

Representing Multidimensional Trees

Earlham Theory Group

September 14, 2004

1 Introduction

Traditionally, the context-free grammars are represented as a set of string rewriting rules: a grammar is a four-tuple $\mathcal{G} = \langle \Sigma, V, S, P \rangle$, where Σ is the terminal alphabet, V is a finite set of non-terminal symbols, $S \in V$ is the initial symbol, and P is a finite set of productions (each of which map some symbol, $x \in V$, to a string of symbols, $y \in (\Sigma \cup V)^*$) [Sip01].

One can modify the above description of \mathcal{G} to represent the context-free grammars using local trees [GS84]. Let P be a set of local trees (a set of trees with height ≤ 1), each having a yield in $(\Sigma \cup V)^*$ and a root in V . Let each member of Σ be the root of a trivial local tree, and let the initial symbol S be the root a trivial derivation tree T_0 . The context-free derivation of any T_{i+1} can then be performed by concatenating local trees licensed by P to the derivation tree T —by replacing a leaf node of T_i , labeled by some x , with a local tree in P with root labeled x . The derived string, at any point in the process, is the yield of T . A string is in $L(\mathcal{G})$ if and only if it is in Σ^* and is the yield of some derivation tree. We can see that, in this representation, the local trees in P are grammatically equivalent to the productions in the traditional CFGs—

they map some root symbol, $x \in V$, to some string yield, $y \in (\Sigma \cup V)^*$. Figure 1 shows an example grammar in local tree form, plus an example derivation tree constructed from rules in this grammar. The example derivation tree yields the string $((\))(\))(\))$.

Tree adjoining grammars (TAG) [JS96] lift the context-free grammars from a string-generating formalism to a tree-generating formalism. Just as context-free grammars can be represented as a set of string rewriting rules, tree adjoining grammars can be represented as a set of tree rewriting rules. Just as CFGs derive strings by repeated symbol replacement, TAGs derive trees by adjunction.

Conceptually, tree adjunction in TAG is the substitution of one tree into another at a specific non-leaf node. Specifically, to adjoin some auxilliary tree β to some derivation tree Γ_i at node x , we split Γ_i into two trees— γ_u (the tree which does not contain x or the subtree below it), and γ_l (the subtree rooted by x). We create Γ_{i+1} with two replacements: we replace $\gamma_l \in \Gamma_i$ with β , and $x \in \beta$ with γ_l . This procedure requires that the auxilliary tree β have both a root labeled by x and a distinguished leaf node labeled by x (the "foot"). Figures 2 and 3 provide a visual representation of this pro-

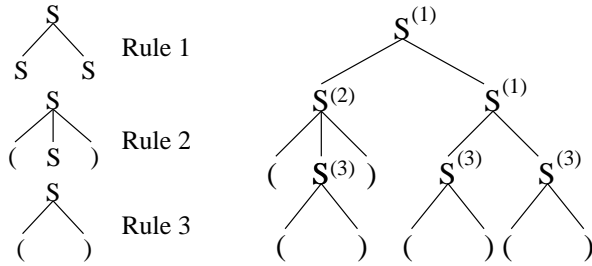


Figure 1: An example context-free grammar in local tree form, plus one possible derivation tree with root labeled by S .

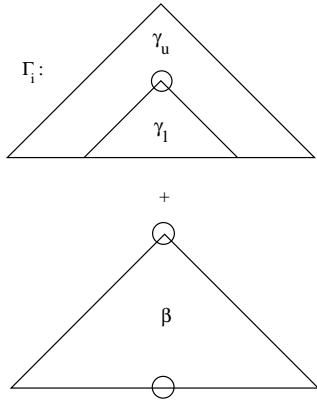


Figure 2: Adjunction of tree β into Γ_i .

cess.

The ability to replace non-frontier nodes with licensed trees differentiates TAGs from our local tree representation of CFGs, and affords TAGs a degree of context-sensitive generative power [Wei88].

In the case of CFGs, we were able to represent string rewriting using concatenation of local trees. That is, we were able to reduce substitution on a structure to concatenation of a structure, purely by raising the dimensionality of the structure. For the CFGs, we moved from substitution on one-dimensional structures (strings) to concatenation of two-dimensional local structures while maintaining equivalent genera-

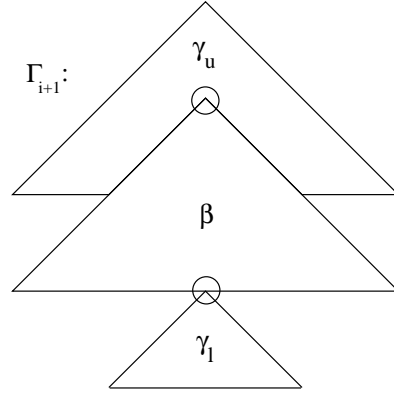


Figure 3: Γ_{i+1} , after adjunction with β .

$\Sigma: \{ a, b \}$
 $V: \{ S, S' \}$
 $S: s$

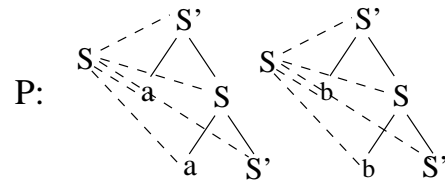


Figure 4: An example tree-adjoining grammar in three dimensions, plus one possible derivation tree.

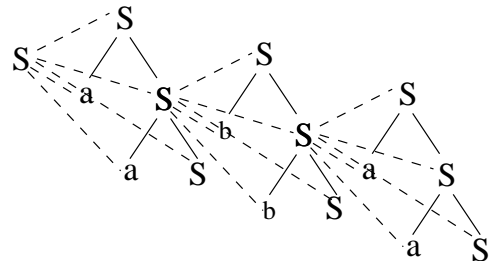


Figure 5: One possible three-dimensional derivation tree.

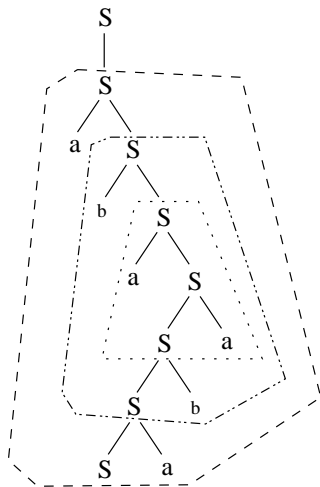


Figure 6: The two-dimensional yield of a three-dimensional derivation tree.

tive power. In a similar fashion, we can also achieve generative power equivalent to that of TAG using only concatenation—by moving from adjunction and substitution on two-dimensional structures to concatenation of three-dimensional local structures [Wei92].

In this paper, we explore a framework for the representation and construction of multi-dimensional trees. These multi-dimensional trees, when used in the context of a tree-licensing grammar, are capable of providing generative power equal to or greater than that of a TAG.

2 Tree Construction

2.1 Tree-building Operations

The classic representation of 2-dimensional trees, a parent with a set of children, is difficult to generalize into a form for a tree in higher dimensions. Since a 2-dimensional tree has arbitrarily many 2-dimensional

children that have a 1-dimensional ordering, a node in a 3-dimensional tree would have arbitrarily many 3-dimensional children that have a 2-dimensional (partial) ordering. For an n -dimensional tree, a node would have arbitrarily many n -dimensional children with an $(n - 1)$ -dimensional (partial) ordering.

Instead, we choose to use a well-known “left child, right sibling” representation [Sed98] for 2-dimensional trees. Instead of maintaining references to arbitrarily many 2-dimensional children, each node contains a reference to a single minimum 2-dimensional successor (the “left child”) and to a single minimum 1-dimensional successor (the “right sibling”). With this representation, the first child of a node is positioned as the 2-dimensional successor of that node. The remaining children are then each placed as the 1-dimensional successor of the preceding child. Note that in full generality, this representation admits the possibility that the root of the tree may have a right sibling. We will interpret such a structure as a linearly ordered forest, and we will interpret the case where the root has no right sibling as the trivial forest, a single tree.

Bottom-up construction for the usual left-child/right-sibling trees has two operations as seen in Figure 7: extending the forest by adding another tree at the beginning of the ordered forest (which we will call EXLEFT) and adding a root as the parent of the trees in an ordered forest (which we will call EXUP).

Definition 1 *Linearly Ordered Forests*

- \sim is an empty forest.

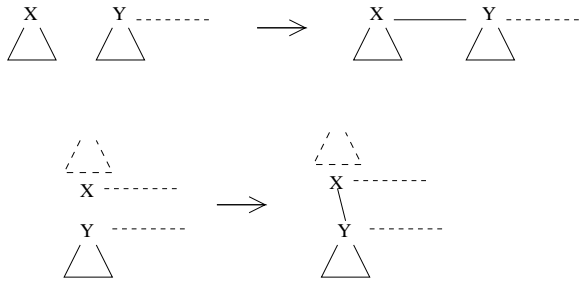


Figure 7: The two tree-building operations.

- If t_1 is a tree and t_2 is a linearly ordered forest then $\text{EXLEFT}(t_1, t_2)$ is a linearly ordered forest.
- If t_1 is a linearly ordered forest and $X \in \Sigma$ then $\text{EXUP}(X, t_1)$ is a tree.
- Nothing else is a linearly ordered forest.

This allows us to construct 2-dimensional trees of arbitrary branching structures using only two links per node.¹ We will later generalize this structure to allow us to construct arbitrary dimensional, arbitrary branching structures requiring only one link per dimension per node.

2.1.1 Example

First we will look at a 2-dimensional example using these two structure building operations. We will build the tree shown in Figure 8. In the figure, solid lines represent the actual links while the dotted lines are there only to better visualize the tree structure. An annotated version of the tree shown in Figure 9 shows the various local

¹An interesting result of using the left-child/right-sibling tree organization is that it allows every n -branching tree to be embedded in a binary branching structure.

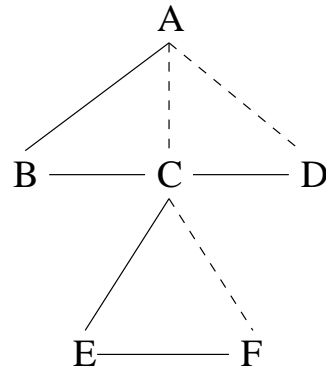


Figure 8: An example two-dimensional tree.

trees and forests that make up the tree as circled groups.

Starting at the bottom, $\text{EXUP}(F, \sim)$ creates tree t_1 . $\text{EXLEFT}(\sim, t_1)$ combines t_1 with an empty tree to form the singleton forest f_1 . Similarly, $\text{EXUP}(E, \sim)$ forms tree t_2 , which is then added to f_1 with $\text{EXLEFT}(f_1, t_2)$ to create f_2 . Repeating this step once again, $\text{EXUP}(D, \sim)$ creates t_3 and $\text{EXLEFT}(\sim, t_3)$ creates f_3 . t_4 is created with $\text{EXUP}(C, f_2)$ and then added to f_3 to create f_4 through $\text{EXLEFT}(f_3, t_4)$. $\text{EXUP}(B, \sim)$ creates t_5 which is used in $\text{EXLEFT}(f_4, t_5)$ to create f_5 . The tree is then completed with $\text{EXUP}(A, f_5)$ to create t_6 and $\text{EXLEFT}(\sim, t_6)$ to construct the final tree.

2.2 Unified Constructor

Generalizing the two tree-building operations to arbitrary dimensions, however, is simpler if we create a new, unified constructor. Instead of first adding siblings to create a forest, and then assigning the forest a root node, we will do this in one step. Taking a label and two forests, we will create a new node that is the root of the first

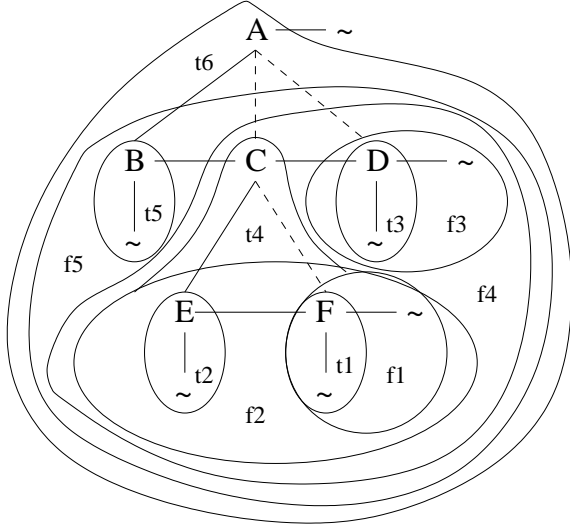


Figure 9: An example tree annotated to show trees and forests.

forest and the leftmost sibling of the second forest. The “root of a forest” is simply the root of the minimum tree in that forest. In contrast, we may refer to the minima of a forest—these are simply the roots of the individual trees of the forest, and they are incomparable to each other in the major dimension of the forest. The root of the forest is the unique minimum of the forest iff the forest is a singleton forest, or a tree.

Using the unified constructor, a 2-dimensional (singleton) forest is formed if there is no 1-dimensional child. Otherwise, a proper forest is formed. Each of the siblings of the new node is a minimum.

Definition 2 *Linearly Ordered Forests (unified constructor)*

- \sim is an empty linearly ordered forest.
- If t_1 and t_2 are linearly ordered forests and $X \in \Sigma$ then $T(X, t_1, t_2)$. Here t_2 is the set of two-dimensional children of

the new node labeled X , and t_1 is the set of its siblings to the right.

- Nothing else is a linearly ordered forest.

2.2.1 Example

Using the unified constructor to construct the tree in Figure 8, every pair of EXUP and EXLEFT terms using the same first argument can be combined with that term as the first argument of the T constructor. The next arguments of the constructor would be the next terms of the EXUP and EXLEFT instances respectively. The terms of the construction would be:

$$\begin{aligned} T(F, \sim, \sim) &= f1 \\ T(E, f1, \sim) &= f2 \\ T(D, \sim, \sim) &= f3 \\ T(C, f3, f2) &= f4 \\ T(B, f4, \sim) &= f5 \end{aligned}$$

and $T(A, \sim, f5)$ completes the tree.

2.3 Extending to Arbitrary Dimensions

As our trees become more complex, it is helpful to define some terminology that allows us to talk about various aspects of an n -dimensional tree without our understanding becoming obfuscated in bulky explanation.

Definition 3 *n -dimensional Local Tree*

A local tree is a tree of height ≤ 1 in each dimension. This consists of a root and an $(n-1)$ -dimensional yield, which we will call the local yield.

Definition 4 $(n - 1)$ -dimensional Local Yield

An $(n - 1)$ -dimensional local yield of a node is the set of all n -dimensional children of that node, which are ordered as an $(n - 1)$ -dimensional singleton forest, a tree—there must be exactly one minimum with respect to the $(n - 1)$ -dimensional ordering.

Definition 5 $(n - 1)$ -dimensional Child Structure

An $(n - 1)$ -dimensional child structure of an n -dimensional node is the forest of trees rooted at that node's $(n - 1)$ -dimensional local yield.

Now we can restate the way our 2-dimensional trees are built. Every node in a 2-dimensional forest roots two local trees: a 2-dimensional local tree, the yield of which are the node's two-dimensional children, and a 1-dimensional local tree whose yield is the node's right sibling. Consequently, the 2-dimensional successor is the minimal point in the node's 1-dimensional child structure, and the 1-dimensional successor is the minimal point in the node's 0-dimensional child structure. Both these child structures are linearly ordered forests. To extend this to arbitrary dimensions, we will say that every node in an n -dimensional tree has n successors. Each successor in the i th dimension is the minimal point in that node's $(i - 1)$ -dimensional child structure for $1 \leq i \leq n$. We extend the unified construction as follows:

Definition 6 Tree-ordered Forests (preliminary)

- \sim is an (empty) d -dimensional forest.

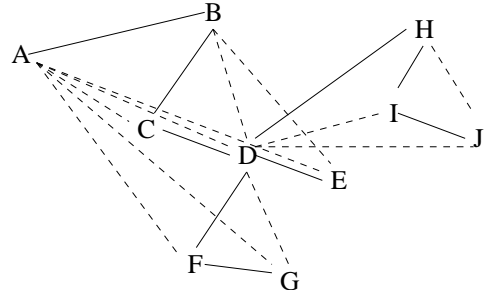


Figure 10: An example three-dimensional tree.

- If t_1, t_2, \dots, t_d are tree ordered forests and $X \in \Sigma$ then $T(X, t_1, t_2, \dots, t_d)$ is a tree-ordered forest. t_i is the set of i -dimensional children of the new node labeled X .
- Nothing else is a tree-ordered forest.

2.3.1 Example

To see the unified constructor for n dimensions, we will look at the 3-dimensional tree shown in Figure 10. As in Figure 8, the solid lines are the actual links while the dotted lines are present to better visualize the tree structure. Again, the tree has been annotated (Figure 11) to show the local trees and forests.

The construction operation for this tree will take four arguments: the label of the node and three forests. We will indicate the construct with $T(W, X, Y, Z)$ where W is the label, X is a forest and the 1-dimensional successor of W , Y is a forest and the 2-dimensional successor, and Z is a forest and the 3-dimensional successor. The empty tree is represented by \sim . In all our constructions of trees, the order that the forests are constructed is flexible. Until two forests are combined, they can be built

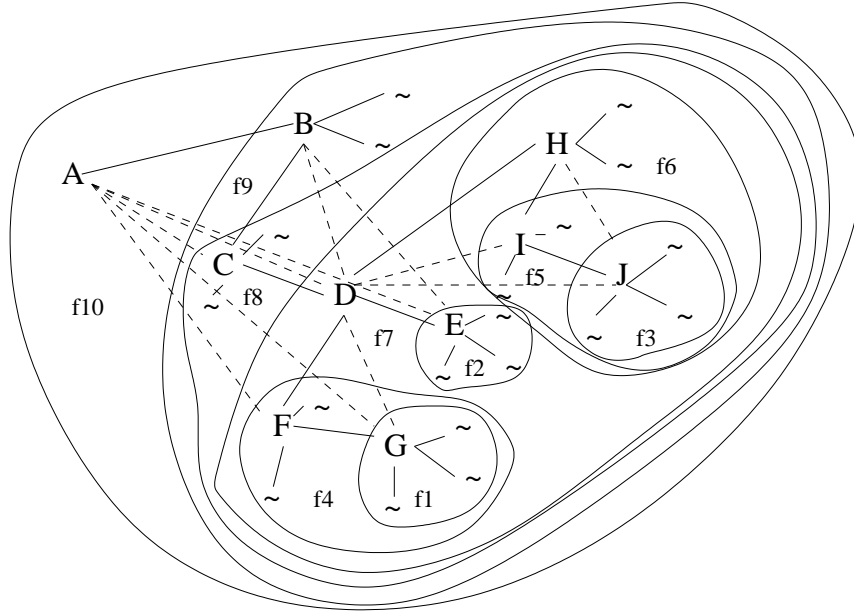


Figure 11: An example three-dimensional tree, annotated to show trees and forests.

independently of one another.

$$T(A, \sim, \sim, f9) = f10$$

First we will construct $f6$:

$$T(J, \sim, \sim, \sim) = f3$$

$$T(I, f3, \sim, \sim) = f5$$

$$T(H, \sim, f5, \sim) = f6$$

Then we create $f4$:

$$T(G, \sim, \sim, \sim) = f1$$

$$T(F, f1, \sim, \sim) = f4$$

And the last child of D is $f2$:

$$T(E, \sim, \sim, \sim) = f2$$

Next we combine them in D :

$$T(D, f2, f4, f6) = f7$$

And then we can continue to build the rest of the tree:

$$T(C, f7, \sim, \sim) = f8$$

$$T(B, \sim, f8, \sim) = f9$$

2.4 (i, d) -forests

Our tree-ordered forest definition is not yet complete. In the above example, note that when assigning $f6$ as the 3-dimensional successor of D , the root of $f6$ does not have a 1-dimensional successor. In fact, it *can't* have a 1-dimensional successor—it is the root of a 2-dimensional local yield, which by our definition must be a singleton 2-dimensional forest. Only the empty tree can be used for this constructor argument, and definition 6 needs to be modified to include this. To denote a d -dimensional forest with a unique minimum (within a local structure, in this sense) in dimension i , we define an (i, d) -forest, where $0 \leq i \leq d$, as a forest whose root has an empty j -dimensional local yield for all $j < i$. A node with no i -dimensional children for all $i < j$ can be interpreted as

the root of a (j, d) -forest. Having done this, we can insist that the only forests accepted as a 3-dimensional successor in the unified constructor be $(2, d)$ -forests, the subset of forests that do not have a 1-dimensional successor.

Lemma 1 *If X is the root of an i -dimensional local yield, it has no j -dimensional successor for $j < i$.*

Proof: When $i = 0$ or $i = 1$, this is trivially true. For $i > 1$, if X has any j -dimensional successor for $j < i$, then it is not the unique minimum of the local yield and hence is not the root of that local yield, which must be a singleton forest. \dashv

Corollary 1 *Every (i, d) -forest is also a (j, d) -forest for $j < i$.*

Definition 7 *Tree-ordered Forests [Fully typed]*

- \sim is an (empty) (i, d) -forest for all $0 \leq i \leq d$
- If t_1, t_2, \dots, t_d are, respectively, $(0, d)$ -, $(1, d)$ -, \dots , $(d-1, d)$ -forests and $X \in \Sigma$ then $T(X, t_1, t_2, \dots, t_d)$ is a (j, d) -forest for all $0 \leq j \leq i$, where i is the smallest dimension such that t_i is not empty, or d if all t_k are empty. Here each t_k is the successor of the new node labeled X in the k th dimension.
- Nothing else is a tree-ordered forest.

By Corollary 1, the set of (i, d) -forest \subset the set of $(i-1, d)$ -forest for all $0 < i \leq d$.

Going back to Figure 11, we can label each tree as an (i, d) -forest. $f1, f2$, and $f3$ are $(0,3)$ -forests. $f4, f5, f7$, and $f8$ are $(1,3)$ -forests and, by Corollary 1, $(0,3)$ -forests. $f6$ and $f9$ are $(2,3)$ -forests (and both $(1,3)$ - and $(0,3)$ -forests). Finally, $f10$ is a $(3,3)$ -forest.

3 Concrete Forms

In order to work with these trees in more concrete applications such as programs, we must develop a notion of these trees both as an abstract data type for storing trees in memory, and as a string to store trees in files and use as input.

3.1 Abstract Data Type

In order to create programs to work with these trees, we will need a way of constructing data structures that can represent an n -dimensional forest. The information we need to store falls directly out of the definition of forests. The forest constructor T takes a label and a sequence of (i, d) -forests. Our data structure will then store the label of the node created and references to the forests that are that node's local yields. With this model, one instance of our structure can represent one node, with the label and pointers to the local yields being stored. Since each local yield is made up of smaller local yields, what is really being pointed to is a node that is the minimum of that local yield.

The constructor for building forests follows Definition 7. This function takes a list of references to forests and a label and returns a new forest rooted at that label.

Each element of the list of forests passed to the constructor represents its successor in a particular dimension designated by the ordering.

Selectors would be needed to determine the label and successor of each node. Mutators could also be included to allow modification of the label and the successors, but this is not necessarily needed since the constructor can be used to set these.

Here is an example of this ADT realized as a C++ class:

```
template<class label_type>
class forest
{
public:
    forest(label_type label,
           vector<forest*> links)

    void set_label(label_type new_label);
    void set_link(std::size_t link_number,
                 forest* new_link);

    label_type get_label( ) const;
    tree* get_successor(
        std::size_t link_number) const;

private:
    label_type label;
    vector<forest*> link;
}

```

Because of the C++ vector class used for the implementation, this class is polymorphic in its dimensionality. The dimensionality of the forests is not explicit in this data type; it is however noted in the size of the vector. While the constructor allows for a d-dimensional forest to reference forests of different dimensionality this may be disallowed by adding assert statements throughout the code to ensure uniformly dimensional trees.

3.2 Flat Form

When working with complex trees as the input or output of a program or attempting to store trees in text files, it becomes necessary to develop a concise way of representing the tree in a flat (string) form. A form of this type can be taken directly from the free algebra given by the construction from Σ and \sim . One variation, to allow for more concise and easier to read trees, is to write the terms as $X(t_1, t_2, \dots, t_d)$ rather than $T(X, t_1, t_2, \dots, t_d)$.

Definition 8 Flat Form

- \sim is an (empty) (i, d) -forest in flat form for all $0 \leq i \leq d$
- If t_1, t_2, \dots, t_d are, respectively, $(0, d)$ -, $(1, d)$ -, \dots , $(d - 1, d)$ -forests in flat form and $X \in \Sigma$ then $X(t_1, t_2, \dots, t_d)$ is a (j, d) -forest in flat form for all $0 \leq j \leq i$, where i is the smallest dimension such that t_i is not empty, or d if all t_k are empty. Here each t_k is the successor of the new node labeled X in the k th dimension.
- Nothing else is a forest in flat form.

3.2.1 Example

The terms of the algebra for the tree in Figure 8 are:

$$\begin{aligned}
 &F(\sim, \sim) \\
 &E(F(\sim, \sim), \sim) \\
 &D(\sim, \sim) \\
 &C(D(\sim, \sim), E(F(\sim, \sim), \sim)) \\
 &B(C(D(\sim, \sim), E(F(\sim, \sim), \sim)), \sim) \\
 &A(\sim, B(C(D(\sim, \sim), E(F(\sim, \sim), \sim)), \sim))
 \end{aligned}$$

For a more complicated example, here are the terms of Figure 10:

$G(\sim, \sim, \sim)$
 $F(G(\sim, \sim, \sim), \sim, \sim)$
 $E(\sim, \sim, \sim)$
 $J(\sim, \sim, \sim)$
 $I(J(\sim, \sim, \sim), \sim, \sim)$
 $H(\sim, I(J(\sim, \sim, \sim), \sim, \sim), \sim)$
 $D(E(\sim, \sim, \sim), F(G(\sim, \sim, \sim), \sim, \sim),$
 $\quad H(\sim, I(J(\sim, \sim, \sim), \sim, \sim), \sim))$
 $C(D(E(\sim, \sim, \sim), F(G(\sim, \sim, \sim), \sim, \sim),$
 $\quad H(\sim, I(J(\sim, \sim, \sim), \sim, \sim), \sim)), \sim, \sim)$
 $B(\sim, C(D(E(\sim, \sim, \sim), F(G(\sim, \sim, \sim), \sim, \sim),$
 $\quad H(\sim, I(J(\sim, \sim, \sim), \sim, \sim), \sim)), \sim, \sim), \sim)$
 $A(\sim, \sim, B(\sim, C(D(E(\sim, \sim, \sim), F(G(\sim, \sim, \sim), \sim, \sim),$
 $\quad H(\sim, I(J(\sim, \sim, \sim), \sim, \sim), \sim)), \sim, \sim), \sim))$

[Wei92] David J. Weir. A geometric hierarchy beyond context-free languages. *Theoretical Computer Science*, 104, 1992.

References

- [GS84] Ferenc Gécseg and Magnus Steinby. *Tree Automata*. Akadémiai Kiadó, Budapest, 1984.
- [JS96] Aravind K. Joshi and Yves Schabes. Tree-adjointing grammars. In A. Salomaa and S. Rozenberg, editors, *Handbook of Formal Languages and Automata*. Springer-Verlag, 1996.
- [Sed98] Robert Sedgewick. *Algorithms in C++*. Addison-Wesley, 3rd edition, 1998.
- [Sip01] Michael Sipser. *Introduction to the Theory of Computation*. PWS Publishing Company, 20 Park Plaza, Boston, MA 02116, 2001.
- [Wei88] David J. Weir. *Characterizing Mildly Context-Sensitive Grammar Formalisms*. PhD thesis, University of Pennsylvania, 1988.