Kontextfreie Hypergraphgrammatiken mit Speicher

Großer Beleg

Falk Hartmann

an der

TECHNISCHEN UNIVERSITÄT DRESDEN

Fakultät Informatik

Verantwortlicher Hochschullehrer:	Prof. Dr.–Ing. habil H. Vogler
Betreuer:	Herr S. Maneth M.Sc./Univ. of Massachusetts
Eingereicht am:	18. Dezember 1996

Zielstellung

In [Eng86] wird das Konzept der kontextfreien Grammatiken mit Speicher definiert. Eine solche Grammatik mit Speicher induziert eine Übersetzung von der Menge der Eingaben des Speichers in die Menge der Ausgaben der zugrundeliegenden kontextfreien Grammatik. In [EV86] werden allgemeine Aussagen über die Übersetzungstärke sogenannter $X(S)$–transducer gegeben (auch S–kontrollierte X–Grammatiken genannt); dabei bezeichnet X den Typ der Grammatik (z.B. X=kontextfrei) und S den Speichertyp.

Kontextfreie Hypergraphgrammatiken ($CFHG$), siehe z.B. [Hab92, DKH96], sind kontextfreie Grammatiken, bei denen linke Regelseiten aus einem Nichtterminalsymbol bestehen und rechte Regelseiten Hypergraphen sind.

In dieser Belegarbeit soll das Konzept der Grammatiken mit Speicher von kontextfreien (String–) Grammatiken auf kontextfreie Hypergraphgrammatiken übertragen werden. Dazu sollen zunächst bekannte Speichertypen wie trivialer Speichertyp, Baumspeichertyp, tree walk Speichertyp und pushdown Speichertyp eingeführt werden. Dann soll das Konzept der S–kontrollierten $CFHG$–Grammatiken formal definiert werden und anhand von Beispielen instantiiert werden.

Die Einschränkung der Klasse der kontextfreien Hypergraphgrammatiken auf solche Grammatiken, die nur Hypergraphen erzeugen, welche Terme repräsentieren ($CFHG_{treegen}$), bietet eine elegante Möglichkeit um Mengen von Termen zu beschreiben. Ein S–kontrollierte $CFHG_{treegen}$–Grammatik induziert dann eine Übersetzung von der Menge der Eingaben von S in eine Menge von Bäumen. Eine Instanz dieser Grammatiken sind die top–down tree–to–graph transducer [EV94], i.e., dies sind baumkontrollierte $CFHG_{treegen}$-Grammatiken. Die Übersetzungsstärke der $CFHG_{treegen}$-Grammatiken soll für unterschiedliche Speichertypen S beispielhaft untersucht werden.

Context–Free Hypergraph Grammars With Storage

Falk Hartmann
University of Dresden, 1996

Abstract

In this paper, the former known concept of storage–controlled grammars — which was defined in [Eng86] for regular and context–free string– and tree–grammars — will be extended to context–free hypergraph grammars with storage. The considerations are restricted to string– and tree–generating hypergraph grammars mainly and are given in form of examples.

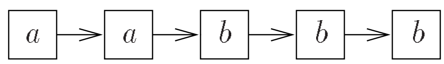
Contents

Chapter 1

Introduction

In this chapter we give a short introduction to the main ingredients of the paper, i.e., the concepts of hypergraph grammars and storage control.

The development of computing machinery goes along with a generalization of information structures. Among the most essential data types are **strings**. Strings allow to present written natural language as well as any other kinds of text. From the syntactic point of view, they are just sequences of elements taken from a given alphabet. The string s in Figure 1.1, for instance, uses the alphabet $A = \{a, b\}$. We can read the string as $aabbb$ — the reading direction is given implicitly from left to right.



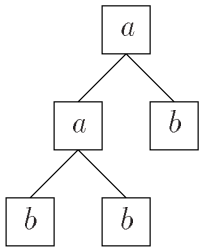
Alphabet $A = \{a, b\}$

String $s = aabbb \in A^*$

Figure 1.1: A string

A natural extension of the data type string is the concept of **trees**. Trees are mostly used to represent **terms**. The term $t = a(a(b, b), b)$ can be viewed as the tree in Figure 1.2. No brackets are needed in this representation, because with the implicit given direction (depth first left–to–right) to read the tree, the term is unambiguously represented. On the syntactic side, terms are built from symbols of a ranked alphabet — the mentioned term t is a term over the ranked alphabet $\Delta = \{a^{(2)}, b^{(0)}\}$, and we write $t \in T_\Delta$. Interpreting the elements as function symbols, the rank gives the number of function arguments — funtions of rank 0 are constants. The notions term and tree are very similar, so they are often used as synonyms. Trees can be interpreted as strings in many ways: **monadic trees** — trees from a ranked alphabet, in which all symbols are of rank 0 or 1 — can be read as strings (e.g., top–down), or the **yield** (sometimes also called **front**) of the tree, which is the sequence of the symbols on the leafs of the tree, read from left to right, is a string, e.g., the yield of the tree in Figure 1.2 is the string bbb.

Being confronted with the task of representing non–hierarchical data gives rise to an even more general data type: the **graph**. From the syntactic point of view, a graph is a collection of **nodes** together with **edges** connecting them. The nodes (or vertices) are represented by a set V, the edges by a so called **edge–relation** $E \subseteq V \times L_E \times V$, where L_E is a set of edge-labels. In this case we speak of **edge–labeled** graphs. There can be a set L_N of node-labels, too,

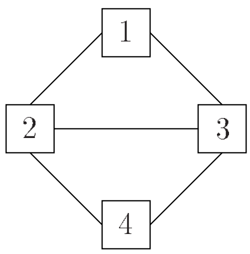


Ranked alphabet $\Delta = \{a^{(2)}, b^{(0)}\}$

Term $t = a(a(b, b), b) \in T_\Delta$

Figure 1.2: A tree

together with a labeling function $label : V \to L_N$, which leads to **node– and edge–labeled** graphs. In the Figure 1.3, the edge–relation is assumed to be symmetric — but there is no reason for this limitation at all. Graphs with a symmetric edge–relation (as in the figure) are called **undirected**, all others **directed**. But although this direction may be given, there is no overall direction in a graph, because it can still have cycles. This makes it difficult to find a unique linear representation for graphs. However, there are still linear representations, like terms denoting the graph, but they are not as natural as the linear representations of strings and terms — see also [BC87] or [Dre96, Eng94, Eng96] for further information on term representations of graphs.



Nodes $V = \{1, 2, 3, 4\}$

Edge-Relation $E = \{(1,2), (1,3), (2,1), (2,3),$
$(2,4), (3,1), (3,2), (3,4),$
$(4,2), (4,3)\}$

Labels $L = V$

Labeling function $lab = id(V)$

Graph $G = (V, E, L_V, lab)$

Figure 1.3: A graph

Because graphs are a generalization of strings and trees, there are of course possibilities to use graphs to represent strings or trees. For example, the string in the left part in Figure 1.1 is a directed node–labeled graph. Obviously it is also possible to view trees as graphs in the same way. But there is a more interesting way to present terms (equivalent to trees). Every term may be viewed as a **directed acyclic graph** (DAG). The DAG in Figure 1.4 represents the term $a(a(b, b), b)$. The operation, which transforms a DAG into a tree is called **unfolding**. This means that starting at the only symbol without outgoing edges, which is by convention at the top, the symbols in the nodes are taken over to the tree. The edges are taken over, too, but if two edges are pointing to the same node, what is not allowed in a tree, the node becomes duplicated and the unfolding proceeds as described. It is necessary to preserve the order (from left to right) of the edges. This is important, otherwise the unfolding is ambiguous. The other way, translating a term into a DAG is done by constructing it bottom up from the leafs of the term–representing tree. Before creating a new node in the DAG, existing nodes are checked, whether they are equal to the desired one. If so, no new node is created, an edge to the existing one will be used instead. This technique is called **sharing**. Sharing makes it necessary to introduce an order on the edges attached to a node. This is the reason for the fact, that DAG's are not just graphs, moreover, they can be simplier represented by hypergraphs as we will see later. In Figure 1.4 the order is from left to right, the left–most edge represents the first argument to the node. DAG's are widely used in computer science. They are a memory efficient way to represent terms. The construction of a DAG from a term eliminates common subexpressions — this is one of the reasons, why DAG's are a very popular mean in compiler construction. The dependency graph of the attributes in a derivation of a non–circular attribute grammar is a DAG, too. Because of the non–circularity of DAG's, they are, in contrast to graphs in general, always sortable in a topological way.

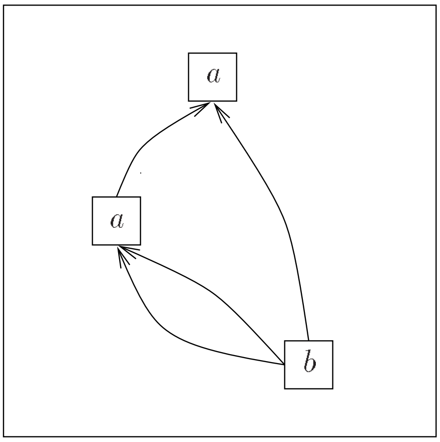


Figure 1.4: A directed acyclic graph

A possible extension of the graph concept are **bipartite graphs**, which are derived from usual graphs by introducing a second set of nodes. The elements of the sets V_1 and V_2 of nodes are usually drawn as boxes and circles, respectively, as in Figure 1.5. The edge relation is restricted to connect only nodes from different sets: $E \subseteq V_1 \times V_2 \cup V_2 \times V_1$. The well–known Petri–nets (see [Pet62] for details) are an application of bipartite graphs and are widely used in the theory of processes. The graph in Figure 1.5 is of practical significance, too: it is a so called **process–resource–graph**. The processes are π_1 and π_2, the resources are ρ_1 and ρ_2. The arrow from π_1 to ρ_1 means, that the process requests the resource, the arrow from ρ_1 to π_2 tells, that the process has the resource.

In Chapter 3, we will present a very popular and elegant generalization of the graph concept — the so–called **hypergraphs**. As usual graphs, a hypergraph is basically a tuple of the set of nodes and a relation, which sets up the edges, which are here called **hyperedges**. Considerating the concept of graphs, there seems to be no reason to limit the number of nodes incident with an edge. In usual graphs this number is always two. For hypergraphs this number can be arbitrary, thus hyperedges can connect zero, one, two or more nodes of the hypergraph. Hyperedges connecting n nodes are said to be of **rank** (or **type**) n. Figure 1.6 shows a hypergraph in a natural way. The nodes are $v_1 \ldots v_5$, they are not labeled, but they are sometimes associated with a number, e.g., node v_2 with the number 1. The nodes with numbers are called **external** nodes, the other **internal**. This distinction is needed for technical reasons to allow a simple definition of hyperedge replacement and will be explained later in Chapter 3.

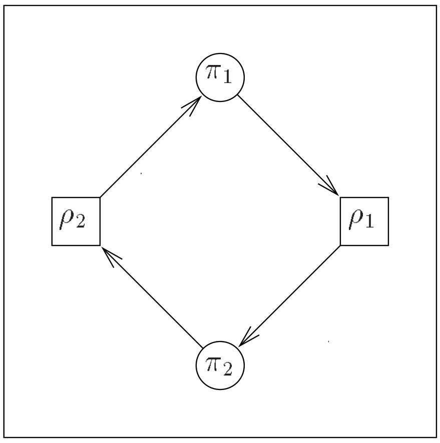


Figure 1.5: A bipartite graph

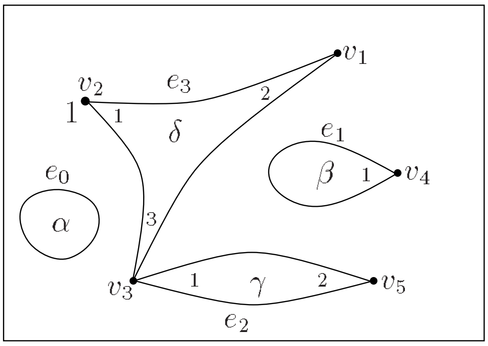


Figure 1.6: A hypergraph in a natural way

Consider the hyperedge e_2 in this figure. It is a hyperedge of rank 2, attached to the nodes v_3 and v_5. Hyperedge e_0 has no attachments at all, it is of rank 0. Within the edge e_2 there are the numbers 1 and 2, which are giving the hyperedge a direction. It can be compared with a directed edge in a usual directed graph.

There exist several approaches to the problem describing a language. A language can be described by giving an algorithm which yields all the elements of the language, or by giving an algorithm, which checks for a given item whether it is element of the language or not or to give an algorithm which describes, how the language can be constructed from another given language. This corresponds to the concepts of **grammars**, **acceptors** and **transducers**, respectively.

In 1956 Chomsky introduced the concept of context–free grammars with the intention to model natural languages (see [Cho56]). As in natural language, grammars are finite descriptions for mostly infinite languages in computer science, too. The basic idea of grammars is to describe the language by producing it from a starting point using replacement recursively. Therefore, the notion of grammars is based on two concepts: **replacement** and **recursion**. The context–free approach of Chomsky to string languages has been defined for a lot of other languages like tree– and graph–languages and is extendable to hypergraph languages, too.

The second ingredient of this paper — the storage types — will be introduced in Chapter 4 and are just a formalization of well–known information structures like pushdowns, trees and input–tapes. We can imagine a storage type as an abstract model of an object. Objects (in the sense of programming technology) has state variables and methods. The state is reflected in the storage type as the **configuration**. The methods are partitioned into **instructions** and **predicates**. The instructions change the configuration of the storage type, while the predicates are provided to get information about the configuration. Usually, objects have some distinguished methods, the so–called constructors, which we call in storage types **encoding**. The allow to set up a storage type, because they are in fact a translation from the **input** into a configuration. Besides that, we use a **meaning function** for the semantic realization of instruction, predicate and encoding symbols. To sum it up, we will have a tuple of sets of configurations, inputs, instruction symbols, predicate symbols, encoding symbols and a meaning function.

An excellent example is the storage type **pushdown**, as it is drawn in Figure 1.7. Three configurations are shown (numbered (1), (2) and (3) for reference). It is easy to see, that this pushdown operates over the alphabet $\Gamma = \{\alpha, \beta, \gamma\}$, thus the configurations of the storage type can be interpreted as strings (just turn the pushdown such that the topmost symbol is left, i.e., 90 counter–clockwise). Between the configuration (1) and (2) the *pop*–instruction was used. Thus, configuration (2) is nearly the same as (1), only the topmost symbol α is removed. Next, a $push(\gamma)$–instruction was applied to (2). As expected, the configuration (3) differs from (2) only in the new topmost symbol γ.

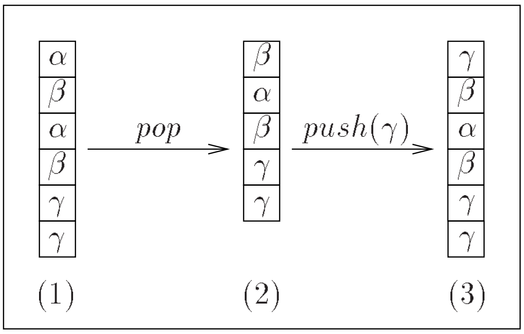


Figure 1.7: A stack and operations on it

The input, encoding symbols and the meaning function, which are for technical reasons primary, are left out here and will be explained later in Chapter 4.

Additionally to the definitions of simple storage types operations with storage types can be defined, so two storage types can be multiplied to get a more complex one. The pushdown, as described before, can be defined and interpreted as a operator, which takes a storage type S and enhances it to a pushdown of S.

The combination of grammars with storage to the **storage–controlled grammars** does not only enhance the generative power of grammars (as it will be shown later), it provides the grammar with an input set, too, thus the controlled grammar can be viewed as a transducer.

Since the introduction of context–free grammars by Chomsky a very great variety of grammars have been developed and examined. Today some of the concepts are known to be equivalent, but there is still need for a general framework. The concept of grammars which are controlled by storage types is such a framework to unify a lot of grammars and grammar–related notions and to classify all these grammars in one way, thus providing computer scientists with a uniform method to denote and compare their explorations. An overview about some relations between controlled grammars and other grammar–like concepts follows in Chapter 4.

The main goal of the paper is the combination of context–free hypergraph grammars with storage types. This will be done in Chapter 5 by giving examples for it. The most examples are tree–generating hypergraph grammars, a special case of hypergraph grammars, which allow a very elegant way of representing trees. There are examples for string–generating and general controlled hypergraph grammars, too.

Chapter 2

Preliminaries

The empty set is denoted by $\emptyset$. The set $\{0, 1, ...\}$ of natural numbers is denoted by $\mathbb{N}$ and $\mathbb{N}_+ = \mathbb{N} - \{0\}$. For $n \geq 0$, $[N] = \{1, ..., n\}$, in particular $[0] = \emptyset$.

The empty string is denoted by ε. Syntactically, strings are sequences of elements of an alphabet. Let A be an alphabet, then A^* denotes the set of all strings over the alphabet A and $A^+ = A^* - \{\varepsilon\}$. Given a string $w = w_1 w_2 ... w_k$ over an alphabet A, the functions $ar : A^* \to A^*$ and $res : A^* \to A$ are defined as

$$ar(w) = \begin{cases} w_1 w_2 ... w_{k-1} & \text{iff } k > 1 \\ \varepsilon & \text{otherwise} \end{cases}$$

$$res(w) = \begin{cases} w_k & \text{iff } k \geq 1 \\ \varepsilon & \text{otherwise} \end{cases}$$

A ranked set (or ranked alphabet) is a set Γ together with a mapping $rank_\Gamma : \Gamma \to \mathbb{N}$. The set Γ_n is the set of elements $\gamma \in \Gamma$ with $rank_\Gamma(\gamma) = n$. For $\gamma \in \Gamma_n$, we also write $\gamma^{(n)}$ for short. The set of terms constructable from this set is denoted by T_Γ and is defined inductively as the smallest set T with the following properties:

1. For every $\gamma \in \Gamma$ with $rank_\Gamma(\gamma) = 0$: $\gamma \in T$.

2. For every $\gamma \in \Gamma$ with $rank_\Gamma(\gamma) = n$ and $\xi_1, ..., \xi_n \in T$: $\gamma(\xi_1, ..., \xi_n) \in T$.

$inc(\Gamma)$ and $dec(\Gamma)$ denote the sets

$$\{\gamma^{(n+1)} \mid \gamma \in \Gamma \text{ and } rank_\Gamma(\gamma) = n\}$$

and

$$\{\gamma^{(n-1)} \mid \gamma \in \Gamma \text{ and } rank_\Gamma(\gamma) = n \text{ and } n \geq 1\}$$

respectively, i.e., the elementwise incrementation/decrementation of the ranks of Γ.

A given term $t \in T_\Gamma$ can be considerated as a tree, too. In this case the function $label_t(\pi) : occ(t) \to \Gamma$ returns to every path $\pi = w_1 ... w_k \in occ(t)$ the label of the corresponding node. The set $occ(t)$ is the smallest set O such that:

1. $\varepsilon \in O$

2. For every $\pi \in O$, $\gamma = label_t(\pi)$ and $rank_\Gamma(\gamma) = n$: $\pi 1, ..., \pi n \in O$.

The full binary tree of the height n over the ranked alphabet $\Gamma = \{\delta^{(2)}, \alpha^{(0)}\}$ is denoted by $fbt_{\delta,\alpha}(n)$ and is defined recursively by the following two equations:

1. $fbt_{\delta,\alpha}(0) = \alpha$ and
2. $fbt_{\delta,\alpha}(n+1) = \delta(fbt_{\delta,\alpha}(n), fbt_{\delta,\alpha}(n))$ for $n \geq 1$.

Given a binary **relation** $T \subseteq D \times R$, D and R are called **domain** (denoted by $dom(T) = D$) and **range** ($ran(T) = R$), respectively. For a relation $R \subseteq A \times A$ we denote by $\equiv_R$ the smallest equivalence relation R', which contains R, i.e., with $R \subseteq R'$. A binary relation R is called an **equivalence relation**, if it satiesfies the following properties:

1. $(a, a) \in R$ for every $a \in A$. *Reflexivity*
2. If $(a, b) \in R$, then $(b, a) \in R$, too. *Symmetry*
3. IF $(a, b) \in R$ and $(b, c) \in R$, then $(a, c) \in R$, too. *Transitivity*

The set of equivalence classes induced in a set V by a equivalence relation R is denoted by V/R. Binary relations $T \subseteq D \times R$ can be considered as functions $F : D \rightarrow R$, too, if $|\{(d, r) \mid r \in R\}| \leq 1$ for every $d \in D$. If $|\{(d, r) \mid r \in R\}| = 1$ for every $d \in D$, we call T **total**, otherwise **partial**.

$BE(P)$ denotes the set of BOOLEan expressions over P, where P is a set of predicates. The set $BE(P)$ is inductively defined as the smallest set B with the following properties:

1. $P \subseteq B$,
2. `true` $\in B$ and `false` $\in B$,
3. For every $b \in B$, `not` $b \in B$, too, and
4. For every $b_1, b_2 \in B$, b_1 `and` $b_2 \in B$ and b_1 `or` $b_2 \in B$, too.

We assume the reader to be familiar with the λ–calculus as introduced in [Bar84].

Furthermore, the symbol $id(S)$ denotes the identity operation on a set S, thus $id(S) = \lambda s \in S.s$ and $id(S) : S \rightarrow S$.

Chapter 3

Context–Free Hypergraph Grammars

3.1 Hypergraphs

In the introduction, we viewed hypergraphs in a way as in Figure 1.6. The approach to draw hyperedges like in the figure is as already mentioned, the natural one, but there are problems to view crossing edges. Therefore, hypergraphs are usual shown as bipartite graphs — this is possible because of the one–to–one correspondence between hypergraphs and bipartite graphs. The hypergraph in Figure 1.6 is shown in Figure 3.1 again. Now there are two sets of nodes, drawn as boxes and circles. Hyperedges are no longer edges in the commen sense. They are represented by the labeled boxes together with the connecting edges, which are now called **tentacles**. Hyperedges of rank 2 are represented as usual by a directed edge with its label beside. This usual way of representation avoids problems with crossing edges and is easier to read.

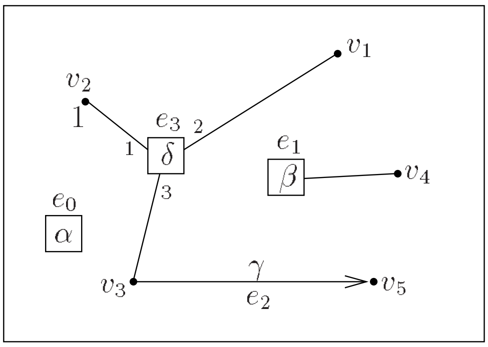


Figure 3.1: A hypergraph in the usual way

Definition 3.1 of hypergraphs is also based on the one–to–one correspondence between bipartite graphs and hypergraphs. The edge relation is replaced by a set of hyperedges — the tentacles are represented by an **attachment function** *att*.

Definition 3.1 *(hyperedge–labeled hypergraph)*
Let Γ be a ranked alphabet. A **(hyperedge–labeled) hypergraph** H over Γ is a tuple $H = (V, E, lab, att, ext)$, where

- V is a finite set of **nodes**,
- E is a finite set of **hyperedges**, distinct from the set of nodes: $V \cap E = \emptyset$,
- $lab : E \to \Gamma$ is a function mapping to every hyperedge $e \in E$ a **label** $lab(c) \in \Gamma$,
- $att : E \to V^*$ is a function mapping to each hyperedge $e \in E$ a sequence $att(e)$ of pairwise distinct **attachment** nodes, such that $|att(e)| = rank_\Gamma(lab(e))$,
- $ext \in V^*$ is a sequence of **external nodes**. □

The hypergraph from Figure 1.6 can of course be written as such a tuple $H = (V, E, lab, att, ext)$ over the ranked alphabet $\Gamma = \{\alpha^{(0)}, \beta^{(1)}, \gamma^{(2)}, \delta^{(3)}\}$ where:

- $V = \{v_1, v_2, v_3, v_4, v_5\}$,
- $E = \{e_0, e_1, e_2, e_3\}$,
- $lab(e_0) = \alpha, lab(e_1) = \beta, lab(e_2) = \gamma, lab(e_3) = \delta$,
- $att(e_0) = \varepsilon$ (this means no attachments),
  $att(e_1) = v_4$,
  $att(e_2) = v_3v_5$,
  $att(e_3) = v_2v_1v_3$ and
- $ext = v_2$

Given a hypergraph H, the components of the defined tuple are denoted by V_H, E_H, att_H, lab_H and ext_H. The **size** of H, denoted $|H|$ is the sum of both the number of nodes and the number of the hyperedges, i.e., $|V_H| + |E_H|$. The set of all hypergraphs which can be constructed over the ranked alphabet Γ is denoted by $HGR(\Gamma)$. A hyperedge e is said to be of rank m, if $rank(lab(e)) = m$ and is called an **m–edge**. The rank n of an hypergraph $H \in HGR(\Gamma)$ is the number of its external nodes, $n = |ext_H|$. The hypergraph is then an **n–hypergraph** and the function $rank$ gives its rank n. The set of all hypergraphs $H \in HGR(\Gamma)$ with $rank(H) \leq n$ is denoted by $n\text{-}HGR(\Gamma)$. Two hypergraphs H and K are called **disjoint**, if $V_H \cap V_K = \emptyset$ and $E_H \cap E_K = \emptyset$.

Equality of two hypergraphs H and K is not the pairwise equality of all components of the tuple. Two hypergraphs are said to be equal, if they are isomorphic, i.e., bijections between the sets of hyperedges and nodes of H and K exist.

Let $\gamma \in \Gamma$ be a symbol of rank m. A hypergraph with exactly m nodes and one γ-labeled hyperedge only, which is distinctly attached to all the nodes, is called a **singular hypergraph labeled by** γ and is denoted by $sing(\gamma)$. Formally, $sing(\gamma)$ is the m–hypergraph $([m], e, lab, att, ext)$ with $lab(e) = \gamma$ and $att(e) = ext = 12...m$ (e is attached to all nodes and in addition to this all nodes are external ones). The singular γ-labeled hypergraph for $m = 3$ is shown in Figure 3.2.

We have mentioned on page 7 in the introduction, that hypergraphs are a very elegant way to represent terms using sharing of common subexpressions. The basis for this representation is understanding hyperedges in a functional way. This is illustrated in Figure 3.3.

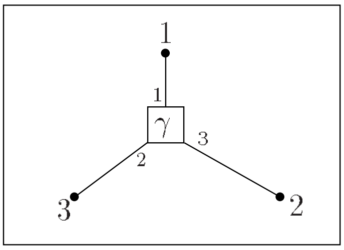


Figure 3.2: The hypergraph $sing(\gamma)$ (where $rank(\gamma) = 3$)

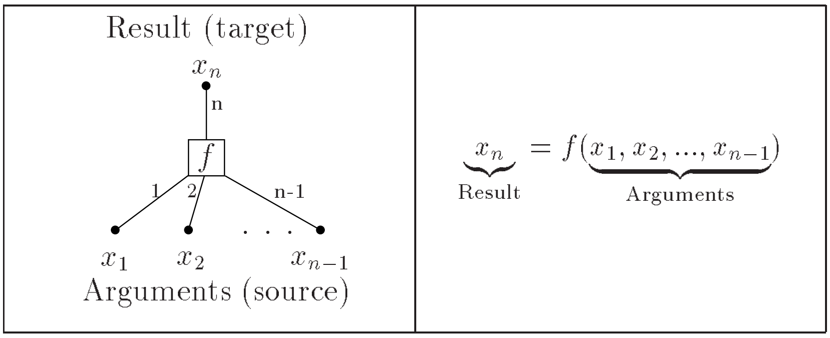


Figure 3.3: A hyperedge and it's interpretation as function

Following the pattern of this analogy, we will call the tentacle n a **result–tentacle** and the others **argument–tentacles**. With this interpretation in mind, the understanding of Figure 3.4 is easy. A hypergraph representing a term in such a way is called a **jungle** (for further details see, e.g., [HKP91]). The external node 1 (at the top) is the starting point for unfolding the jungle into the underlying term. Unfolding proceeds in the same way as the unfolding of DAG's described in the introduction at page 6. The symbol at the first edge reached from the only external node is obviously δ, thus the term is $\delta(x_1, x_2)$. Following the tentacle 1 gives the argument x_1 and tentacle 2 gives the argument x_2. Please recall that the result–tentacle of an edge labeled with $\gamma^{(n)}$ is always the tentacle n, as already seen in Figure 3.3.

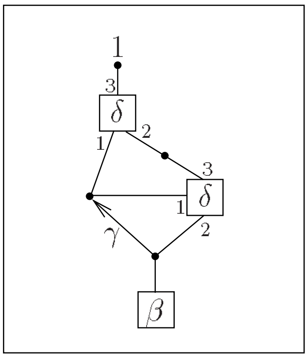


Figure 3.4: A jungle representing the term $\delta(\gamma(\beta), \delta(\gamma(\beta), \beta))$

For an exact definition of a jungle as a special hypergraph, it is necessary to formalize three conditions:

- First, to ensure the possibility of unfolding, it is necessary that the hypergraph is topological sortable, or in other words, is **acyclic**.
- Moreover, a node must get exactly one result from its attached hyperedges. This leads to the notion of **in–degree**. A node's in-degree is the number of result–tentacles it is attached with. Thus, in a jungle, all nodes must have in–degree 1.
- Of course the jungle must have exacty one root, thus it has to be a 1-hypergraph.

Definition 3.2 *(acyclic hypergraphs)*
Let g be a hypergraph $(V_g, E_g, lab_g, att_g, ext_g)$. A **path of g from node $\mathbf{v_0}$ to node $\mathbf{v_k}$** is an element $v_0e_1v_1...e_nv_n$ of $V_g(E_gV_g)^*$, where $v_k \in V_g$ for every $0 \leq k \leq n$, $e_k \in E_g$ for every $1 \leq k \leq n$. Furthermore, $v_{k-1} \in ar(att_g(e_j))$ and $res(att_g(e_j)) = v_j$ for every $1 \leq k \leq n$. The hypergraph g is **acyclic**, if for every path $v_0e_1v_1...e_nv_n$ and every $0 \leq i, j \leq n, i \neq j$ holds $v_i \neq v_j$, i.e., no path contains a node twice. □

Definition 3.3 *(in–degree)*
Let g be a hypergraph $(V_g, E_g, lab_g, att_g, ext_g)$. The **in–degree of a node v** is $|C_v|$, where $C_v = \{e_j | res(att_g(v)) = e_j\}$. □

Definition 3.4 *(jungle)*
A **jungle** is an acyclic 1-hypergraph, in which every node has in–degree 1. The term represented by a jungle is given by the partial function $unfold : HGR(\Gamma) \to T_{dec(\Gamma)}$, which is defined as $unfold(G) = \mu(ext_G(1))$ where the mapping $\mu : V_G \to T_{dec(\Gamma)}$ is inductively defined as $\mu(v) = \gamma(\mu(v_1), ..., \mu(v_k))$ where $\gamma = lab_G(res_G^{-1}(v))$ and $v_1...v_kv = att_G(res_G^{-1}(v))$. □

These definitions provide us with a mean to define string–representing hypergraphs, too. This is very analogous to the definition of jungles, with the difference that the rank of all hyperedges is restricted to two now:

Definition 3.5 *(string–representing hypergraph)*
A hypergraph $g = (V_g, E_g, lab_g, att_g, ext_g)$ is said to represent a string, if it is of rank 2, all of it's edges are of rank 2 and every node has in–degree 1. Let ext_g be v_0v_n. Then the sequence of labels $w = lab(e_1)...lab(e_n)$ along the only path $v_0e_1v_1...e_nv_n$ in g from v_0 to v_n is the string represented by g. On the contrary, we call g the hypergraph induced by w and write $g = w^\bullet$.□

3.2 Context–free hypergraph grammars

The general syntactic form of a grammar is independent from the type of the described language: a context–free grammars is a tuple (N, T, S, R), where

- N is the set of **non–terminal symbols**,
- T is the set of **terminal symbols**,
- S is a single non–terminal, the **starting symbol** and
- R is a set of **productions** (also called **rules**).

In case of context–free grammars, which we restrict our consideration to, productions have the general form $lhs \to rhs$, where lhs — the **left–hand side** — is a single non-terminal and rhs — the **right–hand side** is a sequence of non–terminals and terminals.

The process in which the grammar produces elements of a language, is called **derivation** and is a sequence **derivation steps**. A derivation step always takes one **sentential form** S_1 and transforms it into another, e.g., S_2. The first derivation step starts with the start symbol S, which is per definition a sentential form. In a derivation step, any non–terminal $A \in N$ contained in the sentential form S_1, for which one or more productions with the left–hand side A exist in R is replaced in S_1 by the right–hand side of one of these productions. The result is the new sentential form S_2. The derivation continues until no non–terminals are left, then the sentential form contains only terminals and is a **sentence**, i.e., an element of the language described by the grammar.

In Figure 3.5 example derivations of the context–free string grammar $G_{ab} = (\{S\}, \{a, b\}, S, R)$ (with $R = \{\pi_1, \pi_2\}$, where π_1 is the rule $\pi_1 = S \rightarrow aSb$ and π_2 is the rule $S \rightarrow \varepsilon$) are shown. This grammar is in fact a finite description for the infinite string–language $L = \{a^n b^n | n \geq 0\}$.

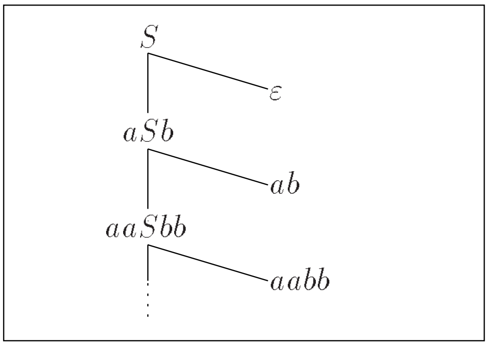


Figure 3.5: Derivations of the context–free string grammar G_{ab}

The basic notions of grammars, replacement and recursion, are often used to distinguish grammars. A sentential form may contain more than one non–terminal. If there are more than one, they can be replaced one after the other (in a **serial** way) or all at the same time (i.e., **parallel**). The first approach is used in the usual context-free grammars, the second leads to the family of **L-systems**, which were introduced by Lindenmayer (see [Lin68]) as a mean for modeling growth of plants. Beside this differentiation of grammars, grammars are of course classified by the mechanism of replacement used. In Figure 3.6 methods of replacement for three grammars are shown. The replacement in column *(a)* is the replacement used in usual context–free (string–)grammars: a non–terminal is replaced by a sequence of non–terminals and terminals. In *(b)* replacement in tree grammars are shown, x_1 and x_2 are placeholder actual values and are only used within the productions. In *(c)* hyperedge replacement is shown.

Intuitively, the meaning of **hyperedge replacement** is easy to describe: taking a hypergraph G, an hyperedge e is substituted by a hypergraph H by deleting the hyperedge, adding the hypergraph H instead and identifying the external nodes of H to the nodes, which are former attached by the hyperedge labeled e. The result of the substitution is written $G[e/H]$. First we formalize the identification of nodes as follows:

Definition 3.6 *(identification of modes)*
Let G be a hypergraph and let $R \subseteq V_G \times V_G$ be a binary relation over the set of nodes of G. Let $\Phi : V_G \rightarrow V_G / \equiv_R$ be the canonical mapping from the set of nodes to the set of equivalence classes of nodes. Then G/R is the hypergraph $(V', E_G, lab_G, att', ext')$, where

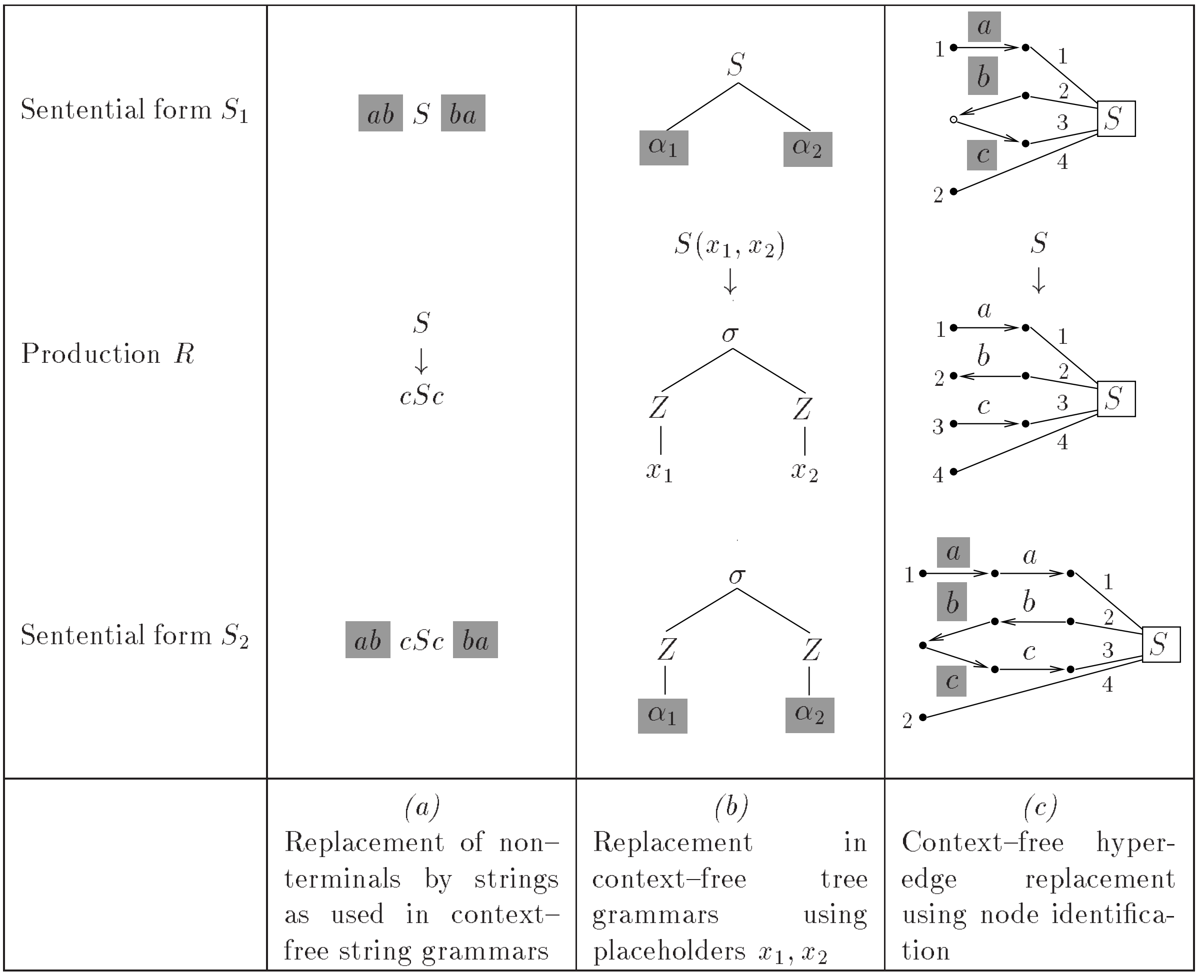


Figure 3.6: Examples for replacement in different kinds of context–free grammars

- $V' = V_G / \equiv_R$,
- $att'(e)(i) = \Phi(att_G(e)(i))$ for every $e \in E_G$ and every $i \in [rank(e)]$ and
- $ext'(i) = \Phi(ext_G(i))$ for every $i \in [rank(G)]$. □

The definition 3.6 allows to identify some nodes in a hypergraph using a binary relation. This is used in the following formal definition 3.7 for the replacement, where a relation R is used to identify all attachment nodes of a hyperedge e to the external nodes of a hypergraph H.

Definition 3.7 *(hyperedge replacement, substition)*
Let G be a hypergraph of rank $n \geq 0$ over the ranked alphabet Γ: $G \in n\text{–}HGR(\Gamma)$. Let $e \in E_G$ be a hyperedge with rank $m \geq 0$ and $H \in m\text{–}HGR(\Gamma)$ be a hypergraph of the same rank m such that G and H are disjoint. The **substitution** of H for e in G, denoted by $G[e/H]$, is the $n-$hypergraph F/R over Γ defined as follows:

- $V_F = V_H \cup V_G$,
- $E_F = (E_G - \{e\}) \cup E_H$,

- lab_F is $lab_H \cup lab_G$ restricted to E_F,
- att_F is $att_H \cup att_G$ restricted to E_F,
- $ext_F = ext_G$ and
- $R = \{(u, v) \in V_F \times V_F | u = att_G(e)(i)$ and $v = ext_H(i)$ for some $i \in [m]\}$. □

In addition to this, $G[e_1/H_1, ..., e_n/H_n]$ is used as an abbreviation for the simultaneous replacement of the pairwise distinct hyperedges $e_1, ..., e_n$ in G by the hypergraphs $H_1, ..., H_n$, respectively. The hypergraph replacement, as defined here, has some nice properties:

- *Sequentialization & parallelization.* First of all, it does not matter, whether the replacement is done simultaneously or one after the other. Formally, if G is a hypergraph with pairwise distinct $e_1, ..., e_n \in E_G$ and H_i is a hypergraph with $rank(H_i) = rank(e_i)$ for all $i \in [n]$, then:
$$G[e_1/H_1, ..., e_n/H_n] = G[e_1/H_1] \cdots [e_n/H_n].$$
- *Confluence.* Furthermore, the order in which replacement is done, does not affect the result. If G is a hypergraph with distinct hyperedges $e_1, e_2 \in E_G$ and H_1, H_2 are hypergraphs of rank $rank(e_1), rank(e_2)$, respectively, then:
$$G[e_1/H_1][e_2/H_2] = G[e_2/H_2][e_1/H_1].$$
- *Associativity.* Last, if in a hypergraph G a hyperedge $e_1 \in E_G$ is replaced by a hypergraph H_1 of rank $rank(e_1)$ and then in the result a hyperedge $e_2 \in E_{H_1}$ is replaced by a hypergraph H_2 of rank $rank(e_2)$, the same result is obtained by first replacing e_2 in H_1 with H_2 and then replacing e_1 in G with the result:
$$G[e_1/H_1][e_2/H_2] = G[e_1/H_1[e_2/H_2]].$$

With this notation of replacement we can now start to define context–free hypergraph grammars. Let N and T be distinct ranked alphabets of non–terminals and terminals, respectively. A **production** over N is an ordered pair $\pi = (\lambda, \rho)$ with $\lambda \in N$, $\rho \in HGR(N \cup T)$ and $rank(\lambda) = rank(\rho)$. As already mentioned, λ and ρ are called the left– and right–hand side of the production π respectively. The notation $lhs(\pi) = \lambda$ and $rhs(\pi) = \rho$ will also be used.

These concepts allow to define the notion of context–free hypergraph grammars as follows:

Definition 3.8 *(context–free hypergraph grammar)*
A context–free hypergraph grammar is a tupel $G = (N, T, S, R)$, where

- N is a set of **non–terminals**,
- T is a set of **terminals** with $N \cap T = \emptyset$,
- $S \in N$ is the **start–symbol**,
- R is a finite set of **productions** of the form (λ, ρ), where $\lambda \in N$, $\rho \in HGR(N \cup T)$ and $rank(\lambda) = rank(\rho)$. □

Now the derivation relation and the language generated by the grammar can be defined as in Definition 3.9. The occurrence of $sing(S)$ is for technical reasons, because $S \in N$ is only a label, but for the derivation a hypergraph is needed.

Definition 3.9 *(hyperedge replacement language)*
Let $G = (N, T, S, R)$ be a context–free hypergraph grammar, let $F \in HGR(N \cup T)$, $e \in E_F$. If $(lab_F(e), \rho) \in R$, then F **directly derives** to $F[e/\rho]$ and we write $F \Rightarrow_G F[e/\rho]$ or just $F \Rightarrow F[e/\rho]$. Based on this **direct derivation**, a sequence of $F_0 \Rightarrow \cdots \Rightarrow F_n$ is called a **derivation of length n**, denoted by $F_0 \Rightarrow_G^n F_n$ or $F_0 \Rightarrow^n F_n$. If the length $n \geq 0$ doesn't matter, this can be written as $F_0 \Rightarrow_G^* F_n$ or $F_0 \Rightarrow^* F_n$. The **hypergraph language** $\lambda(G)$ generated by a hyperedge replacement hypergraph grammar G is the set of all hypergraphs $F \in HGR(T)$, which are derivable from the singular S-labeled hypergraph $sing(S)$, where S is the start–symbol of the grammar G:

$$\lambda(G) = \{F \in HGR(T) \mid sing(S) \Rightarrow_G^* F\}. \qquad \square$$

The class of all hyperedge replacement grammars is denoted by $CFHG$, the subset of $CFHG$ which contains exactly the grammars, which generate trees or strings only, will be denoted by $CFHG_{treegen}$ and $CFHG_{stringgen}$, respectively. The class of all hyperedge replacement languages by $\lambda - CFHG$. A hyperedge replacement grammar is said to be of **order r** ($r \geq 0$), if the ranks of the right–hand sides of all of its productions are less or equal then r — the above introduced classes are then $CFHG_r$ and $\lambda - CFHG_r$.

Example 3.10 *(string–generating hypergraph grammar $G_{abc} \in CFHG_{stringgen}$)*
An example for a string–generating hypergraph grammar is the grammar $G_{abc} = (N, T, S, R)$, where $N = \{S^{(2)}, A^{(4)}\}$, $T = \{a^{(2)}, b^{(2)}, c^{(2)}\}$ and R is the set of productions shown in Figure 3.7.

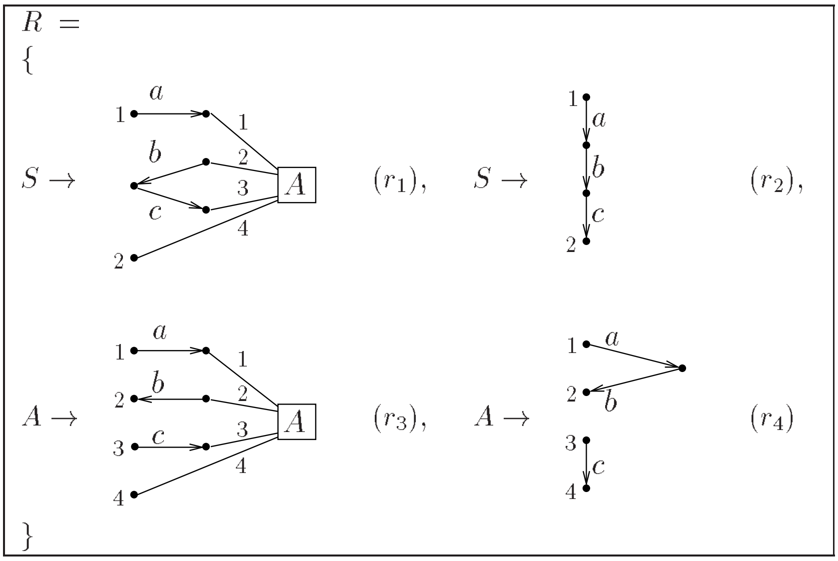


Figure 3.7: Productions of the context–free hypergraph grammar G_{abc}

In Figure 3.8 a derivation of this grammar is shown. It is to be seen, that the grammar produces the language of string–representing hypergraphs $\lambda(G_{abc}) = \{(a^n b^n c^n)^\bullet \mid n \geq 1\}$. The corresponding string language $\{a^n b^n c^n \mid n \geq 1\}$ is well-known for the fact, that it can't be produced by a context–free string grammar, i.e., the notion of context–freeness makes no sense without telling which type of grammar was used.

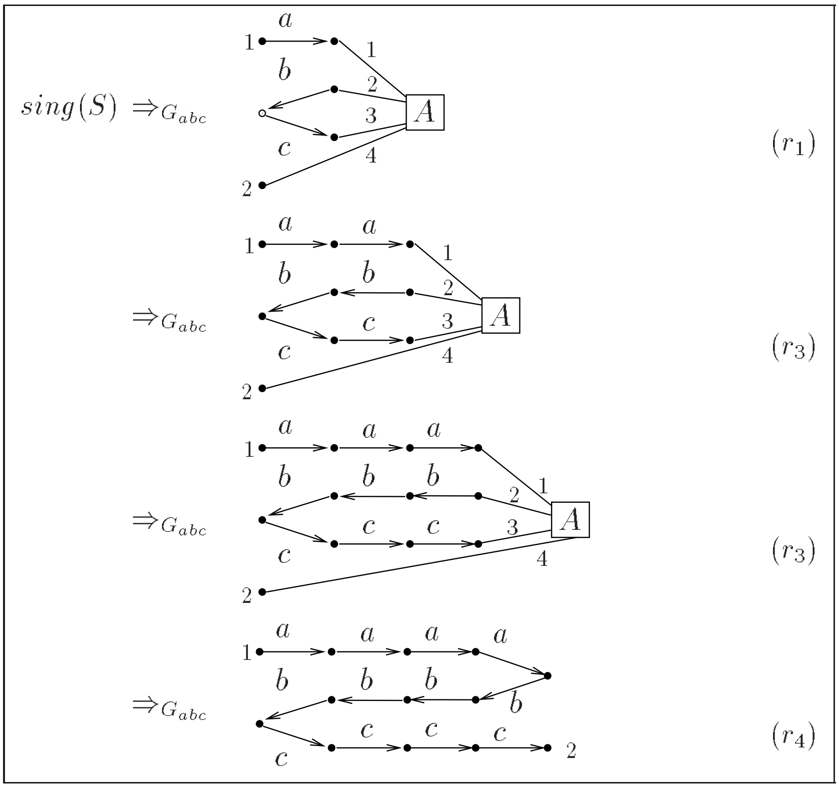


Figure 3.8: An example for a derivation of G_{abc}

Chapter 4

Storage Types

4.1 Storage types

Storage types are an abstract model of information structures like they are used in programming computer machinery, e.g., stacks, lists or even usual numbers. To hide the details of implementation, which are not of interest to the theory at all, a storage type has a uniform structure. Of course a storage is necessary, which can store exactly one element of the so called set of **configurations**. For the manipulation, a set of **instruction** symbols is provided, which elements, the so–called instructions, can be used to change the storage. It is also possible to use the **predicate** symbols to test the storage. A storage type is set up by using an **encoding** symbol on an **input**. The **meaning function** is used to interpret the predicate, instruction and encoding symbols.

Definition 4.1 *(storage type)*
A **storage type S** is a tuple (C, P, F, m, I, E), where

- C is the set of **configurations**,
- P is the set of **predicate** symbols,
- F is the set of **instruction** symbols,
- I is the set of **input** elements,
- E is the set of **encoding** symbols and
- m is the **meaning** function, which associates with every predicate $p \in P$ a mapping $m(p) : C \to \{\texttt{true}, \texttt{false}\}$, with every instruction $f \in F$ a partial function $m(f) : C \to C$ and with every encoding $e \in E$ a partial function $m(e) : I \to C$. $\square$

As to be seen in this definition, the correct term "predicate symbol p" is often replaced by the shorter "predicate p", which is in the correct sense the predicate symbols interpreted by the meaning function, i.e., $m(p)$. The correct meaning will hopefully always be clear from the context.

The simplest case of a storage type is the so called trivial storage type, which is defined as in Definition 4.2.

Definition 4.2 *(trivial storage type)*
The **trivial storage type** $\mathbf{S_0}$ is defined by $S_0 = (C, P, F, m, I, E)$, where

- $C = \{c_0\}$ for some arbitrary, but fixed object c_0,
- $P = \emptyset$,
- $F = \{id\}$,
- $I = C$,
- $E = \{en\}$ and
- $m(id) = m(en) = id(C) = c_0$. □

Obviously, this trivial storage type allows us to store exactly **one** object c_0, but not to test anything (because of $P = \emptyset$) nor to take the object back from storage. The only instruction symbol, $id \in F$, is the identity operator, which leaves the storage in its configuration. Storages containing such an instruction are said to have an identity, which is formally defined in Definition 4.3.

Definition 4.3 *(identity instruction)*
A storage type S **has an identity**, if there is an instruction symbol $id \in F$ such that $m(id) = id(C)$. □

The best way to get an idea about the difference between storages with or without identity is an example. Consider the following storage type, abbreviated P. This is not to be confused with the set of predicate symbols — the correct meaning of P will be always be clear from the context.

Example 4.4 *(pushdown)*
Let Γ be an alphabet. The storage type **Pushdown** is the tuple $P = (C, P, F, I, E, m)$, where

- $C = \Gamma^+$,
- $P = \{top = \gamma \mid \gamma \in \Gamma\} \cup \{bottom\}$,
- $F = \{push(\gamma) \mid \gamma \in \Gamma\} \cup \{pop\}$,
- $I = \{u_0\}$ for a fixed object u_0,
- $E = \Gamma$ and
- $m(top = \gamma)(c) = \mathtt{true}$, iff $\delta = \gamma$,
  $m(bottom)(c) = \mathtt{true}$, iff $\beta = \varepsilon$,
  $m(push(\gamma))(c) = \gamma\delta\beta$,
  $m(pop)(c) = \begin{cases} \beta & \text{iff } \beta \neq \varepsilon, \\ \text{undefined} & \text{otherwise} \end{cases}$
  for every $c = \delta\beta$ with $\delta \in \Gamma$ and $\beta \in \Gamma^+$,
  $m(\gamma)(u_0) = \gamma$ for every $\gamma \in E$.

Obviously, P formalizes a pushdown, which can store elements of the alphabet Γ. To demonstrate this, let Γ be the set $\{\alpha, \beta, \gamma\}$ — then the following table shows the application of instructions, predicates and encodings on the storage type P.

Configuration $c \in C$	Application of predicate	Application of instruction (resp. encoding)
ε	$m(bottom)(c) = \texttt{true}$	$m(\beta)(u_0) = \beta$
$\boxed{\beta}$	$m(top = \gamma)(c) = \texttt{false}$	$m(push(\gamma))(c) = \alpha\beta$
$\boxed{\alpha}\boxed{\beta}$	$m(top = \alpha)(c) = \texttt{true}$	$m(push(\gamma))(c) = \gamma\alpha\beta$
$\boxed{\gamma}\boxed{\alpha}\boxed{\beta}$	$m(bottom)(c) = \texttt{false}$	$m(push(\alpha))(c) = \alpha\gamma\alpha\beta$
$\boxed{\alpha}\boxed{\gamma}\boxed{\alpha}\boxed{\beta}$	$m(top = \alpha)(c) = \texttt{true}$	$m(pop)(c) = \alpha$
$\boxed{\gamma}\boxed{\alpha}\boxed{\beta}$	$m(top = \gamma)(c) = \texttt{true}$	$m(pop)(c) = \gamma$
$\boxed{\alpha}\boxed{\beta}$	$m(bottom)(c) = \texttt{false}$	...

In the table — as in Example 4.4, too — the stack is written from left to right, with the last–pushed symbol as the leftmost one.

Now it can be seen, that there is no operation symbol in this storage type, which leaves it in the same state - every operation changes the configuration of the storage (if we decide to interprete the "*pop*"-operation on the empty stack as an exception). Of course we can add the operation "*stay*" to the pushdown, thus obtaining a pushdown P_{id} with **identity**:

Example 4.5 *(pushdown with identity)*
The store type **Pushdown with identity** $P_{id} = (C, P, F', I, E, m')$ can be obtained from the storage type Pushdown from Example 4.4 by

- extending F to $F' = F \cup \{stay\}$ and
- defining m' to be the same as m on $P \cup F \cup E$ and $m'(stay) = id(C)$.

If we apply the operation "*stay*" to the result of the last instruction in the table above, $\alpha\beta$, the result is $\alpha\beta$, too. This construction can be generalized to provide an storage type with identity for every store type S without. The operation "*stay*" is called "*id*" in the general case:

Definition 4.6 *(storage type with identity)*
For a storage type $S = (C, P, F, I, E, m)$, **S with identity** is the storage type $S_{id} = (C, P, F \cup \{id\}, I, E, m')$, where id is a "new" instruction symbol, m' is the same as m on $P \cup F \cup E$, and $m'(id) = id(C)$. □

Considerating the example pushdown, one can ask how to construct a pushdown of pushdowns of elements of an alphabet Γ. One possibility is to define it "manually", but there is a more elegant way to construct such a storage type. In the following, the pushdown is not a storage type but an operator, which makes a pushdown from a given storage type. This pushdown operator is defined as follows:

Definition 4.7 *(pushdown operator)*
Let S be the storage type (C, P, F, m, I, E). The **pushdown of S**, denoted by $\mathbf{P}(S)$, is the storage type (C', P', F', m', I', E'), where

- $C' = (\Gamma \times C)^+$ and Γ is the fixed infinite set of pushdown symbols,
- $P' = \{top = \gamma\} \cup \{test(p) \mid p \in P\}$,
- $F' = \{push(\gamma, f), stay(\gamma, f) \mid \gamma \in \Gamma, f \in F\} \cup$ $\{stay(\gamma) \mid \gamma \in \Gamma\} \cup \{pop\} \cup \{id\}$
- for every $c' = (\delta, c)$ with $\delta \in \Gamma, c \in C$, and $\beta \in C' \cup \{\varepsilon\}$,
  $m'(top = \gamma)(c') = \mathtt{true}$, if $\delta = \gamma$,
  $m'(test(p))(c') = m(p)(c)$,
  $m'(push(\gamma, f))(c') = \begin{cases} (\gamma, c_1)(\delta, c)\beta & \text{iff } m(f) \text{ is defined on } c \text{ and } c_1 = m(f)(c), \\ \text{undefined} & \text{otherwise,} \end{cases}$
  $m'(stay(\gamma, f))(c') = \begin{cases} (\gamma, c_1)\beta & \text{iff } m(f) \text{ is defined on } c \text{ and } c_1 = m(f)(c), \\ \text{undefined} & \text{otherwise,} \end{cases}$
  $m'(stay(\gamma))(c') = (\gamma, c)\beta$,
  $m'(pop)(c') = \begin{cases} \beta & \text{if } \beta \neq \varepsilon, \\ \text{undefined} & \text{otherwise,} \end{cases}$
  $m'(id)(c') = (\delta, c)\beta$,
- $I' = I$ and
- $E' = \{\lambda u \in I.(\gamma_0, e(u)) \mid \gamma_0 \in \Gamma, e \in E\}$. □

The usual pushdown now can be interpreted as $P = \mathbf{P}(S_0)$, with the first P denoting the usual pushdown and the second standing for the pushdown-operator. If we do so, we will still use the usual simple *push*– and *pop*–notion. It should be clear, that the pushdown operator is really an operation taking one storage type and translating it into another. There is just another very useful operation on storage types — the multiplication — which allows to build tupels of storage types which can be tested and manipulated in parallel, providing the storage type concept with a greater generality.

Definition 4.8 *(product of storage types)*
Let $S_1 = (C_1, P_1, F_1, I_1, E_1, m_1)$ and $S_2 = (C_2, P_2, F_2, I_2, E_2, m_2)$ be storage types. Their **product** $S = S_1 \times S_2$ is the storage type $(C_1 \times C_2, P_1 \times P_2, F_1 \times F_2, I_1 \times I_2, E_1 \times E_2, m)$ where

- $m(p_1, p_2)(c_1, c_2) = m_1(p_1)(c_1) \ \mathtt{and}\ m_2(p_2)(c_2)$
- $m(f_1, f_2)(c_1, c_2) = (m_1(f_1)(c_1), m_2(f_2)(c_2))$
- $m(e_1, e_2)(i_1, i_2) = (m_1(e_1)(i_1), m_2(e_2)(i_2))$

for every $c_k \in C_k, p_k \in P_k, f_k \in F_k, e_k \in E_k$ and $i_k \in I_k$ for $k \in \{1, 2\}$. □

Remark:
Definition 4.8 differs from the product defined in [Eng86], where the sets of predicates are expected to be distinct and are not pairwise combined but added together. Predicates which test both storage types are build using BOOLEan expressions. We decide to go the other way, because it simplifies the build of products of equal storage types, which can be very useful in controlled grammars.

4.2 Controlled grammars

Over the past decades a wide variety of grammar–like generation mechanisms have been defined. Most of them can be defined by enriching two basic types of grammars in a certain way. These two types are the regular and the context–free grammars. The kind of enriching we will consider, is the control of the derivation process. I.e., we will impose certain conditions that must hold in order for a particular rule to be applicable. For this control mechanism we use storage types. Then, rules of a grammar are coupled with tests on the configuration of a storage type. The resulting concept is called storage–controlled grammar. The definition is not limited to a special type of context–free grammar, so they can be used for context–free string– as well as for context–free hypergraph grammars. This makes the storage–controlled grammar to a framework with two degrees of freedom: the type of the used grammar X and the type of the used storage type S. We write $X(S)$–grammar for a grammar of the type X controlled by the storage type S. A lot of relations between $X(S)$–grammars and other grammars are known, Figure 4.1 on page 32 shows some special cases for grammar X and storage S, in which the $X(S)$–grammar is equal or closely related to another well–known grammar–notion (or transducer, too). Possible choices of X are:

- REG: The class of regular grammars, i.e., a context–free string grammar with only one or no non–terminal on the right–hand sides of all rules (see [Cho56] for further information).
- CF: The well–known context–free string grammars (see [Cho56], too).
- RT: The class of regular tree grammars. This means that all non–terminals are of rank 0, thus they are always at the front of a sentential form (for details see [Bra69]).
- CFT: The class of context–free tree grammars, which are also called macro–grammars (for further information see [Fis68]).
- $CFHG$: The class of the context–free hypergraph grammars as introduced in the previous chapter.

Definition 4.9 *(storage–controlled grammar)*
Let $S = (C, P, F, I, E, m)$ be a storage type and let $X \in \{REG, CF, RT, CFT, CFHG\}$ be a grammar type. An **X(S)-grammar** is a tuple $G = (N, T, S_0, R, en)$, where

- N is the alphabet of non–terminals,
- T is the alphabet of terminals,
- S_0 is a single start symbol, taken from N: $S_0 \in N$,
- R is a finite set of productions of the form $A \to \texttt{if}\ cond\ \texttt{then}\ \xi$ with the non–terminal $A \in N$, the condition $cond \in BE(P)$ and $\xi \in L_X(N(F) \cup T)$ and
- en is the encoding symbol: $en \in E$.

The set $L_X(N(C) \cup T)$ is called the set of total configurations. The derivation relation of G, denoted by $\Rightarrow_G$ (abbreviated by $\Rightarrow$, if the grammar is clear from the context), is the binary relation on the set of total configurations with the following property: $(\xi_1 A(c)\xi_2, \xi_1\xi'\xi_2)$ is an element of $\Rightarrow_G$, if and only if:

- There is a rule $A \to$ `if` *cond* `then` ξ in R,
- $m(cond)(c) =$ `true`,
- $m(f)(c)$ is defined for all f occuring in ξ and
- ξ' can be obtained from ξ by replacing every $B(f)$ by $B(m(f)(c))$. For a grammar type X and a storage type S the set of all $X(S)$–grammars is denoted by $X(S)$. $\square$

Remarks:

1. In the definition $L_X(\Gamma)$ is used as abbreviation for the greatest language constructable from the set Γ (without restrictions as regular or context–free mechanism), e.g., $L_{REG}(\Gamma) = L_{CF}(\Gamma) = \Gamma^*$, $L_{RT}(\Gamma) = L_{CFT}(\Gamma) = T_\Gamma$ and $L_{CFHG}(\Gamma) = HGR(\Gamma)$.
2. In the Definition 4.9 can be seen, that only non–terminals get equipped with storage types. The reason is that the desired output is free of configurations and storage types on non–terminals can only be additional information — but it is obvious, that they cannot take influence on the further derivation.
3. A total configuration is defined as a sequence of pairs, but can be interpreted as a sequence of non–terminals with parameter (which are the storage type S), too. So we will write $A(S) \to \rho$ instead of $A \times S \to \rho$ and furthermore the S is dropped from the non–terminal on the left–hand side. On the right–hand side of all rules, the non–terminals are equipped with instruction symbols and the storage type, but because of the fact that the storage type is the everywhere the same, it can be dropped too, so we write $A \to B(F)$ rather than $A(S) \to B(F(S))$. The same is valid for the condition.
4. To support an easy notation of rules, it is allowed to take conditions not only from the set of predicates P, but also from the set of BOOLEan expressions over P, $BE(P)$. Additionaly, two rules $A \to$ `if` p `then` ξ_1 and $A \to$ `if not` p `then` ξ_2 can be written $A \to$ `if` p `then` ξ_1 `else` ξ_2.
5. Furthermore, we allow to take right–hand sides from $L_X(N(F^+) \cup T)$ and not only from $L_X(N(F) \cup T)$. Then a semicolon is used to separate the instruction symbols.

A grammar can be viewed as a transducer, transducing the start–symbol into the generated language. A $X(S)$–grammar is, because of the input from the storage type, in fact both transducer and grammar. The language generated by a $X(S)$–grammar, the set of accepted inputs and the translation of it can now be defined.

Definition 4.10 *(translation, generated language, accepted input set)*
Let G be a $X(S)$–grammar as defined above.
The **translation** of G is the binary relation $\tau(G)$ defined as:
$\tau(G) = \{(i, \phi) \in I \times L_X(T) \mid S_0(m(en)(i)) \Rightarrow_G^* \phi \text{ for some } i \in I\}$.
The **language generated** of G is denoted by $\lambda(G)$ and is defined as follows:
$\lambda(G) = \{\phi \in L_X(T) \mid S_0(m(en)(i)) \Rightarrow_G^* \phi \text{ for some } i \in I\}$.
The **accepted input–set** of G is denoted by $\alpha(G)$ and is defined as follows:
$\alpha(G) = \{i \in I \mid S_0(m(en)(i)) \Rightarrow_G^* \phi \text{ for some } \phi \in L_X(T)\}$.
The set of all languages generatable by grammars of the type $X(S)$ is denoted by $\mathcal{L}(X(S))$:
$\mathcal{L}(X(S)) = \{\lambda(G) \mid G \in X(S)\}$. $\square$

Remarks:

1. For a comment on the notation $L_X(\Gamma)$ see the first remark on Definition 4.9.

2. The translation relation shows, that the $X(S)$–grammar is really closely related to a transducer. The languages are only the range and the domain of this relation, because obviously $\lambda(G) = ran(\tau(G))$ and $\alpha(G) = dom(\tau(G))$.

3. A transducer is called deterministic, if for every state and every input symbol exists **exactly one** production. An **deterministic X(S)–transducer** can have to rules for the same situation, for instance $A \to \texttt{if}\ p_1\ \texttt{then}\ \xi_1$ and $A \to \texttt{if}\ p_2\ \texttt{then}\ \xi_2$, if $m(p_1\ \texttt{and}\ p_2)(c) = \texttt{false}$ for every $c \in C$, i.e., the rules exclude each other because of their conditions.

4. We will use both the $\lambda(G)$ and the $\mathcal{L}X(S)$ notion for usual, uncontrolled grammar concepts, too, and we allow to attach an index to the λ and $\mathcal{L}$ which converts the languages into the desired type, e.g., $\lambda_s(G)$ is the string language generated by G and $\mathcal{L}(X(S))$ is the set of all string languages generatable by $X(S)$–grammars.

Storage types are not only intended to model information structures (i.e., to save something), they also be used to synchronize the derivation of different parts of the sentential form, too. A very simple example is the **countdown** storage type, like it is introduced in Definition 4.11.

Definition 4.11 *(countdown storage type)*
The storage type **Countdown** is the tuple $Countdown = (C, P, F, I, E, m)$, where

- C is the set of nonnegative integers,
- $P = \{null\}$,
- $F = \{dec\}$,
- $I = C$,
- $E = \{en\}$,
- and for every $c \in C$:
  $m(null)(c) = \texttt{true}$ iff $c = 0$,
  $m(dec)(c+1) = \begin{cases} c & \text{iff } c \geq 1 \\ \text{undefined} & \text{otherwise} \end{cases}$ and
  $m(en) = \lambda n \in I.n$ □

Example 4.12 *($E0L \subseteq CF(Countdown)$)*
As an example for the synchronization with storage types, consider the (string–) language $L = \{a^{2^n} b^{2^n} c^{2^n} \mid n \geq 1\}$, which can be generated by an L–system, but not by a context–free string grammar. The following table shows the definitions and a derivation for the word $a^4b^4c^4$. The parallelization inherent in the L–system can be modeled by a serial, but controlled context–free string grammar, because the storage type can be copied in one production, as in the rule $A \to \texttt{if}\ null\ \texttt{then}\ a\ \texttt{else}\ A(dec)A(dec)$, where the storage type binded to the left–side A is given to both non–terminals A on the right–hand side.

L–system	Controlled context–free string grammar
$L = \{N \cup T, R, s\}$ with $N \cup T = \{s, a, b, c\}$ $R = \{s \to abc,$ $\quad a \to aa,$ $\quad b \to bb,$ $\quad c \to cc\}$	$G(Countdown) = \{N, T, R, S, en\}$ with $N = \{S, A, B, C\}$ $T = \{a, b, c\}$ $R = \{S \to A(dec)B(dec)C(dec),$ $\quad A \to$ `if` $null$ `then` a $\qquad$ `else` $A(dec)A(dec),$ $\quad B \to$ `if` $null$ `then` b $\qquad$ `else` $B(dec)B(dec),$ $\quad C \to$ `if` $null$ `then` a $\qquad$ `else` $C(dec)C(dec)\}$
$s \Rightarrow abc$ $\Rightarrow aabbcc$ $\Rightarrow aaaabbbbcccc = a^4b^4c^4$	$S(3) \Rightarrow A(2)B(2)C(2)$ $\Rightarrow A(1)A(1)B(2)C(2)$ $\Rightarrow A(0)A(0)A(0)A(0)B(2)C(2)$ $\Rightarrow^* A(0)\cdots B(0)\cdots B(0)C(0)\cdots C(0)$ $\Rightarrow^* aaaabbbbcccc = a^4b^4c^4$

Another very useful storage type is the **one–way** storage type, which simulates the input tape of an one–way automat.

Definition 4.13 *(One–way storage type)*
The storage type **One-way** is the tuple $One\text{–}way = (C, P, F, I, E, m)$, where

- $C = \Omega^*$ for some fixed set Ω of input symbols,
- $P = \{first = a \mid a \in \Omega\} \cup \{empty\}$,
- $F = \{read\}$,
- for $c = bw \in \Omega^* \cup \{\varepsilon\}$ (with $b \in \Omega \cup \{\varepsilon\}, w \in \Omega^*$)
$$m(first = a)(c) = \begin{cases} \texttt{true} & \text{iff } b = a \\ \texttt{false} & \text{otherwise,} \end{cases}$$
$$m(empty)(c) = \begin{cases} \texttt{true} & \text{iff } b = \varepsilon \\ \texttt{false} & \text{otherwise,} \end{cases}$$
$$m(read)(c) = \begin{cases} w & \text{iff } b \neq \varepsilon \\ \text{undefined} & \text{otherwise,} \end{cases}$$
- $I = \Omega^*$ and
- $E = \{\Sigma \mid \Sigma \text{ is a finite subset of } \Omega\}$ with $m(\Sigma) = id(\Sigma^*)$ for every $\Sigma \in E$.

The following Example table shows the application of predicates and instruction. The used alphabet is $\Gamma = \{\alpha, \beta, \gamma\}$. Note that *One–way* has no identity nor a possibility to write something to the "tape".

Configuration $c \in C$	Application of predicate	Application of instruction (resp. encoding)
ε	$m(empty)(c) = \texttt{true}$	$m(\Gamma)(\alpha\beta\gamma) = \alpha\beta\gamma$
$\boxed{\alpha}\boxed{\beta}\boxed{\gamma}$	$m(first = \gamma)(c) = \texttt{false}$	$m(read)(c) = \beta\gamma$
$\boxed{\beta}\boxed{\gamma}$	$m(first = \beta)(c) = \texttt{true}$	$m(read)(c) = \gamma$
$\boxed{\gamma}$	$m(empty)(c) = \texttt{false}$	$m(read)(c) = \varepsilon$
ε	$m(empty)(c) = \texttt{true}$	...

It is possible to keep trees in storage types too, depending on the provided instructions **Tree** and **Treewalk** can be defined. The first is related to top–down strategies, the second to tree–walking like it is done in attribute grammars.

Definition 4.14 *(Tree storage type)*
The storage type **Tree** is the tuple $TR = (C, P, F, I, E, m)$, where

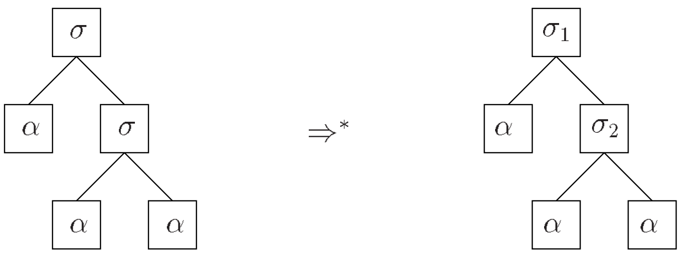
- $C = T_\Omega$ for some fixed ranked set Ω, such that Ω_k is infinite for every $k \geq 0$,
- $P = \{root = \sigma \mid \sigma \in \Sigma\}$,
- $F = \{sel_i \mid i \in \{1, 2, 3, ...\}\}$,
- for every $c = \tau t_1 ... t_k \in T_\Omega$, with $\tau \in \Omega_k$, $k \geq 0$, and $t_1, ..., t_k \in T_\Omega$:
  $m(root = \sigma)(c) = (\tau = \sigma)$,
  $$m(sel_i)(c) = \begin{cases} t_i & \text{iff } 1 \leq i \leq k \\ \text{undefined} & \text{otherwise} \end{cases}$$
- $I = T_\Sigma$ and
- $E = \{\Sigma \mid \Sigma \text{ is a finite subset of } \Omega\}$ with $m(\Sigma) = id(T_\Sigma)$ for every $\Sigma \in E$.

An example for the usage of the instructions and predicates follows. The ranked alphabet is $\Gamma = \{\gamma^{(2)}, \beta^{(1)}, \alpha^{(0)}\}$. Instead of the tree the represented terms are shown.

Configuration $c \in C$	Application of predicate	Application of instruction (resp. encoding)
ε	$m(root = \gamma)(c) = \texttt{false}$	$m(\Gamma)(\gamma(\beta(\alpha), \gamma(\alpha, \alpha))) = \gamma(\beta(\alpha), \gamma(\alpha, \alpha))$
$\gamma(\beta(\alpha), \gamma(\alpha, \alpha))$	$m(root = \gamma)(c) = \texttt{true}$	$m(sel_1)(c) = \beta(\alpha)$
$\beta(\alpha)$	$m(root = \alpha)(c) = \texttt{false}$	$m(sel_1)(c) = \alpha$
α	$m(root = \alpha)(c) = \texttt{true}$	...

The following example compares the work of a top–down tree–transducer with the one of the tree–controlled tree grammar. Because of the decision to write the configuration as a parameter to non–terminals, there is no difference to be seen during the derivation at all. Below the derivations the tree and the translated tree are shown.

Top–down tree transducer	Controlled regular tree grammar
$T = \{Q, \Sigma, \Delta, q_1, R\}$ with $Q = \{q_1^{(1)}, q_2^{(1)}\}$ $\Sigma = \{\sigma^{(2)}, \alpha^{(0)}\}$ $\Delta = \{\sigma_1^{(2)}, \sigma_2^{(2)}, \alpha^{(0)}\}$ $R = \{$ $q_1(\sigma(x_1, x_2)) \to \sigma_1(q_2(x_1), q_2(x_2)),$ $q_2(\sigma(x_1, x_2)) \to \sigma_2(q_1(x_1), q_1(x_2)),$ $q_1(\alpha) \to \alpha,$ $q_2(\alpha) \to \alpha$ $\}$	$G(TR) = \{N, T, R, Q_1, en\}$ with $N = \{Q_1, Q_2\}$ $T = \{\sigma_1^{(2)}, \sigma_2^{(2)}, \alpha^{(0)}\}$ $en = \{\sigma^{(2)}, \alpha^{(0)}\}$ $R = \{$ $Q_1 \to \texttt{if } root = \sigma \texttt{ then } \sigma_1(Q_2(sel_1), Q_2(sel_2)),$ $Q_2 \to \texttt{if } root = \sigma \texttt{ then } \sigma_2(Q_1(sel_1), Q_1(sel_2)),$ $Q_1 \to \texttt{if } root = \alpha \texttt{ then } \alpha,$ $Q_2 \to \texttt{if } root = \alpha \texttt{ then } \alpha$ $\}$
$\underline{q_1(\sigma(\alpha, \sigma(\alpha, \alpha)))}$ $\Rightarrow \sigma_1(\underline{q_2(\alpha)}, q_2(\sigma(\alpha, \alpha)))$ $\Rightarrow \sigma_1(\alpha, \underline{q_2(\sigma(\alpha, \alpha))})$ $\Rightarrow \sigma_1(\alpha, \sigma_2(\underline{q_1(\alpha)}, q_1(\alpha)))$ $\Rightarrow \sigma_1(\alpha, \sigma_2(\alpha, \underline{q_1(\alpha)}))$ $\Rightarrow \sigma_1(\alpha, \sigma_2(\alpha, \alpha))$	$\underline{Q_1(\sigma(\alpha, \sigma(\alpha, \alpha)))}$ $\Rightarrow \sigma_1(\underline{Q_2(\alpha)}, Q_2(\sigma(\alpha, \alpha)))$ $\Rightarrow \sigma_1(\alpha, \underline{Q_2(\sigma(\alpha, \alpha))})$ $\Rightarrow \sigma_1(\alpha, \sigma_2(\underline{Q_1(\alpha)}, Q_1(\alpha)))$ $\Rightarrow \sigma_1(\alpha, \sigma_2(\alpha, \underline{Q_1(\alpha)}))$ $\Rightarrow \sigma_1(\alpha, \sigma_2(\alpha, \alpha))$



This result is no surprise at all, more interesting is the relation between $X(S)$–grammars and attribute grammars. A RT–grammar, which is controlled by the storage type Treewalk (see Definition 4.15), is in fact very closely related to attribute grammars, viewed as tree transducers (usually called **attributed tree transducers**). The main difference between attribut grammar and attributed tree transducer is that the first has a semantic domain, the second one has not.

Definition 4.15 *(Treewalk storage type)*
The storage type **Treewalk** is the tuple $TW = (C, P, F, I, E, m)$, where

- $C = \{(t, \pi) \mid t \in T_\Omega, \pi \in occ(t)\}$ for some fixed ranked set Ω, such that Ω_k is infinite for every $k \geq 0$,
- $P = \{label = \sigma \mid \sigma \in \Sigma\} \cup root \cup \{son = i \mid i \in \{1, 2, 3...\}\}$,
- $F = \{down_i \mid i \in \{1, 2, 3, ...\}\} \cup \{up, stay\}$,
- $m(label = \sigma)((t, \pi)) = label_t(\pi) = \sigma$,
  $m(root)((t, pi)) = \texttt{true}$ iff $\pi = \varepsilon$,
  $m(son = i)((t, \pi)) = \texttt{true}$ iff $\exists \pi' \in occ(t) : \pi = \pi' i$,
  $$m(down_i)((t, \pi)) = \begin{cases} (t, \pi i) & \text{iff } 1 \leq i \leq rank_\Omega(label_t(\pi)) \\ \text{undefined} & \text{otherwise,} \end{cases}$$
  $$m(up)((t, \pi)) = \begin{cases} (t, \pi') & \text{iff } \exists i \in \{1, 2, 3...\} : \pi = \pi' i \\ \text{undefined} & \text{otherwise,} \end{cases}$$

$m(stay)(t,\pi)) = (t,\pi)$,

- $I = T_\Omega$ and
- $E = \{\Sigma \mid \Sigma \text{ is a finite subset of } \Omega\}$ with $m(\Sigma)(t) = (t,\varepsilon)$ for every $\Sigma \in E$. $\square$

The following table shows some major results in expressing well–known concepts in terms of $X(S)$–grammars. There are relationship to automata too, but this is beyound the scope of this work. For further results see [Eng86].

	CF Context–free string grammars	RT Regular tree grammars
Pushdown	The $CF(Pushdown)$ grammars are essentially the **indexed grammar** introduced by Aho in [Aho68] as an extension of context–free grammars.	The $RT(Pushdown)$ grammar equals the context–free tree grammar (or indexed tree grammar), the yields of these tree languages are the indexed languages.
Countdown	$CF(Countdown)$ equals the well-known **E0L–grammar**, as demonstrated in Example 4.12.	
Tree	The $CF(TR)$ transducer is the **top–down tree–to–string** transducer, which is also called **generalized syntax–directed translation scheme** (see [ERS80] for details).	The $RT(TR)$ transducer is the **top–down tree transducer** (see [Tha70, Rou70].
Treewalk		The $RT(TW)$ transducer is very closely related to the **attribute grammar** defined by Knuth (see [Knu68, Knu71]).

Figure 4.1: Controlled grammars as a universal framework

Chapter 5

Storage Controlled Context–free Hypergraph Grammars

It is well–known that node–replacement grammars such as C–edNCE grammars (for a survey see [ER96, Eng96]) are more powerful than context–free hypergraph grammars for generating sets of graphs ([ER90, Eng96]). Thus, it is worth to examine the generative power of hyperedge replacement grammars equipped with additional constructs like storage control. Besides this, controlled grammars can be interpreted as transducers which translate from the entry of the storage type into elements of the hypergraph language which is described by the underlying grammar.

In this chapter the concept of storage control will be added to context–free hypergraph grammars. This is mainly done by giving examples for various storage types. It is of interest, whether there exist hypergraph languages, which can't be constructed with pure context–free hypergraph grammars, but can be constructed by the same grammar type enriched with control. A further point of interest is the question, how $CFT(S)$ and $CFHG_{treegen}(S)$ are related for various storage types.

5.1 String generating context–free hypergraph grammars

The language G_{abc} presented in Example 3.10 illustrates the fact, that string–generating context–free hypergraph grammars have a higher string–generating power than usual context–free string grammars. On the other hand, it is known, that controlled context–free string grammars are more powerful than uncontrolled ones, e.g., the controlled grammar G_{square}, which is introduced in Example 5.3, generates a language unconstructable with a usual context–free grammar and even unconstructable by a context–free hypergraph grammar (this is known from [Hab92], where this language is Example 2.9 of Chapter 4 and is shown to violate the Theorem 2.6 (linear–growth theorem) in the same chapter). If we use $\mathcal{L}_s(A)$ to denote the set of all **string** languages generatable with grammars of type A, we have the following relations:

Lemma 5.1 $\mathcal{L}_s(CF) \subset \mathcal{L}_s(CFHG_{stringgen})$ □

This can be proofed using the following relations:

1. $\mathcal{L}_s(CF) \subseteq \mathcal{L}_s(CFHG_{stringgen})$ (which is obvious, all that can be done by a context–free string–grammar can be done by a context–free hypergraph grammar, too) and

2. $\lambda_s(G_{abc}) \in \mathcal{L}_s(CFHG_{stringgen}) - \mathcal{L}_s(CF)$ (with G_{abc} is the grammar from Example 3.10)

Lemma 5.2 $\mathcal{L}_s(CF) \subset \mathcal{L}_s(CF(Countdown))$ $\square$

This can be proofed with the two relations:

1. $\mathcal{L}_s(CF) \subseteq \mathcal{L}_s(CF(Countdown))$ (which is obvious — if we not test the storage type *Countdown*, $CF(Countdown)$ degenerates to CF, thus $CF(Countdown)$ can generate at least all the languages which can be generated by CF), and

2. $\lambda_s(G_{square}) \in \mathcal{L}_s(CF(Countdown)) - \mathcal{L}_s(CF)$ (where G_{square} is the grammar in Example 5.3)

Considering the following Example 5.3, another lemma can be stated.

Example 5.3 *(*Context–free string grammar G_{square}*)*

Consider the *Countdown*–controlled grammar $G_{square}(Countdown) = (N, T, S, R, en)$ with

- $N = \{S, G\}$,
- $T = \{a\}$ and
- $R = \{$
  $S \rightarrow \texttt{if } null \texttt{ then } \varepsilon \texttt{ else } S(dec)G(dec),$
  $G \rightarrow \texttt{if } null \texttt{ then } a \texttt{ else } aaG(dec)\}$
  $\}$

Derivations using G_{square} look like this:

$$
\begin{array}{rcl}
S(3) & \Rightarrow & S(2)G(2) \\
 & \Rightarrow & S(1)G(1)G(2) \\
 & \Rightarrow & S(0)G(0)G(1)G(2) \\
 & \Rightarrow & G(0)G(1)G(2) \\
 & \Rightarrow & aG(1)G(2) \\
 & \Rightarrow & a^3G(0)G(2) \\
 & \Rightarrow & a^4G(2) \\
 & \Rightarrow & a^6G(1) \\
 & \Rightarrow & a^8G(0) \\
 & \Rightarrow & a^9
\end{array}
$$

Thus, the translation defined by this grammar is $\tau(G_{square}) = \{(n, a^{n^2}) \mid n \in \mathbb{N}\} \subset \mathbb{N} \times A^*$.

Lemma 5.4 $\mathcal{L}_s(CF(Countdown)) - \mathcal{L}_s(CFHG_{stringgen}) \neq \emptyset$ $\square$

This bases on the mentioned fact, that Example 5.3 violates the linear–growth theorem and therefore cannot be generated by $CFHG_{stringgen}$–grammars.

If we would found a language which can be generated by $CFHG_{stringgen}$– and not by $CF(Countdown)$–grammars, we could view this relations as a Hasse diagramm. In fact, it is very likely that G_{ww} from the following Example 5.5 is such a grammar.

Example 5.5 *(grammar producing $\{ww \mid w \in \Sigma^*\}$)*
Consider the context–free hypergraph grammar $G_{ww} = (N, T, S, R)$, where

- $N = \{S^{(2)}, A^{(4)}\}$,
- $T = \{a^{(2)}, b^{(2)}, c^{(2)}\}$ and
- $R = R_a \cup R_b \cup R_c$ where R_x is shown in Figure 5.1 for $x \in \{a, b, c\}$.

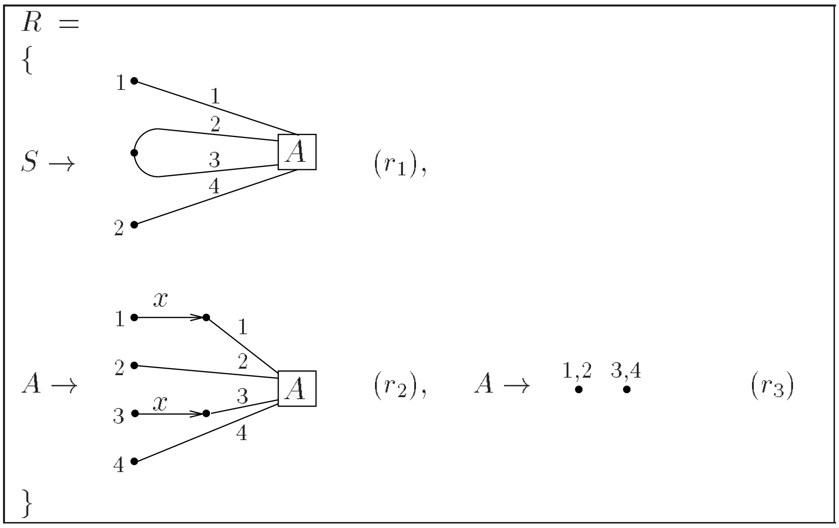


Figure 5.1: Subset R_x of the productions of the context–free hypergraph grammar G_{ww}

An example derivation for this grammar G_{ww} is shown in Figure 5.2.
It is easy to recognize, that the generated language is

$$\lambda(G_{ww}) = \{(ww)^{\bullet} \mid w \in T^*\}$$

Although sharing like that in the grammar G_{abc} from Example 3.10 can be simulated by using the storage type *Countdown*, in this case it seems to be impossible, so we can claim the following:

Claim 5.6 $\mathcal{L}_s(CFHG_{stringgen}) - \mathcal{L}_s(CF(Countdown)) \neq \emptyset$ □

But we can go on and find a grammar which is generatable with a $CFHG_{stringgen}(Countdown)$, but not by a $CF(Countdown)$ or a $CFHG_{stringgen}$: We take the two languages which show the incomparability of $CF(Countdown)$ and $CFHG_{stringgen}$ and concatenate them to the language

$$\{wwa^{n-1^2} \mid w \in \{a,b,c\}^* \text{ and } |w| = n-1 \text{ and } n \geq 1\}$$

This language violates the mentioned linear growth theorem, too, and it is not constructable by a $CF(Countdown)$–grammar, but it is constructable by a $CFHG_{stringgen}(Countdown)$–grammar, as it can be seen in Example 5.7.

Example 5.7 *(A real $CFHG_{stringgen}(Countdown)$ language)*
The grammar $G_{wwsquare}$ is analogous to G_{ww}, so we construct it based on G_{ww}. Let $G_{wwsquare}(Countdown) = (N, T, S, R, en)$ with

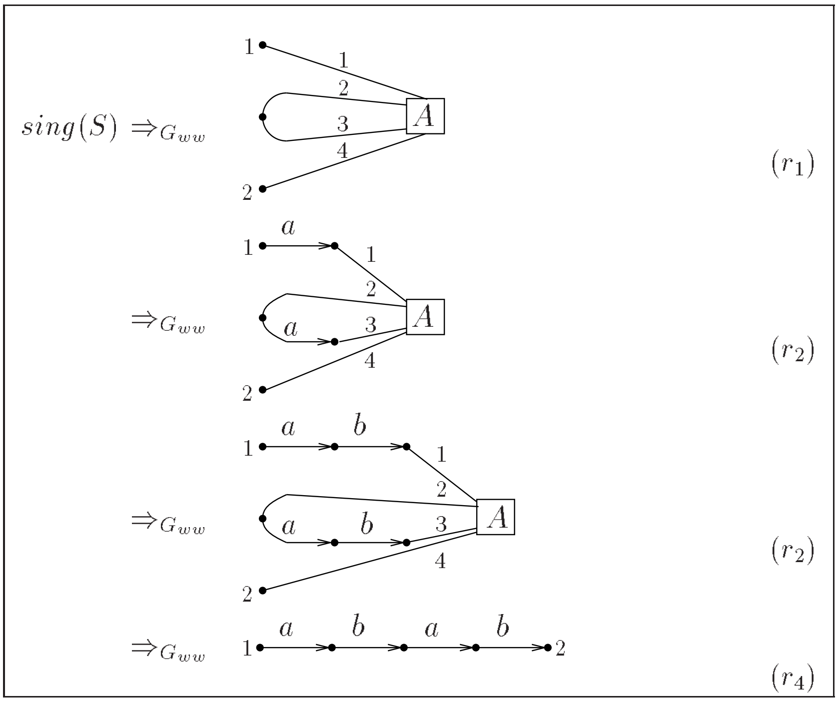


Figure 5.2: An example for a derivation of G_{ww}

- $N = N_{G_{ww}} \cup \{B^{(2)}, C^{(2)}\}$,
- $T = T_{G_{ww}}$,
- R is the same as $R_{G_{ww}} \cup R_B$ with little changes:
  - Rule r_1 is replaced by the rule r_1' from Figure 5.3.
  - Rule r_2 has the condition `if not` *null* `then` ... and an instruction *dec* is attached to the non–terminal A on the right–hand side.
  - Rule r_3 is taken over identically as an unconditioned rule.
  - $R_B = \{B \rightarrow$ `if` *null* `then` ε `else` $B(dec)C(dec)$,
    $C \rightarrow$ `if` *null* `then` a `else` $aaC(dec)\}$
    (because of $rank(B) = rank(C) = 2$ we write strings as an abbreviation)

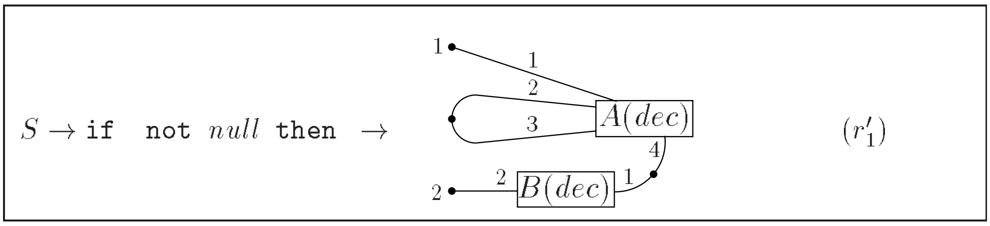


Figure 5.3: Production r_1' of the controlled hypergraph grammar $G_{wwsquare}$

Comparing the grammar with the grammars G_{square} from Example 5.3 and G_{ww} from the Example 5.5, one can see, that the generated langage is really the above mentioned one.

Based on this grammar, we can claim now:

Lemma 5.8 $\mathcal{L}_s(CF(Countdown) \subset \mathcal{L}_s(CFHG_{stringgen}(Countdown))$ $\square$

This can be proofed using the following relations:

1. $\mathcal{L}_s(CF(Countdown) \subseteq \mathcal{L}_s(CFHG_{stringgen}(Countdown))$ (which is obvious, all that can be done by a context–free string–grammar can be done by a context–free hypergraph grammar with the same control, too) and

2. $\lambda_s(G_{wwsquare}) \in \mathcal{L}_s(CFHG_{stringgen}(Countdown)) - \mathcal{L}_s(CF(Countdown))$

Lemma 5.9 $\mathcal{L}_s(CFHG_{stringgen}) \subset \mathcal{L}_s(CFHG_{stringgen}(Countdown))$ $\square$

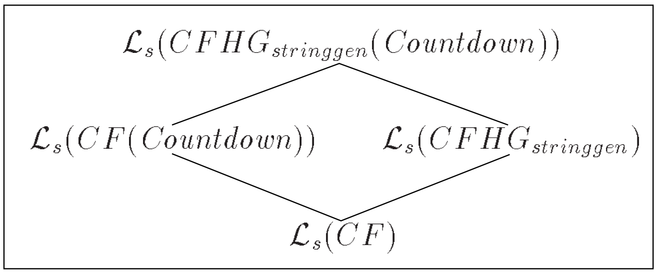
This can be proofed using the following relations:

1. $\mathcal{L}_s(CFHG_{stringgen}) \subset \mathcal{L}_s(CFHG_{stringgen}(Countdown))$ (which is obvious, all that can be done by a context–free string–grammar without control can be done by a context–free hypergraph grammar with control, too) and

2. $\lambda_s(G_{wwsquare}) \in \mathcal{L}_s(CFHG_{stringgen}(Countdown)) - \mathcal{L}_s(CFHG_{stringgen})$

This facts allow us to claim a Hasse diagramm.

Claim 5.10 *(Hasse diagramm of some string–generating concepts)*



$\square$

5.2 Tree generating context–free hypergraph grammars

This section deals with tree–generating hypergraph grammars. As introduced in Chapter 3, such hypergraph grammars produces jungles rather than trees. We focus our attention on two questions:

- How is the set of all jungles over a ranked alphabet Γ, which we denote by $jungles(\Gamma)$ here, related to the set of all output languages generatable with $CFHG(S)$–grammars with the set of terminals Γ? This leads to the task to give $CFHG(S)$–grammars creating subclasses of the set of all jungles, which is done in Section 5.2.1.

- How strong is the generation power of $CFHG(S)$–grammars? This question is the point of interest in Section 5.2.2.

Before considerating this questions, we examine the relation between trees and jungles and define the tree–to–tree translation given by TR– or TW–controlled grammars.

In Figure 5.4 a tree t together with three jungles j_1, j_2, j_3 is shown. It is easy to see, that each of these is representing the same term, actually $t = \delta(\beta(\alpha), \alpha, \gamma(\beta(\alpha), \alpha))$. The jungle j_1 looks very much like the tree t — in fact, topological they are essentially the same. No sharing is done in j_1, nevertheless it is a jungle, it can be proved with Definition 3.4. In the second jungle j_2, the subterm $\beta(\alpha)$ is shared between the γ– and the δ–hyperedge, but the subterm α appears more than once — j_2 needs more nodes to represent the tree than necessary. The jungle j_3 is the jungle for t with **maximum sharing** — the number of nodes in j_3 is minimal for the tree t.

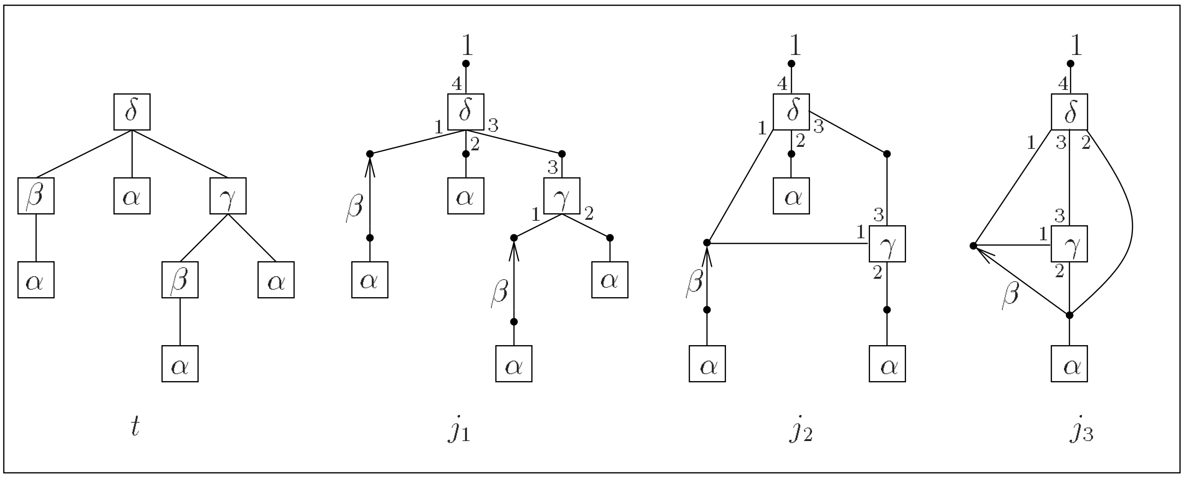


Figure 5.4: Tree and jungles representing just one term

First, we will consider translations from encodings of tree–based storage types (TR, TW) into jungles. Because of the former mentioned ambiguity of jungles, we will consider the tree–to–tree instead of the tree–to–jungle translation.

Definition 5.11 *(tree–to–tree translation of tree–controlled hypergraph grammars)*
Let $G = (N, \Delta, S_0, R, \Gamma)$ be a $CFHG(S)$–grammar, where $X \in \{Tree, Tree - Walk\}$. The tree–to–tree translation induced by G is the binary relation

$\tau_{\text{tr. to tr.}}(G) = \{(t, t') \in T_\Gamma \times T_\Delta \mid S_0(m(\Gamma)(t)) \Rightarrow^*_G \phi$ and $unfold(\phi) = t'$ for some $t \in I\}$. □

Remark:
This translation relation can also be applied to controlled grammars with composed storage types, if there is only one TR/TW contained in the storage type. The definition is analogous to the above one, but we drop the other inputs to the composed storage type, thus restricting the translation's domain to a tree–input again.

The notion of tree–to–tree translation can now be used to compare the tranlation power of this tree–controlled context–free hypergraph grammars with well–known tree translation constructs like top–down or macro tree transducer.

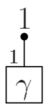
5.2.1 Generation of subclasses of $jungles(\Gamma)$

As a first example, the translation of a tree into the corresponding jungle without sharing can be done by a tree–controlled context–free hypergraph grammar. The set of productions depends on the encoding used, which is on its side the ranked alphabet used to construct the input trees. Thus, the grammar is given in form of a construction, which produces the grammar for a given ranked alphabet Γ.

Construction 5.12 *(Tree–to–jungle translation)*
Let Γ be the ranked alphabet used to construct the input trees. Consider the grammar $G_{t2j} = (N, T, q, R, \Gamma)$ with:

- $N = \{q^{(1)}\}$,
- $T = inc(\Gamma)$ and
- For every $\gamma \in \Gamma$ with $rank_\Gamma(\gamma) = 0$:

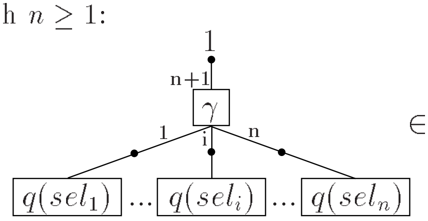
  $q \rightarrow$ `if` $root = \gamma$ `then`

  $\in R$

  For every $\gamma \in \Gamma$ with $rank_\Gamma(\gamma) = n$ with $n \geq 1$:

  $q \rightarrow$ `if` $root = \gamma$ `then`

  $\in R$

To demonstrate the use of the construction, Example 5.13 shows, how a tree–controlled hypergraph grammar can be constructed. The underlying ranked alphabet is chosen to be $\Gamma = \{\delta^{(3)}, \gamma^{(2)}, \alpha^{(0)}\}$.

Example 5.13 *(Tree–to–jungle transducer)*
Consider the ranked alphabet $\Gamma = \{\delta^{(3)}, \gamma^{(2)}, \alpha^{(0)}\}$. Following the Construction 5.12, the grammar $G_{t2jex} = (\{q^{(1)}\}, \{\delta^{(4)}, \gamma^{(3)}, \alpha^{(1)}\}, q, R, \Gamma)$ with R as in Figure 5.5. A example translation of the term $\delta(\alpha, \gamma(\alpha, \alpha), \alpha)$ is shown in Figure 5.6.

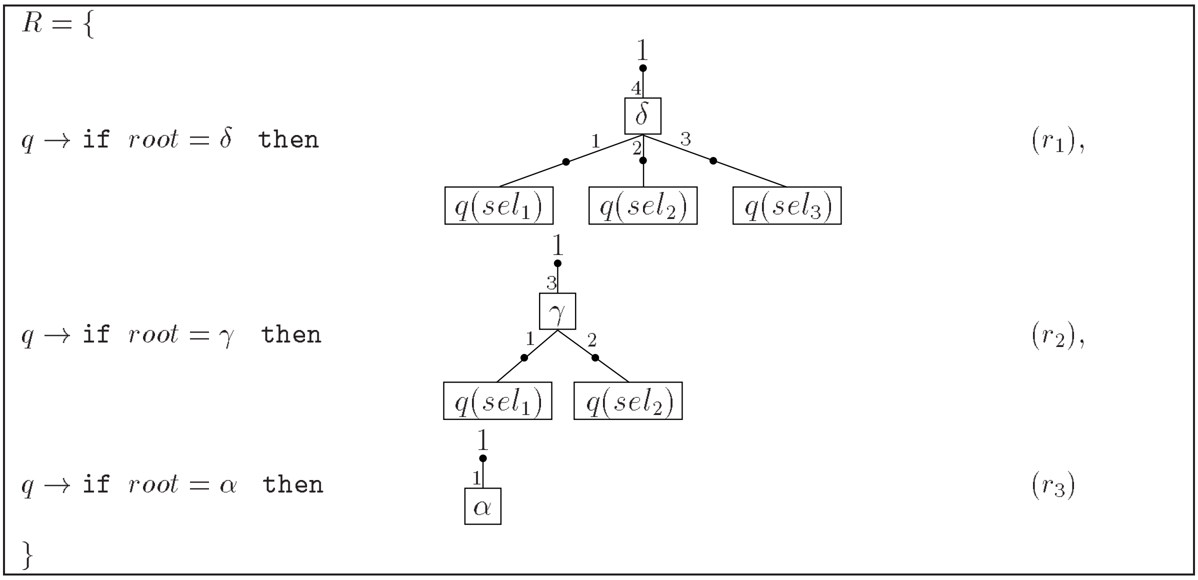


Figure 5.5: The productions of the context–free hypergraph grammar G_{t2jex}

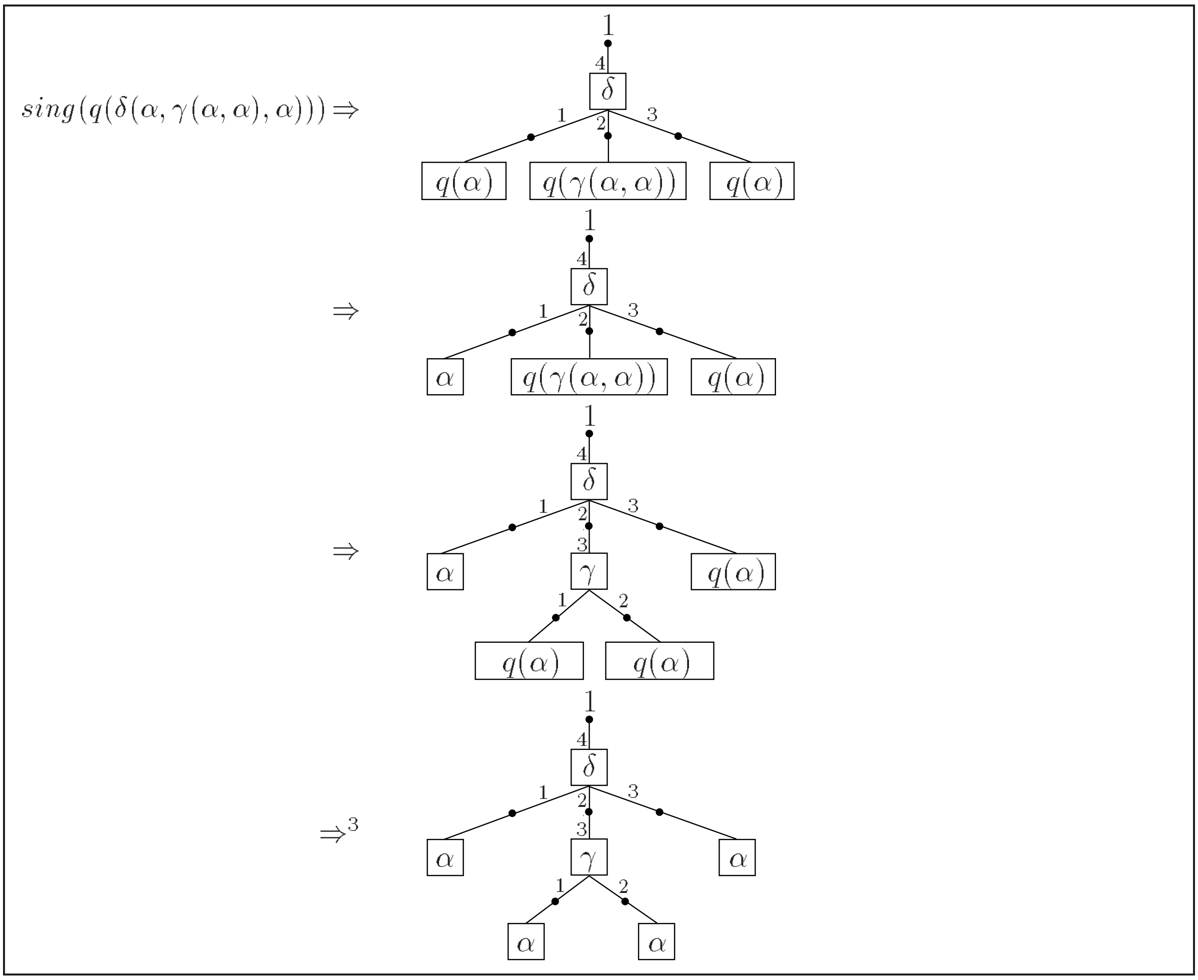


Figure 5.6: A derivation of the context–free hypergraph grammar G_{t2jex}

Considering the example grammar G_{t2jex}, it should be clear, that the translation is just $\tau_{\text{tr. to tr.}}(G_{t2jex}) = id(T_\Gamma)$.

There is just another controlled grammar for translating a tree into a corresponding jungle in Example 5.14, which generates jungles with so–called **front–sharing**, i.e., all the symbols on the front of the tree appear exactly once in the jungle. It works with a more complicated storage type (in fact a product of a tree and two pushdowns), and is therefore a nice example for using pushdown, especially as temporary memory. The front–sharing can be realized in an easier way (using only TR–control), but this leads to garbage in the output, i.e., symbols of rank 1 which are not connected to the only external node of the jungle.

Example 5.14 *(Tree–to–jungle transducer, generating front–shared jungles)*
Consider the grammar S–controlled $G_{fs} = (N, T, D_1, R, en)$, where

- $T = \{\gamma^{(3)}\} \cup \underbrace{\{\alpha_1^{(1)}, ..., \alpha_k^{(1)}\}}_{T_1}$,
- $N = \{D_1^{(1)}, ..., D_{k+1}^{(k+1)}, U_1^{(1)}, ..., U_{k+1}^{(k+1)}, L_1^{(1)}, ..., L_{k+1}^{(k+1)}, L_1^{'(1)}, ..., L_{k+1}^{'(k+1)}, S_1^{(1)}, ..., S_{k+1}^{(k+1)}\}$,
- $R = \bigcup_{i=1}^{k+1}$ with R_i as in Figure 5.7.
- $en = (dec(T), en, en)$.

and $S = TW \times (\mathbf{P}(S_0))^2$. In the rules of the grammar the conditions are 3-tupels of predicates. As a convention, we combine two rules having the condition (p_1, p_2, p_3) and $(p_1, p_2, \texttt{ not } p_3)$ and the same right–hand side ξ to one rule having the condition $(p_1, p_2, \texttt{true})$, what makes sense because of the and–relation used in the multiplication of storage types in Definition 4.8.

Remarks:

1. For a better readability, the tentacle numbers on the right–hand sides are dropped when they are equal to the number of the attached external node.
2. In Rule r_8, we have used a semicolon to unite two instruction symbols together, see the remarks to Definition 4.9 for this convention.

To explain the techniques used here, consider a derivation starting with the tree

$$t = \gamma(\gamma(\alpha_1, \gamma(\alpha_0, \alpha_1)), \alpha_0)$$

and with empty initialized pushdowns. The starting symbol is D_1, thus we can derive the following way, using two simplifications for the representations: we will only write the path into the non–terminals, dropping the tree and we will write the hypergraph as a tree, as long as possible:

$$\begin{array}{rcll}
(D_1(\varepsilon,\varepsilon,\varepsilon))() & \Rightarrow & (D_1(2,\varepsilon,\varepsilon))() & (r_1) \\
 & \Rightarrow & (L_1(2,\varepsilon,\varepsilon))() & (r_2) \\
 & \Rightarrow & (L_2'(2,\alpha_0,\varepsilon))(\alpha_0) & (r_5) \\
 & \Rightarrow & (U_2(2,\alpha_0,\varepsilon))(\alpha_0) & (r_7) \\
 & \Rightarrow & (D_2(1,\alpha_0,\varepsilon))(\alpha_0) & (r_8) \\
 & \Rightarrow & (D_2(12,\alpha_0,\varepsilon))(\alpha_0) & (r_1)
\end{array}$$

$$
\begin{array}{lll}
\Rightarrow & (D_2(122, \alpha_0, \varepsilon))(\alpha_0) & (r_1) \\
\Rightarrow & (L_2(122, \alpha_0, \varepsilon))(\alpha_0) & (r_2) \\
\Rightarrow & (L_2(122, \varepsilon, \alpha_0))(\alpha_0) & (r_4) \\
\Rightarrow & (L'_3(122, \alpha_1, \alpha_0))(\alpha_0, \alpha_1) & (r_5) \\
\Rightarrow & (L'_3(122, \alpha_0\alpha_1, \varepsilon))(\alpha_0, \alpha_1) & (r_6) \\
\Rightarrow & (U_3(122, \alpha_0\alpha_1, \varepsilon))(\alpha_0, \alpha_1) & (r_7) \\
\Rightarrow & (D_3(121, \alpha_0\alpha_1, \varepsilon))(\alpha_0, \alpha_1) & (r_8) \\
\Rightarrow & (L_3(121, \alpha_0\alpha_1, \varepsilon))(\alpha_0, \alpha_1) & (r_2) \\
\Rightarrow & (L'_3(121, \alpha_0\alpha_1, \varepsilon))(\alpha_0, \alpha_1) & (r_3) \\
\Rightarrow & (U_3(121, \alpha_0\alpha_1, \varepsilon))(\alpha_0, \alpha_1) & (r_7) \\
\Rightarrow & (U_3(12, \alpha_0\alpha_1, \varepsilon))(\alpha_0, \alpha_1) & (r_9) \\
\Rightarrow & (D_3(11, \alpha_0\alpha_1, \varepsilon))(\alpha_0, \alpha_1) & (r_8) \\
\Rightarrow & (L_3(11, \alpha_0\alpha_1, \varepsilon))(\alpha_0, \alpha_1) & (r_2) \\
\Rightarrow & (L_3(11, \alpha_1, \alpha_0))(\alpha_0, \alpha_1) & (r_4) \\
\Rightarrow & (L'_3(11, \alpha_1, \alpha_0))(\alpha_0, \alpha_1) & (r_3) \\
\Rightarrow & (L'_3(11, \alpha_0\alpha_1, \varepsilon))(\alpha_0, \alpha_1) & (r_6) \\
\Rightarrow & (U_3(11, \alpha_0\alpha_1, \varepsilon))(\alpha_0, \alpha_1) & (r_7) \\
\Rightarrow & (U_3(1, \alpha_0\alpha_1, \varepsilon))(\alpha_0, \alpha_1) & (r_9) \\
\Rightarrow & (U_3(\varepsilon, \alpha_0\alpha_1, \varepsilon))(\alpha_0, \alpha_1) & (r_9) \\
\Rightarrow & (S_3(\varepsilon, \alpha_0\alpha_1, \varepsilon))(\alpha_0, \alpha_1) & (r_{10})
\end{array}
$$

It is easy to see, that the derivation until here is an right–to–left depth–first searching for symbols from T_1. The symbols D and U (standing for "down" and "up", respectively) are used for the searching, while L and L' are used for a lookup mechanism on the pushdowns. The index on the non–terminals is used to save the number of associated symbols from T_1 (plus 1 for the root). If one symbol α_j is found, the searching stops and starts the lookup. The symbol L pops the elements from the first stack and pushs it onto the second. If the top of the first stack shows the searched symbol α'_j, the lookup state L' is used to reorganize the stacks (see rule r_3). If L doesn't found α_j on the first pushdown, it generates it, increases its index (from i to $i+1$) and calls L' to reorganize the stacks, too (see rule r_5). Then the search continues. If the TW–control gets to the root again, rule r_{10} starts the process of generating the jungle. S is called and has on its first pushdown a list of symbols from T_1 which are in the same order as the terminals associated with S_i on its tentacles $1, ..., i-1$. S passes top–down through the tree, symbols not from T_1 are simply rewritten in rule $r_1 1$, and the symbols from T_1 are not created, instead the attached ones are used.

As already mentioned, the translation relation of G_{fs} is the identity, i.e., $\tau_{\text{tr. to tr.}}(G_{fs}) = id(T)$. The example can only demonstrate the principles of jungle generation and the usage of controls.

$R_i = \{$

$D_i \to$ `if` $(label = \gamma, \texttt{true}, \texttt{true})$ `then` (r_1)

$D_i \to$ `if` $(label = \alpha_j, \texttt{true}, \texttt{true})$ `then` (r_2)
(for every $\alpha_j \in T_1$)

$L_i \to$ `if` $(label = \alpha_j, top = \alpha_j, \texttt{true})$ `then` (r_3)
(for every $\alpha_j \in T_1$)

$L_i \to$ `if` $(label = \alpha_j, top = \alpha_k, \texttt{true})$ `then` (r_4)
(for every $\alpha_j, \alpha_k \in T_1$ with $j \neq k$)

$L_i \to$ `if` $(label = \alpha_j, bottom, \texttt{true})$ `then` (r_5)
(for every $\alpha_j \in T_1$)

$L'_i \to$ `if` $(\texttt{true}, \texttt{true}, top = \alpha_j)$ `then` (r_6)
(for every $\alpha_j \in T_1$)

$L'_i \to$ `if` $(\texttt{true}, \texttt{true}, bottom)$ `then` (r_7)
(for every $\alpha_j \in T_1$)

$U_i \to$ `if` $(son = j, \texttt{true}, \texttt{true})$ `then` (r_8)
(for every $j \in \{2, ..., \max_{\gamma \in T} rank(\gamma) - 1\}$)

$U_i \to$ `if` $(son = 1, \texttt{true}, \texttt{true})$ `then` (r_9)

$U_i \to$ `if` $(root, \texttt{true}, \texttt{true})$ `then` (r_{10})

$S_i \to$ `if` $(label = \gamma, \texttt{true}, \texttt{true})$ `then` (r_{11})

$S_i \to$ `if` $(label = \alpha_j, top = \alpha_k, \texttt{true})$ `then` (r_{12})
(for every $\alpha_j, \alpha_k \in T_1$ with $j \neq k$)

$S_i \to$ `if` $(label = \alpha_j, top = \alpha_j, \texttt{true})$ `then` (r_{13})
(for every $\alpha_j \in T_1$)

$\}$

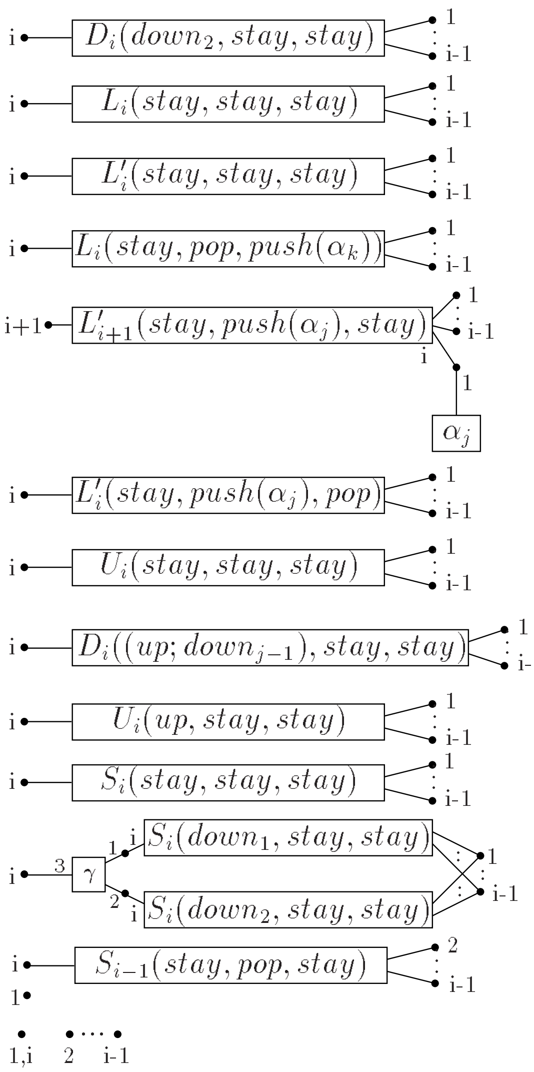


Figure 5.7: A subset of the productions of the context–free hypergraph grammar G_{fs}

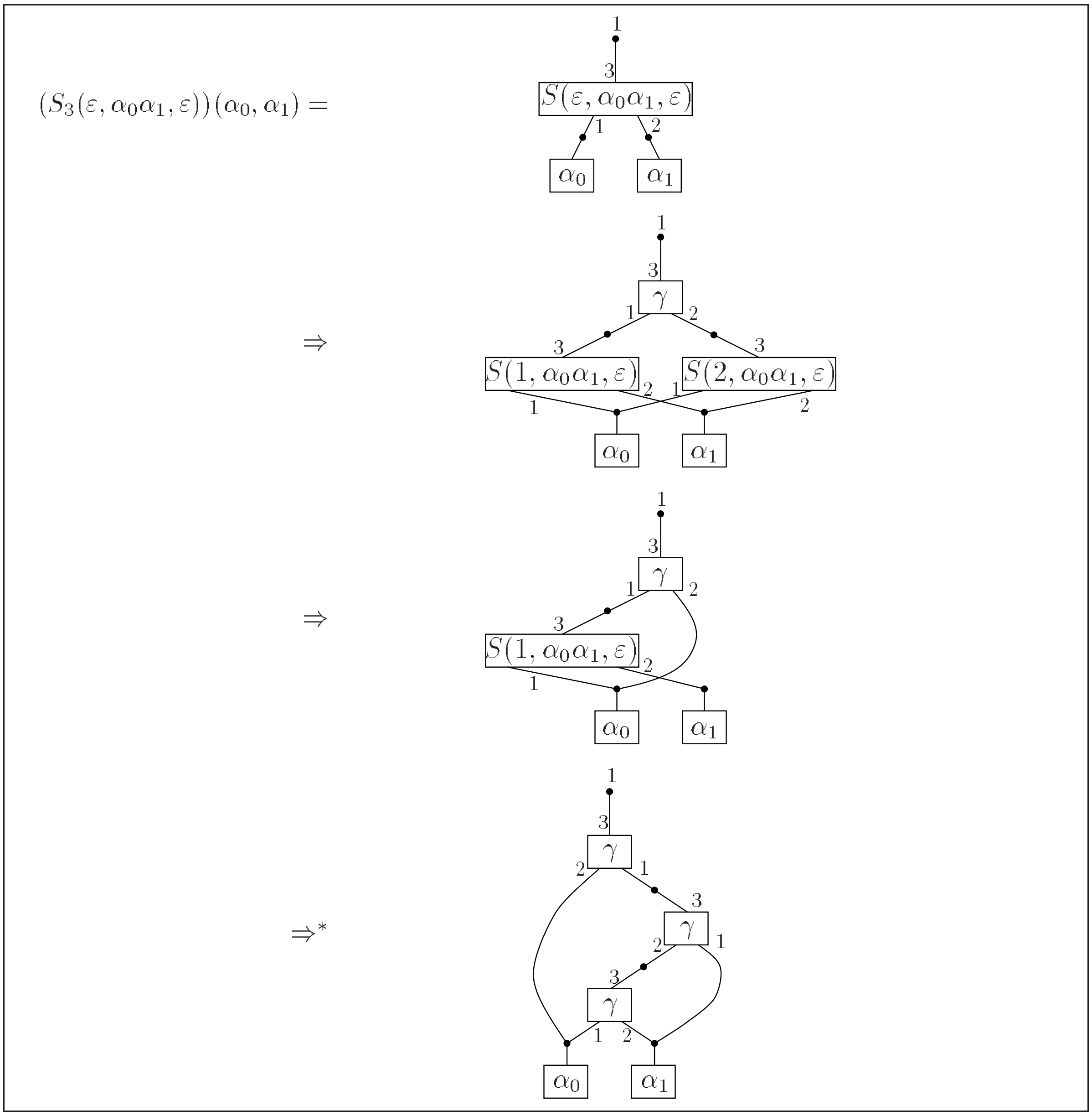


Figure 5.8: A derivation of the context–free hypergraph grammar G_{fs}

Maybe the best example for computing the identity of trees would be a tree–controlled hypergraph grammar generating the jungle with maximum sharing from the input tree, but it look's like this would need more generation power than available in this concept. Such a grammar would be a complete formal description for the common subexpression elimination, which is a very important algorithm in compiler construction.

Nevertheless it is possible to give examples which generate jungles with optimum sharing. The following Example 5.15 is a TR–controlled grammar generating such jungles.

Example 5.15 *(unfolding the flank of a tree into a optimum shared jungle)*
Consider the grammar $G_{symm}(TR) = (N, T, S, R, \Gamma)$ with

- $N = \{S^{(1)}\}$,
- $T = \{\delta^{(4)}, \gamma^{(3)}, \alpha^{(1)}\}$,
- R as in Figure 5.9 and
- $\Gamma = dec(T)$

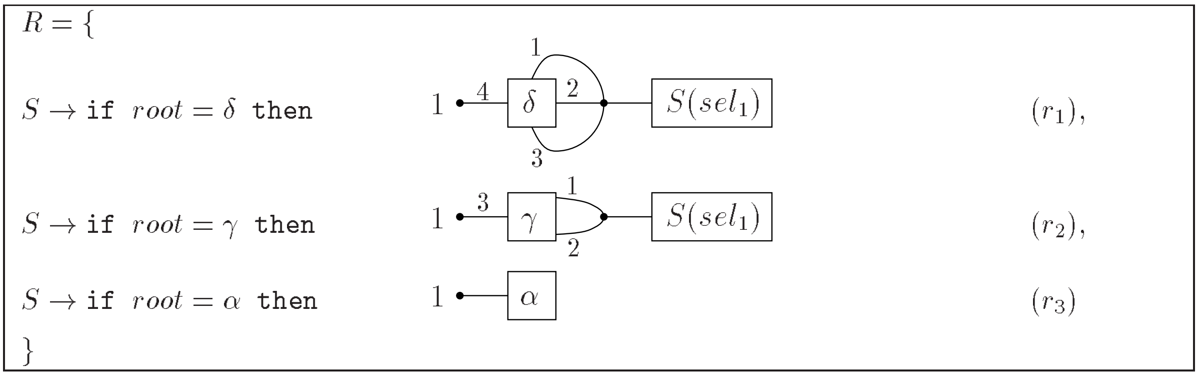


Figure 5.9: The productions of the context–free hypergraph grammar G_{symm}

A derivation of this grammar is shown in Figure 5.10.

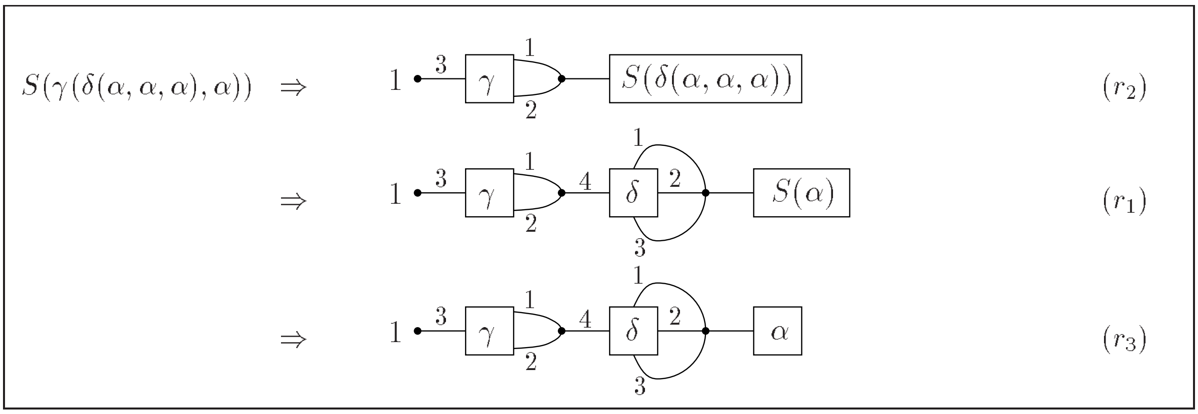


Figure 5.10: A derivation of the context–free hypergraph grammar G_{symm}

If we define a function $symm : T_\Gamma \to T_\Gamma$ inductively with

- For every $\sigma \in \Gamma_n$ with $n \geq 1$: $symm(\sigma(x_1, ..., x_n)) = \sigma(symm(x_1), ..., symm(x_1))$ and
- for every $\alpha \in \Gamma_0$: $symm(\alpha) = \alpha$,

we can express the tree–to–tree translation as $\tau_{\text{tr. to tr.}}(G_{symm}) = \{(t, symm(t)) \mid t \in T_\Gamma\}$. This can be done by a top–down tree transducer, too. But considerating the jungles, which are generated by G_{symm}, one can see, that they are all optimum shared — there is no subexpression appearing twice (because if there would be two such subexpressions, they will be of course of equal height, but the definition of the grammar shows, that there cannot be two subexpression with this property).

5.2.2 Generative power

As already mentioned, the relation between $CFT(S)$ and $CFHG(S)$ seems to be an open question. An example demonstrating that the last concept has the higher generative power hasn't be found yet. The best example found with a high generative power is the following one.

Example 5.16 *(example for high generative power)*
Consider the $CFHG(TR)$–grammar $G_{pow2} = (N, T, S_0, R, en)$ with:

- $N = \{Q_0^{(1)}, Q^{(2)}\}$,
- $T = \{\delta^{(3)}, alpha^{(1)}\}$,
- $en = \{\gamma^{(1)}, \alpha^{(0)}\}$,
- R as in Figure 5.11.

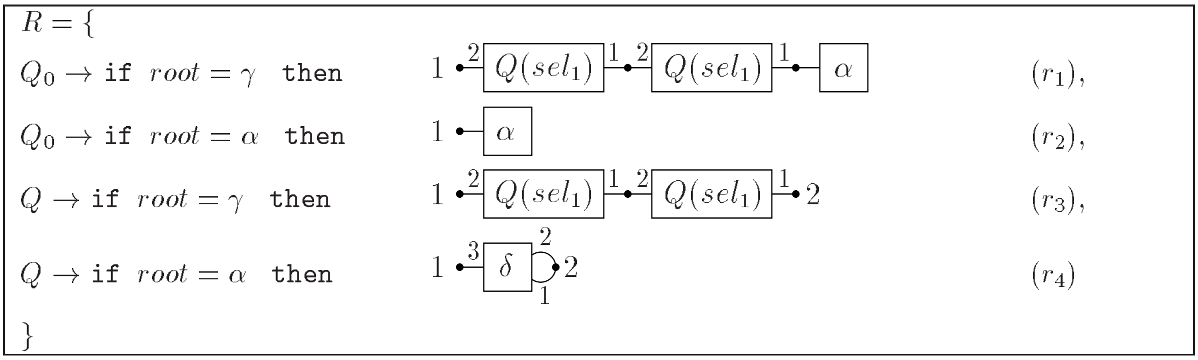


Figure 5.11: The productions of the context–free hypergraph grammar G_{pow2}

The example derivation for the input $i = \gamma(\gamma(\alpha))$, which is drawn in Figure 5.12, shows that the translation relation of G_{pow2} is $\tau_{\text{tr. to tr.}}(G_{pow2})(\gamma^n \alpha) = fbt_{\delta,\alpha}(2^n)$.

These tree–controlled hypergraph grammars are in fact the top–down tree–to–graph transducers (td–tg transducers, for short) introduced in [EV94]. The paper focuses on so–called tree–generating td–tg transducer. The class of translation of this concept is shown to be equal to the class of tranlations of macro tree transducers. Thus, although the td–tg transducer is a

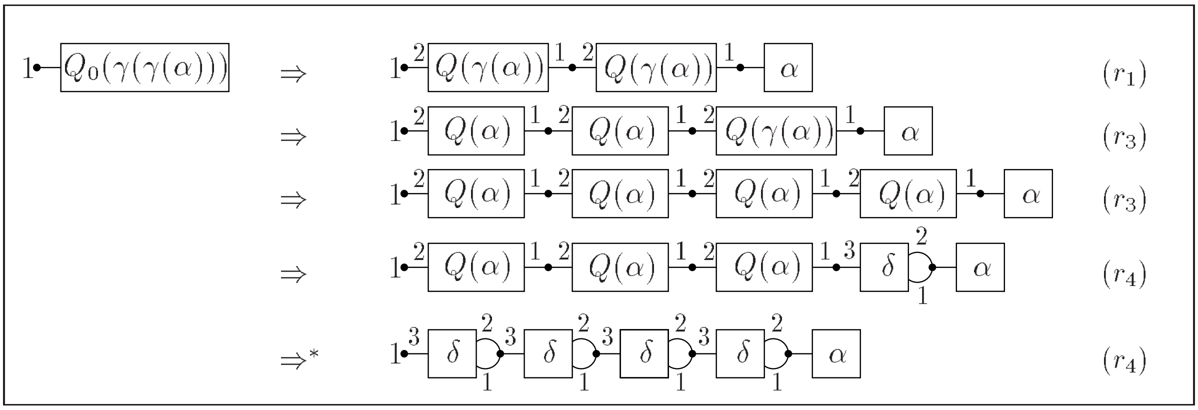


Figure 5.12: A derivation of the context–free hypergraph grammar G_{pow2}

top–down device, it can can contruct a language, which is not constructable by top–down tree transducers.

Several other functions have been tested for definability with $CFHG(S)$–grammars, among them

- A string–to–tree transducer translating the bracket–free prefix–notation of a term into the coressponding tree. Several subcases can be defined, but the general purpose seems to need at least a very complex storage type.
- The double–exponential function, i.e., the translation of trees of height n into such with height 2^{2^n}. This is of course possible with complex storage types, but interesting is the question, whether it is possible with $CFHG(Countdown)$, what seems to be unlikely.
- Other functions which can be calculated by $CFHG(S)$–grammars can be recognized as definable by easier means to, e.g., the translation of trees of height n into such of height n^2.

Nevertheless, even if the translation–/generative power of $CFHG_{treegen}(S)$–grammars does not exceed the power of well–known concepts, the concept can still be interesting. The representation of trees by optimum shared jungles is, as already mentioned, a memory efficient way, thus if there would be a transducer concept which can translate jungles into jungles (without taking the input jungle as a tree), it can be used for large scale symbolic calculation, e.g., for the differentiation of terms. Such a transducer could take care of rules for differentiating to avoid the re–appereance of subterms (the rule for differentiating a fraction $\frac{u}{v}$ contains v and v^2 — where one v could instantly be eliminated by sharing it). Controlled hypergraph grammars would be such transducer, if there can be found an efficient way, to store jungles in storage types in a form which takes care of the sharing.

5.3 Graph generating context–free hypergraph grammars

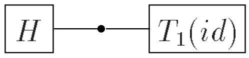
We will only present one suitable example for a graph generating context–free hypergraph grammar consider the grammar $G_{CH} = (N, T, S, R, en)$ with $N = \{S^{(0)}, T_1^{(1)}, T_2^{(2)}, T_3^{(3)}\}$, $T = \{H^{(1)}, C^{(4)}\}$, $en = id_C$ and R is shown in Figure 5.13. The tentacle numbers are dropped from the tentacles, because in G_{CH}, the hyperedges are thought to be undirected. G_{CH} is controlled by the one–way storage type $One\text{–}way_{id}$ with $en = \{1, 2, 3\}$ and with identity.

The set of productions in not complete for all total configurations. Thus, this grammar can be interpreted as an acceptor for a special language, which is in fact $\alpha(G_{CH}) = \{c_0c_1...c_n \mid n \geq 0, \forall i \in \{0,...,n\} : c_n \in \{1,2,3\}, \forall i \in [n] : val(c_{i-1}) + val(c_i) \leq 4\}$, where val denotes the numerical value of the character ($val(1) = 1, ...$). The accepted language may look strange, but looking at Figure 5.14 shows what G_{CH} generates when initialized with the input 2113. The grammar generates a hypergraph, which can be interpreted as the picture of pent-4-en-1-ine. Two tentacles connecting two hyperedges via a node can be interpreted as a bound.

The idea of using graph grammars to model molecular structures appeared in [Fed71], where another grammar concept — the so–called Plex grammars was used.

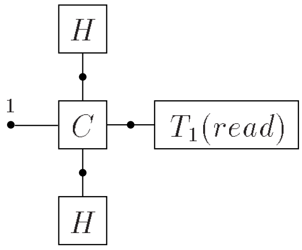
$R = \{$

$S \to$



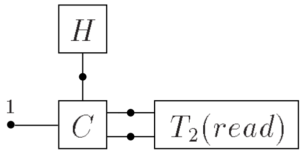
(r_1),

$T_1 \to$ `if` $first = 1$ `then`



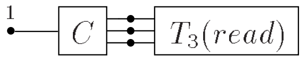
(r_2),

$T_1 \to$ `if` $first = 2$ `then`



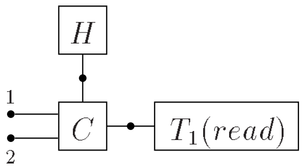
(r_3),

$T_1 \to$ `if` $first = 3$ `then`



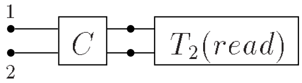
(r_4),

$T_2 \to$ `if` $first = 1$ `then`



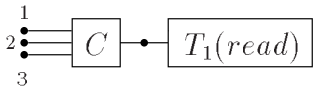
(r_5),

$T_2 \to$ `if` $first = 2$ `then`



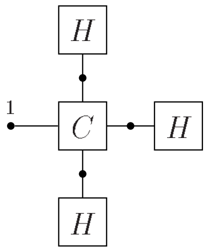
(r_6),

$T_3 \to$ `if` $first = 1$ `then`



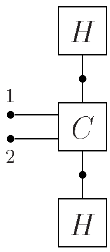
(r_7),

$T_1 \to$ `if` $empty$ `then`



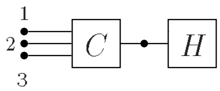
(r_8),

$T_2 \to$ `if` $empty$ `then`



(r_9),

$T_3 \to$ `if` $empty$ `then`



(r_{10})

$\}$

Figure 5.13: The productions of the context–free hypergraph grammar G_{CH}

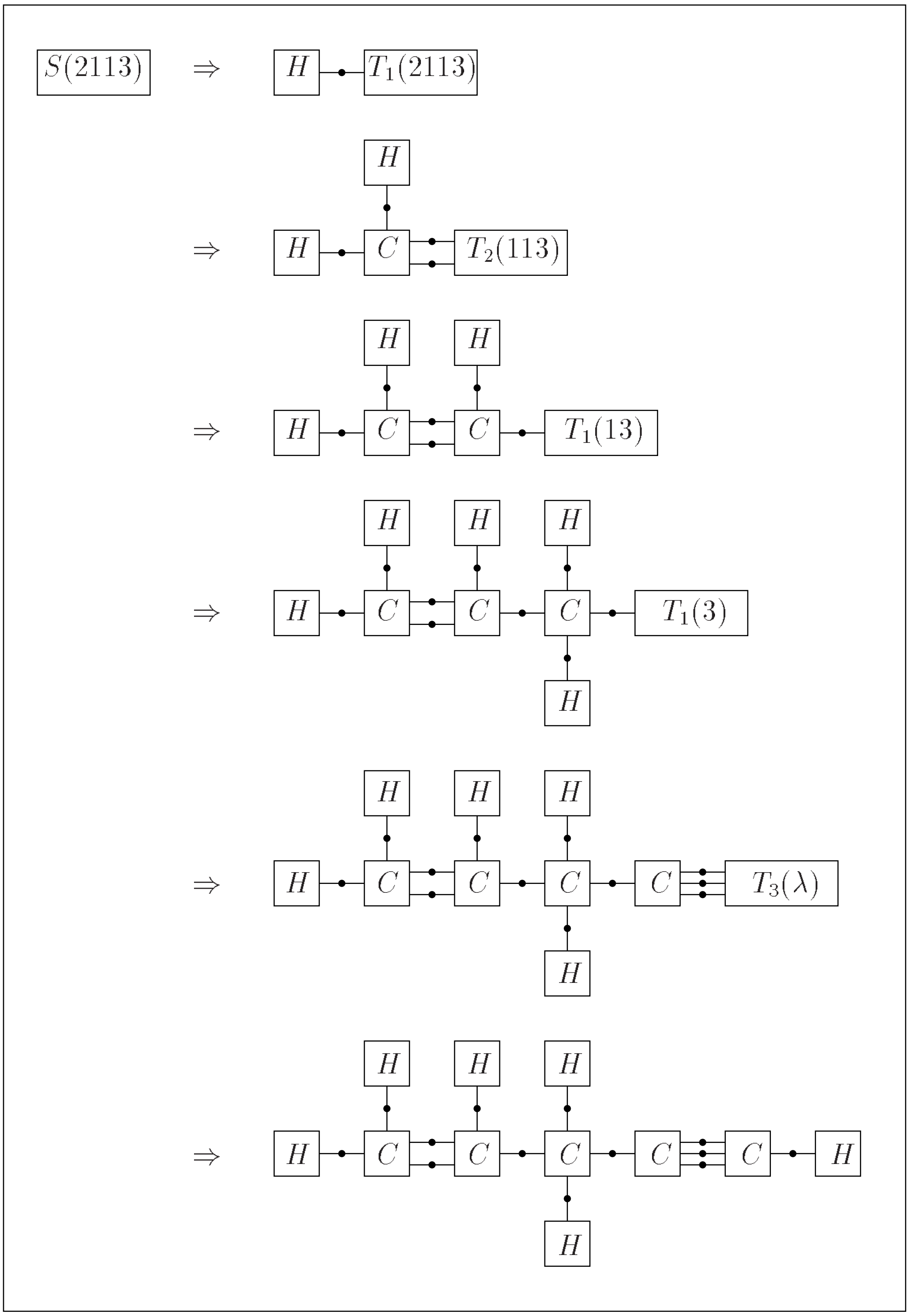


Figure 5.14: An example derivation of the context–free hypergraph grammar G_{CH}

Chapter 6

Conclusions

The work in this paper can be continued in several directions. One of them is to examine the exact translation and generation power of tree–controlled tree–generating context–free hypergraph grammars. The translation power is probably not higher than that of macro transducers.

On the other hand, it is also worth to try to find a tree– or treewalk– controlled context–free hypergraph grammar which generates the largest possible subclass of all jungles over one alphabet. It seems to be unlikely, that the concepts introduced here, have enough generation power for the task to generate all jungles. But the introduced grammar which generates the class of all front–shared jungles should not be expected to be the best possible solution to this problem.

As mentioned in the last chapter, there can be thought of a lot of practical applications for hypergraph grammars in general and especially for tree–generating hypergraph grammars, too. In connection with symbolic computation problems, the question, whether it is possible to do symbolic computation on jungles without unfolding them, gives rise to the question, how to implement a jungle storage type which reflects the sharing to the grammar.

The Hasse diagramm given in Chapter 5.1 must be proved or disproved. Furthermore, equivalent diagramms could probably found using other storage types like pushdowns — it could be an interesting question, how the inclusion relations change with the storage types.

Appendix A

Table of mathematical symbols

	Sets
$\emptyset$	empty set
$\mathbb{N}$	set of the natural numbers $\{0, 1, ...\}$
$\mathbb{N}^+$	$\mathbb{N} - \{0\}$
$[n]$ with $n \geq 0$	set $\{1, ..., n\}$ of positive integers, $[0] = \emptyset$
	Strings
ε	empty string
A^*	set of all words which can be built from alphabet A
A^+	$A^* - \{\varepsilon\}$
$ar(w)$	arguments in w; $ar(w_1 w_2 ... w_k) = \{w_1, ..., w_k - 1\}$
$res(w)$	result in w; $res(w_1 w_2 ... w_k) = w_k$
	Ranked alphabets and trees
$rank_\Gamma(s)$	rank of the symbol $s \in \Gamma$
Γ_n	all elements of the ranked alphabet Γ with rank n
$dec(\Gamma)$	$\{gamma^{n-1} \mid \gamma \in \Gamma_n$ and $n \geq 1$
$inc(\Gamma)$	$\{gamma^{n+1} \mid \gamma \in \Gamma_n\}$
T_Γ	set of all trees (terms) over the ranked alphabet Γ
$occ(t)$	set of all valid pathes in $t \in T_\Gamma$
$label_t(\pi)$	label of the node on the path π in the tree t
$fbt_{\delta,\alpha}(n)$	full binary tree over $\{\delta^{(2)}, \alpha^{(0)}\}$ of height n
	Binary relations
$\equiv_R$	smallest equivalence relation, which contains R
V/R	set of equivalence classes in V induced by R
$ran(R)$	range of the relation R
$dom(R)$	domain of the relation R
	Hypergraphs
$HGR\,(\Gamma)$	set of all hypergraphs over Γ
n–$HGR\,(\Gamma)$	set of all hypergraphs over Γ with rank less or equal n
$sing(\delta)$	singular hypergraph labeled by δ of rank $rank(\delta)$
$w^\bullet$	string hypergraph induced by the string w
$jungles(\Gamma)$	set of all jungles over Γ

Others	
iff	if and only if
$id(S)$	identity operation on S
$BE(p)$	BOOLEan expression over p
$L_X(S)$	$L_{CFT}(S) = L_{RT}(S) = T_S$, $L_{CFHG}(S) = HGR\,(S)$ etc.
$\mathcal{L}(S)$	class of languages generatable by the concept S

Bibliography

[Aho68] A.V. Aho. Indexed grammars, an extension of context–free grammars. *JACM*, 15:647–671, 1968.

[Bar84] H. P. Barendregt. *The Lambda Calculus, its Syntax ans Semantics*. North–Holland, Amsterdam, 1984.

[BC87] M. Bauderon and B. Courcelle. Graph expressions and graph rewritings. *Math. Systems Theory*, 20:83–127, 1987.

[Bra69] W. S. Brainerd. Tree generating regular systems. *Inf. and Control*, 14:217–231, 1969.

[Cho56] N. Chomsky. Three models for the description of language. *IRE Trans. on Information Theory*, 2,3:113–124, 1956.

[DKH96] F. Drewes, H.-J. Kreowski, and A. Habel. Hyperedge replacement graph grammars. to appear in the book on graph grammars edited by G. Rozenberg, 1996.

[Dre96] F. Drewes. *Computation by tree transductions*. PhD thesis, University of Bremen, 1996.

[Eng86] J. Engelfriet. Context–free grammars with storage. Technical report, University of Leiden, 1986.

[Eng94] J. Engelfriet. Graph grammars and tree transducers. In S. Tison, editor, *Proceedings of the 19th Colloquium on Trees in algebra and programming - CAAP 94*, volume 787 of *LNCS*, pages 15–36. Springer-Verlag, 1994.

[Eng96] J. Engelfriet. Context–free graph grammars. In G. Rozenberg and A. Salomaa, editors, *Handbook of Formal Languages, Volume III*. Springer-Verlag, 1996. (to appear).

[ER90] J. Engelfriet and G. Rozenberg. A comparison of boundary graph grammars and context-free hypergraph grammars. *Inform. and Computation*, 84:163–206, 1990.

[ER96] J. Engelfriet and G. Rozenberg. Node replacement graph grammars. In G. Rozenberg, editor, *Handbook of Graph Transformations, Volume I: Foundations*. World Scientific, 1996. (to appear).

[ERS80] J. Engelfriet, G. Rozenberg, and G. Slutzki. Tree transducers, L systems, and two-way machines. *J. Comput. System Sci.*, 20:150–202, 1980.

[EV86] J. Engelfriet and H. Vogler. Pushdown machines for the macro tree transducer. *Theoret. Comput. Sci.*, 42:251–368, 1986.

[EV94] J. Engelfriet and H. Vogler. The translation power of top–down tree–to–graph transducers. *Journal of Computer and System Sciences*, 49:258–305, 1994.

[Fed71] J. Feder. Plex languages. *Inform. Sci.*, 3:225–241, 1971.

[Fis68] M.J. Fischer. *Grammars with macro-like productions*. PhD thesis, Harvard University, Massachusetts, 1968.

[Hab92] A. Habel. *Hyperedge-replacement: Grammars and Languages*, volume 643 of *LNCS*. Springer-Verlag, 1992.

[HKP91] A. Habel, H.-J. Kreowski, and D. Plump. Jungle evaluation. *Fundamenta Informaticae*, 15:37–60, 1991.

[Knu68] D.E. Knuth. Semantics of context–free grammars. *Math. Syst. Theory*, 2:127–145, 1968.

[Knu71] D.E. Knuth. Correction to [Knu68]. *Math. Syst. Theory*, 5:95–96, 1971.

[Lin68] A. Lindenmayer. Mathematical models for cellular interaction in development. *Journal of Theoretical Biology*, 18:280–315, 1968.

[Pet62] C. A. Petri. *Kommunikation mit Automaten*. PhD thesis, Universität Bonn, 1962.

[Rou70] W.C. Rounds. Mappings and grammars on trees. *Math. Systems Theory*, 4:257–287, 1970.

[Tha70] J.W. Thatcher. Generalized^2 sequential machine maps. *J. Comput. Syst. Sci.*, 4:339–367, 1970.