Representing Multidimensional Trees

David Brown, Ian Kelly, Colin Kern, Alex Lemann, and Greg Sandstrom
Earlham College
Department of Computer Science
Richmond, Indiana
{brownda,kellyia,kernco,lemanal,sandsgr}@cs.earlham.edu

October 15, 2004

Abstract

This paper develops a formal definition of multidimensional trees as abstract structures in "left-child, right-sibling" form. After developing this abstract definition, we show how it can be directly implemented as an ADT suitable for use in parsing applications. Additionally, we show how, when viewed in a slightly different way, our definition yields a flat form suitable for serialized input and output.

1 Introduction

Traditionally, the context-free grammars (CFGs) are represented as a set of string rewriting rules: a CFG 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].

We often are interested in the parse trees that derive strings in a grammar, rather than just the strings that are derivable. We can choose to interpret the definition of $\mathcal{G}$ differently, and represent the context-free grammars using local trees [GS84, Rog03]. Interpret each production in P as a local tree (a tree with height $h \leq 1$) having a yield labeled in $(\Sigma \cup V)^*$ and a root labeled in V. Let each member of Σ label the root of a trivial local tree, and let the initial symbol S label the root of a trivial derivation tree T_0. The context-free derivation of any T_{i+1} from T_i can be performed by by replacing a leaf node of T_i, labeled by some x, with a local tree in P that has root labeled x—that is, by concatenating local trees from P to the derivation tree T_i. The derived string, at any point in the process, is the string yield (the left-to-right concatenation of the leaf nodes) of T_i. A string is in the language recognized by $\mathcal{G}$, $L(\mathcal{G})$, if and only if it is in Σ^* and is the string yield of some derivation tree.

We can see that, in this representation, each local tree in P is grammatically equivalent to a production in a traditional CFG—the tree maps some root symbol, $x \in V$, to some ordered set of children $y \in (\Sigma \cup V)^*$. Figure 1 shows an example grammar in local tree form, plus an example derivation

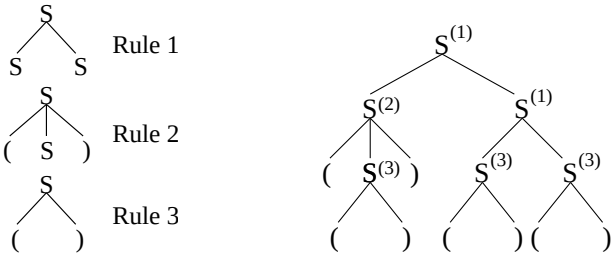

Figure 1: An example context-free grammar in local tree form, plus one possible derivation tree with root labeled by S.

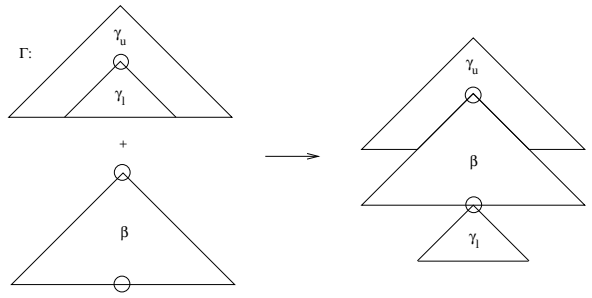

Figure 2: Adjunction of tree β into Γ_i.

tree constructed from rules in the 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. While CFGs derive strings by symbol replacement, TAGs derive trees by a process known as adjunction.

Conceptually, tree adjunction in TAG is the substitution of one tree into another at some node (Figure 2). To adjoin some auxiliary tree β to some derivation tree Γ_i at node x, we require that the auxiliary tree β have a root node with the same label as x, as well as a distinguished leaf node with the same label as x (known as the "foot" node, or foot(β)). To perform the adjunction, we create two trees from Γ_i—γ_u (identical to Γ_i, but with the subtree below x deleted), and γ_l (the subtree rooted by x). We create Γ_{i+1} from Γ_i via two replacements: x in the frontier of γ_u is replaced by β, and foot(β) is replaced by γ_l.

Adjunction, if allowed to occur only at frontier nodes, is exactly equivalent to tree concatenation. The ability to replace non-frontier nodes with trees is what differentiates TAGs from our local tree representation of CFGs. This same ability affords TAGs a degree of context-sensitive generative power not found in CFGs.

Thus far, we have defined a local tree as a traditional parent/child n-branching tree with restricted height ($h \leq 1$). From a different point of view, a local tree is of the exact structure of a single CFG production: it maps one point (the root symbol) to one string of symbols licensed to replace it. From this perspective, a local tree can be thought of as an arbitrary one-dimensional string of symbols, dominated in the second dimension by exactly one symbol. This definition of a local tree can be generalized to arbitrary dimensions, in a way analogous to the construction of topological simplicies. To construct a d-dimensional local tree T^d from an arbitrary $(d-1)$-dimensional tree T^{d-1}, we add exactly one new node. This new node serves as the root of T^d and is the immediate predecessor, in an orthogonal dimension[1], of every node in T^{d-1} (Figure 3). The local yield of T^d is

[1]To represent a multi-dimensional local structure on a two-dimensional medium, we place each d-dimensional root to the left of the $(d-1)$-dimensional structure it dominates, and connect it to each node in that $(d-1)$-dimensional structure with a line. A unique line style is used for each dimension.

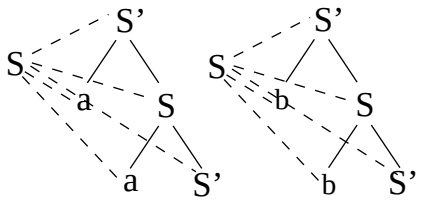

Figure 3: Two three-dimensional local trees. These trees denote a tree-adjoining grammar in local tree form.

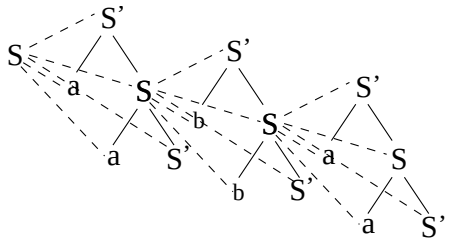

Figure 4: One possible three-dimensional derivation tree.

T^{d-1}.

Concatenation of these local d-dimensional structures allows the construction of arbitrary d-dimensional trees (Figure 4). For an arbitrary d-dimensional tree, we note that the $(d-1)$-dimensional yield is constructible: the yield of an arbitrary d-dimensional tree is obtained by combining, in some well-defined order[2], all of the $(d-1)$-dimensional yields of its d-dimensional local components (Figure 5).

For the CFGs, we were able to represent string rewriting via concatenation of trees—that is, we were able to reduce substitution in a structure to concatenation of structures—purely by raising the dimensionality of the structure. We moved from substitution on one-dimensional structures (strings) to concatenation of two-dimensional local structures, while maintaining equivalent generative power. In a similar fashion, we can achieve generative power equivalent to that of TAG using only concatenation—by moving from adjunction and substitution in two-dimensional structures to concatenation of three-dimensional local structures [Rog03].

[2]For dimensions greater than two, there is an ordering ambiguity that necessitates the storage of extra per-node information.

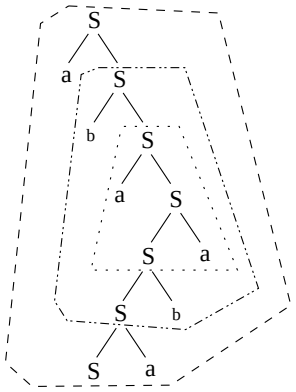

Figure 5: The two-dimensional yield of a three-dimensional derivation tree.

More simply, we can capture the generative power of TAG with a slight variation on our definition of CFGs as sets of local structures: we need only raise the dimensionality of the local structures in P by one. This process may be iterated to higher dimensions, creating an infinite hierarchy of language classes that, in terms of string-generating power, directly corresponds to Weir's control language hierarchy [Wei92].

If viewed as a multidimensional grammar, Figure 3 captures the copy language over $\{a, b\}$ using a set of three-dimensional local trees. Figure 4 shows one possible derivation in this grammar. Figure 5 shows the two-dimensional yield of the derivation tree in Figure 4, with circles around the two-dimensional yield of each three-dimensional local tree. The one-dimensional yield of Figure 5 is the string recognized by the derivation tree in Figure 4.

Dewdney [Dew74] discusses a similar (but

formally distinct) graph-theoretic notion of multidimensional trees. A formally equivalent notion of multidimensional trees, as well as a formally distinct notion of multidimensional grammars, is discussed by Baldwin and Strawn [BS91]. Adaptive k-d trees [Sed98], while useful for searching in a multidimensional space, are not strictly multidimensional in their structure.

Our group is interested in building parsers for grammars based on multidimensional trees. In this paper, we develop a formal definition of these multidimensional trees as abstract structures that can be directly realized as an ADT. We then implement this ADT as a C++ class. Further, by interpreting this abstract structure algebraically we obtain a flat form suitable for file input and output.

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 6: 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.*
- *If* t_1 *is a tree and* t_2 *is a linearly ordered forest then* $\textsc{ExLeft}(t_1, t_2)$ *is a linearly ordered forest.*
- *If* t_1 *is a linearly ordered forest and* $X \in \Sigma$ *then* $\textsc{ExUp}(X, t_1)$ *is a tree.*
- *Nothing else is a linearly ordered forest.*

This allows us to construct arbitrarily branching 2-dimensional trees using only two links per node.[3] We will later generalize this structure to allow us to construct arbitrary dimensional, arbitrary branching structures requiring only one link per dimension per node.

[3] An interesting consequence 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.

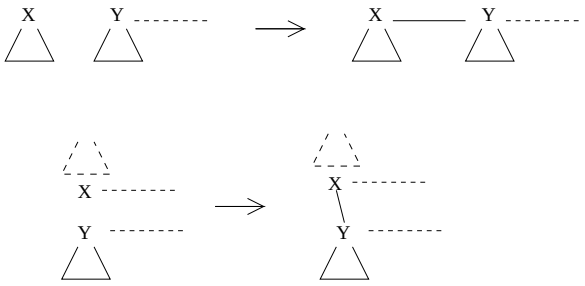

Figure 6: The two tree-building operations.

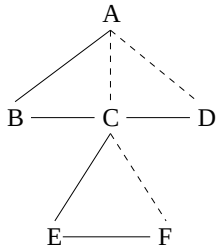

Figure 7: An example two-dimensional tree.

2.1.1 Example

First we will look at a 2-dimensional example using these two tree building operations. We will build the tree shown in Figure 7. 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 in Figure 8 shows the various local trees and forests that make up the tree as circled groups.

Starting at the bottom, ExUp$(F, \sim)$ creates tree $t1$. ExLeft$(\sim, t1)$ combines $t1$ with an empty tree to form the singleton forest $f1$. Similarly, ExUp$(E, \sim)$ forms tree $t2$, which is then added to $f1$ with ExLeft$(f1, t1)$ to create $f2$. Repeating this step once again, ExUp$(D, \sim)$ creates $t3$ and ExLeft$(\sim, t3)$ creates $f3$. $t4$ is created with ExUp$(C, f2)$ and then added to $f3$ to create $f4$ through ExLeft$(f3, t4)$. ExUp$(B, \sim)$ creates $t5$ which is used in ExLeft$(f4, t5)$ to create $f5$. The tree is

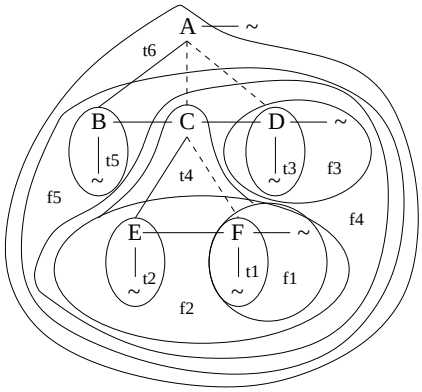

Figure 8: An example tree annotated to show trees and forests.

then completed with ExUp$(A, f5)$ to create $t6$ and ExLeft$(\sim, t6)$ to construct the final tree.

2.2 Unified Constructor

Generalizing the two tree-building operations to arbitrary dimensions 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 parent of the roots of the trees in the first forest, and is 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 if and only if the forest is a singleton forest, i.e. a tree.

Using the unified constructor, a 2-dimensional (singleton) forest is formed if the forest has 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* $T(X, t_1, t_2)$ *is a linearly-ordered forest. 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

Let us use the unified constructor to construct the tree in Figure 7. Every pair of ExUp and ExLeft invocations that use the same first argument can be combined, with that shared term as the first argument of the T constructor. The remaining arguments of the constructor would be the remaining terms of the ExUp and ExLeft instances, respectively. The terms of the construction would be:

$T(F, \sim, \sim) = f1$
$T(E, f1, \sim) = f2$
$T(D, \sim, \sim) = f3$
$T(C, f3, f2) = f4$
$T(B, f4, \sim) = f5$

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

2.3 Extending to Arbitrary Dimensions

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

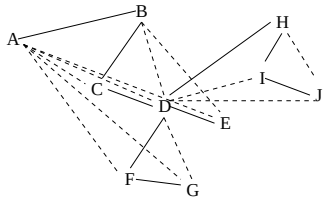

Figure 9: An example three-dimensional tree.

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.*

Note that the local yield of a node is an $(n-1)$-dimensional tree of nodes, while the child structure of the same node is an $(n-1)$-dimensionally ordered forest of n-dimensional trees rooted at those nodes.

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 minimum point in the node's 1-dimensional child structure, and the 1-dimensional successor is the minimum 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 ith dimension is the minimum 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 *(Preliminary) Tree-ordered Forests*

- $\sim$ *is an (empty) d-dimensional forest.*
- *If* $t_1, t_2, \ldots, t_d$ *are tree ordered forests and* $X \in \Sigma$ *then* $T(X, t_1, t_2, \ldots, t_d)$ *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 9. As in Figure 7, 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 10) to show the local trees and forests.

The constructor for this tree will take four arguments: the label of the node, and three forests. We will indicate this construct as $T(W, X, Y, Z)$ where W is the label, X is a forest and the 1-dimensional successor

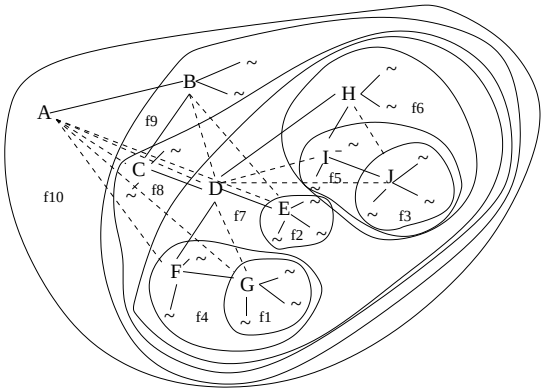

Figure 10: An example three-dimensional tree, annotated to show trees and forests.

of W, Y is a forest and the 2-dimensional successor of W, and Z is a forest and the 3-dimensional successor of W. The empty tree is represented by $\sim$. In all of our constructions of trees, the order that the forests are constructed in is flexible. Until two forests are combined, they can be built independently of one another.

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:

$T(C, f7, \sim, \sim) = f8$
$T(B, \sim, f8, \sim) = f9$

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

2.4 (i, d)-forests

Our definition of tree-ordered forests 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. ⊣

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, ..., t_d are, respectively, $(0, d)$-, $(1, d)$-, ..., $(d-1, d)$-forests and $X \in \Sigma$ then $T(X, t_1, t_2, \ldots, t_d)$ 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 kth dimension.*
- *Nothing else is a tree-ordered forest.*

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

Going back to Figure 10, 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 realization 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 our class uses the the C++ `vector` in its implementation, it is truly 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 assertions to the constructor 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 generated by the constructor from Σ and $\sim$. One variation, to allow for more concise and easier to read trees, is to write the terms as $X(t_1, t_2, \ldots, t_d)$ rather than $T(X, t_1, t_2, \ldots, t_d)$.

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, $\ldots$, t_d *are, respectively,* $(0, d)$*-,* $(1, d)$*-,* $\ldots$*,* $(d-1, d)$*-forests in flat form and* $X \in \Sigma$ *then* $X(t_1, t_2, \ldots, t_d)$ *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 7 are:

$F(\sim, \sim)$
$E(F(\sim, \sim), \sim)$
$D(\sim, \sim)$
$C(D(\sim, \sim), E(F(\sim, \sim), \sim))$
$B(C(D(\sim, \sim), E(F(\sim, \sim), \sim)), \sim)$
$A(\sim, B(C(D(\sim, \sim), E(F(\sim, \sim), \sim)), \sim))$

4 Conclusion

We have given an inductive definition of tree-ordered forests extending the standard "left child, right sibling" form of 2-dimensional trees to multiple dimensions. This abstract class of structures has been interpreted in two ways. First, we used this definition to build a compact abstract data type for multidimensional tree nodes, with size linear in the dimensionality of the tree. An instance of this ADT was realized as a C++ class which is fully polymorphic in its dimensionality. Our definition can also be understood as a term algebra which leads to the definition of a flat form intended for file input and output. Our group is now using these representations in developing parsing algorithms for the multidimensional grammars described in the introduction.

References

[BS91] William A. Baldwin and George O. Strawn. Multidimensional trees. *Theoretical Computer Science*, 84:293–311, 1991.

[Dew74] A. K. Dewdney. Higher-dimensional tree structures. *Journal of Combinatorial Theory*, 17:160–167, 1974.

[GS84] Ferenc Gécseg and Magnus Steinby. *Tree Automata*. Akadémiai Kiadó, Budapest, 1984.

[JS96] Aravind K. Joshi and Yves Schabes. Tree-adjoining grammars. In A. Salomaa and S. Rozenberg, editors, *Handbook of Formal Languages and Automata*. Springer-Verlag, 1996.

[Rog03] James Rogers. wMSO theories as grammar formalisms. *Theoretical Computer Science*, 293(2):291–320, 2003.

[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.

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