Implementing Algebraic Dynamic Programming in the Functional and the Imperative Programming Paradigm

Robert Giegerich, Peter Steffen

Faculty of Technology, Bielefeld University
33501 Bielefeld, Germany
{robert,psteffen}@techfak.uni-bielefeld.de

Abstract. Algebraic dynamic programming is a new method for developing and reasoning about dynamic programming algorithms. In this approach, so-called yield grammars and evaluation algebras constitute abstract specifications of dynamic programming algorithms. We describe how this theory is put to practice by providing a specification language that can both be embedded in a lazy functional language, and translated into an imperative language. Parts of the analysis required for the latter translation also gives rise to source-to-source transformations that improve the asymptotic efficiency of the functional implementation. The multi-paradigm system resulting from this approach provides increased programming productivity and effective validation.

1 Motivation

1.1 Towards a discipline of dynamic programming

Dynamic Programming (DP)[2] is a well-established and widely used programming technique. In recent years, the advances of molecular biology have created thriving interest in dynamic programming algorithms over strings, since genomic data pose sequence analysis problems in unprecedenced complexity and data volume [4]. In this context, it became apparent that there is the lack of a formal method for developing the intricate matrix recurrences that typically constitute a DP algorithm.

1.2 A short review of ADP

Algebraic dynamic programming (ADP) is a technique designed to alleviate this situation. Postponing technical definitions to later sections, the ADP approach can be summarized as follows: Any DP algorithm evaluates a search space of candidate solutions under a scoring scheme and an objective function. The classical DP recurrences reflect the three aspects of search space construction, scoring and choice, and efficiency in an indiscriminable fashion. In the new algebraic approach, these concerns are separated. The search space is described by a so-called

yield grammar, evaluation and choice by an algebra, and efficiency concerns can be persued on a very high level of abstraction. No subscripts, no (subscript) errors.
Based on the abstract concepts of yield grammars and evaluation algebras, ADP is essentially a piece of programming theory. The present paper is concerned with putting this theory into practice.

1.3 Overview of this contribution

Section 2 reviews the central notion of the ADP approach and introduces a domain-specific notation for describing ADP algorithms. The core of implementing ADP algorithms (in either programming paradigm) is the technique of tabulating yield parsers, introduced in Section 3. An embedding of ADP notation in Haskell is given in Section 4, which allows rapid prototyping, but has some methodical and some practical limitations. Section 5 is dedicated to compilation techniques, which allow to generate either optimized ADP/Haskell notation or C code. Finally, Section 6 describes the overall style of algorithm development that arises from this approach.

1.4 Related work

The wide use of DP in bioinformatics is documented in [4], but without methodical guidance. The relation between parsing and dynamic programming is discussed as an open problem in [17].
Yield parsing, as introduced here, takes a string as its input, and therefore, although based on a tree grammar, it is more closely related to string than to tree parsing methods. Its closest relatives are methods for parsing ambiguous context free languages, such as Earley's or the CYK algorithm [1].
The ADP approach has evolved in the recent years in the application context of biosequence analysis. An informal description of the method is found in [6], the first rigorous definition is given in [8], while some applications have appeared earlier: The first application developed in the spirit of the yet-to-be-developed ADP method is a program for aligning recombinant DNA [7]. ADP has further been applied to solve the problem of folding saturated RNA secondary structures, posed by Zuker and Sankoff in 1984 [5, 21]. An application to statistical scoring in pattern matching is reported in [14]. The development of the ADP compiler described here is new and based on ongoing research [19].

2 Tree grammars and yield languages

2.1 Basic terminology

Alphabets. An *alphabet* $\mathcal{A}$ is a finite set of symbols. Sequences of symbols are called strings. ε denotes the empty string, $\mathcal{A}^1 = \mathcal{A}$, $\mathcal{A}^{n+1} = \{aw | a \in \mathcal{A}, w \in \mathcal{A}^n\}$, $\mathcal{A}^+ = \bigcup_{n \geq 1} \mathcal{A}^n$, $\mathcal{A}^* = \mathcal{A}^+ \cup \{\varepsilon\}$. By convention, a denotes a single symbol, w and x a string over $\mathcal{A}^*$.

Signatures and algebras. A (single-sorted) signature Σ over some alphabet $\mathcal{A}$ consists of a sort symbol S together with a family of operators. Each operator o has a fixed arity $o : s_1...s_{k_o} \to S$, where each s_i is either S or $\mathcal{A}$. A Σ-algebra $\mathcal{I}$ over $\mathcal{A}$, also called an interpretation, is a set $\mathcal{S}_\mathcal{I}$ of values together with a function o_I for each operator o. Each o_I has type $o_I : (s_1)_I...(s_{k_o})_I \to S_I$ where $\mathcal{A}_I = \mathcal{A}$.
A *term algebra* T_Σ arises by interpreting the operators in Σ as *constructors*, building bigger terms from smaller ones. When variables from a set V can take the place of arguments to constructors, we speak of a term algebra with variables, $T_\Sigma(V)$, with $V \subset T_\Sigma(V)$.
Trees and tree patterns. Terms will be viewed as rooted, ordered, node-labeled trees in the obvious way. Note that only leaf nodes can carry symbols from $\mathcal{A}$. A term/tree with variables is called a *tree pattern*. A tree containing a designated occurrence of a subtree t is denoted $C[...t...]$. We adopt the view that the tree constructors represent some structure that is associated explicitly with the sequence of leaf symbols from $\mathcal{A}^*$. The nullary constructors that may reside at leaf nodes are not considered part of the yield. Hence the yield function y on $T_\Sigma(V)$ is defined by $y(t) = w$, where $w \in (\mathcal{A} \cup V)^*$ is the sequence of leaf symbols from $\mathcal{A}$ and V in left to right order.

2.2 Tree grammars

A tree language over Σ is a subset of T_Σ. Tree languages are described by tree grammars, which can be defined in analogy to the Chomsky hierarchy of string grammars. Here we use regular tree grammars originally studied in [3], with the algebraic flavour introduced in [9]. Our specialization so far lies solely with the distinguished role of the alphabet $\mathcal{A}$.

Definition 1 (Tree grammar.) A regular tree grammar $\mathcal{G}$ over Σ is given by

- a set V of nonterminal symbols,
- a designated nonterminal symbol Ax called the axiom,
- a set P of productions of the form $v \to t$, where $v \in V$ and $t \in T_\Sigma(V)$.

The derivation relation for tree grammars is $\to^*$, with $C[...v...] \to C[...t...]$ if $v \to t \in P$. The language of $v \in V$ is $\mathcal{L}(v) = \{t \in T_\Sigma | v \to^* t\}$, the language of $\mathcal{G}$ is $\mathcal{L}(\mathcal{G}) = \mathcal{L}(Ax)$. For arbitrary $q \in T_\Sigma(V)$ we define $\mathcal{L}(q) = \{t \in T_\Sigma | q \to^* t\}$. $\square$

2.3 Lexical level and conditional productions

The following two extensions are motivated by the fact that yield grammars are to be used as a programming device. We add a *lexical level* to our grammars. As with context free grammars, this is not necessary from a theoretical point of view, but makes examples less trivial and applications more concise. In the sequel we shall admit strings over $\mathcal{A}^*$ in place of single symbols. By convention, the terminal symbol `char` will denote an arbitrary symbol from $\mathcal{A}$, and the terminal symbol `string` will denote a string from $\mathcal{A}^*$.

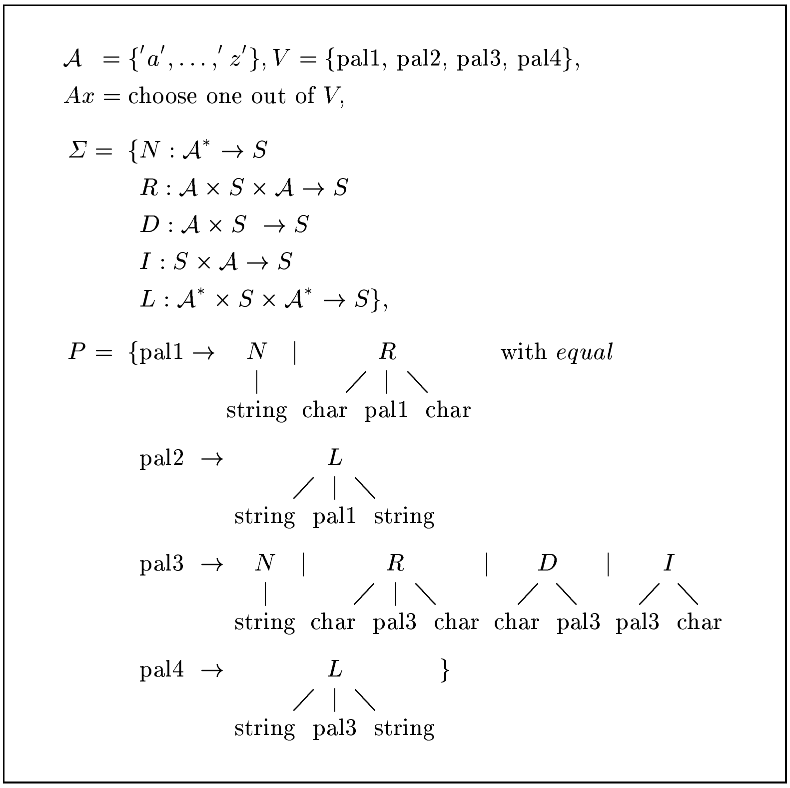
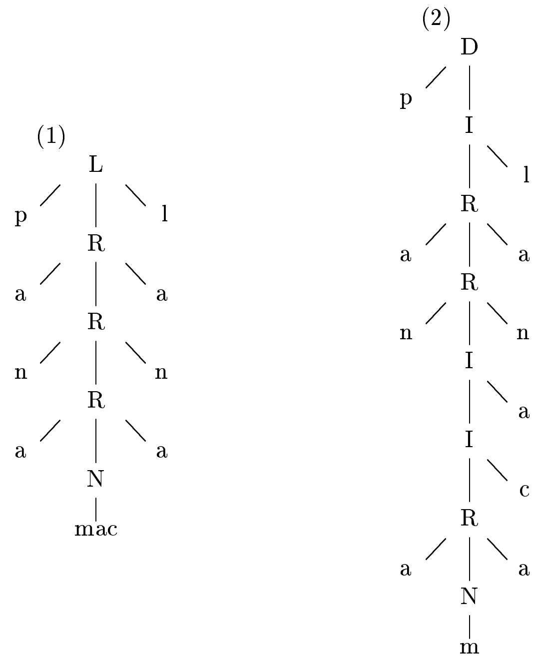


Fig. 1. Tree Grammars Pal_1 through Pal_4, depending on the choice of the axiom.

Our tree grammars will further be augmented with *conditional productions*. Their meaning is explained as a conditional form of derivation:

Definition 2 (Conditional productions.) A conditional production has the form $v \xrightarrow{c} t$ where c is a predicate defined on $\mathcal{A}^*$. A derivation using a conditional production, $v \xrightarrow{c} t \to^* t'$, where $t' \in T_\Sigma$, is well-formed only if $c(y(t'))$ holds. The language of a tree grammar with conditions is the set of trees that can be derived from the axiom by well-formed derivations. □

Note that the use of a conditional production $v \xrightarrow{c} t$ with c at some point in a derivation affects the complete derivation that continues from the subtree t inserted by this production. Only after the derivation is complete, the condition can be checked. A typical example is a condition that imposes a minimal or maximal length on $y(t')$.



(1) L "p" (R 'a' (R 'n' (R 'a' (N "mac") 'a') 'n') 'a') "l"
(2) D 'p' (I (R 'a' (R 'n' (I (I (R 'a' (N "m") 'a') 'c') 'a') 'n') 'a') 'l')

Fig. 2. A local separated palindrome derived from pal2 (1) and a global approximate palindrome derived from pal3 (2).

2.4 Examples: Palindromes and separated palindromes

Figure 1 shows four simple tree grammars for palindromic languages, depending on the choice of the axiom. They describe separated palindromes ($pal1$) of the form uvu^{-1}, and approximate separated palindromes ($pal3$) under the standard edit distance model of single character (R)eplacements, (D)eletions and (I)nsertions [11]. Operator N marks the middle part v of uvu^{-1}. Choosing axioms $pal2$ or $pal4$, we obtain local palindromes, embedded in an arbitrary context string. Note that this example makes use of a syntactic predicate $equal(a_1...a_n) = a_1 \equiv a_n$ and the lexical symbols `char`, `string` as described in Section 2.3. Note that $t \in \mathcal{L}(\mathrm{Pal}_i)$ explicitly describes the internal structure of a palindrome, while the palindromic string by itself is $y(t)$.

Figure 2 shows two examples for the input sequence `panamacanal`, each with term representation and the corresponding tree.

2.5 Yield languages and the yield parsing problem

We can now define the particular parsing problem we shall have to solve:

Definition 3 (Yield grammars and yield languages.) The pair $(\mathcal{G}, y)$ is called a yield grammar, and its yield language $\mathcal{L}(\mathcal{G}, y) = y(\mathcal{L}(\mathcal{G}))$. $\square$

Definition 4 (Yield parsing.) Given a yield grammar $(\mathcal{G}, y)$ over $\mathcal{A}$ and $w \in \mathcal{A}^*$, the yield parsing problem is to construct $P_{\mathcal{G}}(w) := \{t \in \mathcal{L}(\mathcal{G}) | y(t) = w\}$. $\square$

Compared to string languages described by context free grammars – such as programming languages – yield languages that arise in practise are often trivial and could be defined by much simpler means. For example, $\mathcal{L}(\mathrm{Pal}_i, y)$ is $\mathcal{A}^*$ for all four example grammars. Here, all our interest lies in determining the trees $P_{\mathcal{G}}(w)$, which represent various palindromic structures we associate with a given yield string w.
Note that the trees $P_{\mathcal{G}}(w)$ returned by a yield parser are not derivation trees according to $\mathcal{G}$, but terminal trees derivable by $\mathcal{G}$ with yield w.

2.6 Yield languages versus context free languages

For the record, we dwell a moment on aspects of formal language theory and show that yield languages are context free languages, and vice versa. This allows to carry over useful (un)decidability results (such as emptiness or ambiguity) from context free languages to yield languages.

Definition 5 (Flat grammar.) The flat grammar associated with the yield grammar $(\mathcal{G}, y)$, where $\mathcal{G} = (V, Ax, P)$ is the context free string grammar $y(\mathcal{G}) = (V, \mathcal{A}, Ax, \{v \to y(t) | v \to t \in P\})$. $\square$

Note that several distinct productions of the tree grammar may map to the same production in the flat grammar.
By construction, $\mathcal{L}(y(\mathcal{G})) = \mathcal{L}((\mathcal{G}, y))$ – for each derivation in $\mathcal{G}$ there is one in $y(\mathcal{G})$, and vice versa. So, yield languages are context free languages. The converse is also true. Each string grammar $\mathcal{G}'$ can be turned into a corresponding tree grammar by naming its productions, and using these names with suitable arities as operators of Σ. Each tree derived by this tree grammar is a syntax tree in $\mathcal{G}'$, labeled by explicit production names. We conclude:

Theorem 6 The class of yield languages is the class of context free languages. $\square$

2.7 Evaluation algebras

Definition 7 (Evaluation algebra.) Let Σ be a signature with sort symbol Ans. A Σ-evaluation algebra is a Σ-algebra augmented with an objective function $h : [Ans] \to [Ans]$, where $[Ans]$ denotes lists over Ans. $\square$

In most DP applications, the purpose of the objective function is minimizing or maximizing over all answers. We take a slightly more general view here. The objective may be to calculate a sample of answers, or all answers within a certain threshold of optimality. It could even be a complete enumeration of answers. We may compute the size of the search space or evaluate it in some statistical fashion, say by averaging over all answers, and so on. This is why in general, the objective function will return a list of answers. If maximization was the objective, this list would hold the maximum as its only element.

2.8 Algebraic dynamic programming and Bellman's principle

Given that yield parsing traverses the search space, all that is left to do is evaluate candidates in some algebra and apply the objective function.

Definition 8 (Algebraic dynamic programming.)

- An ADP problem is specified by a signature Σ over $\mathcal{A}$, a yield grammar $(\mathcal{G}, y)$ over Σ, and a Σ-evaluation algebra I with objective function h_I.
- An ADP problem instance is posed by a string $w \in \mathcal{A}^*$. The search space it spawns is the set of all its parses, $P_\mathcal{G}(w)$.
- Solving an ADP problem is computing

$$h_\mathcal{I}\{t_\mathcal{I} \mid t \in P_\mathcal{G}(w)\}.$$

□

There is one essential ingredient missing: efficiency. Since the size of the search space may be exponential in terms of input size, an ADP problem can be solved in polynomial time and space only under the condition known as Bellman's principle of optimality. In his own words:

> An optimal policy has the property that whatever the initial state and initial decision are, the remaining decisions must constitute an optimal policy with regard to the state resulting from the first decision. [2]

We can now formalize this principle:

Definition 9 (Algebraic version of Bellman's principle.) For each k-ary operator f in Σ, and all answer lists $z_1, \dots, z_k$, the objective function h satisfies

$$\begin{aligned} & h(\ [\ f(x_1, \dots, x_k) \mid x_1 \leftarrow z_1, \dots, x_k \leftarrow z_k\ ]\ ) \\ =\ & h(\ [\ f(x_1, \dots, x_k) \mid x_1 \leftarrow h(z_1), \dots, x_k \leftarrow h(z_k)\ ]\ ) \end{aligned}$$

Additionally, the same property holds for the concatenation of answer lists:

$$h(\ z_1 +\!\!+\ z_2\ ) = h(\ h(z_1) +\!\!+\ h(z_2)\ )$$

□

The practical meaning of the optimality principle is that we may push the application of the objective function inside the computation of subproblems, thus preventing combinatorial explosion. We shall annotate the tree grammar to indicate the cases where h is to be applied.
Compared to the classic description of DP algorithm via matrix recurrences, we have achieved the following:

– An ADP specification is *more abstract* than the traditional recurrences. Separation between search space construction and evaluation is perfect. Tree grammars and evaluation algebras can be combined in a modular way, and the relationships between problem variants can be explained clearly.
– The ADP specification is also *more complete*: DP algorithms in the literature often claim to be parametric with respect to the scoring function, while the initialisation equations are considered part of the search algorithm [4]. In ADP, it becomes clear that initialisation semantically is the evaluation of empty candidates, and is specified within the algebra.
– Our formalization of Bellman's principle is *more general* than commonly seen. Objectives like complete enumeration or statistical evaluation of the search space now fall under the framework. If maximization is the objective, our criterion implies Morin's formalization (strict monotonicity) [15] as a special case.
– The ADP specification is *more reliable*. The absence of subscripts excludes a large class of errors that are traditionally hard to find.

2.9 The ADP programming language

Yield grammars in ASCII notation For lack of space, we can only show the core of the ADP language. The declarative semantics of this language is simply that it allows to describe signatures, evaluation algebras and yield grammars. The signature Σ is written as an algebraic data type definition in Haskell. Alike EBNF, the productions of the yield grammar are written as equations. The operator `<<<` is used to denote the application of a tree constructor to its arguments, which are chained via the `~~~`-operator. Operator `|||` separates multiple righthand sides of a nonterminal symbol. The axiom symbol is indicated by the keyword `axiom`, and conditions are attached to productions via the keyword `with`. Finally, the application of the objective function `h` is indicated via the `...`-operator. Here is one of our example grammars in ADP language:

```
data Palindrome =
   N  (Int, Int)       |  R  Char Palindrome Char |
   D  Char Palindrome  |  I       Palindrome Char

grammar_Pal3 x  = axiom pal3   where
   pal3  =  N <<< string                     |||
            R <<< char  ~~~ pal3 ~~~ char    |||
            D <<< char  ~~~ pal3             |||
            I <<<           pal3 ~~~ char    ... h
```

The operational semantics of Section 4 turns this little program into a yield parser for Grammar Pal_3.

From yield parsing to algebraic dynamic programming The tree parser becomes a generic dynamic programming algorithm by replacing the signature Σ (`Palindrome` in our example) by an arbitrary Σ-evaluation algebra, which becomes a parameter to the grammar. Tabulation is controlled via the keywords `tabulated` and `p`.[1] `-~~` and `~~-` are variants of the `~~~`-operator, which are equivalent in their declarative semantics (cf. 4.3).

```
> import Array

> type Palindrome_Algebra answer =
>   ((Int,Int)                   -> answer,  -- evaluation function n
>    Char      -> answer -> Char -> answer,  -- evaluation function r
>    Char      -> answer         -> answer,  -- evaluation function d
>    answer    -> Char           -> answer,  -- evaluation function i
>    [answer]  -> [answer])                  -- objective  function h

> grammar_Pal3 alg x  = axiom (p pal3)   where
>   (n, r, d, i, h)   = alg
>   axiom ax   = ax (0,m)
>   (_, m)     = bounds x
>   tabulated  = table m
>   char (i,j) = [x!j | i+1 == j]

>   pal3 = tabulated(
>            n <<< string                     |||
>            r <<< char  -~~ p pal3 ~~- char   |||
>            d <<< char  -~~ p pal3            |||
>            i <<<           p pal3 ~~- char      ... h)
```

3 Tabulating yield parsers

3.1 The yield parsing “paradox“

Yield parsers, to be developed in this section, will serve as the computational engine for implementing ADP algorithms. Our claim is that ADP applies generally to dynamic programming over sequence data. Apparently, there is a contradiction:
On the one hand, there exist DP algorithms of various polynomial complexities, such as diverse $O(n^2)$ algorithms for sequence comparison [4], $O(n^3)$ algorithms for RNA structure prediction [21], or the $O(n^6)$-algorithm of Rivas and Eddy [16] for folding RNA pseudoknots.
On the other hand, we have learned in Section 2.6 that yield languages are context free languages. The trees to be delivered by a yield parser are built from the operators of the underlying signature, but otherwise isomorphic to syntax trees returned by a context free parser for the corresponding flat grammar. We

[1] The initial binding clauses serve the Haskell embedding and are explained below. All lines preceded by > jointly result in an executable Haskell program.

know that languages defined by ambiguous context free string grammars can be parsed in $O(n^3)$ by the CYK algorithm [1] or even slightly faster [18, 20]. Two questions come to mind:

1. Why should we introduce yield languages at all – could we not stick with string grammars and their established parsing methods?
2. Wouldn't this imply that we can solve all dynamic programming problems in the scope of ADP in $O(n^3)$? Or, formulated the other way, does this mean that the scope of ADP is limited to dynamic programming problems within the $O(n^3)$ complexity class?

Let us first note that Question 1 is correct in the sense that there is a simple and general tree grammar transformation that makes the yield parser run in $O(n^3)$. Each tree returned by the parser can be postprocessed in $O(n)$ to correspond to the original grammar. This makes Question 2 even more puzzling.
Well, ADP *is* a general method, not limited to $O(n^3)$ algorithms, so something essential must have been overlooked in Question 1. It is the fact that the yield parser's obligation is not just to determine the parse trees. It must construct them in a special way, in order to provide for amalgamation of recognition and evaluation phase (cf. Section 2.8). Remember that the trees constructed are formulas of T_Σ, to be interpreted in the evaluation algebra I. The parser must construct the trees such that, at any point, a (partial) tree t can be substituted by the (partial) answer value t_I. Only so, the choice function can be applied to partial answers and prevent the combinatorial explosion of their number. This prevents the use of grammar transformations that might speed up the parsing.

3.2 Adopting the CYK parsing algorithm

We shall adopt the so-called Cocke-Younger-Kasami (CYK) parsing technique for ambiguous context free grammars [1] to yield parsing. Two problems arise:

– Two tree productions like $v \to N(w), v \to M(w)$ both correspond to the same string production $v \to w$. In order not to loose results, the yield parser must be based directly on $\mathcal{G}$ rather than a string grammar.
– CYK relies on a transformation of the string grammar into Chomsky normal form. Such a transformation is not allowed in our case, as it would imply a change of the signature underlying the tree grammar.

Neither the explicit transformation to Chomsky normal form, nor the implicit transformation of the Graham-Harrison parser [10] can be used. We shall start with a nondeterministic, top-down parser that is subsequently transformed into a CYK-style parser implemented via dynamic programming.

3.3 Top-down nondeterministic yield parsers

For a given yield string w, the set $P_{\mathcal{G}}(w)$ of all its parses can be defined recursively using the productions of $\mathcal{G}$. For $v \in V$ and $a \in \mathcal{A}^*$ we define the sets of parses

$P_v(w)$ and $P_a(w)$, respectively. Without loss of generality we assume that all productions for a nonterminal v are defined via a single rule with alternative righthand sides, in the form $v \rightarrow q_1|...|q_r$. Possibly different conditions c_i can be associated with each alternative q_i, denoted q_i with c_i.

$$P_v(w) = P_{q_1}(w) \cup ... \cup P_{q_r}(w) \text{ for } \quad v \in V \tag{1}$$

For the tree patterns on the righthand sides, we define the parse sets $P_q(w)$ via structural recursion over pattern q:

$$P_a(w) \ = \ \text{if } w = a \text{ then } \{a\} \text{ else } \emptyset \text{ for } a \in \mathcal{A} \tag{2}$$

$$P_N(w) = \ \text{if } w = \epsilon \text{ then } \{N\} \text{ else } \emptyset \text{ for } N \in \Sigma \tag{3}$$

$$P_q(w) \ = \{N(x_1, ..., x_r)|x_i \in P_{q_i}(w_i) \text{ for } q = N(q_1, ..., q_r), \quad w = w_1...w_r\} \tag{4}$$

$$P_{q \texttt{ with } c}(w) \ = \ \text{if } c(w) \text{ then } P_q(w) \text{ else } \emptyset \tag{5}$$

Note that in Equation 4, w is split into r subwords in all possible ways.
Finally, we define

$$P_{\mathcal{G}}(w) = \widehat{P_{Ax}(w)} \tag{6}$$

where $\widehat{P_{Ax}(w)}$ is the P_{Ax}-component of the least fixpoint solution to the above equation system.
We now turn the equation system into a top-down, recursive yield parser. Subwords of a string $w = a_1...a_n$ are indicated by index pairs: $w_{(i,j)} = a_{i+1}...a_j$. The length of $w_{(i,j)}$ is $j - i$, and $w_{(i,k)}w_{(k,j)} = w_{(i,j)}$. When w is the string to be parsed, we often write (i,j) instead of $w_{(i,j)}$. Parsers are functions defined on yield strings, returning lists (rather than sets) of trees. A parser $p_v(w)$ computes $P_v(w)$, for each $v \in V$. A parser fails by returning an empty list. We shall use list comprehension notation, in analogy to set notation: $[f(x,y)|x \in xs, y \in ys, \phi(x,y)]$ denotes the list of all values $f(x,y)$ such that x is from the list xs, y from the list ys, and the argument pair (x,y) satisfies the predicate ϕ. [ ] denotes the empty list, ++ denotes list concatenation.

$$p_{\mathcal{G}}(w) \ = p_{Ax}(0,n) \tag{7}$$

$$p_v(i,j) \ = p_{q_1}(i,j)\texttt{++}...\texttt{++}p_{q_r}(i,j) \qquad \text{for } v \in V \tag{8}$$

$$p_a(i,j) \ = \text{if } w_{(i,j)} = a \text{ then } [a] \text{ else } [\,] \quad \text{for } a \in \mathcal{A} \tag{9}$$

$$p_N(i,j) = \text{if } w_{(i,j)} = \epsilon \text{ then } [N] \text{ else } [\,] \quad \text{for } N \in \Sigma \tag{10}$$

$$p_q(i,j) = [t(x_1, ..., x_r)|x_1 \in p_{q_1}(i,k_1), ..., x_r \in p_{q_r}(k_{r-1},j)] \quad \text{for } q = t(q_1, ..., q_r),\ t \in \Sigma,\ i \leq k_1... \leq k_{r-1} \leq j \tag{11}$$

$$p_q(i,j) = [t|t \in p_{q'}(i,j), c(i,j)] \qquad \text{for } q = q' \text{ with } c \tag{12}$$

The partial correctness of the yield parsers is obvious, as they are an operational form of the equations defining the parse sets $P_v(w)$. Termination, however, is a problem. The parser does not terminate if the grammar allows an infinite set of parses, resulting from circular productions chains that produce tree nodes, but an empty contribution to the yield (e.g. $v \to N(v)$). The parser may also run into futile recursion when $p_v(i,j)$ recalls itself on the same subword (i,j) in the production $v \to N(v,a), a \in \mathcal{A}$, rather than restricting its efforts to $(i, j-1)$. And even when termination occurs, the parser will surely enjoy the thrills of combinatorial explosion, because its top-down nature leads it to parsing certain subtrees an exponential number of times. Two more steps are needed to turn the yield parser into an effective and efficient program.

3.4 Tabulating yield parsers

We turn the nondeterministic, top-down parser into a bottom-up CYK-style parser in two steps: We add tabulation of parser results, and we provide the necessary bottom-up control structure.
Adding tabulation. Dynamic programming is recursion combined with tabulation of intermediate results. A DP table is nothing but a function mapping a (finite) index domain to some values. When f is a function of a pair of integers, let us denote by $f!$ a table such that $f!(i,j) = f(i,j)$.[2] Reconsider equations 7 - 12. Replace each occurrence of a parser p_v on either side of the equations by the corresponding table $p_v!$, according to the notation introduced above. E. g., a parser call $p_v(i,j)$ now turns into the table lookup $p_v!(i,j)$, and Equation 8 turns into a table definition $p_v!(i,j) = ...$. There is no need to tabulate parsers $p_{\mathcal{G}}$ and p_a, as they perform only a constant amount of computation per call. Parsers p_q need not be tabulated either, as they do not perform redundant work once all the p_v are tabulated. In pasting together the trees they construct, the parsers p_q use pointers to subtrees already stored in the tables $p_v!$, rather than copying subtrees.
Adding control structure. Unless we assume a data-flow oriented implementation language, we must organize the calculation of table entries in a way such that all entries are computed before they are used. Furthermore, the splitting of subwords in Equation 11 should be restricted to subwords of appropriate size.

Definition 10 (Yield size.) The yield size of a nonterminal symbol v of grammar $\mathcal{G}$ is the pair $(\inf_{x \in y(\mathcal{L}(v))} |x|, \sup_{x \in y(\mathcal{L}(v))} |x|)$ if $\mathcal{L}(v) \neq \emptyset$, and $(\infty, 0)$ otherwise. $\square$

Since all parser calls only access parser results on the same or a smaller subword of the input, all the recurrences as derived in the previous paragraph are arranged in a double for-loop, such that shorter subwords are parsed before longer ones. Parsers that do not need tabulation are defined outside these for-loops.
In the loop body, recurrences for all tabulated parsers are arranged in dependency order. The notation $p_{q_i}?(k_{i-1}, k_i)$ denotes $p_v!(k_{i-1}, k_i)$ if q_i is a variable v and hence p_v is tabulated, and otherwise it denotes $p_{q_i}(k_{i-1}, k_i)$. Equation 16 gives rise to inner loops, with subscript ranges determined by yield size

[2] This notation is borrowed from Haskell, where $(f!)$ actually turns an array into a function.

analysis. For $q \in T_\Sigma(V)$ this analysis computes $low'(q) = \inf_{t \in \mathcal{L}(q)} |y(t)|$ and $up'(q) = \sup_{t \in \mathcal{L}(q)} |y(t)|$. This is detailed in Section 5.2. Altogether, we obtain the following refined definition for the tabulating yield parser:

$$p_\mathcal{G}(w) \quad = p_{Ax}(0, n) \tag{13}$$
$$p_a(i,j) \quad = \text{if } w_{(i,j)} = a \text{ then } [a] \text{ else } [\,] \qquad \text{for } a \in \mathcal{A}^* \tag{14}$$
$$p_N(i,j) \quad = \text{if } w_{(i,j)} = \epsilon \text{ then } [N] \text{ else } [\,] \qquad \text{for } N \in \Sigma \tag{15}$$

$$p_q(i,j) \quad = [t(x_1, ..., x_r) | x_1 \in p_{q_1}?(i, k_1), ..., x_r \in p_{q_r}?(k_{r-1}, j)]$$
$$\text{for } q = t(q_1, ..., q_r), t \in \Sigma \tag{16}$$
$$\text{for } k_1, ..., k_{r-1} \quad \text{such that } \; k_0 = i, \; k_r = j,$$
$$max(k_{i-1} + low'(q_i))(k_{i+1} - up'(q_{i+1})) \leq k_i \leq$$
$$min(k_{i-1} + up'(q_i))(k_{i+1} - low'(q_{i+1}))$$
$$p_q(i,j) = [t | t \in p_{q'}(i,j), c(i,j)] \qquad \text{for } q = q' \; \text{ with } c \tag{17}$$
$$\text{for } j = 0 \text{ to } n$$
$$\text{for } i = 0 \text{ to } j$$
$$p_v!(i,j) = p_{q_1}(i,j){+}{+}...{+}{+}p_{q_r}(i,j) \qquad \text{for } v \in V \tag{18}$$

3.5 Asymptotic efficiency of tabulating yield parsers

The number of parses for $w \in \mathcal{A}^*$ can be exponential in $|w|$. An example is the grammar $v \to N(a,v) \mid M(a,v) \mid b$, where the yield string $a^n b$ has 2^n parses. In such a case, the size of the answer dominates the computational cost both in terms of time and space. In the subsequent analysis, we assume that the number of parses computed is bounded by a constant k. This is achieved by application of the objective function. Often only one or the k-best parses are retained, subject to some criterion of optimality.
The space requirements of the yield parser are determined by the table sizes. There are at most $|V|$ tables of size $(n+1)\times(n+1)$, yielding $O(n^2 * |V| * k)$ overall. For the runtime efficiency of a yield parser, the critical issue is the number of moving subword boundaries when w is split into subwords $w_1...w_r$ according to Equation 16. If for all i, $|w_i|$ is proportional to $|w|$, then this splitting introduces an $(r-1)$-fold nested for-loop in addition to the double for-loop iterating over subwords (i,j). Yield size analysis serves to avoid this worst case where possible.

Definition 11 (Width of productions and grammar.) Let t be a tree pattern, and let k be the number of nonterminal or lexical symbols in t whose yield size is not bounded from above. We define $width(t) = k - 1$. Let π be a production $v \to q_1 | \dots | q_r$. $width(\pi) = max\{width(q_1, \dots, q_r)\}$, and $width(\mathcal{G}) = max\{width(\pi) \mid \pi \text{ production in } \mathcal{G}\}$. $\Box$

Theorem 12 The execution time of a tabulating yield parser for tree grammar $\mathcal{G}$ on input w of length n is $O(n^{2+width(\mathcal{G})})$.

Proof: The outer for-loops lead to a minimal effort of $O(n^2)$. Inner for-loops generated from the righthand side patterns do not contribute to asymptotic efficiency if their index range is independent of n. The maximum nesting of for-loops with index range proportional to n arising from a given production is equal to the width of the production. Hence, the width of the grammar determines overall asymptotic efficiency as stated. □

Grammar transformations can greatly improve efficiency, but their feasibility depends on properties of the evaluation algebra. Two such transformations are described with the program development methods in [8].

4 Embedding ADP in Haskell

The ADP notation was designed such that its operational semantics can be defined adapting the technique of parser combinators [13].

4.1 Parser and combinator definitions

The input is an array x with bounds $(1,n)$. A subword $x_{i+1},...,x_j$ of x is represented by the subscript pair `(i,j)`. Functions `char` and `axiom` explicitly depending on input x and $n = \mathit{length}\ x$ are given for documentation here; they must actually be defined within the main function. A parser is a function that given a subword of the input, returns a list of all its parses. The lexical parser `char` recognizes any subword of length 1 and returns it, while `string` recognizes a possibly empty subword.

```
> type Parser b = (Int,Int) -> [b]

  char :: Parser Char
  char (i,j) = [x!j | i+1 == j]

> string :: Parser (Int,Int)
> string (i,j) = [(i,j) | i <= j]
```

The nonterminal symbols are interpreted as parsers, with the productions serving as their mutually recursive definitions. The operators introduced in ADP notation are defined as parser combinators: `|||` concatenates result lists of alternative parses, and `<<<` grabs the results of subsequent parsers connected via `~~~` and successively "pipes" them into the tree constructor. Combinator `...` applies the objective function to a list of answers.

```
> infixr 6 |||
> (|||)           :: Parser b -> Parser b -> Parser b
> (|||) r q (i,j) = r (i,j) ++ q (i,j)

> infix  8 <<<
> (<<<)           :: (b -> c) -> Parser b -> Parser c
> (<<<) f q (i,j) =  map f (q (i,j))
```

```
> infixl 7 ~~~
> (~~~)            :: Parser (b -> c) -> Parser b -> Parser c
> (~~~) r q (i,j) =  [f y | k <- [i..j], f <- r (i,k), y <- q (k,j)]

> infix  5 ...
> (...)             :: Parser a -> ([a] -> [a]) -> Parser a
> (...) p h (i,j)  = h (p (i,j))
```

The operational meaning of a `with`-clause can be defined by turning `with` into a combinator, this time combining a parser with a filter. Finally, the keyword `axiom` of the grammar is interpreted as a function that returns all parses for the axiom symbol `ax` and the complete input.

```
> type Filter    =  (Int, Int) -> Bool
> with          :: Parser a -> Filter -> Parser a
> with p c (i,j) =  if c (i,j) then p (i,j)  else []

  axiom           :: Parser a -> [a]
  axiom ax        =  ax (0,n)
```

Although these functions are called parsers, they do not necessarily deliver trees. The answers are solely computed via the functions of the evaluation algebra, whatever their type is.

4.2 Tabulation

As in Section 3, adding tabulation is merely a change of data type. The function `table` records the results of a parser `p` for all subwords of an input of size `n`. The function `p` is table lookup. Note the invariance `p (table n f) = f`.

```
> type Parsetable a = Array (Int,Int) [a]

> table     :: Int -> Parser a -> Parsetable a
> table n q =  array ((0,0),(n,n))
>                   [((i,j),q (i,j)) | i<- [0..n], j<- [i..n]]

> p         :: Parsetable a -> Parser a
> p t (i,j) =  if i <= j then t!(i,j) else []
```

4.3 Yield parser combinators for bounded yields

If the length of the yield of a nonterminal v is restricted to a fixed interval known from yield size analysis, the `~~`-combinator may be used to restrict the parsing effort to subwords of appropriate length range. Note the direct correspondence to the calculation of the loop bounds k_i defined in Equation 16.

```
> infixl 7 ~~,-~~ , ~~-
> (~~) :: (Int,Int) -> (Int,Int)
```

```
>       -> Parser (b -> c) -> Parser b -> Parser c
> (~~) (l,u) (l',u') r q (i,j)
>       = [x y | k <- [max (i+l) (j-u') .. min (i+u) (j-l')],
>                x <- r (i,k), y <- q (k,j)]

> (-~~) q r (i,j) = [x y | i<j, x <- q (i,i+1), y <- r (i+1,j)]
> (~~-) q r (i,j) = [x y | i<j, x <- q (i,j-1), y <- r (j-1,j)]
```

The combinators `-~~` and `~~-` are special cases of the `~~~` combinator in another way: they restrict the lefthand (respectively righthand) parser to a single character. The parser `pal3` in Section 2.9 avoids all uses of `~~~` and runs in $O(n^2)$.

5 Compiling and optimizing ADP algorithms

5.1 Experience with the Haskell implementation

For about two years, we have been using the Haskell embedding for reformulation and unification of classical DP algorithms, for teaching, and for development of new applications. While this has been a worthwhile effort intellectually, there are two serious shortcomings of this approach from the practical point of view.
The first concern is efficiency. Although the Haskell prototype has the same asymptotic efficiency as an imperative implementation, its space requirements prohibit application to large size biosequence data.
The second concern is a methodical one: In sophisticated examples, we strive for best runtime efficiency by using the special combinators for bounded yields wherever possible. This is sometimes nontrivial, and always error-prone. Based on this experience, it appears most beneficial to automate yield size and dependecy analysis.

5.2 Yield size analysis

Let $\mathbb{N}^\infty = \mathbb{N} \cup \{\infty\}$. The minimal and maximal yield sizes for all nonterminal symbols are described by a pair of functions $low, up : V \to \mathbb{N}^\infty$ and their extensions to arbitrary tree patterns $low', up' : T_\Sigma(V) \to \mathbb{N}^\infty$. They are computed using the following equation system: Let $v \to q_1|...|q_r$ be the production defining v.

$$(low(v),\ up(v)) \quad = (\min_{i=1}^{r} low'(q_i),\ \max_{i=1}^{r} up'(q_i)) \qquad \text{for } v \in V \tag{19}$$

$$(low'(w),\ up'(w)) = (|w|, |w|) \qquad \text{for } w \in \mathcal{A}^* \tag{20}$$

$$(low'(q),\ up'(q)) \quad = (\sum_{i=1}^{r} low'(q_i),\ \sum_{i=1}^{r} up'(q_i)) \qquad \text{for } q = t(q_1, ..., q_r),\ t \in \Sigma \tag{21}$$

$$(low'(q),\ up'(q)) \quad = (\max\ low'(q')\ c_l,\ \min\ up'(q')\ c_u) \quad \text{for } q = q' \text{ with } c \tag{22}$$

$$(low'(v),\ up'(v)) \quad = (low(v),\ up(v)) \qquad \text{for } v \in V \tag{23}$$

In Equation 22, the bounds c_l and c_u associated with a syntactic predicate c are defined as $c_l = min\{|x| \mid c(x)\}$ and $c_u = max\{|x| \mid c(x)\}$ if the maximum exists, and $c_u = \infty$ otherwise. In general, they cannot be determined automatically, but must be specified explicitly by the designer of the grammar.
These equations are monotonically decreasing in the first component, and monotonically increasing in the second. The solution can be computed by Kleene fixpoint iteration, starting with the initial value $(low(x), up(x)) = (\infty, 0)$. The low component always converges, since all strictly decreasing chains in $\mathbb{N}^\infty$ are finite. The strictly increasing chains of $up(x)$ are not necessarily finite. In the absence of syntactic conditions one can show that if $up(v)$ is still increasing after $|V|$ iterations, then $up(v) = \infty$ is the least fixpoint solution. The handling of conditions in full generality is an open problem. In any case, $(c_l, c_u) = (0, \infty)$ is a safe approximation.

5.3 Dependency analysis

The nested for-loops of the parser guarantee that when a word $w_{(i,j)}$ is to be parsed, all of its proper subwords have been parsed already. A problem arises with chain productions: In yield grammars, the analogue to chain productions $u \to v$ found in string grammars is the situation where $u \to C[...v...] \to^* C[...t...]$ such that $y(C[...t...]) = y(t)$. In other words, the tree context generated from u around v does not contribute to the yield. We denote this $u \to_{chain} v$. In this situation, a parser must reduce the input word $w_{(i,j)}$ to v before reducing it to u in the same iteration of the nested loop. The relation $\to_{chain}$ can be determined directly using the results from yield size analysis:

$$\text{Let } u \to q_1|...|q_r.$$

$$u \to_{chain} v \text{ iff } \bigvee_{i=1}^{r} d(q_i) \quad \text{where} \tag{24}$$

$$\begin{aligned}
d(q) &= \bigvee_{i=1}^{r'} \Big(\sum_{j=1}^{i-1} low'(q'_j) \equiv 0 \wedge d(q'_i) \wedge \sum_{j=i+1}^{r'} low'(q'_j) \equiv 0\Big) && \text{for } q = t(q'_1, ..., q'_{r'}),\ t \in \Sigma \\
d(q) &= d(q') && \text{for } q = q' \text{ with } c \\
d(q) &= v' \equiv v && \text{for } q = v' \in V \\
d(q) &= false && \text{for } q = w \in \mathcal{A}^*
\end{aligned}$$

low' is the above extension of low. The order of equations in the loop body (cf. Section 3.4) is chosen according to a topological sort with respect to $\to_{chain}$. Should $\to_{chain}$ be circular, a grammar design error is reported (see Section 6.2). The results of yield size and dependency analysis complete the definition of the tabulating yield parser.
Stepping back mentally from these technicalities for a moment, we observe the following: Anyone developing DP recurrences in the traditional way implicitly must solve the problems of yield size and dependency analysis, in order to define the control structure and the subscripts in the table accesses. Moreover, she must

do so without the guiding help of a tree grammar. This explains much of the technical difficulty of developing correct recurrences.

5.4 Translation to C

We are developing a compiler translating ADP algorithms to C. Aside from parsing the ADP program and producing C code, the core of the compiler is implementing the grammar analyses described in Section 5. With respect to the evaluation algebra we follow the strategy that simple arithmetic functions are inlined, while others must be provided as native C functions. Compiler options provide a simplified translation in the case where the evaluation algebra computes scalar answers rather than lists. As an example, the code produced for the grammar Pal_3 is shown in Appendix A.

5.5 Haskell source-to-source compilation

Yield size analysis determines the entire information required for minimal index ranges in all loops. We added a *source-to-source option* to the compiler, reproducing ADP input with all ~~~ operators replaced by variants bound to exact yield sizes. Hence, the user is no longer committed to delicate tuning efforts.

6 The ADP multi-paradigm programming system

6.1 Working with ADP

As a programming methodology, the ADP approach gives a clear five step guidance for developing a new algorithm [8]. Prior to the work reported here, the ADP program had to be translated into C by hand, following the definitions of Section 3. The C program was then tested systematically against the Haskell prototype, a procedure that guarantees much higher reliability than ad-hoc testing. This has been applied to non-trivial problems in RNA structure prediction [5], DNA sequence comparison [6] and gene prediction (ongoing work).
Still, the main difficulties with this approach were twofold: It proved to be time consuming to produce a C program equivalent to the Haskell prototype. Furthermore, for sake of efficiency developers were tempted to perform ad-hoc yield size analysis and used special combinators in the prototype. This introduced through the backdoor the possibility of subscript errors otherwise banned by the ADP approach. The compiler now developed eliminates both problems.

6.2 Grammar analysis support for improved prototyping

Grammar analysis further supports the prototyping phase by *reporting design errors* reflected by grammar anomalies. *Infinitely many derivations* of a given yield string are possible iff the relation $\rightarrow_{chain}$ is circular. This is detected during dependency analysis. An error message can be produced instead of a parser that may not terminate on some input. *Useless nonterminals*, which cannot produce a finite terminal tree and hence do not contribute to the language, are recognized by yield size analysis. u is useless, iff $low(u) = \infty$. They indicate oversights in the designer's case-analysis. No parser needs to be generated for nonterminal u, and the designer might appreciate a warning in this situation.

6.3 Future work

Our prevalent goal is to create a stable elementary ADP programming system that, thanks to the C compilation, can be utilized to speed up program development in large-scale applications. A number of advanced DP techniques have already experimented with, like attributed nonterminals and parsers which use precomputed information. The current formulation of ADP is directed towards string and applies to (single) string analysis and (pairwise) string comparison. Beyond strings, we have first results showing that ADP can be extended to trees [12], while other data domains have not yet been considered.

References

1. A.V. Aho and J.D. Ullman. *The Theory of Parsing, Translation and Compiling*. Prentice-Hall, Englewood Cliffs, NJ, 1973. I and II.
2. R. Bellman. *Dynamic Programming*. Princeton University Press, 1957.
3. W.S. Brainerd. Tree generating regular systems. *Information and Control*, 14:217–231, 1969.
4. R. Durbin, S. Eddy, A. Krogh, and G. Mitchison. *Biological Sequence Analysis*. Cambridge University Press, 1998.
5. D. Evers and R. Giegerich. Reducing the conformation space in RNA structure prediction. In *German Conference on Bioinformatics*, pages 118–124, 2001.
6. R. Giegerich. A systematic approach to dynamic programming in bioinformatics. *Bioinformatics*, 16:665–677, 2000.
7. R. Giegerich, S. Kurtz, and G. F. Weiller. An algebraic dynamic programming approach to the analysis of recombinant DNA sequences. In *Proc. of the First Workshop on Algorithmic Aspects of Advanced Programming Languages*, pages 77–88, 1999.
8. R. Giegerich and C. Meyer. Algebraic dynamic programming. In *Proc. of the 9th International Conference on Algebraic Methodology And Software Technology*, 2002. To appear.
9. R. Giegerich and K. Schmal. Code selection techniques: Pattern matching, tree parsing and inversion of derivors. In *Proc. European Symposium on Programming 1988*, Lecture Notes in Computer Science **300**, Springer Verlag, pages 247–268, 1988.
10. S.L. Graham and M.A. Harrison. An improved context-free recognizer. *ACM Transactions on Programming Languages and Systems*, 2(3):415–462, 1980.
11. D. Gusfield. *Algorithms on Strings, Trees, and Sequences*. Computer Science and Computational Biology. Cambridge University Press, 1997.
12. M. Höchsmann. Tree and Forest Alignments - An Algebraic Dynamic Programming Approach for Aligning Trees and Forests. Master's thesis, Bielefeld University, Mai 2001.

13. G. Hutton. Higher order functions for parsing. *Journal of Functional Programming*, 3(2):323–343, 1992.
14. C. Meyer and R. Giegerich. Matching and Significance Evaluation of Combined Sequence-Structure Motifs in RNA. *Z.Phys.Chem.*, 216:193–216, 2002.
15. T.L. Morin. Monotonicity and the principle of optimality. *Journal of Mathematical Analysis and Applications*, **86**:665–674, 1982.
16. E. Rivas and S. Eddy. A dynamic programming algorithm for RNA structure prediction including pseudoknots. *J. Mol. Biol.*, 285:2053–2068, 1999.
17. D.B. Searls. Linguistic approaches to biological sequences. *CABIOS*, 13(4):333–344, 1997.
18. K. Sikkel and M. Lankhorst. A parallel bottom-up tomita parser. In G. Görz, editor, *1. Konferenz Verarbeitung natürlicher Sprache (KONVENS'92), Nürnberg, Germany*, Informatik Aktuell, pages 238–247. Springer-Verlag, 1992.
19. P. Steffen. Basisfunktionen für die Übersetzung von Programmen der Algebraischen Dynamischen Programmierung. Master's thesis, Bielefeld University, February 2002. In German.
20. M. Tomita. *Efficient Parsing for Natural Language — A Fast Algorithm for Practical Systems*. Int. Series in Engineering and Computer Science. Kluwer, Hingham, MA, 1986.
21. M. Zuker and S. Sankoff. RNA secondary structures and their prediction. *Bull. Math. Biol.*, 46:591–621, 1984.

Appendix A: C-Code for Pal_3 Example

```
void calc_pal3(int i, int j) {
   struct t_result  *v[8];
   if ((j-i) >= 0) { v[0] = allocMem(0); }
     else { v[0] = NULL; };                          /*  n s       = 0 */
   if ((j-i) >= 2) { v[1] = allocMem(pal3[i+1][j-1]
                          + isEqual(x[i+1], x[j])); }
     else { v[1] = NULL; };             /*  r a s b  = s + isEqual(a,b) */
   if ((j-i) >= 1) { v[2] = allocMem(pal3[i+1][j]); }
     else { v[2] = NULL; };                          /*  d _ s    = s  */
   if ((j-i) >= 1) { v[3] = allocMem(pal3[i][j-1]); }
     else { v[3] = NULL; };                          /*  i   s _  = s  */

   v[4] = append(v[2], v[3]);                                   /* ||| */
   v[5] = append(v[1], v[4]);                                   /* ||| */
   v[6] = append(v[0], v[5]);                                   /* ||| */

   v[7] = maximum_v(v[6]);                 /*  h x      = [maximum x] */
   freemem_result(v[6]);
   pal3[i][j] = (*v[7]).value;
   freemem_result(v[7]);
};

void mainloop() {
   int i; int j;
   for (j=0; j<=n; j++)
      for (i=j; i>=0; i--)
         calc_pal3(i, j);
};
```