symbol* S *and ending with the string* s*. In each step, denoted* $s_i \Rightarrow s_{i+1}$*, some nonterminal in the intermediate string* s_i *is replaced by one of its right-hand sides in the grammar* $\mathcal{G}$*, thus yielding* s_{i+1}*. We usually denote the derives relation as* $S \stackrel{*}{\Rightarrow} s$*. (When needed, the notion of derivation can also be extended from the start symbol* S *to arbitrary nonterminals.) A* ***reduction*** *is the converse of a derivation.*[1] □

Definition 2.1.3 *(Language) The* ***language*** $\mathcal{L}_t(\mathcal{G})$ *of a given grammar* $\mathcal{G}$ *is the set of all terminal strings derivable from the start symbol* S*, that is,* $\mathcal{L}_t(\mathcal{G}) = \{u \in \mathcal{T}^* \mid S \stackrel{*}{\Rightarrow} u\}$*. The strings* $u \in \mathcal{L}_t(\mathcal{G})$ *are called* ***sentences*** *(or* words*).*

As a straightforward generalization one also considers ***sentential forms*** $u \in \mathcal{V}^*$*, which may contain terminals as well as nonterminals;*[2] *the language of sentential forms is denoted by* $\mathcal{L}(\mathcal{G})$*.* □

[1] In [29], these two concepts are formalized as dual forms of grammars: *generative* and *analytic*.

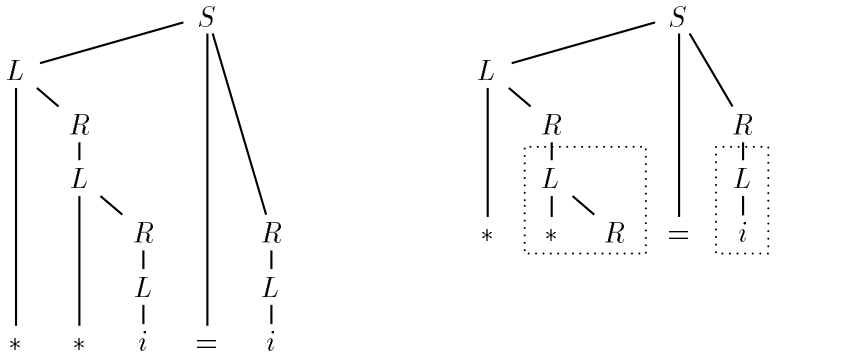
[2] This is not only necessary for *LR*-style parsing, but is also useful e.g. in connection with certain tools such as syntax-directed editors, incremental editors [38] or program transformation systems, where one works with so-called *program schemes*. Such schemes are nothing but sentential forms.

To illustrate our concepts we use the following grammar[3] as a running example throughout most of the paper. It generates sentences of the kind "$***i = **i$", with arbitrarily many stars.

$$\begin{array}{lcl} \text{S} & \rightarrow & \text{L} = \text{R} \\ & & \text{R} \\ \text{L} & \rightarrow & * \text{R} \\ & & \text{i} \\ \text{R} & \rightarrow & \text{L} \end{array}$$

2.2 Parse Trees and Postfix Notation

The essence of derivations and reductions is captured in so-called **parse trees**: Parse tree (a) in Figure 2.1 represents the full derivation of the *sentence* "$**i = i$", while the tree (b) represents a partial derivation, that is, a *sentential form*.



(a) full tree (sentence) (b) intermediate tree (sentential form)

Figure 2.1: Variations of parse trees

Note: For the subsequent specifications it will be convenient to employ the "apply-to-all" operator that is well-known from functional programming: the application of a function f to all elements of a set S is denoted by

$$f * S \stackrel{def}{=} \{\ f(x) \mid x \in S\ \}$$

Definition 2.2.1 (Parse trees) *The parse trees, in short* **trees**, *for a grammar $\mathcal{G}$ are defined as the set $\mathcal{T}(\mathcal{G}) = \{\ t \mid t$ is a tree fulfilling condition $(*)\}$ (see also [32])*

$(*)$ *Each subtree t' contained in t has the following property: if we take the root of t' as left-hand side and the sequence of the roots of the subtrees of t' as right-hand side, then we obtain a production of $\mathcal{G}$. Formally:*

$$(root(t') \rightarrow root * subtrees(t')) \in \mathcal{P}$$

using obvious functions root and subtrees. $\square$

The following properties are trivial consequences of this definition (see also Figure 2.1).

Lemma 2.2.1 *The leaves of $t \in \mathcal{T}(\mathcal{G})$, read from left to right, yield a sentence or sentential form $s \in \mathcal{L}(\mathcal{G})$. Formally: leaves $* \mathcal{T}(\mathcal{G}) = \mathcal{L}(\mathcal{G})$, with the obvious function leaves.*

[3] This example is taken from the standard textbook [1] because it is one of the smallest grammars illustrating the difference between SLR and LALR parsers.

Lemma 2.2.2 *The redices occurring in the reduction process correspond to complete subtrees of height 1.*

2.3 Adding Rule Numbers

Note: There is a technique that is helpful in many circumstances; it is based on the concept of so-called ***null nonterminals***: one can add to the grammar new nonterminals that derive the empty string ε, since this does not change the language. These additional nonterminals can then be used for various purposes: they may carry semantic actions or they may act as "assertions" that guide reductions or enable correctness proofs. We employ this technique for two purposes: In this section we use it for integrating the tree generation more homgeneously into the parsing process. And in the next section we use it for coping more easily with the idea of "follow sets".

The standard notion of production rules transfers too little information into the parse trees. Let us therefore briefly redesign our grammars and parse trees. First of all, we add production numbers[4] to our grammar, as illustrated by the modified Grammar $\mathcal{G}_0$ in Table 2.1.

Grammar $\mathcal{G}_0$		
S	$\rightarrow$	L = R ①
		R ②
L	$\rightarrow$	* R ③
		i ④
R	$\rightarrow$	L ⑤

Table 2.1: Grammar with rule annotations

Now we no longer label the inner nodes of the parse tree using the nonterminals, rather use the numbers ⓘ of the corresponding production rules. The result is illustrated by the tree (b) in Figure 2.2 (where we have additionally attributed the nodes by the nonterminals as their "types"). Note that this is actually a better representation than the traditional form (a), since it contains more information. The nonterminals can be trivially retrieved from the rule numbers, but the rule numbers also differentiate between the different productions for the nonterminal.[5]

However, the crucial point is that this representation not only conveys *more* information than the classical parse trees, it also conveys *essential* information: if we carry out a *postorder traversal* of the tree (b), we obtain the following string:

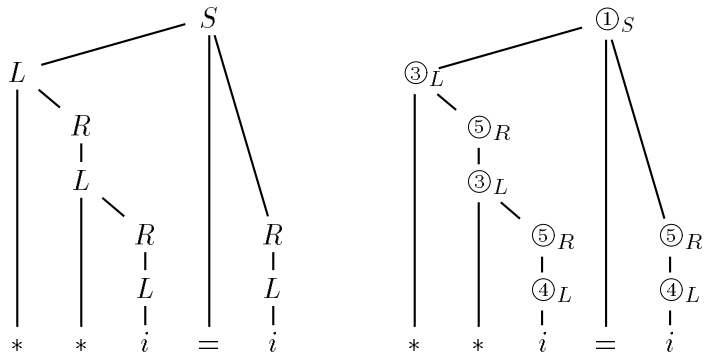
$$* \; * \; i \; ④ \; ⑤ \; ③ \; ⑤ \; ③ \; = i \; ④ \; ⑤ \; ①$$

The symbols ⓘ therefore have a dual nature. On the one hand, they are considered to be the *empty string*; hence, they are "invisible". On the other hand, they tell us which reductions are to be applied where. To see this, consider the two corresponding derivations of our running example in the two variants of the grammar.

$$\begin{array}{lllll} \text{Original Grammar:} & * \; * \; i & & = & i \\ \text{Modified Grammar:} & * \; * \; i \; ④ \; ⑤ \; ③ \; ⑤ \; ③ & & = & i \; ④ \; ⑤ \; ① \end{array}$$

[4] I was told that a numbering of this kind was also used by Don Knuth in an early paper on parsing; unfortunately, I have not been able to track this paper down.

[5] Incidentally, this illustrates very clearly the close connection between term algebras over a signature Σ and abstract syntax trees for grammars, as has already been discussed in great detail in [35]: our numbers correspond to the function symbols of the signature ("named production rules").



(a) original tree (b) tree with rule numbers

Figure 2.2: Parse trees with numbers

The first view considers the ⓘ as null nonterminals, the second one as some set $\mathcal{A}$ of "pseudo terminals", called *actions*.

Lemma 2.3.1 *Consider a grammar $\mathcal{G} = (\mathcal{T}, \mathcal{N}, S, \mathcal{P})$ and its two number-augmented variants $\mathcal{G}' = (\mathcal{T}, \mathcal{N} \cup \mathcal{A}, S, \mathcal{P}')$, where $\mathcal{A}$ is the set of actions viewed as null nonterminals, and $\mathcal{G}'' = (\mathcal{T} \cup \mathcal{A}, \mathcal{N}, S, \mathcal{P}')$, where the actions from $\mathcal{A}$ are viewed as pseudo terminals.*

- *The languages coincide up to a filtering of the actions ⓘ:*

$$\begin{array}{l} \mathcal{L}_t(\mathcal{G}) = \mathcal{L}_t(\mathcal{G}') = \mathit{filter} * \mathcal{L}_t(\mathcal{G}'') \\ \mathcal{L}(\mathcal{G}) = \mathit{filter} * \mathcal{L}(\mathcal{G}') = \mathit{filter} * \mathcal{L}(\mathcal{G}'') \end{array} \quad \textit{where filter eliminates the symbols } ⓘ$$

- *The strings of the extended grammar $\mathcal{G}''$ are in a one-to-one correspondence to the parse trees for the original grammar: $\mathcal{L}(\mathcal{G}'') = \mathit{postorder} * \mathcal{T}(\mathcal{G})$*

Proof: Obvious from the definitions (by a trivial induction on the height of the trees). □

As a matter of fact, this extension conveys even more information. If the numbers in a $\mathcal{G}''$ string are used — from left to right — to determine the succession of reduction rules (and the positions where they are applied), then we obtain an LR parsing of the sentence. This will become evident in the course of the paper. Actually the relationship is even more elucidating (which we mention here without proof):

- the *postorder* traversal of the tree corresponds to LR parsing;
- the *preorder* traversal corresponds to LL parsing.

Effect on grammar transformations: The addition of these numbers has essential advantages for our approach: *we can now freely transform our grammars (see Section 2.5)*. Since the numbers (which determine the tree generation) have become part of the rules, they are kept in the proper places under all transformations.

Based on the well-known fact that the postorder traversals are an isomorphic representation of the corresponding trees, we will base most of our subsequent discussions on these number-augmented grammars.

2.4 Continuations "(Lookahead")

In many parsing situations – both in *LL*- and *LR*-parsing – we need to know, which symbols can possibly follow *after* some point (usually after an action symbol ⓘ) that we have reached. Traditionally one uses the concept of "follow sets" for providing this knowledge. However, the whole treatment becomes much smoother when we integrate this knowledge more homogeneously into our approach, again using suitable *null nonterminals*: The ***continuation symbol*** $\boxed{A}\rangle$ intuitively stands for "anything that can follow after the nonterminal A in any sentential form".

Definition 2.4.1 *For any nonterminal A we define its* ***continuation language*** *as the set*

$$\mathcal{L}(\boxed{A}\rangle) \quad \stackrel{def}{=} \quad \{\ w \mid S \stackrel{*}{\Rightarrow} vAw\ \}$$

For the start symbol S we employ the convention that $\mathcal{L}(\boxed{S}\rangle)$ contains the special "end-of-input" symbol '$\natural$', that is, $\natural \in \mathcal{L}(\boxed{S}\rangle)$. □

The occurrence of a continuation symbol in some derivation of the kind

$$S \stackrel{*}{\Rightarrow} u\ \boxed{A}\rangle\ v$$

is to be considered as an *assertion* that the following substring v is contained in $\mathcal{L}(\boxed{A}\rangle)$. Such an assertion is at certain points in the parsing process needed to disambiguate conflicting productions (as we will see later on).

The integration of these continuations has to tackle two problems: (1) *Where shall continuation symbols be attached to the grammar?* (2) *How are the continuation languages described?* The first aspect is trivial: We simply append the pertinent continuation symbol $\boxed{A}\rangle$ to each right-hand side of the nonterminal A. This is admissible, since we consider $\boxed{A}\rangle$ as a null nonterminal. This extension is illustrated for our running example in the left part of Table 2.2.

Grammar $\mathcal{G}_0$		
S	→	L = R ① $\boxed{S}\rangle$
		R ② $\boxed{S}\rangle$
L	→	* R ③ $\boxed{L}\rangle$
		i ④ $\boxed{L}\rangle$
R	→	L ⑤ $\boxed{R}\rangle$

Continuation Grammar		
$\boxed{S}\rangle$	--→	$\natural$
$\boxed{L}\rangle$	--→	= R $\boxed{S}\rangle$
		$\boxed{R}\rangle$
$\boxed{R}\rangle$	--→	$\boxed{S}\rangle$
		$\boxed{L}\rangle$

Table 2.2: Grammar with continuations

The second aspect is not very hard either (as is illustrated by the right part of Table 2.2): The continuation grammar can be directly read off the given grammar. This is entailed by the following observation: consider a production

$A \rightarrow u_1 B u_2$

and a derivation $S \stackrel{*}{\Rightarrow} vAw \Rightarrow vu_1Bu_2w$. This entails the relationship $\mathcal{L}(\boxed{B}\rangle) \supseteq \mathcal{L}(u_2\ \boxed{A}\rangle)$, which is equivalent to the production

$\boxed{B}\rangle$ --→ $u_2\ \boxed{A}\rangle$

We use a different derivation symbol '--→' for the description of the continuation grammar in order not to confuse the two aspects of the continuation symbols: (a) they act as null nonterminals in regular derivations; (b) they act as assertions representing continuation languages. When computing these continuation languages, we have to treat the $\boxed{A}\rangle$ as "normal" nonterminals defined by the continuation grammar. To symbolize this change of view, we use the symbol '$\stackrel{*}{\dashrightarrow}$' for such derivations.

To obtain the continuation grammar we essentially have to scan the extended grammar for all occurrences of the nonterminals $A \in \mathcal{N}$ in order to obtain the definitions of the $\boxed{A}$. The following principles then determine the continuation grammar:

1. If $\mathcal{G}$ contains the rule $B \to v\, A\, w\, \boxed{B}$, then the continuation grammar contains the rule $\boxed{A} \dashrightarrow w\, \boxed{B}$.
2. Action symbols $\textcircled{i}$ are ignored. (They are null nonterminals and thus do not contribute any visible symbols.)
3. Trivial rules of the kind $\boxed{A} \dashrightarrow \boxed{A}$ are eliminated. (They do not contribute to the continuation language.)
4. For the axiom S the rule $\boxed{S} \dashrightarrow \sharp$ is added. (When we address $LR(k)$ grammars, we have to add k such 'end-of-input' symbols, i.e. $\boxed{S} \dashrightarrow \sharp^k$, in order to avoid undefined lookaheads.[6])

The right half of Table 2.2 shows the continuation grammar for our running example. In principle, this grammar could be further simplified (using the transformations to be presented in a moment). But in general we will only need to extract very little information from it. Therefore such a simplification is in general not worth the effort.

The following remarks should further elucidate our use of these continuation symbols:

- They mainly serve the purpose of simplifying and unifying some of the later considerations: In connection with $LR(k)$ parsing we will encounter references to "the first k elements of the string s generated by α". In these situations it will sometimes require continuation symbols for α to generate a sufficiently long string at all.
- The occurrence of a continuation symbol in some string, e.g. $u\, \boxed{A}\, v$, is an *assertion* meaning "v (or an extension vw of it) is in the continuation language $\mathcal{L}(\boxed{A})$". Consequently, a string like $u\, \boxed{A}\, v\, \boxed{B}$ can usually be shortened to $u\, v\, \boxed{B}$, since the information conveyed by $\boxed{A}$ is a weaker assertion than that provided by $\boxed{B}$.

Since the continuation symbols $\boxed{A}$ are uniquely determined by the action symbols $\textcircled{i}$, we will usually omit them unless they are explicitly needed.

2.5 Equivalence-Preserving Transformations on Grammars

As mentioned in the introduction, the essence of our approach lies in suitable *transformations of the grammars*. These transformations are all composed of a few very elementary rules that we present in the sequel. (The fact that these rules are so trivial is the clue to the simplicity, comprehensibility and flexibility of our approach.)

Definition 2.5.1 *(Equivalence of grammars) We call two grammars $\mathcal{G}_1$ and $\mathcal{G}_2$* ***equivalent*** *if they generate the same language, i.e. $\mathcal{L}_t(\mathcal{G}_1) = \mathcal{L}_t(\mathcal{G}_2)$.* $\square$

Note that this definition implies that the two grammars have the same set $\mathcal{T}$ of terminal symbols. But the nonterminals and the productions may be different.

In the course of the paper we will also encounter a generalized notion of "equivalence" (which actually isn't an equivalence relation and therefore needs another name):

[6] This has already been observed by Knuth in his original paper [18].

Definition 2.5.2 *(Quasi-equivalence) We say that the grammar $\mathcal{G}'$* ***models*** *the grammar $\mathcal{G}$, if $\mathcal{L}_t(\mathcal{G}') = \mathcal{L}(\mathcal{G})$; that is, the sentences of $\mathcal{G}'$ are the sentential forms of $\mathcal{G}$. We sometimes also say that the two grammars are* ***quasi-equivalent****.*

Now we introduce a few elementary operations that produce equivalent grammars.

Definition 2.5.3 *(Unfolding, folding) Let a grammar $\mathcal{G}$ be given which contains two productions of the form given below (where u, v or w may be empty). Then* ***unfolding*** *of B yields a new production (and thus a new grammar $\mathcal{G}'$)*

$$\boxed{\begin{array}{lcl} A & \to & uBw \\ B & \to & v \end{array}} \quad \rightsquigarrow \quad \boxed{\begin{array}{lcl} A & \to & uvw \\ B & \to & v \end{array}} \qquad \textit{(unfolding)}$$

If the production $(A \to uBw)$ is kept, then the new grammar also retains the same sentential forms. The converse transformation is called ***folding****.*[7] □

Definition 2.5.4 *(Deletion, addition) Let $\mathcal{G}$ be a grammar. We call a nonterminal B* nonreachable *if there is no derivation $S \stackrel{*}{\Rightarrow} u$ such that u contains the symbol B. If B is nonreachable, we may* ***delete*** *any production of the kind $B \to v$. Conversely, we may* ***add*** *arbitrary productions for nonreachable nonterminals.*

Of course, we can also add or delete null nonterminals*, that is, nonterminals that only derive the empty string.* □

Definition 2.5.5 *(Left factoring) Let a grammar $\mathcal{G}$ be given that contains one or more productions of the form shown below (where some of the w_i may be empty). Then* ***left factoring****, short* ***factoring****, extracts the common part v into a new production using a* new *nonterminal Z.*

$$\boxed{\begin{array}{lcl} A & \to & vw_1 \\ & \vdots & \\ A & \to & vw_n \end{array}} \quad \rightsquigarrow \quad \boxed{\begin{array}{lcl} A & \to & vZ \\ Z & \to & w_1 \\ & \vdots & \\ Z & \to & w_n \end{array}} \qquad \textit{(left factoring)}$$

□

For top-down parsers the presence of *left-recursive* productions is disastruous. We can eliminate a left-recursive production by introducing a new nonterminal with suitable production rules. Alternatively, we can also use the Kleene star. The traditional form of this transformation is the following:

$$\begin{array}{lcl} A & \to & A\,u \\ A & \to & v \end{array} \quad \rightsquigarrow \quad A \to v\,u^* \quad \rightsquigarrow \quad \begin{array}{lcl} A & \to & v\,Z \\ Z & \to & u\,Z \\ Z & \to & \varepsilon \end{array}$$

However, in connection with $LR(k)$ parsing the generated ε-production may cause problems. Therefore we use a slightly more complex form of left-recursion elimination. This extended form considers not only the definition of the nonterminal A, but also all its application points $B \to w_1 A w_2$. Let us take the Kleene star version of the above transformation and unfold it. This leads to $B \to w_1\, v u^*\, w_2$. If we now reduce this Kleene star to normal productions, we obtain the following rule.

[7] One only has to prohibit the pathological situation of folding the right-hand side of a production with itself. That is, $B \to u$ must not be converted into $B \to B$.

Definition 2.5.6 *(Left-recursion elimination) We can replace left-recursive symbols at their application points by the following rule:*

$$\boxed{\begin{array}{lcl} B & \to & w_1\,A\,w_2 \\ A & \to & A\,u \\ A & \to & v \end{array} \quad \rightsquigarrow \quad \begin{array}{lcl} B & \to & w_1\,v\,Z \\ Z & \to & w_2 \\ Z & \to & u\,Z \end{array}}$$

(left-recursion removal)

This way, the left-recursion of A is replaced by the right-recursion of Z. $\square$

The decisive advantage of this form is that it does *not* generate ε-productions. Moreover, as will be seen later on, this retains the $LR(k)$ property also in those cases, where there is a conflict between w_2 and u. (In these cases the simple variant of left-recursion removal would fail.) The price to be paid is that we need a new Z for each *application* of A.

Sometimes we want to keep the nonterminal A (for example, because we need all sentential forms, not just the sentences). Then we can use the following variant of the transformation:

$$\begin{array}{lcl} B & \to & w_1\,A\,w_2 \\ A & \to & A\,u \\ A & \to & v \end{array} \quad \rightsquigarrow \quad \begin{array}{lcl} B & \to & w_1\,A\,Z \\ A & \to & v \\ Z & \to & w_2 \\ Z & \to & u\,Z \end{array}$$

However, this variant can only be used, when *all* occurrences of A are transformed simultaneously (since the productions of A are actually changed).

The correctness of our whole approach is based on the following obvious property:

Lemma 2.5.1 *The above operations — viz. unfolding, folding, factoring, addition, deletion and left-recursion elimination — generate equivalent grammars: $\mathcal{L}_t(\mathcal{G}') = \mathcal{L}_t(\mathcal{G})$.*

The addition of *rule numbers* as decribed in Section 2.3 becomes particularly important in connection with these transformations. This shows very clearly e.g. in the rule for recursion removal:

$$\begin{array}{lcll} B & \to & w_1\,A\,w_2 & ① \\ A & \to & A\;u & ② \\ A & \to & v & ③ \end{array} \quad \rightsquigarrow \quad \begin{array}{lcll} B & \to & w_1\,v & ③\,Z \\ Z & \to & w_2 & ① \\ Z & \to & u & ②\,Z \end{array}$$

Without the rule numbers the transformed grammar would lead to a completely different kind of parse tree; but the rule numbers – which represent tree constructors – are transformed accordingly with the rest of the productions such that the original tree productions are kept in place. (The string for the extended grammar $\mathcal{G}'$ remains unaffected by the transformations, and this string is the postfix representation of the original tree.)

Lemma 2.5.2 *Since we only work with number-augmented grammars $\mathcal{G}$, the above transformations also retain the generated trees: $\mathcal{T}(\mathcal{G}') = \mathcal{T}(\mathcal{G})$.*

A similar observation holds for the *continuation symbols* $\boxed{A\rangle}$. According to the principles laid out in Section 2.4 they are integrated into all transformations without problems. For example, suppose that in the left-recursion removal all right-hand sides end in continuations. Then we obtain:

$$\begin{array}{lcl} B & \to & w_1\,A\,w_2\,①\,\boxed{B\rangle} \\ A & \to & A\,u\,②\,\boxed{A\rangle} \\ A & \to & v\,③\,\boxed{A\rangle} \end{array} \quad \rightsquigarrow \quad \begin{array}{lcl} B & \to & w_1\,v\;③\;Z \\ Z & \to & w_2\,①\,\boxed{B\rangle} \\ Z & \to & u\;②\;Z \end{array}$$

Note that the continuations after ② and ③ are no longer needed, because they are now followed by a true string, which conveys more information than the assertion $\boxed{A}$.

This example also illustrates why it is usually the easiest way of proceeding to first apply all transformations to a grammar without continuations and afterwards add the (uniquely determined) continuations to all those productions that end in action symbols.

Part I

Towards $LR(k)$ Grammars

Chapter 3

Normal Forms for Grammars

The main idea of our approach is to transform a given grammar into an equivalent form that is, however, better suited for parsing. Incidentally, the forms into which we want to bring our grammars are very close to well-known normal forms: the *Chomsky normal form* and the *Greibach normal form.*

We split the overall transformation into three consecutive subtransformations, which bring the grammar closer and closer to the intended final form. All transformations are solely based on the rules from Section 2.5, thus immediately guaranteeing the correctness of the overall transformation process.

3.1 First Normal Form: Bounded Production Length

The first transformation is a trivial customization; it is inspired by the well-known *Chomsky normal form.*[1]

The idea. LR-style parsing owes its speed to some extent to the fact that it utilizes an induced *regular grammar.* Therefore, our first goal is to bring our grammar a bit closer to this form. In order to generate the envisaged normal form, we introduce auxiliary nonterminals and production rules so that we finally obtain a grammar in which the right-hand sides of all productions have at most length 2. We also consider the variant, where continuation symbols are admitted.

Definition 3.1.1 *(First normal form / Chomsky normal form*[2]*) A grammar $\mathcal{G}$ is said to be in* ***first normal form****, short:* ***1NF****, if every production is of one of the three forms*

$$\begin{array}{lcl} X \to Y\,Z & \text{or} & \\ X \to Y\,\textcircled{i} & \text{or} & \text{respectively} \\ X \to \textcircled{i} & & \end{array} \qquad \begin{array}{lcl} X \to Y\,Z & \text{or} & \\ X \to Y\,\textcircled{i}\,\boxed{A\rangle} & \text{or} & \\ X \to \textcircled{i}\,\boxed{A\rangle} & & \end{array}$$

with $X \in \mathcal{N} \cup \mathcal{Z}$, $Z \in \mathcal{Z}$, $Y \in \mathcal{N} \cup \mathcal{T}$. Note that we constrain the symbols Z on the second position of the right-hand sides to a special set $\mathcal{Z}$ of newly introduced auxiliary nonterminals. □

The following construction transforms a given grammar into its 1NF. We formulate it for the variant without continuation symbols.

[1] This normal form is also the basis of the Cocke-Younger-Kasami algorithm [16, 40], which — like *LR*-style parsing — is a bottom-up parser.

[2] Actually, this is not quite the Chomsky normal form, because there we had to convert productions $A \to bZ_i$ with terminal $b \in \mathcal{T}$ into $A \to BZ_i$ and $B \to b$. This, however, would not be suited for our later design.

Construction 3.1.1 *(First normal form) Apply the following process as long as possible: For any production*

$X \rightarrow Y_1 \cdots Y_n \quad (n \geq 3)$

introduce a new nonterminal Z_i *and transform the production into the two productions*

$X \rightarrow Y_1\ Z_i \qquad Z_i \rightarrow Y_2 \cdots Y_n$.

For the start symbol S *we introduce a special "start" production*

$Z_0 \rightarrow S$ ⓪

The action ⓪ *indicates the successful parse of the whole string.* □

For our running example (see Table 2.1), Construction 3.1.1 leads to the following grammar.

			Z_0	→	S ⓪			
S	→	L Z_1	Z_1	→	$= Z_2$	Z_2	→	R ①
		R ②						
L	→	$* Z_3$	Z_3	→	R ③			
		i ④						
R	→	L ⑤						

For reasons of easier readability in some of the later stages, we rearrange the layout of the grammar a bit, leading to the version in Table 3.1, which will serve as our reference point in the following.

Grammar $\mathcal{G}_1$		
$Z_0 \rightarrow$ S ⓪	$Z_1 \rightarrow = Z_2$ $Z_2 \rightarrow$ R ①	$Z_3 \rightarrow$ R ③
S → L Z_1 R ②	L → $* Z_3$ i ④	R → L ⑤

Table 3.1: First normal form (rearranged layout)

Theorem 3.1.1 *(Correctness of the 1NF construction) Construction 3.1.1 transforms a given grammar* $\mathcal{G} = (\mathcal{T}, \mathcal{N}, S, \mathcal{P})$ *into an equivalent grammar* $\mathcal{G}' = (\mathcal{T}, \mathcal{N} \cup \mathcal{Z}, Z_0, \mathcal{P}')$*, which is in 1NF.*

Proof: Trivial: we only use equivalence-preserving transformations. Moreover, the construction obviously terminates, since the right-hand sides become stricly shorter. Since every production in $\mathcal{G}$ ends with a symbol ⓘ, the 1NF form follows immediately. □

Discussion. There is an obvious relationship between the size of the given grammar and the number of new nonterminals required for the 1NF. Here, the *size* of the grammar is the sum of the lengths of all its right-hand sides (not counting the action symbols ⓘ).

Lemma 3.1.1 *The 1NF construction introduces auxiliary nonterminals* $Z_0, \ldots, Z_{n-k}$*, where* n *is the* size *of the grammar* $\mathcal{G}$ *and* k *is the number of the productions in* $\mathcal{G}$ *(= the number of actions* ⓘ*).*

This observation is interesting, because it is shown in [5] that for an $LL(1)$ grammar this size coincides with the number of states of the corresponding $LALR(1)$ grammar (which is $n + 1$). This will turn out to be a much deeper connection in Section 7.1.

3.2 Second Normal Form: Leading Terminals

LR-style parsing also obtains its efficiency to a great extent from the fact that decisions between conflicting reductions can be made by only inspecting the first input symbol(s). In *LL* parsers the same principle is embodied in the so-called first (and follow) sets. This motivates our second customization of the grammar (the crucial step), which is inspired by the well-known *Greibach normal form.*

The idea. All nonterminals which start a rule have to be unfolded because we ultimately want to arrive at a grammar where all rules start with a terminal symbol t or an action symbol $ⓘ$. Again, we have to distinguish the variants with and without continuation symbols. So we strive for productions of the kind

$$\begin{array}{lcl} Z & \to & t \cdots Z_i \cdots ⓙ \cdots \\ Z & \to & ⓘ \cdots Z_i \cdots ⓙ \cdots \end{array} \qquad \text{or also} \qquad \begin{array}{lcl} Z & \to & t \cdots Z_i \cdots ⓙ \cdots ⓚ \boxed{A} \\ Z & \to & ⓘ \cdots Z_i \cdots ⓙ \cdots ⓚ \boxed{A} \end{array}$$

where the $\cdots$ indicate arbitrary mixtures of Z_i and/or $ⓘ$. This is formalized in the following definition.

Definition 3.2.1 *(Second normal form / Greibach normal form[3]) A grammar $\mathcal{G}$ is said to be in* ***second normal form****, short:* ***2NF****, if every production is of one of the two forms*

$$\begin{array}{lcll} Z & \to & t\ z & |z| \geq 1 \\ Z & \to & ⓘ\ z & |z| \geq 0 \end{array}$$

with $Z \in \mathcal{Z}$, $t \in \mathcal{T}$ (or $t \in \mathcal{T} \cup \mathcal{N}$), $z \in (\mathcal{Z} \cup \mathcal{A})^*$. If we also admit continuation symbols, the strings z that end in an action symbol $ⓘ$ are followed by a continuation symbol $\boxed{A}$. □

Note that as a byproduct the so-called "first sets" used in many of the traditional approaches are now immediately evident from the grammar.

Example: In our grammar $\mathcal{G}_1$ (see Table 3.1), we start by unfolding all applications of R. (In principle, the order in which the nonterminals are unfolded does not matter, but for reasons of efficiency some orders are better than others.) Note that R can be eliminated, since it is no longer needed.

$$\begin{array}{lcl} Z_0 & \to & \mathrm{S}\ ⓪ \end{array} \qquad \begin{array}{lcl} Z_1 & \to & = Z_2 \\ Z_2 & \to & \mathrm{L}\ ⑤\ ① \end{array} \qquad \begin{array}{lcl} Z_3 & \to & \mathrm{L}\ ⑤\ ③ \end{array}$$

$$\begin{array}{lcl} \mathrm{S} & \to & \mathrm{L}\ Z_1 \\ & & \mathrm{L}\ ⑤\ ② \end{array} \qquad \begin{array}{lcl} \mathrm{L} & \to & * Z_3 \\ & & \mathrm{i}\ ④ \end{array}$$

At this point we should apply left factoring to the two productions for the nonterminal S. This leads to the introduction of an auxiliary nonterminal Z_4:

$$\begin{array}{lcl} \mathrm{S} & \to & \mathrm{L}\ Z_4 \end{array} \qquad \begin{array}{lcl} Z_4 & \to & Z_1 \\ & & ⑤\ ② \end{array}$$

Since we do not want rules that have Z_i's as their leftmost symbol, we unfold Z_1 immediately. (Z_1 could now be eliminated, because it is no longer reachable).

$$\begin{array}{lcl} \mathrm{S} & \to & \mathrm{L}\ Z_4 \end{array} \qquad \begin{array}{lcl} Z_4 & \to & = Z_2 \\ & & ⑤\ ② \end{array}$$

We call this transformation ***full left factoring***; that is, left factoring with subsequent unfolding of the leading Z_i (if such Z_i are generated). The analogous principle is used for ***full left-recursion removal.***

[3] Actually only the first kind of production is admitted in the true Greibach normal form, but our — unavoidable — generalization is close enough.

Now we have to unfold the next nonterminals, that is, L and S. Since this does not entail a need for further left factoring, we are done (in this example). The result is shown in Table 3.2.

Grammar $\mathcal{G}_2$		
$Z_0 \to$ * Z_3 Z_4 ⓪	$Z_2 \to$ * Z_3 ⑤ ①	$Z_3 \to$ * Z_3 ⑤ ③
i ④ Z_4 ⓪	i ④ ⑤ ①	i ④ ⑤ ③
		$Z_4 \to$ = Z_2
		⑤ ②

Table 3.2: Second normal form

Including Sentential Forms. The extension to sentential forms can be achieved by a minor adaption of our 2NF construction: after unfolding a nonterminal we simply *keep* the pertinent production instead of eliminating it.[4] With this modification our running example from Table 3.2 obtains the form presented in Table 3.3.

Grammar $\mathcal{G}_{2a}$		
$Z_0 \to$ S ⓪	$Z_2 \to$ R ①	$Z_3 \to$ R ③
R ② ⓪	L ⑤ ①	L ⑤ ③
L Z_4 ⓪	* Z_3 ⑤ ①	* Z_3 ⑤ ③
* Z_3 Z_4 ⓪	i ④ ⑤ ①	i ④ ⑤ ③
i ④ Z_4 ⓪		$Z_4 \to$ = Z_2
		⑤ ②

Table 3.3: Second normal form with sentential forms

In this grammar the former nonterminals S, L and R are treated like terminals (since there no longer exist productions for them). Hence, the language generated by this grammar comprises all sentential forms of the language of our original grammar.

As illustrated by the above example, we apply the following construction. However, by contrast to the example, we do ***not*** apply left factoring yet.

Construction 3.2.1 *(Second normal form) Let $\mathcal{G}$ be a grammar in 1NF. Then do the following as often as possible:*

Pick some nonterminal $A \in \mathcal{N}$ (this means that A occurs as leftmost symbol in at least one production).

1. *If A is left-recursive, apply "full left-recursion elimination".*
2. *Unfold **all** occurrences of A in the grammar. (This is feasible due to the absence of left-recursive productions.)*
3. *Eliminate the productions for A from the grammar (since it has become unreachable).*

We can employ this construction in two different variants, depending on whether we want only terminal sentences or all sentential forms:

[4] These additional productions correspond to the so-called GOTO table in traditional LR-style parsers, whereas our original productions correspond to the ACTION table (see Section 7.1).

Variant a: *Eliminate the productions that contain occurrences of A.*

Variant b: *Keep the productions.*

□

Theorem 3.2.1 *(Correctness of the 2NF construction) Variant a of Construction 3.2.1 transforms a given grammar $\mathcal{G} = (\mathcal{T}, \mathcal{N} \cup \mathcal{Z}, Z_0, \mathcal{P})$, which is in 1NF, into an equivalent grammar $\mathcal{G}' = (\mathcal{T}, \mathcal{Z}, Z_0, \mathcal{P}')$, which is in 2NF.*

Proof: The proof requires the verification of three facts: equivalence, the 2NF property, and termination.

Equivalence is trivial, since we only apply equivalence-preserving transformations.

The 2NF property requires two observations: (1) In each step, the complete unfolding lets the pertinent nonterminal A vanish from the grammar. So, ultimately, there will be only nonterminals $Z \in \mathcal{Z}$ left. (2) The rules invariantly have the required forms: initially, this is true, because $\mathcal{G}$ is in 1NF (and 1NF is a special case of 2NF). The unfolding obviously retains the patterns.

Termination is easily seen: In each major step one nonterminal $A \in \mathcal{N}$ vanishes. And each nonterminal A has only finitely many occurrences that need to be unfolded or considered in left-recursion elimination. □

Corollary 3.2.1 *Variant b of Construction 3.2.1 yields a grammar $\mathcal{G}' = (\mathcal{T} \cup \mathcal{N}, \mathcal{Z}, Z_0, \mathcal{P}')$ that models $\mathcal{G}$ (in the sense of Def. 2.5.2), i.e. it generates all sentential forms of $\mathcal{G}$: $\mathcal{L}(\mathcal{G}) = \mathcal{L}_t(\mathcal{G}')$.*

Remark: Our auxiliary nonterminals $Z_i \in \mathcal{Z}$ essentially correspond to the $LR(0)$ items from traditional approaches. (This will be discussed in greater depth in Section 5.2.)

3.3 Third Normal Form: Unique Shifting

A major deficiency of the above 2NF construction is the omission of left factoring. Left factoring is *not* mandatory for our normal forms. Therefore the correctness theorem works out nicely. But it is crucial for efficiency both of the construction itself and the resulting parser. As a matter of fact, it will turn out to be the key issue for establishing the $LR(k)$ property. However, in certain pathological cases the left factoring process may run into termination problems. Therefore it is technically simpler to separate the presentation of this aspect from the rest of the normalization process. But in practice it should be amalgamated with the other transformations in order to increase the efficiency of the overall process.

In our running example we have used left factorings (plus subsequent unfoldings) such as

$$\begin{array}{lcl} \mathrm{S} & \to & \mathrm{L}\ Z_1 \\ & & \mathrm{L}\ ⑤\ ② \end{array} \quad \rightsquigarrow \quad \begin{array}{lcl} \mathrm{S} & \to & \mathrm{L}\ Z_4 \\ Z_4 & \to & Z_1 \\ & & ⑤\ ② \end{array} \quad \rightsquigarrow \quad \begin{array}{lcl} \mathrm{S} & \to & \mathrm{L}\ Z_4 \\ Z_4 & \to & =\ Z_2 \\ & & ⑤\ ② \end{array}$$

before eliminating the nonterminal L through unfolding. This is obviously reasonable, since it avoids the duplication of all L-productions in the right-hand sides of S, which would otherwise lead to

$$\begin{array}{lcl} \mathrm{S} & \to & *\ Z_3\ Z_1 \\ & & \mathrm{i}\ ④\ Z_1 \\ & & *\ Z_3\ ⑤\ ② \\ & & \mathrm{i}\ ④\ ⑤\ ② \end{array}$$

Moreover, since left factoring is an equivalence-preserving transformation, there seems to be no reason for not integrating it into the process from the very beginning.

Yet, there is one subtle problem: *The process may not terminate.* Even though this can only happen in a few pathological cases, it is nonetheless unpleasant. However, in order not to interrupt the derivation process at this point, we defer the discussion of this termination problem (to Section 6) and proceed with the normalizing transformations.

Definition 3.3.1 *(Third normal form) A grammar $\mathcal{G}$ is said to be in* ***third normal form****, short:* ***3NF****, if it is in 2NF and there are no two productions*

$$\begin{array}{lll} Z & \rightarrow & x\ u \\ & & x\ v \end{array}$$

the right-hand sides of which start with the same symbol.

Except for the aforementioned rare termination problem this normal form can obviously be obtained by the following principle:

- Apply left factoring wherever possible.

However, we can improve the efficiency by delaying the left factorings as long as possible. This may be called "***lazy left factoring***".

Construction 3.3.1 *(Third normal form) Modify the 2NF construction 3.2.1 by adding the following transformation before step (1):*

0. Apply full left factoring to all occurrences *of the nonterminal A.*

Finally, when all nonterminals have been unfolded, apply full left factoring to all terminals in all productions. □

Note: if we would always apply left factoring as soon as possible, we could create unnecessarily many additional Z_i symbols. Consider a situation like

$$\begin{array}{lll} Z_i & \rightarrow & \text{i} \ldots \\ & & \text{A} \ldots \\ & & \text{B} \ldots \end{array} \qquad\qquad \begin{array}{lll} \text{A} & \rightarrow & \text{i} \ldots \\ \\ \text{B} & \rightarrow & \text{i} \ldots \end{array}$$

Due to the successive unfolding of A and B we will create two auxiliary Z-symbols instead of the one that suffices, if we factorize all three occurrences of i in one sweep. So we wait with the left factoring as long as possible. However, we must not forget to do it at the latest before unfolding a nonterminal $A \in \mathcal{N}$ in the 2NF construction (as has been demonstrated for the nonterminal L in our running example).

Theorem 3.3.1 *(Correctness of the 3NF construction) For any given grammar $\mathcal{G}$, Construction 3.3.1 yields an equivalent grammar $\mathcal{G}'$, which is in 3NF – provided that the construction terminates.*

Proof. Follows directly from the correctness of the 2NF construction and the fact that left factoring is an equivalence-preserving transformation. □

Chapter 4

Main Theorem: 3NF + $LL(k)$ = $LR(k)$

So far our derivation has led to a construction that transforms a given grammar $\mathcal{G}$ into an *equivalent* grammar $\mathcal{G}'$. Moreover, we have argued on intuitive grounds that this transformed grammar is better suited for parsing purposes, because it has the $LR(k)$ property. Now we have to make this informal claim and argument precise. This is done in our main theorem.

Theorem 4.0.2 (Main theorem) *Consider a grammar $\mathcal{G}$ and its transformed 3NF version $\mathcal{G}'$. The original grammar $\mathcal{G}$ is $LR(k)$ iff the transformed grammar $\mathcal{G}'$ is $LL(k)$.*

The proof of this theorem also sheds light on the close relationship between the $LR(k)$ and $LL(k)$ property. Recall that the only difference between 2NF and 3NF is that the former does not yet include left factoring. It turns out that this is also the characterisitc difference between LL- and LR-style parsers.

Corollary 4.0.1 *Consider a grammar $\mathcal{G}$ without left recursion and its transformed 2NF version $\mathcal{G}'$. If $\mathcal{G}'$ is "conflict-free" (in the sense of Def. 4.0.5 below), then $\mathcal{G}$ is $LL(k)$.*

The remainder of this section is devoted to the proof of Theorem 4.0.2. Yet, there is an unfortunate difficulty: we can find several different definitions of "LR parser" in the literature.[1] This forces us to first choose one of these definitions, before we can prove any equivalence at all. In such a situation it is advisable to employ standard textbooks as the point of reference. Unfortunately, there still is a choice: one class of books treats the problem from the point of view of practical compiler construction; the main reference here is, of course, the famous "dragon book" [1] (to which most other books refer anyhow, e.g. [2, 39]). The other class takes the viewpoint of formal-language theory [19, 29, 32]; here the choice is not so obvious, but we basically employ [29]. Knuth in his original paper [18] actually addresses both variants.

There is a second problem that we have to take into consideration: as was mentioned above, our construction does – in its current form – not necessarily terminate. In order not to obfuscate the following discussion we ignore this problem for the moment. (The formulation " ... its transformed 3NF version $\mathcal{G}'$ " in the above theorem entails the successful completion of the construction.)

[1] It is telling that DeRemer and Pennello [8] point out in connection with $LALR$ parsing that several papers actually define what they call "NQLALR(1)", that is, "not quite LALR(1)". It also comes as a surprise that some of the books and papers in the literature do not make any attempt to establish the equivalence of their definitions.

Before we go into the details of the proof we repeat the outcome of the normalization process for our running example. Table 3.3 already contains the essence of the 3NF – albeit without the continuation symbols. Since we have to refer to them in the course of the proof, it is now time to explicitly show them, which is done in Table 4.1.

Grammar $\mathcal{G}_3$		
$Z_0 \rightarrow$ S ⓪ $\boxed{S}$ R ② ⓪ $\boxed{S}$ L Z_4 ⓪ $\boxed{S}$ * Z_3 Z_4 ⓪ $\boxed{S}$ i ④ Z_4 ⓪ $\boxed{S}$	$Z_2 \rightarrow$ R ① $\boxed{S}$ L ⑤ ① $\boxed{S}$ * Z_3 ⑤ ① $\boxed{S}$ i ④ ⑤ ① $\boxed{S}$	$Z_3 \rightarrow$ R ③ $\boxed{L}$ L ⑤ ③ $\boxed{L}$ * Z_3 ⑤ ③ $\boxed{L}$ i ④ ⑤ ③ $\boxed{L}$ $Z_4 \rightarrow$ $= Z_2$ ⑤ ② $\boxed{S}$
$\boxed{S}$ --> ↯	$\boxed{L}$ --> Z_4 $\boxed{S}$ $\boxed{S}$	

Table 4.1: Third normal form with continuations

As can be seen here, the information $\boxed{S}$ --> ↯ is mandatory for disambiguating the two productions for Z_4.

Proof of the Main Theorem

Remark: This is the point, where the straightforward simplicity of our approach ends, because now we have to link it to the traditional notions and notations.

Most books and papers follow – in more or less adapted notations – the original definitions given by Knuth in his seminal paper [18] (see e.g. [19, 29, 39]). We summarize these concepts to the extent needed for our proof. The focus of our attention are the points, where reductions can take place, that is, the occurrences of right-hand sides of productions $(A \rightarrow \alpha)$ in the given strings.

Definition 4.0.2 (Handle, viable prefix) *Consider a rightmost derivation that ends with an application of the production $(A \rightarrow \alpha)$:*

$$S \quad \stackrel{*}{\Rightarrow} \quad v\,A\,w \quad \Rightarrow \quad v\,\alpha\,w \qquad \text{with } w \in \mathcal{T}^*$$

Then α is called the ***handle*** *of the string $v\,\alpha\,w$. Moreover, any prefix of $v\,\alpha$ is called a* ***viable prefix****.* □

Remark 1: Since *LR* parsing is only concerned with *rightmost* derivations, we presume this property for all derivations in the remainder of this section without mentioning it explictily every time anew.

Remark 2: The introduction of the action symbols ⓘ in our approach yield a nice characterization of the handles: Any handle α ends with an action symbol ⓘ. (Unfortunately this is only a conceptual characterization, since the ⓘ are "invisible", i.e. not present in the input string.)

The quest for efficient parsing suffers from a fundamental problem: at any given point in time we have in general only seen an initial fragment of the given string; that is, we have to deal with a *limited horizon*. Therefore we are interested in languages, where the next action – i.e. the handle – can always be uniquely determined in spite of this limited horizon. The most important class

of this kind are the $LR(k)$ grammars: here the horizon extends k symbols beyond the handle.[2] Similar to the functions for left contexts in [2] or [19] we can introduce a corresponding function that makes this horizon property more explicit:

Definition 4.0.3 (Horizon) *We define the* ***k-horizon*** *of a production $(A \to \alpha)$ by the following function:*

$$horizon_k(A \to \alpha) = \{\ v\,\alpha\,u \mid S \stackrel{*}{\Rightarrow} v\,A\,u\,w,\ |u| = k\ \}$$

□

Let the "starts" relation $s_1 \sqsubseteq s_2$ express the fact that the string s_1 is a prefix of the string s_2, that is, $s_2 = s_1 u$ for some suitable (possibly empty) u. Extend this relation to sets of strings by letting $A \sqsubseteq B$ express the fact that some string in A is a prefix of some string in B, that is, $\exists\ a \in A, b \in B :\ a \sqsubseteq b$. Then the $LR(k)$ property can be expressed as

Definition 4.0.4 ($LR(k)$ grammar) *A grammar $\mathcal{G}$ is an* ***$LR(k)$ grammar****, if the following condition holds:*

$$horizon_k(A \to \alpha) \sqsubseteq horizon_k(B \to \beta) \quad \Rightarrow \quad (A \to \alpha) = (B \to \beta)$$

□

This means: whenever there are two strings with rightmost derivations of the form

$$\begin{array}{lcll} S & \stackrel{*}{\Rightarrow} & v_1\,A\,u\,w_1 \ \Rightarrow\ v_1\,\alpha\,u\,w_1 & (u\,w_1 \in \mathcal{T}^*,\ |u| = k) \\ S & \stackrel{*}{\Rightarrow} & v_2\,B\,w_2 \ \Rightarrow\ v_2\,\beta\,w_2 & (w_2 \in \mathcal{T}^*) \end{array}$$

with different productions $(A \to \alpha) \neq (B \to \beta)$, then

$$v_1\,\alpha\,u \not\sqsubseteq v_2\,\beta\,w_2$$

Put into other words: if we have a string s_1 with a handle α, and if we consider a horizon that extends k symbols beyond the handle, then any other string s_2 with a different handle has to differ from s_1 already within the horizon.

Example: To see the meaning of these definitions we consider one of the simplest possible counterexamples (see Table 4.2).[3]

Grammar C_1			
S	→	a S a	①
		a	②

Table 4.2: A simple non-$LR(k)$ grammar

Consider two different strings and their rightmost derivations:

$$\begin{array}{lcll} S & \stackrel{*}{\Rightarrow} & a\,a\,\boxed{S}\,a\,a \ \Rightarrow\ a\,a\,\boxed{a}\,a\,a & (= s_1) \\ S & \stackrel{*}{\Rightarrow} & a\,a\,a\,\boxed{S}\,a\,a\,a \ \Rightarrow\ a\,a\,a\,\boxed{a}\,a\,a\,a & (= s_2) \end{array}$$

[2] In order to avoid problems at the right end of the input strings, we have to provide at least k "end-of-file" symbols '↯' as continuation $\boxed{S}$ of the start symbol (as it was already done by Knuth).

[3] This example is a simplified variant of examples given e.g. in [29, 30]. It is nice, because it provides a counterexample without being ambiguous.

This example shows that the grammar cannot be $LR(1)$. Consider the string s_1. The handle ends after the third symbol, the $LR(1)$-horizon therefore comprises the first four symbols. However, these first four symbols are identical to those of s_2, which has a different handle. (In this example it happens to be the same production, but at a differrent point.) Actually, the example even shows that the grammar is not $LR(2)$. And by using arbitrarily long strings it can be easily seen that it actually is not $LR(k)$ for any k.

Impact on our construction. The transformation of the above grammar generates the version in Table 4.3. Here we have a so-called shift/reduce conflict for Z_3, because $\boxed{S}$ contains the

Grammar C_2		
$Z_0 \rightarrow$ S ⓪ $\boxed{S}$ a Z_3 ⓪ $\boxed{S}$	$Z_1 \rightarrow$ S Z_2 a Z_3 Z_2 $Z_2 \rightarrow$ a ① $\boxed{S}$	$Z_3 \rightarrow$ S Z_2 a Z_3 Z_2 ② $\boxed{S}$
$\boxed{S}$ ⇢ ↯ a $\boxed{S}$		

Table 4.3: The transformed non-$LR(k)$ grammar

lookahead symbol a. Hence, the grammar is *not* $LR(1)$. As a matter of fact, the continuation of the shift production and the continuation of the reduce production both generate arbitrarily long sequences '$aaa\ldots$', which demonstrates that the grammar is not $LR(k)$ for any k. □

The notion of "conflict" used in the example has to be made precise. (Recall that '$\overset{*}{\dashrightarrow}$' denotes the derivation relation, where the continuation symbols $\boxed{A}$ actually generate the continuation strings.)

Definition 4.0.5 (Conflict) *Let $\mathcal{G}$ be a grammar in 3NF. A nonterminal Z has a* ***shift/reduce conflict*** *of length k, if it has two productions of the kind $(Z \rightarrow t\, z_1)$ and $(Z \rightarrow ⓘ\, z_2)$ such that $z_1 \overset{*}{\dashrightarrow} uw_1 \ldots$ and $z_2 \overset{*}{\dashrightarrow} tuw_2$ with $|u| = k-1$.*

Z has a ***reduce/reduce conflict*** *of length k, if it has two productions of the kind $(Z \rightarrow ⓘ\, z_1)$ and $(Z \rightarrow ⓙ\, z_2)$ such that $z_1 \overset{*}{\dashrightarrow} uw_1$ and $z_2 \overset{*}{\dashrightarrow} uw_2$ with $|u| = k$.* □

This concept coincides – for the special case of grammars in 3NF – with the classical notion of strong $LL(k)$ grammar (as e.g. formulated in [2] or [19]).

Lemma 4.0.1 *If a 3NF grammar $\mathcal{G}$ has no k-conflicts, it is a strong $LL(k)$ grammar.* □

Now we are ready to prove our main theorem.

Proof of Main Theorem 4.0.2. We want to show that the original grammar $\mathcal{G}$ is $LR(k)$ iff the transformed 3NF grammar $\mathcal{G}'$ is $LL(k)$, i.e. has no k-conflicts.

From the correctness theorems for our normal forms we know that $\mathcal{G}$ and $\mathcal{G}'$ are "quasi-equivalent", that is, $\mathcal{L}(\mathcal{G}) = \mathcal{L}_t(\mathcal{G}')$. Based on this fact we show the two directions of the theorem separately.

1. *$\mathcal{G}'$ is conflict-free $\Rightarrow$ $\mathcal{G}$ is $LR(k)$* .

   This is equivalent to the negated form "*$\mathcal{G}$ is not $LR(k)$ $\Rightarrow$ $\mathcal{G}'$ has k-conflicts*", which we will actually use. If $\mathcal{G}$ is *not* $LR(k)$, then we have two rightmost derivations

$$\begin{array}{lcl} S & \overset{*}{\Rightarrow} & v_1 \boxed{\alpha\, ⓘ} u_1 \; w_1 \\ S & \overset{*}{\Rightarrow} & v_2 \boxed{\beta\, ⓙ} u_2 \; w_2 \end{array} \quad \text{sucht that} \quad \begin{cases} ⓘ \neq ⓙ \\ v_1\alpha u_1 = v_2\beta u_2 \\ |u_1| = k \end{cases} \qquad (*)$$

(Actually, for large β the horizon may also end inside of β; but this does not influence the argument. W.l.o.g. we also assume that v_2 is at least as long as v_1.) Due to the aforementioned equivalence there is a corresponding derivation in $\mathcal{G}'$

$$Z_0 \quad \stackrel{*}{\Rightarrow} \quad v_1 \alpha Z_i z \quad \Rightarrow \quad v_1 \alpha \, ⓘ z' z \quad \stackrel{*}{\Rightarrow} \quad v_1 \alpha \, ⓘ u_1 w_1$$

with a production $(Z_i \to ⓘ z')$ and $z' z \stackrel{*}{\Rightarrow} u_1 w_1$.

Since we are talking about rightmost derivations, ⓘ is the leftmost action symbol. Due to the left factorings that entail the uniqueness property of the 3NF, the derivation of $v_1\alpha$ in $\mathcal{G}'$ is uniquely determined. Hence, Z_i is the first point, where a choice may occur and we have $v_1 = v_2$. We distinguish two cases:

(a) $|\alpha| = |\beta|$.
Then $(*)$ together with $v_1 = v_2$ entails also $\alpha = \beta$ and $u_1 = u_2$. Hence there must be two productions

$$\begin{array}{lll} Z_i & \to & ⓘ \; z' \\ & & ⓙ \; z'' \end{array}$$

such that we obtain

$$\begin{array}{lllll} Z_0 & \stackrel{*}{\Rightarrow} & v_1 \alpha Z_i z & \not\rightarrow & v_1 \alpha \, ⓘ z' z \quad \stackrel{*}{\Rightarrow} \quad v_1 \alpha \, ⓘ u_1 w_1 \\ & & & \not\rightarrow & v_1 \alpha \, ⓙ z'' z \quad \stackrel{*}{\Rightarrow} \quad v_1 \alpha \, ⓘ u_1 w_2 \end{array}$$

This entails $z' \stackrel{*}{:\Rightarrow} u_1 w_1$ and $z'' \stackrel{*}{:\Rightarrow} u_1 w_2$ with $|u_1| = k$. Therefore Z_i has a reduce/reduce conflict.

(b) $|\alpha| \neq |\beta|$.
W.l.o.g. we take $|\alpha| < |\beta|$. By the uniqueness of the 3NF $(*)$ entails $\beta = \alpha t u$ and $u_1 = t u u_2$ with suitable $t \in \mathcal{V}, u \in \mathcal{V}^*$. By the same argument as above we obtain the need for two productions

$$\begin{array}{lll} Z_i & \to & ⓘ \; z' \\ & & \mathrm{t} \; z'' \end{array}$$

such that $z' \stackrel{*}{:\Rightarrow} u_1 w_1$ and $z'' \stackrel{*}{:\Rightarrow} u u_2 w_2$ with $|u_1| = k$. This is a shift/reduce conflict.

2. *$\mathcal{G}$ is LR(k) $\Rightarrow$ $\mathcal{G}'$ is conflict-free.*

This is equivalent to the negated form "*$\mathcal{G}'$ has k-conflicts $\Rightarrow$ $\mathcal{G}$ is not LR(k)* ", which we will actually use. We consider the case of a reduce/reduce conflict:

$$\begin{array}{lllll} Z_i \to & ⓘ z' & \text{with} & z' \stackrel{*}{:\Rightarrow} u w_1, & |u| = k \\ & ⓙ z'' & \text{with} & z'' \stackrel{*}{:\Rightarrow} u w_2 & \end{array}$$

As above, consider two derivations which exhibit this conflict:

$$\begin{array}{lllll} Z_0 & \stackrel{*}{\Rightarrow} & v_1 \alpha Z_i z & \not\rightarrow & v_1 \alpha \, ⓘ z' z \quad \stackrel{*}{\Rightarrow} \quad v_1 \alpha \, ⓘ u w_1 \\ & & & \not\rightarrow & v_1 \alpha \, ⓙ z'' z \quad \stackrel{*}{\Rightarrow} \quad v_1 \alpha \, ⓘ u w_2 \end{array}$$

Without loss of generality we can assume that ⓘ is the leftmost action symbol in the string. (If it were not, we could use the corresponding sentential form, where all action symbols further left have been reduced to their corresponding nonterminals.) Due to the

3NF property, $v_1\alpha$ is uniquely determined. Because of the equivalence of the grammars we also have corresponding derivations in $\mathcal{G}$:

$$\begin{array}{l} S \overset{*}{\Rightarrow} v_1\alpha\,\textcircled{i}\,uw_1 \\ S \overset{*}{\Rightarrow} v_1\alpha\,\textcircled{j}\,uw_2 \end{array} \qquad \text{with} \qquad |u| = k$$

which *violates* the $LR(k)$ condition.

The case of a shift/reduce conflict is shown analogously.

This concludes the proof of our Main Theorem. $\square$

Part II

From Grammars To Parsers

Chapter 5

Emulated Versus Classical *LR* Parsing

Our first observation is that our transformed grammars induce parsing algorithms that *behave like* classical $LR(k)$ parsers in the sense that they make the same decisions and thus have the same expressive power. However, they are *programmed differently*: While the induced parsers of our grammars are realized in a recursive-descent style, the traditional $LR(k)$ parsers are table-driven, based on the paradigm of stack automata. In the following we want to study this relationship more closely and show, how our approach can also be used to produce classical *LR* parsers.

5.1 A Simple (Top-Down) Parsing Scheme

For the following discussion we have to refer to the *parser* that is induced by a grammar. To make these references sufficiently precise we briefly sketch one way of obtaining such parsers. Even though our approach can be used for parsers written (or generated) in any language, we feel that the paradigm of functional programming provides by far the best and most elegant framework for expressing the pertinent concepts. Therefore, we work with a *recursive-descent parser* based on higher-order functions.[1]

The programming technique is based on the philosophy that the grammar already *is* the parser. Basically, every nonterminal becomes a function, and its productions provide the corresponding function definitions.[2] This is illustrated for our standard grammar in Table 5.1. As can be seen here, we merely introduce a few connectors between the symbols in the grammar, but otherwise leave the structure of the grammar completely unchanged. The sequential composition '`A;B`' stands for "*first apply parser* `A` *and then apply parser* `B`", and the choice '`A|B`' for "*apply parser* `A` *or parser* `B`". In addition, we need a lifting operator '!' to convert terminal symbols into parsers. At the end of each production, we add an action $\texttt{action}_i$ which takes care of the tree constructor ⓘ.

Since this is not a paper on functional programming, we defer the actual code of these higher-

[1] That higher-order functions are an elegant means for directly implementing parsers from given grammars has been legendary in the functional-programming community for quite a while. We have actually used this principle successfully in the teaching of parsing concepts to students for many years [23]. Detailed descriptions of such techniques are given e.g. in [13, 14, 12, 22] (where further references can be found as well).

[2] We essentially use the notation of the functional language Opal [24, 10, 11, 26], slightly enriched in order to increase flexibility. But it should be noted that the code would look essentially the same in other functional languages such as Ml, Haskell and Miranda.

```
FUN S L R: Parser
DEF S ==   (L ; "="! ; R ; action_1)          --  S  →  L = R ①
       |   (R ; action_2)                     --  S  →  R ②
DEF L ==   ("*"! ; R ; action_3)              --  L  →  * R ③
       |   ("i"! ; action_4)                  --  L  →  i ④
DEF R ==   (L ; action_5)                     --  R  →  L ⑤
```

Table 5.1: Grammar $\mathcal{G}$ as a functional program

order functions to the Appendix (even though it's less than 20 lines). Suffice it to list the functionalities:

```
FUN ; : Parser × Parser → Parser    -- sequential composition
FUN | : Parser × Parser → Parser    -- alternative parsers (not commutative)
FUN ! : String → Parser             -- lifting of tokens
```

with

```
TYPE Parser == Tree × String → Tree × String
```

It should be mentioned that the equivalence-preserving grammar transformations considered in Section 2.5 are also valid in the functional interpretation:

Lemma 5.1.1 *The operations unfolding, folding, factoring, addition and deletion are equivalence-preserving transformations for the functions associated with the grammar.*

Note: since left-recursive grammars are undefined in the functional interpretation, a similar result cannot hold for them. On the contrary, it is only through left-recursion elimination that the functional interpretation makes any sense.

Even though the above parsing scheme uses a recursive-descent strategy and contains backtracking, it works very well and without any reservations in our framework, since left recursion has been eliminated and backtracking due to "shift/shift" conflicts has been minimized by left factoring.

5.2 "Emulated" *LR* Parsing

When we apply the above parser generation to a 3NF grammar, we obtain what may be called "emulated LR parsing". That is, the parser has $LR(k)$ power, because it makes the same decisions and performs the same actions as a traditional $LR(k)$ parser, but it is programmed in the style of a recursive-descent parser. But it is a relatively straightforward exercise to *derive* from this parser a "classical LR parser".

Consider the induced parser for the 3NF of our running example (see Table 4.1). If we parse the simple string $"*\ \ i = i\ \sharp"$ then we obtain the following process. (Recall that a function composition like $Z_3; Z_4$ applies from left to right, i.e. first Z_3 and then Z_4. By '`a::L`' we denote the prepending of the token `a` to the list `L`. And we write ⓘ instead of $\texttt{action}_\texttt{i}$ to increase

readability.)

$$\begin{aligned}
& Z_0(''* \; i = i \; \natural'') \\
= \;& * :: (Z_3; Z_4; ⓪)(''i = i \; \natural'') \\
= \;& * :: i :: (④ ; ⑤ ; ③ ; Z_4; ⓪)(''= i \; \natural'') \\
= \;& * :: i :: ④ :: ⑤ :: ③ :: (Z_4; ⓪)(''= i \; \natural'') \\
= \;& * :: i :: ④ :: ⑤ :: ③ :: = :: (Z_2; ⓪)(''i \; \natural'') \\
= \;& * :: i :: ④ :: ⑤ :: ③ :: = :: i :: (④ ; ⑤ ; ① ; ⓪)(''\natural'') \\
= \;& * :: i :: ④ :: ⑤ :: ③ :: = :: i :: ④ :: ⑤ :: ① :: ⓪
\end{aligned}$$

As can be seen here, the function compositions of the Z_i and ⑦ act essentially like a stack. This becomes even more evident, if we introduce a more compact notation for representing the above calculation process:

$$[Z_0] \; * \; \begin{bmatrix} ⓪ \\ Z_4 \\ Z_3 \end{bmatrix} \; i \; \begin{bmatrix} ⓪ \\ Z_4 \\ ③ \\ ⑤ \\ ④ \end{bmatrix} \; ④ \; \begin{bmatrix} ⓪ \\ Z_4 \\ ③ \\ ⑤ \end{bmatrix} \; ⑤ \; \begin{bmatrix} ⓪ \\ Z_4 \\ ③ \end{bmatrix} \; ③ \; \begin{bmatrix} ⓪ \\ Z_4 \end{bmatrix} \; = \; \begin{bmatrix} ⓪ \\ Z_2 \end{bmatrix} \; i \; \begin{bmatrix} ⓪ \\ ① \\ ⑤ \\ ④ \end{bmatrix} \; ④ \; \begin{bmatrix} ⓪ \\ ① \\ ⑤ \end{bmatrix} \; ⑤ \; \begin{bmatrix} ⓪ \\ ① \end{bmatrix} \; ① \; [⓪] \; ⓪ \qquad (*)$$

We can give this notation two kinds of meaning:

- We can consider it as a shorthand for describing the above functional computation process. In this interpretation we simply record at every point in the string the (composed) function that is applied at this point to the reststring.
- Or we can consider it as a shorthand for some classical mathematical formula, where $[\alpha]x[\beta]$ means $\alpha \stackrel{*}{\Rightarrow} x\beta$. This view provides a nice calculus for doing correctness calculations.

In any case this representation provides the formal basis, on which we can argue about the dynamic execution of the induced parser.

Definition 5.2.1 (Parsing process) *We call a term such as* $(*)$ *above a* **parsing process***: between any two elements of the string the (composed) parsing function* φ *that is applied at this point to the reststring is recorded in the form* $[\varphi]$*.*

5.3 Towards "True" *LR* Parsing

The transition from emulated to classical *LR* parsing is based on a simple observation, which is hinted at in $(*)$ by the gray coloring.

Lemma 5.3.1 *In a parsing process like* $(*)$ *above only the bottom elements of the composed functions are actually needed. All other functions are redundant.*

Sketch of proof: This is shown relatively easily: as an intermediate step for the better understanding of this effect consider a variant, where we actually apply the reductions ⓘ rather than only recording them. For example, the production $L \to i \; ④$ leads to the following reduction:

$$[Z_0] \; * \; \begin{bmatrix} ⓪ \\ Z_4 \\ Z_3 \end{bmatrix} \; \underbrace{i \; \begin{bmatrix} ⓪ \\ Z_4 \\ ③ \\ ⑤ \\ ④ \end{bmatrix} \; ④}_{} \; \begin{bmatrix} ⓪ \\ Z_4 \\ ③ \\ ⑤ \end{bmatrix} \; \ldots \qquad \rightsquigarrow \qquad [Z_0] \; * \; \begin{bmatrix} ⓪ \\ Z_4 \\ Z_3 \end{bmatrix} \; \underline{L} \; \begin{bmatrix} ⓪ \\ Z_4 \\ ③ \\ ⑤ \end{bmatrix} \; \ldots \qquad (*_1)$$

By inspecting the 3NF in Table 4.1 we can see that the production $Z_3 \to L$ ⑤ ③ also yields the action ⑤ (and even ③) that becomes the next bottom element of the function composition. That is, the information above ④ is *not needed* for deducing that the next bottom element is ⑤.

To continue the example, suppose we have also applied the next reduction ⑤ and proceed to ③. That is, we have the situation:

$$[Z_0] \underbrace{* \begin{bmatrix} ⓪ \\ Z_4 \\ Z_3 \end{bmatrix} R \begin{bmatrix} ⓪ \\ Z_4 \\ ③ \end{bmatrix} ③} \begin{bmatrix} ⓪ \\ Z_4 \end{bmatrix} \ldots \qquad \rightsquigarrow \qquad [Z_0] \underline{L} \begin{bmatrix} ⓪ \\ Z_4 \end{bmatrix} \ldots \qquad (*_2)$$

Again the 3NF grammar provides through its production $Z_0 \to L\, Z_4$ ⓪ all necessary information to deduce the next bottom function.

Of course, it is not necessary to actually perform the reductions as it has been done above for didactic purposes. To see this consider the following stage of the process $(*)$:

$$[Z_0] \; * \begin{bmatrix} \vdots \\ Z_3 \end{bmatrix} i \begin{bmatrix} \vdots \\ ④ \end{bmatrix} ④ \begin{bmatrix} \vdots \\ ⑤ \end{bmatrix} ⑤ \begin{bmatrix} \vdots \\ ③ \end{bmatrix} ③ \begin{bmatrix} \vdots \\ ? \end{bmatrix} = \ldots \qquad (‡)$$

Reduction ③ tells us (see Table 2.1) to go two elements back (not counting ③ itself). In doing so we encounter ⑤, which recursively forces us to go one element back. Analogously the ④ brings us beyond the 'i'. The second element requested by ③ then is the '$*$'. So we end at $[Z_0]$. Since the reduction ③ generates an 'L', the production $Z_0 \to L\, Z_4 \ldots$ in the 3NF yields $[Z_4]$ as the next bottom element. This is formally expressed in the following lemma, which also entails the proof of Lemma 5.3.1. □

Lemma 5.3.2 *Consider a production $A \to \alpha$ ⓘ from the original grammar. The handle α can occur in the parsing process in a form like (where we omit the intermediate functions)*

$$\ldots \begin{bmatrix} \vdots \\ Z_i \end{bmatrix} \; \alpha \; ⓘ \begin{bmatrix} \vdots \\ Z_j \end{bmatrix} \ldots$$

iff the 3NF contains the production $Z_i \to A Z_j \ldots$. Analogously for ⓘ instead of Z_j.

The proof follows directly from the construction of the 3NF (as illustrated by the above examples). □

Note 1: This way of proceeding gives us complete freedom to either do the reductions right away or to delay them until some better suited time — a freedom that we wouldn't like to lose.

Note 2: The recursive backward search can, of course, be shortened by using an appropriate data structure (actually a stack) for "memoization". However, since this paper is not about programming techniques we defer this technical amelioration to the Appendix. *It should be mentioned, however, that this implementation leads to a program that directly mimics the classical stack automata used in the literature.*

Modification of the 3NF Construction. When we do no longer look at the higher Z_i and ⓘ in the stacks, then there is no reason to keep them in the grammar. Hence we could shorten the grammar in Table 4.1 to the form presented in Table 5.2 by cutting the right-hand sides after the first Z_i or ⓘ. Note that such a cutting would be the first time in this paper that we apply a transformation that is *not* equivalence-preserving, but derives its correctness from an according

Grammar $\mathcal{G}_{LR}$											
Z_0	$\rightarrow$	S ⓪					Z_3	$\rightarrow$	R ③		
		R ②		Z_2	$\rightarrow$	R ①			L ⑤		
		L Z_4				L ⑤			* Z_3		
		* Z_3				* Z_3			i ④		
		i ④				i ④	Z_4	$\rightarrow$	$= Z_2$		
									⑤ $\boxed{S}$		

Table 5.2: Traditional LR-style grammar

alteration of the corresponding algorithm. Actually, this grammar describes the language of the so-called *viable prefixes*. However, in our further proceeding this cutting is actually not helpful (except for the above reference to the notion of "viable prefix" found in the literature).

Chapter 6

Termination of the 3NF Construction

We have essentially completed the presentation of our construction and its correctness – except for the unpleasant termination problem. Up to now we have allowed ourselves the comfort of simply ignoring this problem. And there are good reasons, why this neglection could be a reasonable way of proceeding. From a pragmatic viewpoint[1] the nontermination problem may actually not be worth the effort of a special treatment:

- The nontermination does *not* endanger the correctness of our approach, but at most its efficiency. This is a very strong reason.
- There are several possibilities for overcoming the nontermination problem (which we will discuss in a moment). This is a strong reason as well.
- The nontermination happens only in very few and quite pathological situations (as will be illustrated below). But this is a weak reason.
- We note in passing that for an $LL(k)$ grammar the 2NF construction suffices (and this one always terminates). However, this criterium is weak in the context of LR parsing.

6.1 What Is The Problem?

Before we discuss the solutions, we want to see more clearly where the problem actually arises. The following example illustrates (by way of the simplest possible grammar) how a transformation process may run into infinite cycling.

Example: Consider the toy grammar in Table 6.1. In this grammar, the nonterminal A generates an odd number of stars, while the nonterminal B generates an even number of stars.

When we transform this grammar into its 1NF, we arrive at the following intermediate form:

[1] There is also a nice gedanken experiment from a theoretical viewpoint: the following considerations – in particular the possibility for a "lazy grammar transformation" – also allow us to perceive *infinite grammars*. The left factoring transforms a given grammar into a new one, which we can consider to be "larger" in a suitably chosen ordering (induced by unfolding). This enables the classical method of ideal completion etc., which makes the notion of infinite grammars well-defined. On the basis of these grammars all our theorems hold without any reservations.

Grammar L		
S → A ①	A → * B ③	B → * A ④
B ②		i ⑤

Table 6.1: A pathological counterexample

S	→	A ①	A	→	* Z_1	B	→	* Z_2
		B ②						i ⑤
Z_0	→	S ⓪	Z_1	→	B ③	Z_2	→	A ④

Now we move towards the 2NF by unfolding the nonterminals S, A and B:

Z_0	→	...	Z_1	→	B ③	Z_2	→	A ④
		* Z_1 ① ⓪			* Z_2 ③			* Z_1 ④
		* Z_2 ② ⓪			i ⑤ ③			
		...						

Left factoring of Z_0 leads to the situation

Z_0	→	...	Z_3	→	Z_1 ① ⓪
		* Z_3			Z_2 ② ⓪
		...			

Now we have to unfold Z_1 and Z_2 (in order to re-establish the 2NF form) leading to

Z_3	→	...
		* Z_2 ③ ① ⓪
		* Z_1 ④ ② ⓪
		...

This requires another left factoring:

Z_3	→	...	Z_4	→	Z_2 ③ ① ⓪
		* Z_4			Z_1 ④ ② ⓪
		...			

Unfolding in turn leads to

Z_4	→	...
		* Z_1 ④ ③ ① ⓪
		* Z_2 ③ ④ ② ⓪
		...

Evidently, Z_4 repeats the problem that we just "solved" for Z_3 – albeit with longer right-hand sides. Hence we have a sequence of growing elements Z_0, Z_3, Z_4, ... and thus *the process will never terminate.* □

The problem with this example clearly lies in the fact that the auxiliary nonterminals Z_1 and Z_2 occur in a combined left factoring situation. This very strange setting makes it plausible that the situation is indeed pathological and will virtually never happen in practice. Nevertheless, since the above grammar is $LR(1)$ – actually even $SLR(1)$ – we have to accept the fact that our construction does not always work — if we include left factoring.

As illustrated by the above example (and shown by the normal-form theorems), the termination problem may only arise through iterated full left factoring. The clue to all solutions lies in the following fact.

***Fact**: We can stop the grammar transformation process*

at any point in time without losing correctness.

By stopping the process we obtain partly non-normalized grammars, that is, grammars which are "amost" in 3NF (except for a very few productions).

Definition 6.1.1 *We say that a grammar is* **almost in 3NF**, *if it is in 2NF and only a few productions violate the uniqueness requirement of the 3NF.*

An equally useful variant of this definition could be given by admitting a few productions which violate the 2NF property by still having productions of the form $(Z_i \rightarrow Z_j \ldots)$, *i.e. the leading* Z_j *has not yet been unfolded.*

The only problem is to decide, *when* the factor-unfold process should be stopped, that is, when we should content ourselves with an almost-3NF grammar. Fortunatley, there is a very simple criterium for this decision: we initially do the full left factorings for the "block" of all Z_i productions generated by the 2NF. This leads in general to a further block of new Z_j productions; these in turn call for more left factorings, and so forth. That is, we obtain a series of blocks of new Z_i productions. Our empirical data indicates that these consecutive blocks should be strictly decreasing in size. Otherwise the construction is very likely to be infinite. (As a matter of fact, this was the case in all our experiments.)

Note that the danger of prematurely stopping the transformation process due to the coarse block-size criterium only concerns the efficiency of the resulting parser, not its correctness!

Construction 6.1.1 *(Modified 3NF Construction) We alter the 3NF Construction 3.3.1 such that it performs the left factorings in the blockwise manner described above. We stop the factor-unfold process as soon as the blocks are no longer strictly decreasing.*

Lemma 6.1.1 *The modified 3NF construction always terminates and generates a 3NF grammar or an almost-3NF grammar.*

The proof is trivial. In practice, the result will mostly be in 3NF. But in a few pathological cases (as illustrated by the above example) some non-3NF productions will remain. But they still are 2NF.

In the following Sections 6.2 – 6.4 we consider three ways of dealing with these almost-3NF grammars.

6.2 A Simple Solution: Backtracking

If we have an almost-3NF grammar containing unnormalized productions such as

$$\begin{array}{lcl} Z_4 & \rightarrow & Z_2 \ \textcircled{3} \ \textcircled{1} \ \textcircled{0} \\ & & Z_1 \ \textcircled{4} \ \textcircled{2} \ \textcircled{0} \end{array}$$

in the above example we can still apply all our parsing schemes from Section 5. The only effect now is that the *backtracking* facility built into this programm will actually occur in the parsing process. This is "only" a question of efficiency, not of correctness. (In Section 9.3 we will briefly sketch implementation techniques that make this backtracking even negligeable in "99%" of all practical cases.)

6.3 Another Simple Solution: Lazy Grammar Transformation

We consider again an almost-3NF grammar $\mathcal{G}$. But if we now encounter a production like

$$\begin{array}{lcl} Z_4 & \to & Z_2 \ \textcircled{3}\ \textcircled{1}\ \textcircled{0} \\ & & Z_1 \ \textcircled{4}\ \textcircled{2}\ \textcircled{0} \end{array}$$

we do *not* apply backtracking in the parser. Rather, we perform *one* further unfolding/factorization step to the grammar, leading to a modified grammar $\mathcal{G}'$. Then we continue the parsing with $\mathcal{G}'$. We call this technique ***lazy grammar transformation***.[2]

Since $\mathcal{G}'$ allows us to proceed at least one symbol ahead in the input string, the number of possibly needed transformation steps is bound by the length of the input string. *This guarantees termination.*

We note in passing that this also allows a nice adaption to *incremental* parser generation [28].

6.4 A Solution Based on Equivalence Classes

The two solutions presented above both have the same disadvantage: they burden the runtime of the parser. Therefore we now consider another solution that is slightly more intricate but only burdens the parser generator. Moreover, it is closer to classical LR-style parsing.

Unfortunately this solution has a major *drawback* as well: it only works for the classical LR parsing of Section 5.3 and not for the emulated LR parsing of Section 5.2. So our freedom of choosing the one or the other algorithmic realization gets lost.[3]

The clue to the solution is the observation (expressed in Lemma 5.3.1) that in a parsing process only the bottom elements are needed. Or, put into other words, all elements after the first Z_i or $\textcircled{i}$ in a production can be ignored. This allows us to cut the infinite unfold-factor cycles.

To understand the underlying idea let us consider a phenotypical production that requires left factoring. That is, we resume the example from Section 6.1, but now in a more schematic form: we are in a situation, where the original 3NF construction stops with an almost-3NF grammar. This means that we have a few unnormalized productions of the following kind. (To make the explanation of our arguments easier we prefer to use letters Y_i instead of Z_i for the new symbols.)

$$\begin{array}{lcl} Z & \to & t\ Z_1\ z_1 \\ & & t\ Z_2\ z_2 \\ & & t\ \textcircled{i}\ z_3 \\ & & \ldots \end{array} \quad \rightsquigarrow \quad \begin{array}{lcl} Z & \to & t\ Y_1 \\ & & \ldots \\ Y_1 & \to & Z_1\ z_1 \\ & & Z_2\ z_2 \\ & & \textcircled{i}\ z_3 \end{array}$$

The z_i stand for sequences of Z_j and $\textcircled{j}$ possibly with trailing continuations $\boxed{A}$. The resulting new production calls for immediate unfolding, where the only problem arises, when two identical terminals are created by Z_1 and Z_2, necessitating a further left factoring. For illustration purposes let us assume that we do not yet run into the cycle.

$$\begin{array}{lcl} Y_1 & \to & Z_1\ z_1 \\ & & Z_2\ z_2 \\ & & \textcircled{i}\ z_3 \end{array} \quad \rightsquigarrow \quad \begin{array}{lcl} Y_1 & \to & t'\ Z_3\ z_4\ z_1 \\ & & \ldots \\ & & t'\ Z_4\ z_5\ z_2 \\ & & \ldots \\ & & \textcircled{i}\ z_3 \end{array} \quad \rightsquigarrow \quad \begin{array}{lcl} Y_1 & \to & t'\ Y_2 \\ & & \ldots \\ & & \ldots \\ & & \textcircled{i}\ z_3 \\ Y_2 & \to & Z_3\ z_4\ z_1 \\ & & Z_4\ z_5\ z_2 \end{array}$$

[2] In accordance with the popular terminology of JAVA we might also baptize this as "just-in-time transformation".

[3] Note that traditional LR parsing only realizes one of these possibilities anyhow.

Now we have to unfold the productions for Y_2, where the need for left-facorization may arise again. Let us assume that this is the point, where the cycle occurs.

$$\begin{array}{lcl} Y_2 & \to & Z_3\ z_4\ z_1 \\ & & Z_4\ z_5\ z_2 \end{array} \quad \rightsquigarrow \quad \begin{array}{lcl} Y_2 & \to & t''\ Z_1\ z_7\ z_4\ z_1 \\ & & \ldots \\ & & t''\ Z_2\ z_8\ z_5\ z_2 \\ & & \ldots \\ & & t''\ ⓘ\ z_9\ z_5\ z_2 \\ & & \ldots \end{array} \quad \rightsquigarrow \quad \begin{array}{lcl} Y_2 & \to & t''\ Y_3 \\ & & \ldots \\ & & \ldots \\ & & \ldots \\ Y_3 & \to & Z_1\ z_7\ z_4\ z_1 \\ & & Z_2\ z_8\ z_5\ z_2 \\ & & ⓘ\ z_9\ z_5\ z_2 \end{array}$$

The left factoring of t'' leads to Y_3, which is a repetition of the situation of Y_1, albeit with longer trailers. Processes of this kind are the (only) source of nontermination.

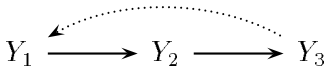
But the classical *LR* parser has a decisive feature that distinguishes it from the emulated *LR* parser: in the productions of Y_1, Y_2 and Y_3 anything beyond the leading Z_is and ⓙs will be cut off. We have indicated this by the light printing of z_3z_1, z_4z_2 etc. Therefore we may consider two productions as being equivalent, whenever they are equal after this cutting.

Definition 6.4.1 (Core, equivalence) *In a production $(A \to Z_i v)$ the part Z_i is called the* ***core*** *and v the* ***extension****. Analogously for $(A \to ⓘ\ v)$.*
We call two nonterminals ***equivalent****, if the cores of their productions are identical.* □

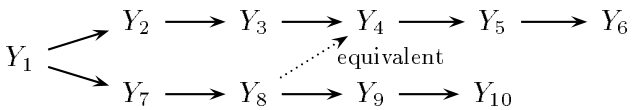
In our schematic example we have the cores $Y_1 \widehat{=} \{Z_1, Z_2, ⓘ\}$, $Y_2 \widehat{=} \{Z_3, Z_4\}$ and $Y_3 \widehat{=} \{Z_1, Z_2, ⓘ\}$, which shows the equivalence of Y_1 and Y_3.

So we may push the modified 3NF construction a bit further: we continue the unfolding/factorization process except for those Y_i that are equivalent to an existing Y_k. This way the brute-force stopping criterium from Section 6.1 (i.e. the blocks are not strictly decreasing in size) is extended by the finer criterium that equivalent productions are not treated anew.[4]

As a matter of fact, we could refine this even further: only those equivalences are dangerous that lead to a cycle. In our above example we have this situation



But when the equivalence only generates a dag-like situation it is harmless:



Here we may continue the transformation of Y_8. Even though this creates additional nonterminals Y_i it has the great advantage that we keep the freedom of choosing among all parsing algorithms.

The thus extended transformation process has two possible outcomes: either it fully succeeds with a 3NF grammar or it still ends with an almost-3NF grammar. But in this grammar each unnormalized nonterminal Y_i is known to be equivalent to an existing (normalized) one – and it is known to be indeed responsible for a cycle.

Construction 6.4.1 *(Modified 3NF construction – first attempt) We modify the 3NF construction as follows: when the left factoring would lead to a new nonterminal Y_j that is equivalent to an existing Y_i, then we use the old Y_i instead.* □

[4] We could use this finer criterium from the very beginning, but it it is much more costly than the block-size criterium and it is only needed in a few pathological cases.

Lemma 6.4.1 *With the above modification the construction always terminates.*

Proof. We are given a set of Z_i productions resulting from the 2NF and start the process of left factoring. Since there are only finitely many Z_i and ⓘ, the number of sets over these Z_i and ⓘ and thus the number of possible new Y_k is also finite. □

We now have guaranteed termination for our construction. And also correctness is still guarantteed – provided that we use the classical LR parsing variant, which only looks at the bottom elements in the parsing process (see Section 5.3). *However, we may lose the $LR(k)$ property.* The reason is that our above notion of equivalence does not take the lookahead into consideration.

To formulate this definition the notion of *k-frontier* is convenient: the **k-frontier** of a string v is the set of the prefixes of length k of all strings derivabel from v:

$$frontier_k(v) = \{\ u \mid v \stackrel{*}{\dashrightarrow} u\,w,\ |u| = k\ \}$$

(We use the derivation '$\stackrel{*}{\dashrightarrow}$', since we may have to employ continuation symbols in order to achieve the required length k.)

Definition 6.4.2 (k-equivalence) *Consider two equivalent nonterminals Y_1 and Y_2 (i.e. they have the same core). Y_1 and Y_2 are called **k-equivalent** if each pair of corresponding productions*

$$\begin{array}{lcl} Y_1 & \to & \ldots \\ & & Z\,z_1 \\ & & \ldots \end{array} \qquad\qquad \begin{array}{lcl} Y_2 & \to & \ldots \\ & & Z\,z_2 \\ & & \ldots \end{array}$$

generates the same prefixes of length k: $frontier_k(Z\,z_1) = frontier_k(Z\,z_2)$.

Theorem 6.4.1 *(Adaption of Main Theorem) If we use k-equivalence in the modified 3NF construction above, then the Main Theorem 4.0.2 still holds.*

Proof: We only need to show that the identification of k-equivalent nonterminals does not introduce additional conflicts. This can be seen in our schematic example above. Consider Y_3 that – without the identification with Y_1 – would have to be transformed further into

$$\begin{array}{lcl} Y_3 & \to & t'\ Y_4 \\ & & \ldots \\ & & \ldots \\ & & ⓘ\ z_9\ z_5\ z_2 \end{array} \qquad\qquad \begin{array}{lcl} Y_4 & \to & Z_3\ z_4\ z_7\ z_4\ z_1 \\ & & Z_4\ z_5\ z_8\ z_5\ z_2 \end{array}$$

Since Y_1 and Y_3 are k-equivalent, Y_3 has a shift/reduce or reduce/reduce conflict of length k if and only if Y_1 has the same conflict. □

Note: The emulated LR parser also uses the extension parts in the parsing process. Therefore Y_1 and Y_3 could not be identified in this setting. But for the classical LR parser the extension parts are only needed for the k-lookahead in the case of conflicts. And this is respected by the notion of k-equivalence.

Chapter 7

Relationship to Classical *LR*-Style Parser Generation

Our Main Theorem 4.0.2 demonstrates that the *result* of our 3NF construction is an $LR(k)$ grammar. Now we want to study, how the *construction itself* relates to the classical ways of obtaining LR-style parsers. This will provide further insights into both our method *and* the traditional techniques. Moreover, it will turn out (somewhat surprisingly) that – and why – our approach combines the power of $LR(k)$ parsers with the efficiency of $LALR(k)$ parsers – at least to some extent.

7.1 The Classical "Sets-of-Items" Construction: $LR(0)$ Parser

The classical construction for $LR(k)$ parsers is the "sets-of-items" construction as described in the standard textbook [1]. It appears worthwhile to study the connections between this traditional construction and our method more deeply. Roughly speaking, we have the following correspondences:

- The 1NF construction produces the so-called "kernel items".
- The 2NF construction corresponds to the "closure forming".
- The 3NF construction generates the so-called "sets of items". More precisely, this holds for the 3NF construction modulo the core-based equivalence relations of Def. 6.4.1 and 6.4.2.

In order to ease understanding we illustrate the comparison by a classical example[1] (see [1], p. 222): Table 7.1 presents the well-known grammar for arithmetical expressions.

Proposition. *Consider the Grammar of Table 7.1 and its $LR(0)$ items (as given in [1], p. 225). There is a one-to-one correspondence between the kernel items and the Z_i and ⓘ in our 1NF. More precisely, the Z_i and ⓘ represent exactly the parts following the dot.*

Proof: The lemma can be shown by a simple induction. Due to its straightforward simplicity we do not present this induction here formally but rather illustrate it on the basis of the 1NF of the above grammar $\mathcal{K}_0$ (see Table 7.2).

[1] Unfortunately we cannot take our running example here, because we need "serious left recursion" in order to demonstrate some complexities.

Grammar $\mathcal{K}_0$		
S	$\rightarrow$	E ①
E	$\rightarrow$	E + T ②
		T ③
T	$\rightarrow$	T * F ④
		F ⑤
F	$\rightarrow$	(E) ⑥
		i ⑦

Table 7.1: A left-recursive grammar

Grammar $\mathcal{K}_1$								
S	$\rightarrow$	E ①	Z_0	$\rightarrow$	S ⓪			
E	$\rightarrow$	E Z_1	Z_1	$\rightarrow$	+ Z_2	Z_2	$\rightarrow$	T ②
		T ③						
T	$\rightarrow$	T Z_3	Z_3	$\rightarrow$	* Z_4	Z_4	$\rightarrow$	F ④
		F ⑤						
F	$\rightarrow$	(Z_5	Z_5	$\rightarrow$	E Z_6	Z_6	$\rightarrow$	) ⑥
		i ⑦						

Table 7.2: First normal form

The *"canonical collection of sets of LR(0) items"* in [1] is based on two functions, *closure* and *goto*. We start the process by forming the closure of the productions for S (where a '$\cdot$' is added in front of the right-hand sides). The *kernel items* are the original productions, the *nonkernel items* are added by the *closure* forming: the productions of all nonterminals that are directly preceded by a '$\cdot$' are added (recursively, if necessary) to the set. (Evidently, this corresponds to the unfolding process in our 2NF construction.)

I_0 :	$S \rightarrow \cdot E$	Z_0	*kernel item*
	$E \rightarrow \cdot E + T$		
	$E \rightarrow \cdot T$		
	$T \rightarrow \cdot T * F$		*nonkernel items*
	$T \rightarrow \cdot F$		
	$F \rightarrow \cdot (E)$		
	$F \rightarrow \cdot i$		

At the right of each kernel item we point out its corresponding Z_i or ⓘ from the above 1NF.

Now the operation $goto(I_0, A)$ is applied to the set I_0 and every nonterminal A that is preceded by a dot: this operation shifts the '$\cdot$' over the nonterminal A and thus generates the kernel items of the next set. If necessary, the *closure* operation is then applied to these kernel items. The final result of this process is shown in Table 7.3.

From this illustration the above lemma is evident: the shifting of the dot in the *goto* operation corresponds to the introduction of the new Z_i in the 1NF construction. Therefore the kernel items indeed are in a one-to-one correspondence to the Z_i and ⓘ. □

As mentioned above, the 2NF construction corresponds to the closure forming.

Finally, the sets with *several kernel items* are those points, where our 3NF construction performs the left-factorization. This essentially yields the sets of (kernel) items.

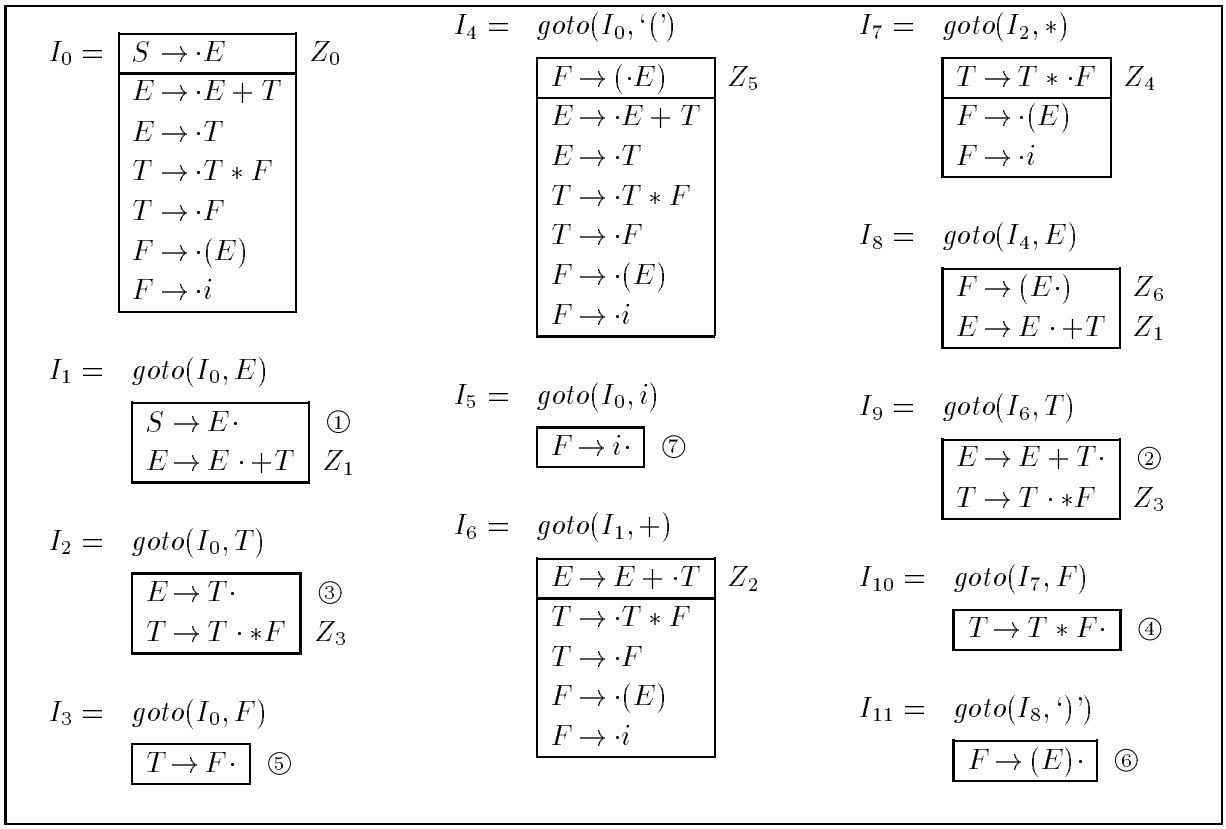


Table 7.3: The sets of items of grammar $\mathcal{K}_0$

Proposition. *The sets of items I_k are in a one-to-one correspondence to the equivalence classes of the 3NF according to the core-based equivalence relation of Def. 6.4.1 and 6.4.2.*

Proof: We illustrate the proof again in terms of the above grammar. For example, in our 3NF construction we have – among others – the following situations (see also Table 7.4):

$Z_0 \to \ldots$ T ③ ① ⓪ T Z_3 ③ ① ⓪ $\ldots$

$Z_5 \to \ldots$ T ③ Z_6 T Z_3 ③ Z_6 $\ldots$

By left-factorization we obtain the new Z_i-productions

$Z_8 \to$ ③ ① ⓪ Z_3 ③ ① ⓪

$Z_{11} \to$ ③ Z_6 Z_3 ③ Z_6

which then undergo the usual unfolding and (if necessary) iterated factorization process leading to the 3NF. However, since Z_8 and Z_{11} have the same core, they are equivalent and thus belong to the same set of items. This set is determined by the core (in this case I_2). □

For example, in our above grammar the left factorizations yields new Z_i with the following cores: $Z_7 \hat{=} \{①, Z_1\}$, $Z_8 \hat{=} \{③, Z_3\}$, $Z_9 \hat{=} \{②, Z_3\}$, $Z_{10} \hat{=} \{Z_6, Z_1\}$ and $Z_{11} \hat{=} \{③, Z_3\}$. So we obtain the according correspondences $I_1 \hat{=} Z_7$, $I_2 \hat{=} \{Z_8, Z_{11}\}$, $I_8 \hat{=} Z_{10}$ and $I_9 \hat{=} Z_9$.

Remark 1: This reference to the core-based equivalence classes demonstrates that the traditional *LR*-style parsing indeed corresponds to what we baptized classical *LR* parsing.

Grammar $\mathcal{K}_2$

$Z_0 \rightarrow$ S ⓪	$Z_2 \rightarrow$ T Z_9	$Z_5 \rightarrow$ E Z_{10}
E Z_7	F ⑤ ②	T Z_{11}
T Z_8	(Z_5 ⑤ ②	F ⑤ ③ Z_6
F ⑤ ③ ① ⓪	i ⑦ ⑤ ②	(Z_5 ⑤ ③ Z_6
(Z_5 ⑤ ③ ① ⓪	$Z_3 \rightarrow$ * Z_4	i ⑦ ⑤ ③ Z_6
i ⑦ ⑤ ③ ① ⓪	$Z_4 \rightarrow$ F ④	$Z_6 \rightarrow$) ⑥
$Z_1 \rightarrow$ + Z_2	(Z_5 ④	
	i ⑦ ④	
$Z_7 \rightarrow$ ① ⓪ ⟨S⟩	$Z_9 \rightarrow$ ② ⟨E⟩	$Z_{10} \rightarrow$) ⑥
+ Z_2 ① ⓪	* Z_4 ②	+ Z_2 Z_6
$Z_8 \rightarrow$ ③ ① ⓪ ⟨S⟩		$Z_{11} \rightarrow$ ③ Z_6
* Z_4 ③ ① ⓪		* Z_4 ③ Z_6

Table 7.4: Second normal form

Remark 2: The plain notion of equivalence leads to the sets of $LR(0)$ items, that is, to an $LR(0)$ parser. If we use k-equivalence instead, we obtain an $LR(k)$ parser. The relationship of this fact to the traditional techniques will be discussed in the following.

7.2 Stronger Than $LALR(k)$

It is known from literature that $LALR(k)$ is as strong as $LR(k)$ with respect to shift/reduce conflicts. (It may, however, perform a few more reductions before it realizes a conflict.) But it is strictly weaker with respect to reduce/reduce conflicts. Since we have already shown that our method leads to $LR(k)$ parsing, we have the following corollary to Theorem 4.0.2:

Corollary 7.2.1 *The parsing induced by the 3NF is strictly stronger than $LALR(k)$ parsing.*

But we also want to illustrate this property by a concrete example in order to gain further insights into the *reasons* for this greater power. In the literature one can find various little grammars that demonstrate the superiority of $LR(1)$; as usual we pick here an example from [1] (p. 238), which is given in Table 7.5.[2]

Grammar $\mathcal{H}_0$

S → a A d ①	A → c ⑤	B → c ⑥
a B e ②		
b A e ③		
b B d ④		

Table 7.5: An $LR(1)$ but non-$LALR(1)$ grammar

As usual we derive the 2NF of this grammar, which in particular creates the following two productions

[2]Incidentally, Beatty [5] (p.1019) uses this grammar as an example for an $LL(k)$ grammar, which is *not* $LALR(k)$ for any k.

$$
\begin{array}{lcl@{\qquad\qquad}lcl}
Z_1 & \to & \text{A } Z_2 & Z_4 & \to & \text{A } Z_5 \\
 & & \text{B } Z_3 & & & \text{B } Z_6 \\
 & & \text{c } ⑤\ Z_2 & & & \text{c } ⑤\ Z_5 \\
 & & \text{c } ⑥\ Z_3 & & & \text{c } ⑥\ Z_6
\end{array}
$$

This calls for left factorings of c, which create Z_7 and Z_8. The final result is presented in Table 7.6.

Grammar $\mathcal{H}_3$		
$Z_0 \to$ S ⓪	$Z_1 \to$ A Z_2	$Z_4 \to$ A Z_5
a Z_1 ⓪	B Z_3	B Z_6
b Z_4 ⓪	c Z_7	c Z_8
	$Z_2 \to$ d ①	$Z_5 \to$ e ③
	$Z_3 \to$ e ②	$Z_6 \to$ d ④
	$Z_7 \to$ ⑤ Z_2	$Z_8 \to$ ⑤ Z_5
	⑥ Z_3	⑥ Z_6

Table 7.6: Third normal form

The potential reduce/reduce conflicts in Z_7 and Z_8 are resolved here by inspecting the one-symbol lookahead of the productions Z_2 and Z_3 in the first case and Z_5 and Z_6 in the second case. Since this conflict is *not* resolved by the $LALR(1)$ parser (see e.g. [1], p. 238), this demonstrates that our parser is strictly stronger.

The explanation is simple: The $LALR$ technique puts Z_is with the same core into an equivalence class and *merges their extensions*. In the above example this applies to Z_7 and Z_8, which leads to a situation of the form

$$
\begin{array}{lcl}
Z_{7/8} & \to & ⑤\ \{Z_2, Z_5\} \\
 & & ⑥\ \{Z_3, Z_6\}
\end{array}
$$

The resulting union does no longer disambiguate the two reductions.

Note: In this example our method successfully terminates already in the basic 3NF construction and therefore never has to look at the cores and their induced equivalences. But even if we would use that criterium from the very beginning (which is admissible but inefficient), we would have to employ the notion of 1-equivalence in order to obtain an $LR(1)$ parser. Under this equivalence Z_7 and Z_8 still would *not* be identified!

7.3 Less Expensive Than Classical $LR(1)$: Empirical Results

The big disadvantage of $LR(1)$ parsers is the huge size of their tables. Since their superiority over $LALR(1)$ parsing only shows in a few relatively pathological cases, this size prohibits their practical use. We claim that our approach actually produces (much?) smaller tables than $LR(1)$ parsers, even though in general they are not quite as small as $LALR(1)$ tables.

It is well-known that the difference in sizes between $LALR(1)$ tables and $LR(1)$ tables can be orders of magnitude (a few hundred as opposed to several thousands). This naturally raises the question, whether the size of our grammars is closer to the one or to the other.

An open problem. Unfortunately we do not have a formal proof that gives a precise cost analysis, but there are two reasons to conclude that the size will be closer to that of an $LALR(1)$ parser: the first one is some "higher-level reasoning" (a weak one) and the second one is evidence gained from experimental test data (a strong one).

(1) Plausibility. The classical construction of $LR(1)$ parsers essentially duplicates all productions (i.e. sets of items) for all possible follow symbols. It is this duplication that causes the explosive growth in size. By contrast, our 3NF construction works with *continuation symbols* that represent continuation *languages*, i.e. whole groups of follow symbols. Hence, the duplication only happens for these groups and not for each symbol in each group.

This is essentially the same principle as that used for the equivalence classes of $LALR(1)$ parsers. However, our approach is not quite as coarse as the $LALR$ approach and only equivates cores to an extent that does not violate the $LR(k)$ property.

Example: Again, we illustrate this claim by a concrete example. In [1] (p. 231–236) the grammar from Table 7.7 is used to illustrate the derivation of an $LR(1)$ parser.

Grammar $\mathcal{I}_0$		
S	$\rightarrow$	C C ①
C	$\rightarrow$	a C ②
		b ③

Table 7.7: Initial grammar

Again we generate the 3NF of this grammar, which is presented in Table 7.8.

Grammar $\mathcal{I}_3$					
Z_0	$\rightarrow$	S ⓪	Z_1	$\rightarrow$	C ①
		C Z_1 ⓪			a Z_2 ①
		a Z_2 Z_1 ⓪			b ③ ①
		b ③ Z_1 ⓪	Z_2	$\rightarrow$	C ②
					a Z_2 ②
					b ③ ②

Table 7.8: Second normal form

This grammar has 7 "states" (three symbols Z_i and four actions ⓘ), whereas the corresponding $LR(1)$ grammar has 10 "states" I_j (see [1], p. 235). This shows that our approach actually produces smaller tables – at least in one example. □

This illustrates the "meta-level reasoning" that is based on the following observations:

- First, recall from Lemma 3.1.1 that the number of symbols Z_i and ⓘ in the 1NF is equal to the size of the original grammar $\mathcal{G}$, where the "size" s of a grammar is the sum of the lengths of its right-hand sides. (Moreover, this is also the number of kernel items in the sets of $LR(0)$ items.)
- The 2NF does not get much larger than the 1NF: most of our transformations are unfoldings, which leave the number of the Z_is invariant. The only increase could occur through left-recursion removals. But these are rare – and if we use the classical LR parsing variant, they are not needed at all.
- Only the left factorings in the transition to the 3NF introduce new Z_js. This should, however, not do too much harm:
  - Left factorings are not too frequent.

– In many cases the left factoring entails an unfolding by which some other Z_k vanishes, thus leaving the overall count invariant. As a matter of fact, only when left factoring becomes necessary due to the unfolding of two different nonterminals *and* when the following Z_k are also needed in productions elsewhere, the number increases.

It is telling that in all of our examples the 3NF is equal or even smaller than the 1NF! So we may expect the number of Z_is of our 3NF to be in the order of the size s of the original grammar $\mathcal{G}$. By contrast, the $LR(1)$ tables are in the order $t * s$, where t is the number of terminal symbols in the grammar and s is again the size of $\mathcal{G}$. This is based on the reasoning that only left factorings lead to a reduction of the sets of items I_j and that every $LR(0)$ set is duplicated for all its follow symbols.

(2) Empirical evidence. The above "meta-level reasoning" needs backing by empirical measurements; these are presented in Figure 7.1. We have used three major grammars: The C

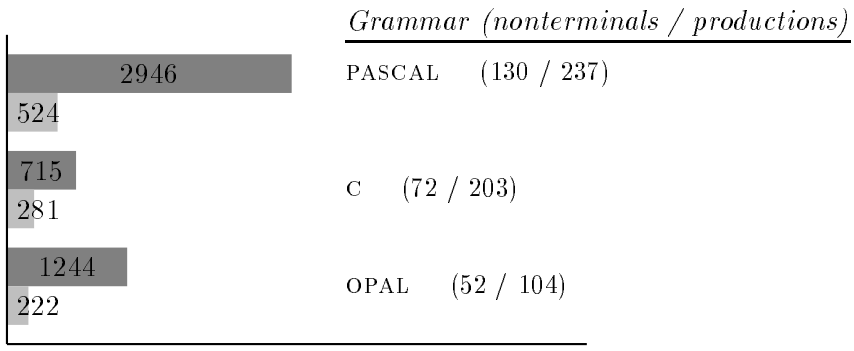


Figure 7.1: Comparison of table sizes

grammar as specified in the standard book [17], the PASCAL grammar in its original version published in [15] and a simplified version of our revised language OPAL 2α [11].

The dark shaded areas represent the number of states generated by the traditional $LR(1)$ method (as described in [1] p. 232), whereas the light shaded areas represent the number of states generated by our method. As can be seen from these figures, our method creates tables that are by a factor $3-5$ smaller than the traditional tables.

These empirical results back the above "meta-level reasoning".

Part III

Variations, Observations and Applications

Chapter 8

Facts About Grammars and Languages

Even though it is not part of our derivation, we note in passing that our approach also provides nice insights into general properties of grammars and their languages. For example, the following facts are known from the literature (see e.g. [29, 19, 2]):

Proposition. *The following properties hold for the various* grammar *classes:*

1. *Every $LL(k)$ grammar is also an $LR(k)$ grammar.*
2. *For a given grammar $\mathcal{G}$ it is decidable, whether it is an $LR(k)$ grammar for given k. Analogously for $LL(k)$ grammars.*
3. *Every $LR(k)$ grammar is unambiguous. Analogously for $LL(k)$ grammars.*
4. *There are $LR(1)$ grammars which are not $SLR(k)$ or $LALR(k)$ for any k.*

The first property is evident from our construction, since the k symbols in $LL(k)$ refer to the handle *plus* some lookahead (which therefore has length $\leq k$), whereas the k in $LR(k)$ refers only to the lookahead. Actually, we can see quite clearly here, *why* LR-style parsing is more powerful than LL-style parsing: suppose a nonterminal A has two productions of the form

$$\begin{array}{lcl} A & \rightarrow & B \ldots \\ A & \rightarrow & C \ldots \end{array}$$

In such a situation a recursive-descent parser has to choose which of the two productions to take. Suppose that the two nonterminals B and C have productions

$$\begin{array}{lcl} B & \rightarrow & t \ldots \\ & \vdots & \\ C & \rightarrow & t \ldots \\ & \vdots & \end{array}$$

with the same leading terminal t. (This means that the language does *not* have the $LL(1)$ property.) In this case — which may be baptized a ***shift/shift conflict*** — the recursive-descent parser for A cannot make the choice by simply looking at the next token: it needs

backtracking. At such points our construction *delays* the need for making the decision by applying left factoring. And this delaying of decisions makes the LR paradigm more powerful.

This coincides with the observation (see Section 2.3) that LL parsing corresponds to a *preorder* traversal of the tree. This means that the parser has to decide, whether to insert an (invisible) action symbol $\textcircled{i}$, *before* it has seen the handle α, whereas in LR parsing the decision is taken *after* having seen the handle.

Properties (2) and (3) follow directly from the fact that our construction works. The last property has actually been shown by the example in Section 7.2.

These properties show, how strong the LR grammars are. But for the weaker $SLR(k)$ and $LALR(k)$ grammars the relationships are more complicated.

Proposition. *For grammars with ε-productions the $LL(1)$ and $SLR(1)$ classes are incomparable [2, 5] But for ε-free grammars the following properties are shown in [5]:*

1. *If $\mathcal{G}$ is $LL(1)$ then it is also LR(0).*
2. *There are $LL(k)$ grammars which are not $LALR(k)$ for any $k \geq 2$*
3. *For p-reduced grammars (i.e., grammars without null nonterminals) the following property holds:*

   If $\mathcal{G}$ is $LL(1)$ then it is also $LALR(1)$.

The first property follows directly from the fact that we need no left factoring. This and the absence of ε-productions guarantees that no $\textcircled{i}$ occurs as first element of a right-hand side. Hence the grammar is $LR(0)$. The same kind of reasoning also shows the last property. The second property is actually shown by the grammar in Section 7.2.

Proposition. *The following properties hold for the various* language *classes:*

1. *Every $LR(k)$ language is also an $LR(1)$ language.*
2. *Every $LR(k)$ language actually is also an $LR(0)$ language (see [19] p. 143).*
3. *Every $LR(k)$ language is $SLR(1)$.*
4. *Every $LL(1)$ language is $LR(1)$ and thus also $SLR(1)$.*

Note that we now talk about the induced languages, not about the grammars themselves.

Sketch of proof. Since we are no longer interested in the (grammar-dependent) trees but only in the pure strings, we can ignore the actions $\textcircled{i}$. This means that we can continue the left factoring process beyond these $\textcircled{i}$, which is guaranteed to terminate at some point due to the $LR(k)$ property. The thus constructed grammar clearly is an $LR(0)$ grammar. □

These short sketches shall suffice to indicate that our modified approach not only yields useful techniques for practical parser generation, but also provides insights into theoretical questions.

Chapter 9

Variations

In the previous sections we have presented an approach that is more flexible and easier to comprehend than traditional LR-style parsers, but besides that exhibits the same expressive power. Now we want to show, how this increased flexibility can be utilized to ease certain features (such as error handling) and to increase the power (such as adaption to non-$LR(1)$ languages).

9.1 Variations on the Grammar Presentation

Throughout the paper we have used a very puristic presentation of the grammars: all productions obey the classical form of context-free grammars and are teminated by action symbols. But in practice we need more notational comfort and flexibility.

9.1.1 Features From EBNF

In practical grammars one wants to have the convenience of the so-called EBNF form. Its features can be trivially translated into elementary forms of productions. As a representative example we consider the Kleene star. The question here is, whether opting for a left- or a right-recursive solution makes any difference.

left-recursive solution

$$\begin{array}{lcl} A \to u^* \; v \; ① \end{array} \rightsquigarrow \begin{array}{lcl} A \to B \; v \; ① \; \boxed{A} \\ B \to B \; u \; ② \; \boxed{B} \\ \qquad ③ \; \boxed{B} \end{array} \rightsquigarrow \begin{array}{lcl} A \to ③ \; Z \\ Z \to u \; ② \; Z \\ \qquad v \; ① \; \boxed{A} \end{array}$$

right-recursive solution

$$\begin{array}{lcl} A \to u^* \; v \; ① \end{array} \rightsquigarrow \begin{array}{lcl} A \to C \; v \; ① \; \boxed{A} \\ C \to u \; C \; ② \; \boxed{C} \\ \qquad ③ \; \boxed{C} \end{array}$$

This shows that both solutions *are equally good*: in the left-recursive solution problems arise, when $u\,Z$ and $v\,\boxed{A}$, that is, $u^+v\,\boxed{A}$ and $v\,\boxed{A}$ exhibit a k-conflict. In the right-recursive solution such problems occur, when $u\,C\,\boxed{C}$ and $\boxed{C}$, that is, $u^+\,\boxed{C}$ and $\boxed{C}$ exhibit a k-conflict. Since $\boxed{C} = v\,\boxed{A}$ these two situations are identical.

The same considerations apply to the case of u^+ and other features such as optional terms or repetitions with separators.

9.1.2 Action-less Productions

We have used the action symbols ⓘ throughout this paper as a central means for organizing the overall transformation, parsing and tree-formation process. However, there are situations when one actually wants to do without these action symbols. And this can be done without problems – at least if some precautions are observed.

The major reason, why one may want to omit action symbols, is nicely illustrated by the classical grammar for arithmetic expressions (see Table 7.1). We can write it in modified form as in Table 9.1, where we retain the numbering from the old version.

Grammar $\mathcal{K}_a$		
S	→	E
E	→	E + T ②
		T
T	→	T * F ④
		F
F	→	(E) ⑥
		i ⑦

Table 9.1: A left-recursive grammar

Here we have the action symbols only in those places, where we actually want to perform semantic actions. So we spare a number of pure adaption actions. This illustrates the main reason for omitting action symbols: it is often desirable to introduce auxiliary nonterminals into a grammar only for modularization purposes. These nonterminals do not play a role in the abstract syntax tree but only make the grammar "better": it may express precedences, avoid ambiguities or simply be more readable. This typically happens in two kinds of situations:

1. A nonterminal with many variants can be made more readable by making each right-hand side into a separate production with a new nonterminal. This does not cause problems as can can be seen from an application of A in a production such as $Z \to A$ ⓘ:

A	→	B	Z	→	A ⓘ	$\rightsquigarrow$	Z	→	A ⓘ
		C							B ⓘ
B	→	u ①							u ① ⓘ
C	→	v ②							C ⓘ
									v ② ⓘ

   The effect here is that *the nonterminal A does not show* up in the abstract syntax tree (because there is no action that creates it), but only B and C. Therefore the production $Z \to A$ ⓘ can be omitted.

2. There is also the converse situation, where one wants to have the nonterminal A in the syntax tree, but not the variants B and C:

A	→	B ①	Z	→	A ⓘ	$\rightsquigarrow$	Z	→	A ⓘ
		C ②							B ① ⓘ
B	→	u							u ① ⓘ
C	→	v							C ② ⓘ
									v ② ⓘ

   The effect here is that *the nonterminals B and C do not show* up in the abstract syntax tree, but only A. Therefore the pertinent right-hand sides B①ⓘ and C②ⓘ can be eliminated.

These schematic examples illustrate that our techniques go through without problems in such situations. However, one should be careful, when recursive productions are involved. To see this consider the following productions and their transformed form.

A	→	B ①	⇝	A	→	v Z
B	→	B u		Z	→	u Z
		v				①

Here it is almost impossible to give any useful code for the semantic action ①, because all information about how many u's there are has been lost.

9.1.3 Precedence And Associativity

Almost every language encounters the problem that the standard operators in arithmetic and logical expressions shall be written in infix notation and obey certain precedence and associativity rules. This complicates the grammar considerably. Moreover, it makes it quite different from the actually intended abstract syntax tree. (In Section 10 we will show how to handle applications, where the programmer is allowed to invent his own infix operators.)

The most famous example for such a grammar is probably that for arithmetic expressions given in Table 9.2. It is charming, because it expresses the essence of the envisaged abstract syntax tree. And it is only because of precedence rules that we have to switch to the much less obvious grammar used in practice (see Table 7.1). We will now sketch, how such an ambiguous grammar can actually be used for parsing, if we add some additional information. All we need to say is that the operator '$*$' binds stronger than '$+$', and that both shall be treated as being left-associative.

Grammar $\mathcal{M}_0$

S	→	E ①	
E	→	E + E ②	[left-assoc, $* \succ +$]
		E * E ③	[left-assoc]
		i ④	

Table 9.2: A highly ambiguous grammar

The 3NF construction for this grammar yields the grammar in Table 9.3.

Grammar $\mathcal{M}_1$

Z_0	→	S ⓪	Z_2	→	E Z_7	Z_4	→	E Z_8
		E Z_6			i ④ Z_7			i ④ Z_8
		i ④ Z_6						
Z_6	→	① $\boxed{S}$	Z_7	→	② $\boxed{E}$	Z_8	→	③ $\boxed{E}$
		+ Z_2 Z_6			+ Z_2 Z_7			+ Z_2 Z_8
		* Z_4 Z_6			* Z_4 Z_7			* Z_4 Z_8
$\boxed{S}$	--→	↯	$\boxed{E}$	--→	Z_6			
					Z_7			
					Z_8			

Table 9.3: The 3NF of $\mathcal{M}_0$

This is obviously highly ambiguous, since $\boxed{E}$ leads to $+$, $*$ and ↯ as possible first symbols.

But now we want to employ the information that '$*$' has a higher precedence than '$+$' and that they shall both be left-associative. (We only consider a 1-symbol lookahead.)

- The *precedence* information says that no reduction of $E + E$ ② may take place, when the next symbol is '$*$'. So the production for Z_7 is modified to $(Z_7 \rightarrow$ ② $\boxed{E_1\rangle})$, where $\boxed{E_1\rangle}$ is $\boxed{E\rangle}$ minus the symbol '$*$'.

- The *precedence* information also says that the reduction $E * E$ ③ must take place, when the next symbol is '+'. That is, in Z_8 the reduction has precedence over the shift of '+'. This means that we have to eliminate the production $(Z_8 \rightarrow + Z_2 Z_8)$.

- The *left-associativity* says that a shift/reduce conflict is to be resolved in favor of the reduction. This means that the two productions $(Z_7 \rightarrow + Z_2 Z_7)$ and $(Z_8 \rightarrow * Z_4 Z_8)$ have to be eliminated from the grammar.

- If we would have *right-associativity* of, say '+', then we would have to take the symbol '+' out of $\boxed{E_1\rangle}$ as well. That is, we would have to act as if the precedence $+ \succ +$ would hold.

The results of these considerations are presented in Table 9.4. As can be seen immediately, this

Grammar $\mathcal{M}_2$		
$Z_0 \rightarrow$ S ⓪ E Z_6 i ④ Z_6 $Z_6 \rightarrow$ ① $\boxed{S\rangle}$ $+ Z_2 Z_6$ $* Z_4 Z_6$	$Z_2 \rightarrow$ E Z_7 i ④ Z_7 $Z_7 \rightarrow$ ② $\boxed{E_1\rangle}$ $* Z_4 Z_7$	$Z_4 \rightarrow$ E Z_8 i ④ Z_8 $Z_8 \rightarrow$ ③ $\boxed{E\rangle}$
$\boxed{S\rangle}$ --> ↯	$\boxed{E_1\rangle}$ --> ↯ +	$\boxed{E\rangle}$ --> ↯ + *

Table 9.4: The 3NF of $\mathcal{M}_0$ after disambiguation

grammar has no conflicts!
So we have here a systematic process by which we can disambiguate grammars based on associativity and precedence information. This is a very convenient feature:

- It makes the specification of many grammars much simpler, in particular for people who are not so familiar with the art of syntax specification.

- It is mandatory in applications, where programmers shall be able to define their own infix- or mixfix operators (see Section 10).

9.2 Variations on the Grammar Transformation Process

The description of our transformation process in Section 3 was very concise. Now we want to point out a few possibilities for optimizations and extensions. Moreover, we want to give an a-posteriori motivation for certain design choices.

9.2.1 Optimizations

In order not to interrupt the presentation of our approach in Section 3, we have not mentioned two straightforward means for improving the efficiency:

- *Chain productions.* Our construction generates as a byproduct an optimization that — according to [39] — may speed up the final LR-style parser considerably. As can be seen e.g. in Table 3.3 we obtain at various places sequences of actions such as $Z_3 \to i$ ④ ⑤ ③. These sequences result from so-called chain productions.

  In non-optimized LR parsing the above sequence ultimately leads to three individual state transistions plus the corresponding reductions, whereas in the optimized version it is handled by one state transition and one (combined) reduction. This latter variant is generated automatically in our approach.

- *Optimization.* In Construction 3.2.1 we allow an arbitrary order in which the nonterminals are unfolded (and thus eliminated). However, by using a specific ordering we can improve the process such that it will need fewer left factorings and thus produces a smaller result grammar.

  When determining the order, we should start with the "topmost" nonterminals. In this context, a nonterminal A is "above" a nonterminal B if B occurs (as first symbol) in a rule for A. That is,

  $$A \text{ "is above" } B \text{ if there is a rule } A \to B \ldots$$

  This was the reason why we chose the nonterminal R before L in our earlier example. "Lazy" left factoring is what makes it better: we collect all pertinent right-hand sides before we treat them together in a single transformation.

  In connection with left-recursive productions we can apply the classical algorithm for finding the maximal strongly connected components of a directed graph.

9.2.2 Incremental Parser Generation

For many applications (see Section 10) it would be helpful, if we could build up our parsers incrementally.[1] Our 3NF construction provides this facility relatively easily. We distinguish two cases:

1. Suppose we want to add a new nonterminal B and its productions. Since this nonterminal would be unreachable, we have to extend at least one of the existing nonterminals, say A, by the new production $(A \to B$ ⓘ$)$.

   This situation is easily handled by our construction, provided that no left recursion is introduced.

   - The 1NF construction for B is independent from the existing grammar (except for determining, which new Z_is are available).
   - The 2NF construction proceeds through the nonterminals from $\mathcal{N}$ successively anyway. So we simply treat B as the last nonterminal to be unfolded.

     The only modification is that we must remember the productions for the other nonterminals, because we have to unfold them in the right-hand sides of B, before we unfold B in A.

[1] From the literature it is known that this can be done within the LR paradigm [7], but that it is simpler within the LL paradigm [3].

2. The story becomes more intricate, if we admit more complex extensions of an existing nonterminal A. Most of these extensions can be reduced to the situation (1) above, with one exception: if the addition leads to (direct or indirect) left recursion, we have to be careful. To see the underlying principle let us consider the simplest possible situation. Suppose that we have the grammar of Table 9.5.

Grammar $\mathcal{N}_0$		
S	$\rightarrow$	A ①
A	$\rightarrow$	x ②

Table 9.5: A trivial nonrecursive grammar

The 3NF of this grammar is given in Table 9.6.

Grammar $\mathcal{N}_1$		
Z_0	$\rightarrow$	S ⓪
		A ① ⓪
		x ② ① ⓪

Table 9.6: The 3NF of $\mathcal{N}_0$

Now assume that we want to add the production

A $\rightarrow$ A y ③

If we add this production to the original grammar $\mathcal{N}_0$ and then apply the 3NF construction, we obtain the grammar in Table 9.7. If we now compare the grammars in Table 9.6 and

Grammar $\mathcal{N}_2$					
Z_0	$\rightarrow$	S ⓪	Z_1	$\rightarrow$	① ⓪
		A Z_1			y ③ Z_1
		x ② Z_1			

Table 9.7: The 3NF of the modified grammar $\mathcal{N}_0$

Table 9.7, we find the following correspondences:

- Z_1 is defined exactly as required by the left-recursion removal in Def. 2.5.6 – applied to the 3NF grammar in Table 9.6!
- The only complication is that we have to replace the trailer ①⓪ by Z_1 also in all those productions of Z_0 that have been obtained by unfolding A. But this is obviously doable with little effort.

These short considerations demonstrate that the 3NF construction indeed can be adapted to incremental parser generation.

9.2.3 Why Simple Left-Recursion Elimination Fails

In Section 2.5 we have opted against the traditional form of left-recursion elimination that is mostly used in the literature and chosen a slightly more complex version instead (see Def. 2.5.6). Now we want to demonstrate that this choice is indeed mandatory.

1. The simple form does not guarantee the $LR(k)$ property.
2. The simple form only works for the *classical* but not for the *emulated LR parsing* algorithm.

Aspect (1) can be seen from the following counterexample:

$$\begin{array}{lcl} \mathrm{S} & \to & \mathrm{A\ b}\ \textcircled{1}\ \boxed{S} \\ \mathrm{A} & \to & \mathrm{A\ b}\ \textcircled{2}\ \boxed{A} \\ & & \mathrm{a}\ \textcircled{3}\ \boxed{A} \end{array}$$

According to Def. 4.0.4 this grammar is $LR(1)$, since $\boxed{S} = \{ \natural \}$ and $\boxed{A} = \{ b \}$. If we transform the grammar using the *simple* recursion removal, we obtain

$$\begin{array}{lcl} Z_0 & \to & \mathrm{S}\ \textcircled{0}\ \boxed{S} \\ & & \mathrm{A}\ Z_1\ \textcircled{0}\ \boxed{S} \\ & & \mathrm{a}\ \textcircled{3}\ Z_3\ Z_1\ \textcircled{0}\ \boxed{S} \end{array} \qquad \begin{array}{lcl} Z_1 & \to & \mathrm{b}\ \textcircled{1}\ \boxed{S} \\ Z_2 & \to & \mathrm{b}\ \textcircled{2}\ \boxed{A} \end{array} \qquad \begin{array}{lcl} Z_3 & \to & \mathrm{b}\ \textcircled{2}\ Z_3\ \boxed{A} \\ & & \varepsilon\ \boxed{A} \end{array}$$

Hence, Z_3 exhibits a conflict. By contrast, the complex variant of recursion removal generates the grammar

$$\begin{array}{lcl} Z_0 & \to & \mathrm{S}\ \textcircled{0}\ \boxed{S} \\ & & \mathrm{a}\ \textcircled{3}\ Z_3\ \textcircled{0}\ \boxed{S} \end{array} \qquad \begin{array}{lcl} Z_1 & \to & \mathrm{b}\ \textcircled{1}\ \boxed{S} \\ Z_2 & \to & \mathrm{b}\ \textcircled{2}\ \boxed{A} \\ Z_3 & \to & \mathrm{b}\ Z_4\ \boxed{S} \end{array} \qquad \begin{array}{lcl} Z_4 & \to & \textcircled{1}\ \boxed{S} \\ & & \textcircled{2}\ Z_3\ \boxed{S} \end{array}$$

Z_4 avoids the conflict. The explanation is simple: The symbol b follows A in the recursive case *and* in some application position. Hence, the simple variant cannot decide, when the recursion is finished (i.e., when the ε-production has to be used). The complex variant resolves this problem by way of left factoring in Z_3, which leads to the conflict-free Z_4.

Aspect (2) can also be seen from a little counterexample. In the *classical-LR variant* we could actually do *without the left-recursion removal*. However, by keeping it we retain the option to switch between the different parsing methods at will.

Example: Consider a maximally simplified variant of the classical grammar for arithmetic expressions. This grammar is shown in Table 9.8.

Grammar $\mathcal{E}_0$		
S	$\to$	E ①
E	$\to$	E + i ②
		i ③

Table 9.8: A left-recursive grammar

(a) The 3NF is shown in Table 9.9. (The intermediate Z_1 is no longer reachable and has therefore been eliminated.)

Grammar $\mathcal{E}_2$					
Z_0	$\to$	S ⓪	Z_2	$\to$	i ②
		E Z_3 ⓪	Z_3	$\to$	① $\boxed{S}$
		i ③ Z_3 ⓪			+ Z_2 Z_3

Table 9.9: Second normal form (using left-recursion removal)

If we apply any of the two parsing methods to the sample string "$i+i\,\natural$" we obtain the following derivation (where the *classical LR parsing* ignores the upper elements):

$$[Z_0]\; i \begin{bmatrix} ⓪ \\ Z_3 \\ ③ \end{bmatrix} \; ③ \begin{bmatrix} ⓪ \\ Z_3 \end{bmatrix} \; + \begin{bmatrix} ⓪ \\ Z_3 \\ Z_2 \end{bmatrix} \; i \begin{bmatrix} ⓪ \\ Z_3 \\ ② \end{bmatrix} \; ② \begin{bmatrix} ⓪ \\ Z_3 \end{bmatrix} \; \begin{bmatrix} ⓪ \\ ① \end{bmatrix} \; ① \; [⓪] \; ⓪ \; \natural$$

(b) On the other hand, if we apply a variant of the 2NF construction *without* left-recursion removal, we obtain the grammar in Table 9.10. Note that Z_3 now comes from the left factoring of

$$\begin{array}{lll} Z_0 & \rightarrow & \text{S } ⓪ \\ & & \text{E } ① \; ⓪ \\ & & \text{E } Z_1 \; ① \; ⓪ \\ & & \text{i } ③ \; ① \; ⓪ \end{array}$$

Grammar $\mathcal{E}_{2a}$					
Z_0	$\rightarrow$	S ⓪	Z_2	$\rightarrow$	i ②
		E Z_3	Z_3	$\rightarrow$	① ⓪ $\boxed{S}\rangle$
		i ③ ① ⓪			$+\; Z_2$ ① ⓪

Table 9.10: Second normal form (without left-recursion removal)

This variant leads essentially to the same *classical LR parsing* process as the grammar in Table 9.9. However, if we use this grammar with *emulated LR parsing*, it would not work, since the process would get stuck immediately:

$$[Z_0]\; i \begin{bmatrix} ⓪ \\ ① \\ ③ \end{bmatrix} \; ③ \begin{bmatrix} ⓪ \\ ① \end{bmatrix} \; ① \; [⓪] \; ⓪ + i \; \natural$$

This demonstrates that the variant with recursion removal is superior in the sense that it enables all parsing methods, whereas the variant without recursion removal only works for one of the variants.

9.3 Variations on the Parsing Process

Not only the presentation of the grammars and the transformation process of the grammars – that is, the parser generation – allow variations. Also the parsing algorithms can be implemented in different ways and be adapted to changing requirements. Most of these variations have to do in one way or another with "ambiguity".

1. When a grammar is $LR(k)$ with $k \geq 2$ we may consider using backtracking instead of k-lookahead.

2. For truly ambiguous grammars we need backtracking.

3. Errors are in some sense the worst form of ambiguity, since we do not even know, where in the string we should restart with what Z_i.

All these problems can essentially be handled with the same techniques. We demonstrate these for the case (1) and only sketch the pertinent adaptions for the other two cases.

9.3.1 k-Lookahead or Backtrack?

It is well-known that $LR(k)$ parsers are efficient for the case $k = 1$ but may become expensive for $k > 1$. To see how this affects our approach we consider an example that is derived from the (slightly modified) syntax of import declarations in OPAL [10]:

```
IMPORT A ONLY a1 a2 a3
       B ONLY b1 b2
```

When we encounter the identifier `B`, it is not yet clear, whether this is the next identifier on the ONLY-list of `A` or the name of the new import. The essence of this situation is captured by the grammar of Table 9.11.

Grammar $\mathcal{I}_0$		
S	$\rightarrow$	L ①
L	$\rightarrow$	I L ②
		I ③
I	$\rightarrow$	i o N ④
N	$\rightarrow$	i N ⑤
		i ⑥

Table 9.11: A Non-$LR(1)$ Grammar

The 3NF is presented in Table 9.12. (Note that Z_1 and Z_4 are not reachable and therfore have been eliminated.)

Grammar $\mathcal{I}_3$								
Z_0	$\rightarrow$	S ⓪	Z_2	$\rightarrow$	o Z_3	Z_5	$\rightarrow$	L ②
		L ① ⓪	Z_3	$\rightarrow$	N ④			I Z_5 ②
		I Z_5 ① ⓪			i Z_6 ④			i Z_2 Z_5 ②
		i Z_2 Z_5 ① ⓪						③ $\boxed{L}$
						Z_6	$\rightarrow$	N ⑤
								i Z_6 ⑤
								⑥ $\boxed{N}$
$\boxed{S}$	-->	$\natural$	$\boxed{I}$	-->	Z_5 $\boxed{S}$	$\boxed{N}$	-->	$\boxed{I}$
$\boxed{L}$	-->	$\boxed{S}$			Z_5 $\boxed{L}$			

Table 9.12: Third normal form

As can be seen here, the continuation information suffices for Z_5 to disambiguate all productions, but in the rules for Z_6 the shift/reduce conflict still cannot be resolved, since i is also a leading symbol of the continuation $\boxed{N}$.

However, when we look further ahead then we find that in the production $Z_6 \rightarrow iZ_6$⑤ the next terminal symbol is again 'i' or '$\natural$ ', whereas in the second production $Z_6 \rightarrow$ ⑥ $\boxed{N}$ the next terminal symbol is 'o', since this is the first symbol generated by Z_2. Hence, with *two* symbols lookahead the conflict is resolved: We have an $LR(2)$ parser. This shows the following property.

Proposition. *Our method automatically generates LR(k) parsers (for arbitrary k) without any additional effort!* □

All we have to do is to analyze the continuation grammar further ahead. This is easy, since its description already exists in the 3NF. But above all, this additional effort only has to be

invested, if a 1-conflict is actually encountered! This property contrasts our approach nicely from the traditional way of proceeding.

In spite of this simplicity it is, however, still questionable, whether the effort for this extra analysis is worth the cost. Firstly, the problem occurs very rarely. Secondly, our parser is a full backtrack parser. Hence, in those cases where the two-symbol lookahead is needed it will simply employ some backtracking in order to perform a successful parse. This one-step backtrack has negligeable cost.

We illustrate this effect by a parse of the simple term '$i\ o\ i\ i\ o\ i\ i\ \natural$'. We look at the parsing process as defined in Def. 5.2.1 with an obvious generalization for ambiguous parses: when an ambiguous production is encountered, all possibilities are recorded. Whenever there is a potential reduce action, we list it as a candidate (with ε, that is, "nothing" as the other alternative).

$$[Z_0]\ i\ \begin{bmatrix}⓪\\①\\Z_5\\Z_2\end{bmatrix}\ o\ \begin{bmatrix}⓪\\①\\Z_5\\Z_3\end{bmatrix}\ i\ \begin{bmatrix}⓪\\①\\Z_5\\④\\Z_6\end{bmatrix}\ \{\varepsilon \mid ⑥\}\ \left[\begin{array}{c|c}⓪&\\①&⓪\\Z_5&①\\④&Z_5\\Z_6&④\end{array}\right]\ \{\varepsilon \mid ④\}\ \left[\begin{array}{c|c}⓪&\\①&\\Z_5&⓪\\④&①\\Z_6&Z_5\end{array}\right]\ i\ \left[\begin{array}{c|c}⓪&\\①&⓪\\Z_5&①\\④&②\\⑤&Z_5\\Z_6&Z_2\end{array}\right]\ o\ \ldots$$

At this point it is evident that the first alternative is not valid, because 'o' is an impossible symbol for Z_6. Hence, we can eliminate all first choices before we proceed:

$$i\ o\ i\ ⑥\ ④\ i\ o\ \begin{bmatrix}⓪\\①\\②\\Z_5\\Z_3\end{bmatrix}\ i\ \begin{bmatrix}⓪\\①\\②\\Z_5\\④\\Z_6\end{bmatrix}\ \{\varepsilon \mid ⑥\}\ \left[\begin{array}{c|c}⓪&\\①&⓪\\②&①\\Z_5&②\\④&Z_5\\Z_6&④\end{array}\right]\ \{\varepsilon \mid ④\}\ \left[\begin{array}{c|c}⓪&\\①&\\②&⓪\\Z_5&①\\④&②\\Z_6&Z_5\end{array}\right]\ i\ \left[\begin{array}{c|c}⓪&\\①&⓪\\②&①\\Z_5&②\\④&②\\⑤&Z_5\\Z_6&Z_2\end{array}\right]\ \natural$$

This time the second variant is excluded, because the symbol '$\natural$' is impossible for Z_2. Hence we can eliminate all second choices before we proceed.

$$i\ o\ i\ ⑥\ ④\ i\ o\ i\ i\ \begin{bmatrix}⓪\\①\\②\\Z_5\\④\\⑤\\Z_6\end{bmatrix}\ ⑥\ \begin{bmatrix}⓪\\①\\②\\Z_5\\④\\⑤\end{bmatrix}\ ⑤\ \begin{bmatrix}⓪\\①\\②\\Z_5\\④\end{bmatrix}\ ④\ \begin{bmatrix}⓪\\①\\②\\Z_5\end{bmatrix}\ ③\ \begin{bmatrix}⓪\\①\\②\end{bmatrix}\ ②\ \begin{bmatrix}⓪\\①\end{bmatrix}\ ①\ [⓪]\ \natural$$

The realization is slightly more complex than indicated by this example: it can happen that one of the branches is split further, in which case we cannot simply "flatten" the choice from pairs to triples but rather have to use nested choices in order to allow a correct backwards elimination.

We note in passing that the internal organization of lists in a functional language like OPAL effects the desired sharing of sublists automatically. So the implementation becomes extremely simple in such languages.

9.3.2 Multi-Pass Parsing Versus Generalized *LR* Parsing

The above realization of the parsing process has a major disadvantage: the simultaneous performance of the multiple parses will frequently encounter situations, where several threads lead to

the same Z_i. This happens e.g. for if-then-else constructs and the like, where all possible parses perform the same local subparse. There are several means to avoid this waste of work.

- In order not to duplicate the work one could realize the various threads as a more complex data structure, viz. a dag. This was e.g. worked out by M. Tomita under the concept of *Genralized LR parsing* in [36] (see also [32], where further references are given). Bates and Lavie [4] use a similar technique, termed forest-structured stacks.
- In practice the Genralized LR parsing with its complex data handling might not be the most efficient way of proceeding. Using a *multi-pass parsing* scheme may actually be more efficient. This technique will be sketched in the following.

The essential point in ***multi-pass parsing*** is that we actually perform the reductions – at least those that are unambiguously possible. Whenever we encounter an ambiguity we proceed – as demonstrated in Section 9.3.1 above – with the set of possible states Z_i. From then on many steps will create new sets of states, but very frequently we will encounter locally unambiguous substrings, where the sets are reduced to simple nonterminals.

This way we can perform all unique subparses once and for all. As a result we obtain a maximally shortended sentential form, which focusses only on the actual ambiguities. Since this sentential form will in general be very short, we can easily accept the overhead of backtrack parsing for it.

This idea can actually be improved further: as has been shown in Section 9.3.1 above we will find out after a while, that certain members of the set of possible states are inconsistent with the next input symbol(s). This allows a backward elimination of all their predecessors. A more efficient variant is, however, a second pass after the first one has been finished. In this backward pass *all* inconsistencies are removed in one sweep.

As has been illustrated by the example of the previous section this second pass will solve all problems in the case of $LR(k)$ grammars also for $k \geq 2$. So it will only happen for some truly ambiguous grammars that real backtracking is needed.

We note in passing that the same principles are also applicable to the design of *parallel parsers* (see [25]).

Ambiguous grammars. These principles also apply to truly ambiguous grammars. The important aspect from the point of view of efficiency is that the splitting into potential branches is managed in such a way that common subparses still are done only once.

We illustrate this by an example that is derived from the (slightly modified) syntax of LET-declarations in languages such as OPAL, HASKELL or ML.

```
LET b     = f a
    h x y = g (f x) y
    z     = h a b
IN ...
```

Without additional conventions such as HASKELL's "offside rule" there is no way of parsing this program fragment uniquely. For example, it is by no means clear, whether `a`, `h` and `x` are still arguments of `f` or the leading function symbol of the next declaration.

The essence of this situation is captured in the grammar of Table 9.13.

When we transform this grammar into its 3NF, then we obtain one Z_i that exhibits a conflict:

Grammar $\mathcal{L}_0$		
S	$\rightarrow$	L ①
L	$\rightarrow$	D L ②
		D ③
D	$\rightarrow$	E = E ④
E	$\rightarrow$	T E ⑤
		T ⑥
T	$\rightarrow$	(E) ⑦
		i ⑧

Table 9.13: An ambiguous grammar

$$
\begin{array}{lcl}
Z_8 & \rightarrow & \mathrm{E}\ ⑤ \\
 & & \mathrm{T}\ Z_8\ ⑤ \\
 & & (\ Z_5\ Z_8\ ⑤ \\
 & & \mathrm{i}\ ⑧\ Z_8\ ⑤ \\
 & & ⑥\ \boxed{E}\rangle
\end{array}
$$

Since $\boxed{E}\rangle \overset{*}{\dashrightarrow} iZ_8\ldots$, there is a shift/reduce conflict. Moreover, we can immediately see that this conflict cannot be resolved by any k-lookahead. In other words, the grammar is *not* $LR(k)$ for any k.

Even though the backtracking is unavoidable here, it has at least been reduced to an absolute minimum: all other Z_i are uniquely determined! So the multi-pass paradigm will shorten the input string to a very short sentential form, before it enters the true backtracking process.

This is the clue for the feasability of a simple method for type-dependent overload resolution: for the ambiguous constructs one may generate *all* possible parses, apply the typing algorithm to all resulting trees and throw away those that are not type-correct.

9.3.3 Error Handling

In the literature there are many treatments of the integration of error handling into parsing (see e.g. [33]). In connection with the paradigm of functional parsing the subject has been extensively treated in [22]. The special form of our 3NF allows a very systematic and straightforward error handling.

- As has already been pointed out in Section 9.3.1 the error handling can be performed along the same lines as the treatment of non-$LR(1)$ grammars.

  As a matter of fact, we might even consider generating a second 3NF for the inverse direction, that is, for parsing from right to left. Then the second pass would encapsulate the error positions with high accuracy.

- But we also have other possibilities. For example, [1] points out that during error recovery we need "stop sets", that is, collections of terminals, the occurrence of which indicates that we have to pop up from a deeper error level. A typical instance is the occurrence of an unexpected '`else`' during the parsing of a `then`-clause. This might indicate that we are e.g. lacking a closing paranthesis. This error is recognized, if the symbol '`else`' is in the stop set (where it has been put upon the parsing of the symbol '`then`').

  In our parsing process as described by Def. 5.2.1 in Section 5.2 this stop set is directly obtainable from the intermediate (composed) functions $[\varphi]$.

9.3.4 Integrating Context-Sensitive Aspects

In the functional-programming community it is well-known that the flexibility provided by the functional parsing as defined in Section 5 also allows – at least to some extent – the integration of context-dependent grammars. We cannot go into details here but merely want to point out the pertinent principle. The parsers as defined in Section 5 convert a production such as

$$A \quad \rightarrow \quad B\ C\ \textcircled{i}$$

into a function definition such as

DEF $\texttt{A} \ == \ \texttt{B} \ ; \ \texttt{C} \ ; \ \texttt{action}_\texttt{i}$

If we want to integrate context-dependent information (such as typing information, potential offside positions etc.), we may extend the above function by an appropriate parameter. This leads to definitions of the kind

DEF $\texttt{A(t)} \ == \ \texttt{B}(\texttt{h}_1(\texttt{t})) \ ; \ \texttt{C}(\texttt{h}_2(\texttt{t})) \ ; \ \texttt{action}_\texttt{i}(\texttt{h}_3(\texttt{t}))$

The auxiliary functions $\texttt{h}_\texttt{i}$ propagate the context-dependent information to the various parts of the parser.

It is a simple exercise to extend the notations for grammars such that "nonterminals" of the form $A(t)$, $B(h(t))$ etc. are allowed. For these extended forms our constructions go through analogously. Even though this is far from providing the full power of attribute grammars, it is a *simple* tool that solves a number of practical problems.

Chapter 10

Sketch of a Non-standard Application: Type-Dependent Overload Resolution and Mixfix Operators

As a nonstandard and nontrivial application of our approach we consider the use of parsing for type analysis combined with user-defined mixfix operations. We want to allow the programmer to write down declarations of the form

DEF $\_ + \_ : \texttt{Int} \times \texttt{Int} \to \texttt{Int}$ $[\![\texttt{left-associative}]\!]$ $[\![+ \prec *]\!]$
DEF $\_ + \_ : \texttt{Matrix} \times \texttt{Matrix} \to \texttt{Matrix}$ $[\![\texttt{left-associative}]\!]$ $[\![+ \prec *]\!]$
DEF $\texttt{\_with\_at\_} : \texttt{Array}[\alpha] \times \alpha \times \texttt{Nat} \to \texttt{Array}[\alpha]$
DEF $\_ \circ \_ : (\beta \to \gamma) \times (\alpha \to \beta) \to (\alpha \to \gamma)$

That is, we need mixfix operators, higher-order functions, overloading and polymorphism. Moreover, there have to be modularisation concepts available such as classes, modules or structures.[1]

The fundamental principle underlying this concept is given by an observation that has already been made by the ADJ group in 1977 [35]: every grammar induces the *signature* of an algebraic structure (which makes the notion of *abstract syntax* more precise). This observation also works the other way round: *every signature of a structure induces a grammar.*

The correspondence is simple: the types of the signature correspond to the nonterminals of the grammar. And the operations (in the above mixfix examples: the fragments of the operations) correspond to the terminal symbols.

From the above wish list we can deduce a number of requirements:

- The analyzer cannot be constrained to $LR(1)$ grammars, since the programmers of such modules will not be willing to obey such constraints and often not even be capable of recognizing them. Hence, we need the *full power of backtrack parsing.* This is provided by our parser as described in Section 5.
- The modularization entails that a module A can import a module B. At this moment the grammar for B is extended by the grammar for A. Hence, we need *incremental*

[1] With the exception of the arbitrary mixfix notations the parser of our language OPAL [24, 26] can cope with these features.

parser generation. Section 9.2.2 has shown that this requirement can be easily met in our framework.

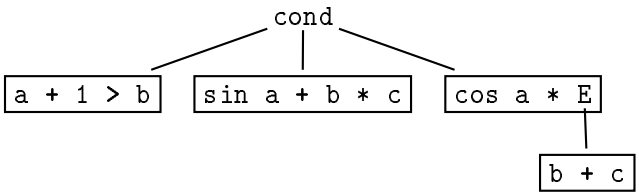
- In some situations it is not known a priori, which role a certain identifier plays. This happens, for example, in pattern-based definitions that are characterisitic for functional programming. Here it may not be immediately clear, whether an identifier is a parameter or a fragment of a mixfix operator. So we obtain something like *"fuzzy" terminal symbols.* The techniques presented in Sections 9.3.1 and 9.3.2 can be adapted easily to this slight complication.

- When we translate the above ADJ-correspondences to the situation of polymorphic functions, we run into a concept of *"generic nonterminals"*. This is best dealt with by using context-dependent parsing (as sketched in Section 9.3.4).

In practice such a system will be based on a "coarse" grammar that takes care of the basic structure of a program: if-then-else, parantheses and the like. As a result we obtain a coarse syntax tree that contains many sequences of yet unparsed fragments. For example, a program fragment such as

IF a + 1 > b THEN sin a + b * c ELSE cos a * (b + c) FI

may lead to a tree of the kind



Then we have to parse the subexpressions in this tree using the grammar that is deduced from the function declarations in the program modules.

Fortunately, the general need for backtracking does not hurt too much, since it is only applied to very short terms. This is the advantage of first applying the coarse structure parse.

However, it can also be seen that this approach requires an intricate interaction between the parser and the Hindley-Milner type algorithm.

Even though the various aspects of our approach – as discussed in the previous sections – can be used to enable this application, the concrete realiziation still requires some research effort.

Chapter 11

Conclusion

We have presented a technique for the simple and flexible generation of (LL and) LR parsers. The idea is to transform the grammar into a particularly suitable form and then use the means of recursive-descent parsing.

The recursive-descent parsing from Section 5.1 is an extremely simple mechanism, which is attractive, short and elegant. But we have to keep a few *problems* in mind:

- To start with the biggest drawback, *LL parsing does not work for grammars with left-recursive productions.*

  However, our construction eliminates left recursion!

- The parsing process may be *slow*, because it may perform some *backtacking* before finding the right production for a given nonterminal.

  However, our 3NF construction eliminates all possible shift/shift conflicts. And it also performs the lookahead for reductions. Therefore, true backtracking may only occur, when the grammar is not $LR(k)$ – and then we actually want the backtrack power!

- We have not provided a decent treatment of errors.

  But this can be easily added (as described in [22]). Moreover, our brief discussion in Section 9.3.3 indicates that the error handling is even fostered by our approach.

Simple as it may be, the technique has a number of important *advantages*:

- We have a *full backtrack parser* that works for arbitrary context-free grammars, not just for $LL(k)$ or $LR(k)$ grammars.

- Our parser yields the first successful parse, but it can easily be varied such that it yields the set of all parses instead.

- One can easily change the viewpoint of a grammar from a parsing function that is directly executed to a data structure that is interpreted. Then one can switch back and forth between transformations *of* the grammar and parsing *with* the grammar. This way, we can apply "lazy" grammar transformations triggered by the parsing process itself (as sketched in Section 6).

- The higher-order functions ';', '|', etc., are defined once and for all and thereafter can be fetched from a library. Therefore, all one has to do in order to obtain a parser is transliterate the given grammar.

- Since the parsing operators are part of the library, we can exploit the full power of the programming language, e.g. for writing the semantic actions. This is much more flexible than working with external tools such as parser generators. For instance, we can easily add attribute computations to the parser (see Section 10).
- Due to its simplicity, our parser can be easily verified (or even be considered "obviously correct".) The verification is in fact outlined in the Appendix.
- Finally, the special form of our 3NF allows us to simplify the implementation of our parser even further (see Appendix).

Therefore we feel that the combination of our grammar transformations with this highly elegant, simple and flexible implementation scheme provides an extremely powerful parsing concept.

Acknowledgement. The ideas for this approach evolved over many years, also stimulated by teaching principles of compiler construction to students. During that time several people have provided critical assessments of the proceeding, notably Wolfram Schulte, Andreas Fett and, above all, Carola Gerke. I am especially grateful to Doug Smith, Cordell Green and other members of Kestrel Institute; the stimulating research environment at this place and the intensive discussions about the formalization of software development lead to the final shaping of this paper. A number of valuable comments were provided by the members of IFIP WG 2.1. Niamh Warde helped to formulate parts of the paper.

Appendix A

Appendix: More About (Functional) Parsing

Since this is not a paper on functional parsing, we have sketched its basic principles only very briefly in Section 5. For the sake of completeness we want to fill in the missing details in this Appendix.

Essentially a parser should take an input string and yield a corresponding tree (or indicate failure). However, this is only true for the overall parsing of the full input; in all intermediate parsing stages we only turn some initial fragment of the input into a tree. The unread remainder of the input therefore has to be returned as a result as well. Finally, programming is slightly facilitated if we add — for reasons of symmetry — the type `Tree` also as an argument. Hence, a parser takes as input a partial tree and a string and yields as output a properly extended tree plus the unread remainder of the string:

```
TYPE Parser == Tree × String → Tree × String
```

Since this is a function type, all operations working on the type `Parser` are higher-order functions. Next we look at the definitions of the operators ";", "|", etc. We can explain them on the basis of a more elementary operation '`A fby (S,F)`', which essentially connects the first parser `A` with two possible continuation parsers: `S` for the success and `F` for the failure of `A`.

```
FUN ; : Parser × Parser → Parser    -- sequential composition
DEF A;B == A fby (B,fail)

FUN | : Parser × Parser → Parser    -- alternative parsers (not commutative)
DEF A|B == A fby (id,B)
```

Using these connectors we can also introduce the Kleene star and its "at-least-once" variant:

```
FUN * : Parser → Parser    -- Kleene star
DEF A* == A fby (A*,id)

FUN + : Parser → Parser    -- "at least once"
DEF A+ == A ; A*
```

So all we need to understand are three operations, viz. '`fby`', '`fail`' and the lifting operator '!'. On this basis we can now program the basic operations.

```
FUN fby : Parser × Parser × Parser → Parser                -- continuations
DEF A fby (SuccCont, FailCont)(OldTree, Input) ==
    LET (NewTree, Remainder) == A(OldTree, Input)
    IN
    IF okay?(NewTree) THEN SuccCont(NewTree, Remainder)
    IF fail?(NewTree) THEN FailCont(OldTree, Input)       -- backtrack!
    FI
```

The parser '`fail`' merely returns a failure indication (note the overloading between the parser called `fail` and the tree called `fail`). Note: we need to return some string for reasons of type conformity; for this purpose, the empty string is as good as any other.

```
FUN fail : Parser
DEF fail(OldTree, Input) == (fail, ◊)
```

Terminal symbols are "lifted" to parsers which simply look at the next input token `x` to check whether it is the expected one; if not, a failure indication is returned. (The operation '`::`' on trees will be discussed in a moment.)

```
FUN ! : Token → Parser                          -- lift tokens
DEF (t!)(OldTree, ◊) == (fail, ◊)               -- unexpected end of input
DEF (t!)(OldTree, x::Remainder) ==
    IF t = x THEN (OldTree::x, Remainder)       -- input symbol shifted to tree
    IF t ≠ x THEN (fail, ◊)                     -- unexpected input symbol
    FI
```

We do not want to go into the details of the representation or the construction of trees. If we adopt the view from Section 2.3, we can actually realize the operation '`t::x`' by simply appending the token `x` to the tree (i.e. string) `t`. The operations $\texttt{action}_i$ also do nothing but append their number to the tree, that is, $\texttt{action}_i(\texttt{Tree}, \texttt{Input}) = (\texttt{Tree}::ⓘ, \texttt{Input})$.[1]

For the special form of the 3NF we could actually simplify the operator '`A | B`': we write a case distinction for the leading terminal symbols and thus avoid the one-step backtrackings with the subseqeunt `fail` tests.

We conclude this brief sketch by noting that the concrete programming that we have chosen is not obligatory; the details can easily be varied to suit different tastes and styles. (For example, Hutton and Meijer [14] design their operators such that they fulfil the monad laws that recently have become so popular in functional programming).

Correctness. Even though the above functional parser is so straightforward that its correctness appears to be evident, we present a verifying derivation for the sake of completeness.

We start from an initial specification that directly represents the correspondence between functional parsers and derivations: A parser `P` transforms a string `s` into a tree `t`. To ease the presentation we consider trees in their postorder form, i.e. as strings for the number-augmented grammar $\mathcal{G}'$ (as described in Section 2.3). Moreover, for this initial specification we actually employ a type `Parser` as we would like to have it: $\texttt{Parser} = \texttt{String} \to \texttt{Tree}$. With the help of the filter function φ that eliminates the numbers ⓘ we can then formulate the initial specification:

AXM $\texttt{P}(\texttt{s}) = \texttt{t} \Longrightarrow \texttt{t} \in \mathcal{L}_{\texttt{P}}(\mathcal{G}') \wedge \varphi(\texttt{t}) = \texttt{s}$

From this definition we can e.g. deduce the following properties:

[1] Of course, in paractical implementations one will realize the actions ⓘ by actually building the trees right away.

$$
\begin{aligned}
&(\mathtt{A};\mathtt{B})(\mathtt{s}) \;==\; \mathtt{t} \\
&\Rightarrow\; \mathtt{t} \in \mathcal{L}_{\mathtt{AB}}(\mathcal{G}') \wedge \varphi(\mathtt{t}) = \mathtt{s} \\
&\Rightarrow\; \exists\, \mathtt{t}_1 \in \mathcal{L}_{\mathtt{A}}(\mathcal{G}'),\ \mathtt{t}_2 \in \mathcal{L}_{\mathtt{B}}(\mathcal{G}')\ . \\
&\quad \mathtt{t} = \mathtt{t}_1 \dashv\vdash \mathtt{t}_2 \;\wedge\; \varphi(\mathtt{t}) = \varphi(\mathtt{t}_1 \dashv\vdash \mathtt{t}_2) = \varphi(\mathtt{t}_1) \dashv\vdash \varphi(\mathtt{t}_2) = \mathtt{s} \\
&\Rightarrow\; \exists\, \mathtt{t}_1 \in \mathcal{L}_{\mathtt{A}}(\mathcal{G}'),\ \mathtt{t}_2 \in \mathcal{L}_{\mathtt{B}}(\mathcal{G}'),\ \mathtt{s}_1,\ \mathtt{s}_2\ . \\
&\quad \mathtt{t} = \mathtt{t}_1 \dashv\vdash \mathtt{t}_2 \;\wedge\; = \mathtt{s}_1 = \varphi(\mathtt{t}_1) \;\wedge\; \mathtt{s}_2 = \varphi(\mathtt{t}_2) \;\wedge\; \mathtt{s} = \mathtt{s}_1 \dashv\vdash \mathtt{s}_2 \\
&\Rightarrow\; \exists\, \mathtt{s}_1,\ \mathtt{s}_2\ . \\
&\quad \mathtt{s} = \mathtt{s}_1 \dashv\vdash \mathtt{s}_2 \;\wedge\; \mathtt{t} = \mathtt{A}(\mathtt{s}_1) \dashv\vdash \mathtt{B}(\mathtt{s}_2)
\end{aligned}
$$

Summarizing: When there is a parse $\mathtt{AB} \stackrel{*}{\Rightarrow} \mathtt{s}$ then this parse fulfils the property

$$
\begin{aligned}
(\mathtt{A};\mathtt{B})(\mathtt{s}) = \mathtt{t} \Rightarrow \quad & \exists\, \mathtt{s}_1,\ \mathtt{s}_2\ .\ \mathtt{s} = \mathtt{s}_1 \dashv\vdash \mathtt{s}_2 \;\wedge \\
& (\mathtt{A};\mathtt{B})(\mathtt{s}) = \mathtt{A}(\mathtt{s}_1) \dashv\vdash \mathtt{B}(\mathtt{s}_2)
\end{aligned}
$$

To ease readability we consider from now on only the well-defined case and therefore omit the premise $(\mathtt{A};\mathtt{B})(\mathtt{s}) = \mathtt{t} \Rightarrow \ldots$ in all further specifications.

This specification evidently suffers from the existential quantifier, which makes it non-constructive. Existential quantifiers can be converted into Skolem functions, which leads to the new specification

$$
(\mathtt{A};\mathtt{B})(\mathtt{s}) = \mathtt{A}(\mathtt{s}_1) \dashv\vdash \mathtt{B}(\mathtt{s}_2) \;\wedge\; \mathtt{s} = \mathtt{s}_1 \dashv\vdash \mathtt{s}_2 \quad \text{WHERE } \mathtt{s}_2 = h_A(\mathtt{s})
$$

Now we embed every parser $\mathtt{P}$ into a new parser $\widehat{\mathtt{P}}$ that also returns the value of the corresponding function h_P. Moreover, $\widehat{\mathtt{P}}$ does not yield a simple tree, but it rather extends a given tree:

$$
\text{DEF } \widehat{\mathtt{P}}(\mathtt{t},\mathtt{s}) \;==\; (\mathtt{t} \dashv\vdash \mathtt{P}(\mathtt{s}_1),\ h_P(\mathtt{s})) \quad \text{WHERE } \mathtt{s} = \mathtt{s}_1 \dashv\vdash h_P(\mathtt{s})
$$

Note that this specification is constructive if h_P is.

This clearly is an embedding of $\mathtt{P}$, because we have $\mathtt{P}(\mathtt{s}) = \pi_1(\widehat{\mathtt{P}}(\Diamond, \mathtt{s}))$ requiring $\pi_2(\widehat{\mathtt{P}}(\Diamond, \mathtt{s})) = \Diamond$. That is, the first component is the result of the full parse (provided that the second result $h_P(\mathtt{s})$ is empty).

Now take $\mathtt{P} = \mathtt{A};\mathtt{B}$. Then we have the derivation

$$
\begin{array}{lll}
\text{DEF } \widehat{\mathtt{P}}(\mathtt{t},\mathtt{s}) & == & \widehat{(\mathtt{A};\mathtt{B})}(\mathtt{t},\mathtt{s}) \\
& == & (\mathtt{t} \dashv\!\vdash \mathtt{P}(\mathtt{s}_1),\ h_P(\mathtt{s})) \qquad \text{WHERE } \mathtt{s} = \mathtt{s}_1 \dashv\!\vdash h_P(\mathtt{s}) \\
& == & (\mathtt{t} \dashv\!\vdash (\mathtt{A};\mathtt{B})(\mathtt{s}_1),\ h_P(\mathtt{s})) \qquad \text{WHERE } \mathtt{s} = \mathtt{s}_1 \dashv\!\vdash h_P(\mathtt{s}) \\
& == & (\mathtt{t} \dashv\!\vdash \mathtt{A}(\mathtt{s}_{1_1}) \dashv\!\vdash \mathtt{B}(\mathtt{s}_{1_2}),\ h_P(\mathtt{s})) \qquad \text{WHERE } \mathtt{s} = \mathtt{s}_1 \dashv\!\vdash h_P(\mathtt{s}) \\
& & \qquad\qquad\qquad\qquad\quad\ \ \mathtt{s}_1 = \mathtt{s}_{1_1} \dashv\!\vdash \mathtt{s}_{1_2} \\
& & \qquad\qquad\qquad\qquad\quad\ \ \mathtt{s}_{1_2} = h_A(\mathtt{s}_1) \\
& == & (\mathtt{t}_1 \dashv\!\vdash \mathtt{B}(\mathtt{s}_{1_2}),\ h_P(\mathtt{s})) \qquad \text{WHERE } \mathtt{s} = \mathtt{s}_1 \dashv\!\vdash h_P(\mathtt{s}) \\
& & \qquad\qquad\qquad\qquad\quad\ \ (\mathtt{t}_1,\mathtt{s}') = \widehat{\mathtt{A}}(\mathtt{t},\mathtt{s}) \\
& & \qquad\qquad\qquad\qquad\quad\ \ \mathtt{s}' = h_A(\mathtt{s}) \\
& & \qquad\qquad\qquad\qquad\quad\ \ \mathtt{s} = \mathtt{s}_{1_1} \dashv\!\vdash \mathtt{s}' \\
& & \qquad\qquad\qquad\qquad\quad\ \ \mathtt{s}_1 = \mathtt{s}_{1_1} \dashv\!\vdash \mathtt{s}_{1_2} \\
& & \qquad\qquad\qquad\qquad\quad\ \ \mathtt{s}_{1_2} = h_A(\mathtt{s}_1) \\
& == & (\mathtt{t}_1 \dashv\!\vdash \mathtt{B}(\mathtt{s}_{1_2}),\ h_P(\mathtt{s})) \qquad \text{WHERE } \mathtt{s} = \mathtt{s}_{1_1} \dashv\!\vdash \mathtt{s}_{1_2} \dashv\!\vdash h_P(\mathtt{s}) \\
& & \qquad\qquad\qquad\qquad\quad\ \ (\mathtt{t}_1,\mathtt{s}') = \widehat{\mathtt{A}}(\mathtt{t},\mathtt{s}) \\
& & \qquad\qquad\qquad\qquad\quad\ \ \mathtt{s}' = h_A(\mathtt{s}) \\
& & \qquad\qquad\qquad\qquad\quad\ \ \mathtt{s} = \mathtt{s}_{1_1} \dashv\!\vdash \mathtt{s}' \\
& == & (\mathtt{t}_1 \dashv\!\vdash \mathtt{B}(\mathtt{s}_{1_2}),\ h_P(\mathtt{s})) \qquad \text{WHERE } (\mathtt{t}_1,\mathtt{s}') = \widehat{\mathtt{A}}(\mathtt{t},\mathtt{s}) \\
& & \qquad\qquad\qquad\qquad\quad\ \ \mathtt{s}' = \mathtt{s}_{1_2} \dashv\!\vdash h_P(\mathtt{s}) \\
& == & (\mathtt{t}_2,\ h_P(\mathtt{s})) \qquad \text{WHERE } (\mathtt{t}_1,\mathtt{s}') = \widehat{\mathtt{A}}(\mathtt{t},\mathtt{s}) \\
& & \qquad\qquad\qquad\qquad\quad\ \ \mathtt{s}' = \mathtt{s}_{1_2} \dashv\!\vdash h_P(\mathtt{s}) \\
& & \qquad\qquad\qquad\qquad\quad\ \ (\mathtt{t}_2,\mathtt{s}'') = \widehat{\mathtt{B}}(\mathtt{t}_1,\mathtt{s}') \\
& & \qquad\qquad\qquad\qquad\quad\ \ \mathtt{s}'' = h_B(\mathtt{s}') \\
& & \qquad\qquad\qquad\qquad\quad\ \ \mathtt{s}' = \mathtt{s}_{1_2} \dashv\!\vdash \mathtt{s}'' \\
& == & (\mathtt{t}_2,\ h_P(\mathtt{s})) \qquad \text{WHERE } (\mathtt{t}_1,\mathtt{s}') = \widehat{\mathtt{A}}(\mathtt{t},\mathtt{s}) \\
& & \qquad\qquad\qquad\qquad\quad\ \ (\mathtt{t}_2,\mathtt{s}'') = \widehat{\mathtt{B}}(\mathtt{t}_1,\mathtt{s}') \\
& & \qquad\qquad\qquad\qquad\quad\ \ \mathtt{s}'' = h_P(\mathtt{s}) \\
& == & (\mathtt{t}_2,\ \mathtt{s}'') \qquad \text{WHERE } (\mathtt{t}_1,\mathtt{s}') = \widehat{\mathtt{A}}(\mathtt{t},\mathtt{s}) \\
& & \qquad\qquad\qquad\qquad\quad\ \ (\mathtt{t}_2,\mathtt{s}'') = \widehat{\mathtt{B}}(\mathtt{t}_1,\mathtt{s}')
\end{array}
$$

This way we have formally derived the definition that was given above – via the auxiliary function `fby` – for the operator ';'. The other operators can be derived analogously.

Improved implementation. In Section 5.3 we have employed a backward search in order to find out, which Z_i or $\textcircled{i}$ has to be taken as the next bottom element of the function $[\varphi]$. This is, of course, unnecessarily inefficient and should be implemented in a more elaborate way. The clue to the improvement is – as in many cases – the introduction of a data structure for "remembering" the necessary information.

Consider the reductions $(*_1)$ and $(*_2)$ in Section 5.3. Their effect is actually twofold: they reduce a substring (the handle α) to a symbol. *And* they also eliminate all intermediate functions inside α. Even if we do not apply the reduction explicitly (but rather only add the action symbol $\textcircled{i}$ to the output), we still have to perform the elimination of the intermediate functions.

Therefore the idea suggests itself to keep these functions (actually only the bottom ones) in a separate list. The resulting process is illustrated by the following example; the corresponding

rules will be given in a moment.

$$
\begin{array}{ll}
 & [Z_0] * i = i \natural \\
\rightsquigarrow & * [Z_0/Z_3]\, i = i \natural \\
\rightsquigarrow & * i\, [Z_0/Z_3/④]\, = i \natural \\
\rightsquigarrow & * i\, ④\, [Z_0/Z_3/⑤]\, = i \natural \\
\rightsquigarrow & * i\, ④\, ⑤\, [Z_0/Z_3/③]\, = i \natural \\
\rightsquigarrow & * i\, ④\, ⑤\, ③\, [Z_0/Z_4]\, = i \natural \\
\rightsquigarrow & * i\, ④\, ⑤\, ③ = [Z_0/Z_4/Z_2]\, i \natural \\
\rightsquigarrow & * i\, ④\, ⑤\, ③ = i\, [Z_0/Z_4/Z_2/④]\, \natural \\
\rightsquigarrow & * i\, ④\, ⑤\, ③ = i\, ④\, [Z_0/Z_4/Z_2/⑤]\, \natural \\
\rightsquigarrow & * i\, ④\, ⑤\, ③ = i\, ④\, ⑤\, [Z_0/Z_4/Z_2/①]\, \natural \\
\rightsquigarrow & * i\, ④\, ⑤\, ③ = i\, ④\, ⑤\, ①\, [Z_0]\, \natural \\
\rightsquigarrow & * i\, ④\, ⑤\, ③ = i\, ④\, ⑤\, ①\, [Z_0/⓪]\, \natural \\
\rightsquigarrow & * i\, ④\, ⑤\, ③ = i\, ④\, ⑤\, ①\, ⓪
\end{array}
$$

The specification of these steps is quite simple; it consists of three main rules:

1. We can perform the step (with terminal symbol t and reststring u)

$$
\begin{array}{rrcl}
 & \ldots\, [\ldots/Z_i]\, t\, u & \rightsquigarrow & \ldots\, t\, [\ldots/Z_i/Z_j]\, u \\
\textit{or} & \ldots\, [\ldots/Z_i]\, t\, u & \rightsquigarrow & \ldots\, t\, [\ldots/Z_i/ⓘ]\, u
\end{array}
$$

   if there is a production $(Z_i \to t Z_j \ldots)$ or $(Z_i \to t\, ⓘ \ldots)$ in the 3NF grammar $\mathcal{G}'$.

2. We can perform the step (with reststring u)

$$
\begin{array}{rrcl}
 & \ldots\, [Z_{i_1}/\ldots/Z_{i_n}/\ldots/Z_{i_{n+k-1}}/ⓙ]\, u & \rightsquigarrow & \ldots\, ⓙ\, [Z_{i_1}/\ldots/Z_{i_n}/Z_j]\, u \\
\textit{or} & \ldots\, [Z_{i_1}/\ldots/Z_{i_n}/\ldots/Z_{i_{n+k-1}}/ⓙ]\, u & \rightsquigarrow & \ldots\, ⓙ\, [Z_{i_1}/\ldots/Z_{i_n}/ⓘ]\, u
\end{array}
$$

   if there is a production $(A \to \alpha\, ⓙ)$ in the original grammar. The number k of elements to be popped off the list is the length of the handle α. The corresponding nonterminal A then determines the newly added Z_j or ⓘ: it comes from the production $(Z_{i_n} \to A Z_j \ldots)$ or $(Z_{i_n} \to A\, ⓘ \ldots)$, respectively, in the 3NF grammar $\mathcal{G}'$.

3. We can perform the step (with reststring u)

$$
\begin{array}{rrcl}
 & \ldots\, [Z_{i_1}/\ldots/Z_{i_n}/\ldots/Z_{i_{n+k-1}}]\, u & \rightsquigarrow & \ldots\, ⓙ\, [Z_{i_1}/\ldots/Z_{i_n}/Z_j]\, u \\
\textit{or} & \ldots\, [Z_{i_1}/\ldots/Z_{i_n}/\ldots/Z_{i_{n+k-1}}]\, u & \rightsquigarrow & \ldots\, ⓙ\, [Z_{i_1}/\ldots/Z_{i_n}/ⓘ]\, u
\end{array}
$$

   if there is an applicable production $(Z_{i_{n+k-1}} \to ⓙ \ldots)$ in the 3NF grammar $\mathcal{G}'$. The rest is analogous to case (2).

Evidently these rules just mimic the effect of the reduction of the handle α to the symbol A without actually performing the reduction itself. So they are obviously correct.

Discussion. The above transformation process has major influences on our programming techniques. The main aspects of the resulting changes are briefly discussed in the following.

- *List of functions.* If we want to adhere to the functional programming style, we can no longer use function composition (because "popping" functions off compositions is impossible). We therefore need a list of functions $[Z_{i_1}/\ldots/Z_{i_n}]$, from which the appropriate number of functions can be popped by the actions ⓘ. But otherwise the programming remains the same.

- *Interpretative parsing.* Since we work with a list $[Z_{i_1}/\ldots/Z_{i_n}]$ anyhow, we may also consider changing the Z_i from functions to simple values (such as numbers). Then they play exactly the role of "states" in traditional *LR*-style parsers.

  The programming is not much different from the above functional version, though: instead of direct applications of the functions Z_i we now have an operation $\texttt{apply}(Z_\texttt{i})$, that is, we work in the style of an interpreter.

- *Efficiency.* An interesting issue is the efficiency of the different variants.

  - In our *emulated-LR* concept we carry around a (composed) function that needs to be applied to the rest string. If one considers the overhead of higher-order functions as being too costly, one can easily implement this composition as a stack of functions (or as a stack of numbers to be "interpreted"). A shift operation adds one or more elements Z_j to this stack, whereas a reduce operation ⓘ takes one element off this stack.
  - In our *true-LR* version we also carry around a stack, albeit a different one.[2] A shift operation adds one element Z_j to this stack, whereas a reduce operation ⓘ takes one or more elements off this stack (and adds another one).
  - The traditional *LR*-style technique actually amalgamates this stack management with the explicit generation of the tree (fragments).

  From this conceptual comparison it should be clear that the various implementation techniques should actually not cause very different costs.

[2] If we consider the stacks of the original version over time, we obtain a stack of stacks. And the bottom elements of these stacks form the stack in the second approach.

Bibliography

[1] Alfred A. Aho, Ravi Sethi, and Jeffrey D. Ullmann. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, 1985.

[2] Roland C. Backhouse. *Syntax of Programming Languages*. Prentice-Hall, 1979.

[3] R. Bahlke and G. Snelting. The PSG system: From formal language definitions to interactive programming environments. *ACM Transactions on Programming Languages*, 8(4):547–576, 1986.

[4] J. Bates and A. Lavie. Recognizing substrings of lr(k) languages in linear time. *ACM Transactions on Programming Languages*, 16(3):1051–1077, 1994.

[5] J. C. Beatty. On the relationship between the LL(1) and LR(1) grammars. *Journal of the ACM*, 29(4):1007–1022, 1982.

[6] M. E. Bermudez and K. M. Schimpf. Practical arbitrary lokkahead lr parsing. *Journal of Computer and System Sciences*, 41:230–250, 1990.

[7] P. Degano, S. Mannucci, and B. Mojana. Efficient incremental LR parsing for syntax-directed editors. *ACM Transactions on Programming Languages*, 10(3):345–373, 1988.

[8] F. L. DeRemer and Th. Pennello. Efficient computation of LALR(1) look-ahead sets. *ACM Transactions on Programming Languages*, 4(4):615–649, 1982.

[9] F.L. DeRemer. Simple LR(k) grammars. *Comm. ACM*, 14:453–460, 1971.

[10] K. Didrich, A. Fett, C. Gerke, W. Grieskamp, and P. Pepper. Design and implementation of an algebraic programming language. In J. Gutknecht, editor, *Programming Languages and System Architectures.*, Lecture Notes in Computer Science 782, pages 228–244. Springer Verlag, 1994.

[11] K. Didrich, W. Grieskamp, J. Exner, Ch. Maeder, M. Südholt, C. Gerke, and Pepper P. Towards a redesign of OPAL. Technical Report 96 – 3, Fachbereich Informatik, Technische Universität Berlin, February 1997.

[12] S. Hill. Combinators for parsing expressions. *J. Functional Programming*, 6(3):445–464, May 1996.

[13] G. Hutton. Higher-order functions for parsing. *J. Functional Programming*, 2(3):323–343, July 1992.

[14] G. Hutton and E. Meijer. Monadic parser combinators. *J. Functional Programming*, 1(1), May 1993.

[15] K. Jensen and N. Wirth. *Pacal User Manual and Report.* Springer, 1978.

[16] T. Kasami. An efficient recognition and syntax analysis algorithm for context-free languages. Technical Report AFCRL-65-758, Air Force Cambridge Research Laboratory, 1965.

[17] B. W. Kernighan and D. M. Ritchie. *The C Programming Language.* Prentice-Hall, 1978.

[18] D. E. Knuth. On the translation of languages from left to right. *Information and Control*, 8:607–639, 1965.

[19] R. N. Moll, M. A. Arbib, and A.J. Kfoury. *An Introduction to Formal Language Theory.* Springer Verlag, 1988.

[20] A. Nijholt. *Deterministic Top-Down and Bottom-Up Parsing: Historical Notes and Bibliographies.* Mathematical Centre, Amsterdam, 1983.

[21] R. Nozohoor-Farshi. GLR parsing for ε-grammars. In M. Tomita, editor, *Generalized LR Parsing*, pages 61–75. Kluwer Academic Publishers, 1991.

[22] A. Partridge and D. Wright. Predictive parser combinators need four values to report errors. *J. Functional Programming*, 6(2):355–364, March 1996.

[23] P. Pepper. Grundlagen des Übersetzerbaus. Course notes, Fachbereich Informatik, Technische Universität Berlin, 1990.

[24] P. Pepper. The programming language OPAL. Technical Report 91 – 10, Fachbereich Informatik, Technische Universität Berlin, June 1991.

[25] P. Pepper. Deductive derivation of parallel programs. In R. Paige, J. Reif, and R. Wachter, editors, *Parallel Algorithm Derivation and Program Transformation*, pages 1–54. Kluwer Academic Publishers, 1993.

[26] P. Pepper. *Funktionale Programmierung in* OPAL, ML, HASKELL *und* GOFER. Springer Verlag, 1999.

[27] R. Plasmeijer and M. van Eckelen. *Functional Programming and Parallel Graph Rewriting.* Addison-Wesley, 1990.

[28] Jan Rekers. *Parser generation for Interactive Environments.* PhD thesis, University of Amsterdam, 1992.

[29] A. Salomaa. *Formal Languages.* Academic Press, 1973.

[30] L. Schmitz. *Syntaxbasierte Programmierwerkzeuge.* Teubner, 1995.

[31] Th. Schöbel-Theuer. *Ein Ansatz für eine allgemeine Theorie kontextfreier Spracherkennung.* PhD thesis, Fakultät Informatik der Universität Stuttgart, 1996.

[32] K. Sikkel. *Parsing Schemata.* Springer Verlag, 1997.

[33] S. Sippu and E. Soisalon-Soininen. A syntax-error-handling technique and its experimental analysis. *ACM Transactions on Programming Languages*, 5(4):656–679, 1983.

[34] S. Sippu and E. Soisalon-Soininen. *Parsing Theory, Vol.II: LR(k) and LL(k) Parsing.* EATCS Monographs on Theoretical Computer Science. Springer Verlag, 1990.

[35] J. W. Thatcher, E. G. Wagner, and J. B. Wright. Initial algebra semantics and continuous algebras. *Journal of the ACM*, 24(1):68–95, 1977.

[36] M. Tomita. *Efficient Parsing for Natural Languages*. Kluwer Academic Publishers, 1985.

[37] T.A. Wagner and S.L. Graham. Incremental analysis of real programming languages. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 31–43, 1997.

[38] T.A. Wagner and S.L. Graham. Efficient and flexible incremental parsing. *ACM Transactions on Programming Languages*, 20:980–1013, 1998.

[39] W. W. Waite and G. Goos. *Compiler Construction*. Springer Verlag, 1984.

[40] D.H. Younger. Recognition of context-free languages in time n^3. *Information and Control*, 10:189–208, 1967.