A Discipline of Dynamic Programming over Sequence Data

Robert Giegerich, Carsten Meyer, Peter Steffen

Faculty of Technology, Bielefeld University
Postfach 10 01 31
33501 Bielefeld, Germany
{robert,cmeyer,psteffen}@techfak.uni-bielefeld.de

Abstract. Dynamic programming is a classical programming technique, applicable in a wide variety of domains such as stochastic systems analysis, operations research, combinatorics of discrete structures, flow problems, parsing of ambiguous languages, and biosequence analysis. Little methodology has hitherto been available to guide the design of such algorithms. The matrix recurrences that typically describe a dynamic programming algorithm are difficult to construct, error-prone to implement, and, in nontrivial applications, almost impossible to debug completely. This article introduces a discipline designed to alleviate this problem. We describe an algebraic style of dynamic programming over sequence data. We define its formal framework, based on a combination of grammars and algebras, and including a formalization of Bellman's Principle. We suggest a language used for algorithm design on a convenient level of abstraction. We outline three ways of implementing this language, including an embedding in a lazy functional language. The workings of the new method are illustrated by a series of examples drawn from diverse areas of computer science.

1 Power and scope of dynamic programming

1.1 Dynamic programming: a method "by example"

Computer science knows a handful of programming methods that are useful across many domains of application. Such methods are, for example, Structural Recursion, Divide-and-Conquer, Greedy Algorithms and Genetic Algorithms. Dynamic Programming (DP) is another classical programming method, introduced even before the term Computer Science was firmly established. When applicable, DP often allows one to solve combinatorial optimization problems over a search space of exponential size in polynomial space and time. Bellman's "Principle of Optimality" [Bel57] belongs to the core knowledge of every computer science graduate. Significant work has gone into formally characterizing this principle [Sni92,Mor82,Mit64], formulating DP in different programming paradigms [Moo99,Cur97] and studying its relation to other general programming methods such as greedy algorithms [BM93].

The scope of DP is enormous. Much of the early work was done in the area of physical state transition systems and operations research [BD62]. Other, simpler examples (more suited for computer science textbooks) are optimal matrix chain multiplication, polygon triangulation, or string comparison. The analysis of molecular sequence data has fostered increased interest in DP. Protein homology search, RNA structure prediction, gene finding, and interpretation of mass spectrometry data pose combinatorial optimization problems unprecedented in variety and data volume. A recent textbook in biosequence analysis [DEKM98] lists 11 applications of DP in its introductory chapter, and many more in the sequel.

Developing a DP algorithm for an optimization problem over a nontrivial domain has intrinsic difficulties. The choice of objective function and search space are interdependent, and inextricably linked to questions of efficiency. Once completed, all DP algorithms are expressed via recurrence relations between tables holding intermediate results. These recurrences provide a very low level of abstraction, and subscript errors are a major nuisance even in published articles. The recurrences are difficult to explain, painful to implement, and almost impossible to debug: A subtle error gives rise to a suboptimal solution every now and then, which is virtually undetectable by human inspection.

In this situation it is remarkable that neither the literature cited above, nor many other computer science textbooks ([Gus97,Meh84,BB88,AHU83,Sed89], to name but a few) provide guidance in the development of DP algorithms. It appears that giving some examples and an informal discussion of Bellman's Principle is all the methodology we can offer to our students and practitioners. Notable exceptions are the textbooks by Cormen et al. and Schöning [CLR90,Sch01], which recognize this deficiency and formulate guiding rules on how to approach a new DP problem. We shall critically review these rules in our conclusion section. This state of the art is nicely summarized in a quote from an (anonymous) referee commenting on an initial version of this work, who wrote: "The development of successful dynamic programming recurrences is a matter of experience, talent, and luck."

1.2 Basic ideas of Algebraic Dynamic Programming

Algebraic dynamic programming (ADP) is a new style of dynamic programming that gives rise to a systematic approach to the development of DP algorithms. It allows one to design, reflect upon, tune and even test DP algorithms on a more abstract level than the recurrences that used to be all that was available to deal with dynamic programming algorithms. Four steps based on mathematical concepts guide the algorithm design. Many tricks that have been invented by practitioners of DP can be expressed as general techniques in ADP. The common aspects of related algorithms can be cleanly separated from their differences. On the implementation side, ADP exactly reproduces the classical DP recurrences. In principle[1], nothing is lost in terms of efficiency. All this together makes us

[1] Asymptotic efficiency is preserved, while the constant factors depend on the mode of implementation, discussed in Chapter 5.

feel that Dynamic Programming is more and more becoming a discipline, rather than a "matter of experience, talent and luck". How can this be achieved?

Any DP algorithm evaluates a search space of candidate solutions under a scoring scheme and an objective function. The classical DP recurrences reflect the four aspects of search space construction, scoring, choice, and efficiency in an indiscriminate fashion. In any systematic approach, these concerns must be separated. The algebraic approach to be presented here proceeds as follows:

The search space of the problem at hand is described by a *yield grammar*, which is a tree grammar generating a string language. The ADP developer takes the view that for a given input sequence, "first" the search space is constructed, leading to an enumeration of all candidate solutions. This is a parsing problem, solved by a standard device called a *tabulating yield parser*. The developer can concentrate on the design of the grammar.

Evaluation and choice are captured by an *evaluation algebra*. It is important (and in contrast to traditional presentations of DP algorithms) that this algebra comprises *all* aspects relevant to the intended objective of optimization, but is independent of the description of the search space. The ADP developer takes the view that a "second" phase evaluates the candidates enumerated by the first phase, and makes choices according to the optimality criterion.

Of course, the interleaving of search space construction and evaluation is essential to prevent combinatorial explosion. This interleaving is contributed by the ADP method in a way transparent to the developer.

By the separate description of search space and evaluation, ADP produces modular and therefore re-usable algorithm components. Often, related optimization problems over the same search space can be solved merely by a change of the algebra. More complex problems can be approached with a better chance of success, and there is no loss of efficiency compared to ad-hoc approaches. Avoiding the formulation of explicit recurrences is a major relief, an effect captured by early practitioners of ADP in the slogan "No subscripts, no errors!". We hope that the application examples presented in this article will convince the reader that following the guidance of ADP in fact brings about a boost in programming productivity and program reliability.

The ADP approach has emerged recently in the context of biosequence analysis, where new dynamic programming problems arise almost daily. In spite of its origin in this application domain, ADP is relevant to dynamic programming over sequential data in general. "Sequential data" does not mean we only study string problems – a chain of matrices to be multiplied, for example, is sequential input data in our sense, as well as the peak profiles provided by mass spectrometry. An informal introduction to ADP, written towards the needs of the bioinformatics community, has appeared in [Gie00a]. The present article gives a complete account of the foundations of the ADP method, and shows its application to several classical combinatorial optimization problems in computer science.

Like any methodological work, this article suffers from the dilemma that for the sake of exposition, the problems treated here have to be rather simple, such that the impression may arise that methodological guidance is not really required. The ADP method has been applied to several nontrivial problems in the field of biosequence analysis. An early application is a program for aligning recombinant DNA [GKW99], when the ADP theory was just about to emerge. Two recent applications are searching for sequence/structure motifs in DNA or RNA [MG02], and the problem of folding saturated RNA secondary structures, posed by Zuker and Sankoff in [ZS84] and solved in [EG01]. We shall give a short account of such “real world” applications.

1.3 Overview of this article

In Section 2 we shall review some new and some well known applications of dynamic programming over sequence data, in the form in which they are traditionally presented. This provides a common basis for the subsequent discussion. By the choice of examples, we illustrate the scope of dynamic programming to a certain extent. In particular, we show that (single) sequence analysis and (pairwise) sequence comparison are essentially the same kind of problem when viewed on a more abstract level. The applications studied here will later be reformulated in the spirit and notation of ADP.

In Section 3 we introduce the formal basis of the ADP method: Yield grammars and evaluation algebras. We shall argue that these two concepts precisely catch the essence of dynamic programming, at least when applied to sequence data. Furthermore, we introduce a special notation for expressing ADP algorithms. Using this notation an algorithm is completely described on a very abstract level, and can be designed and analyzed irrespective of how it is eventually implemented. We discuss efficiency analysis and point to other work concerning techniques to improve efficiency.

In Section 4 we formulate the ADP development method and develop yield grammars and evaluation algebras for the applications described in Section 2. Moreover we show how solutions to problem variants can be expressed transparently using the ADP approach.

Section 5 indicates three ways of actually implementing an algorithm once it is written in ADP notation: The first alternative is a direct embedding and execution in a functional programming language, the second is manual translation to the abstraction level of an imperative programming language. The third alternative, still under development, is the use of a system which directly compiles ADP notation into C code.

In the conclusion, we discuss the merits of the method presented here, evaluate its scope, and glance at its possible extensions.

This article may be read in several different ways. Readers familiar with standard examples of dynamic programming may jump right away to the theory in Section 3. Readers mainly interested in methodology, willing to take for granted that ADP can be implemented without loss of efficiency, may completely skip Section 5.

2 Dynamic programming in traditional style

In this section we discuss four introductory examples of dynamic programming, solved by recurrences in the traditional style. Three will be reformulated in algebraic style in Section 4. We begin our series of examples with an algorithmic fable.

2.1 The oldest DP problem in the world

Our first example dates back to the time at around 800. Al Chwarizmi, today known for his numerous important discoveries in elementary arithmetic and dubbed as the father of algorithmics, was a scholar at the House of Wisdom in Baghdad. At that time, the patron of the House of Wisdom was El Mamun, Calif of Baghdad and son of Harun al Raschid. It is told, that one day the Calif called for Al Chwarizmi for a special research project. He presented the formula $1 + 2 * 3 * 4 + 5$, which had its origins in a bill for a couple of camels, as he remarked. Unfortunately, the formula was lacking the parentheses. The task was to find a general method to redraw the parentheses in the formula (and any similar one) such that the outcome was either minimized or maximized – depending on whether the Calif was on the buying or on the selling side.

We now provide a DP solution for El Mamun's problem. Clearly, explicit parentheses add some internal structure to a sequence of numbers and operators. They tell us how subexpressions are grouped together – which are sums, and which are products. Let us number the positions in the text t representing the formula:

$$t = \ {}_0\, 1\, {}_1\ +\ {}_2\, 2\, {}_3\ *\ {}_4\, 3\, {}_5\ *\ {}_6\, 4\, {}_7\ +\ {}_8\, 5\, {}_9 \tag{1}$$

such that we can refer to substrings by index pairs: $t(0, 9)$ is the complete string t, and $t(2, 5)$ is $2*3$. A substring $t(i, j)$ that forms an expression can, in general, be evaluated in many different ways, and we shall record the best value for $t(i, j)$ in a table entry $T(i, j)$. Since addition and multiplication are strictly monotone functions on positive numbers, an overall value $(x + y)$ or $(x * y)$ can only be maximal if both subexpressions x and y are evaluated to their maximal values. So it is in fact sufficient to record the maximum in each entry. This is our first use of Bellman's Principle, to be formalized later.

More precisely, we define

$$T(i, i+1) = n, \text{if } t(i, i+1) = n \tag{2}$$
$$T(i, j) \quad = \max\{T(i, k) \otimes T(k+1, j) | i < k < j, t(k, k+1) = \otimes\} \tag{3}$$

where $\otimes$ is either $+$ or $*$. Beginning with the shortest subwords of t, we can compute successively all defined table entries.

In $T(0, 9)$ we obtain the maximal possible value overall. If, together with $T(i, j)$, we also record the position k within (i, j) that leads to the optimal value, then we can reconstruct the reading of the formula that yields the optimal value.

It is clear that El Mamun's minimization problem is solved by simply replacing *max* by *min*. Figure 1 gives the results for maximization and minimization of El Mamun's bill.

	0	1	2	3	4	5	6	7	8	9
0	/	(1,1)		(3,3)		(7,9)		(25,36)		(30,81)
1	/	/								
2	/	/	/	(2,2)		(6,6)		(24,24)		(29,54)
3	/	/	/	/						
4	/	/	/	/	/	(3,3)		(12,12)		(17,27)
5	/	/	/	/	/	/				
6	/	/	/	/	/	/	/	(4,4)		(9,9)
7	/	/	/	/	/	/	/	/		
8	/	/	/	/	/	/	/	/	/	(5,5)
9	/	/	/	/	/	/	/	/	/	/

Fig. 1. Results for the maximization and minimization of El Mamun's bill denoted as tuple (x, y) where x is the minimal value and y the maximal value.

Note that we have left open a few technical points: We have not provided explicit code to compute the table T, which is actually triangular, since i is always smaller than j. Such code has to deal with the fact that an entry remains undefined when $t(i, j)$ is not a syntactically valid expression, like $t(1, 4) =$ "+ 2 *". In fact, there are about as many undefined entries as there are defined ones, and we may call this a case of sparse dynamic programming and search for a more clever form of tabulation. Another open point is the possibility of malformed input, like the non-expression "1 + $*$ 2". The implementation shown later will take care of all these cases.

The first discovery Al Chwarizmi made, was that there were 14 different ways to evaluate the bill. In Section 4 we will see that the solution for this problem closely follows the recurrences just developed, except that there is no maximization or minimization involved. This is a combinatorial counting problem. Although DP is commonly associated with optimization problems, we will see that its scope is actually wider.

2.2 Matrix chain multiplication

A classical dynamic programming example is the matrix chain multiplication problem [CLR90]. Given a chain of matrices $A_1, ..., A_n$, find an optimal placement of parentheses for computing the product $A_1 * ... * A_n$. Since matrix multiplication is associative, the placement of parentheses does not affect the final value. However, a good choice can dramatically reduce the number of scalar multiplications needed. Consider three matrices A_1, A_2, A_3 with dimensions $10 \times 100, 100 \times 5$ and 5×50. Multiplication of $(A_1 * A_2) * A_3$ needs $10 * 100 * 5 + 10 * 5 * 50 = 7500$

scalar multiplications, in contrast to $10 * 100 * 50 + 100 * 5 * 50 = 75000$ when choosing $A_1 * (A_2 * A_3)$.

Let M be a $n \times n$ table. Table entry $M(i,j)$ shall hold the minimal number of multiplications needed to calculate $A_i * ... * A_j$. Compared to the previous example, the construction of the search space is considerably easier here since it does not depend on the structure of the input sequence but only on its length. $M(i,j) = 0$ for $i = j$. In any other case there exist $j - i$ possible splittings of the matrix chain $A_i, ..., A_j$ into two parts $(A_i, ..., A_k)$ and $(A_{k+1}, ..., A_j)$. Let (r_i, c_i) be the dimension of matrix A_i, where $c_i = r_{i+1}$ for $1 \leq i < n$. Multiplying the two partial product matrices requires $r_i c_k c_j$ operations. Again we observe Bellman's Principle. Only if the partial products have been arranged internally in an optimal fashion, can this product minimize scalar multiplications overall. We order table calculation by increasing subchain length, such that we can look up all the $M(i,k)$ and $M(k+1,j)$ when needed for computing $M(i,j)$. This leads to the following matrix recurrence:

```
for $j = 1$ to $n$ do                                                        (4)
  for $i = j$ to 1 do
    $M(i,j) = \begin{cases} 0 & \text{for } i = j \\ \min\{M(i,k) + M(k+1,j) + r_i c_k c_j \mid i \leq k < j\} & \text{for } i < j \end{cases}$
return $M(1,n)$                                                              (5)
```

Minimization over all possible splittings gives the optimal value for $M(i,j)$.

This example demonstrates that dynamic programming over sequence data is not necessarily limited to (character) strings, but can also be used with sequences of other types, in this case pairs of numeric values denoting matrix dimensions.

2.3 Global and local similarity of strings

We continue our series of examples by looking at the comparison of strings. The measurement of similarity or distance of two strings is an important operation applied in several fields, for example spelling correction, textual database retrieval, speech recognition, coding theory, or molecular biology.

A common formalization is the string edit model [Gus97]. We measure the similarity of two strings by scoring the different sequences of character deletions (denoted by D), character insertions (denoted by I) and character replacements (denoted by R) that transform one string into the other. If a character is unchanged, we formally model this as a replacement by itself. Thus, an edit operation is applied at each position.

Figure 2 shows some possibilities to transform the string `MISSISSIPPI` into the string `SASSAFRAS`, visualized as an alignment.

A similarity scoring function δ associates a similarity score of 0 with two empty strings, a positive score with two characters that are considered similar, a negative score with two characters that are considered dissimilar. Insertions and deletions also receive negative scores. For strings x of length m and y of length

```
MISSI--SSIPPI      MISSISSIPPI-      MISSI---SSIPPI
SASSAFRAS----      ---SASSAFRAS      SASSAFRAS-----

RR  RIIR DDDD      DDD R  RRRRI      RR  RIII DDDDD
```

Fig. 2. Three out of many possible ways to transform the string `MISSISSIPPI` into the string `SASSAFRAS`. Only deletions, insertions, and proper replacements are marked.

n, we compute the similarity matrix $E_{m,n}$ such that $E(i,j)$ holds the similarity score for the prefixes $x_1, \ldots, x_i$ and $y_1, \ldots, y_j$. $E(m,n)$ therefore holds the overall similarity value of x and y.

E is calculated by the following recurrences:

$$E(0,0) = 0 \tag{6}$$

$$\text{for } i = 0 \text{ to } m-1 \text{ do} \quad E(i+1,0) = E(i,0) + \delta(D(x_{i+1})) \tag{7}$$

$$\text{for } j = 0 \text{ to } n-1 \text{ do} \quad E(0,j+1) = E(0,j) + \delta(I(y_{j+1})) \tag{8}$$

$$\text{for } i = 0 \text{ to } m-1 \text{ do}$$

$$\quad \text{for } j = 0 \text{ to } n-1 \text{ do}$$

$$E(i+1,j+1) = max \left\{ \begin{array}{ll} E(i,j+1) & + \delta(D(x_{i+1})) \\ E(i+1,j) & + \delta(I(y_{j+1})) \\ E(i,j) & + \delta(R(x_{i+1}, y_{j+1})) \end{array} \right\} \tag{9}$$

$$\text{return } E(m,n) \tag{10}$$

The space and time efficiency of these recurrences is $O(mn)$.

Often, rather than computing the (global) similarity of two strings, it is necessary to search for local similarities within two strings. In molecular sequence analysis, we study DNA sequences, given as strings from four types of nucleotides, or protein sequences, given as strings from a twenty-letter alphabet of amino acids. In DNA, we often have long non-coding regions and small coding regions. If two coding regions are similar, this does not imply that the sequences have a large global similarity. If we investigate a protein with unknown function, we are interested in finding a 'similar' protein with known biological function. In this situation, the functionally important sequence parts must be similar while the rest is arbitrary.

Local similarity asks for the best match of a subword of x with a subword of y. The Smith and Waterman algorithm [SW81] requires $O(mn)$ time and space to solve this problem. We compute a matrix $C_{m,n}$ where the entry $C(i,j)$ contains the best score for all pairs of suffixes of $x_1 \ldots x_i$ and $y_1 \ldots y_j$.

$$C(i,j) = max\{score(x', y') | x' \text{ suffix of } x_1 \ldots x_i \text{ and } y' \text{ suffix of } y_1 \ldots y_j\} \tag{11}$$

Since we are looking for a local property, it must be possible to match arbitrary subwords of x and y without scoring their dissimilar prefixes and suffixes. These subwords are named x' and y' in (11), and their score is computed as in the global similarity case. In order to reject prefixes with negative scores, all we need to change in comparison to the recurrences for global similarity (see Equations $6 - 10$) is to fix the first line and column to zero-values and to add a zero-value case in the calculation of the entry $C(i+1, j+1)$. This zero amounts to an empty prefix pair joining the competition for the best match at each point (i, j).

```
for $i = 0$ to $m-1$ do   $C(i,0) = 0$                                  (12)
for $j = 0$ to $n-1$ do   $C(0,j) = 0$                                  (13)
for $i = 0$ to $m-1$ do
    for $j = 0$ to $n-1$ do
        $C(i+1,j+1) = max \left\{ \begin{array}{ll} 0 & \\ C(i,j+1) & + \delta(D(x_{i+1})) \\ C(i+1,j) & + \delta(I(y_{j+1})) \\ C(i,j) & + \delta(R(x_{i+1}, y_{j+1})) \end{array} \right\}$   (14)
return $max_{i,j}\ C(i,j)$                                              (15)
```

Equation 15 performs an extra traversal of table C to obtain the highest score overall.

2.4 Fibonacci numbers and the case of recursion versus tabulation

In this last introductory example, we make our first deviation from the traditional view of dynamic programming. There are many simple cases where the principles of DP are applied without explicit notice. Fibonacci numbers are a famous case in point. They are defined by

$$F(1) = 1 \tag{16}$$

$$F(2) = 1 \tag{17}$$

$$F(i+2) = F(i+1) + F(i) \tag{18}$$

Fibonacci numbers may seem atypical as a DP problem, because there is no optimization criterion. We have already remarked (cf. Section 2.1) that optimization is an important, but not a mandatory constituent of a dynamic programming problem.

Every student of computer science knows that computing F as a recursive function is very inefficient – it takes $2F(n) - 1$ calls to F to compute $F(n)$. Although we really need only the n values $F(1)$ through $F(n-1)$ when computing $F(n)$, each value $F(n-k)$ is calculated not once, but $F(k+1)$ times. The textbook remedy to this inefficiency is strengthening the recursion – define

$$F(n) = Fib(0, 1, n) \tag{19}$$

$$Fib(a, b, i) = \text{ if } (i = 1) \text{ then } b \text{ else } Fib(b, a + b, i - 1) \tag{20}$$

Here we shall consider another solution. This one requires no redefinition of F at all, just a change of data type: Consider F as an integer array, whose elements are defined via Equations 16 – 18. In a data driven programming language, its elements will be calculated once when first needed. In an imperative language, since F is now data rather than a function, we need to add explicit control structure – an upper bound for n and a for-loop to actually calculate the array elements in appropriate order.

The lesson here is the observation that a table (matrix, array) over some index domain, defined by recurrences, is mathematically equivalent to a recursive function defined by the very same recurrences. The difference lies solely in the efficiency of the actual computation. What requires exponential effort in the case of recursive functions may require only polynomial effort when using tabulation. This gives us a first hint at a more systematic development of DP algorithms: Think of a DP algorithm as a family of recursive functions over some index domain. Don't worry about tabulation and evaluation order, this can always be added when the design has stabilized.

2.5 Summary of Section 2

In all the examples studied, there was first some informal reasoning, including a consideration of Bellman's Principle, and subsequently we wrote down the recurrences. (In the Fibonacci example, only a single value is generated for each subproblem. Hence, the objective function degenerates to the identity, and Bellman's Principle is trivially satisfied.) While these examples are simple enough to be solved this way, for more ambitious tasks it would be nice to have a more abstract level that supports formal reasoning about the problem. For example, none of our examples provides a description of the search space, and asking for its size or the (non-)redundancy of its traversal requires to develop another set of – similar, but not identical – recurrences.

Typically, the scoring scheme is meant to be a parameter of the algorithm, but this separation is not perfect. This applies to the score function δ used in the string comparison example. Consider the zero case added in the Smith-Waterman recurrences: It appears to be a part of the algorithm, but we shall see later, that it actually belongs to the scoring scheme. The same holds for the fact that scores from individual edit operations are added up rather than combined in some other fashion.

In the sequel, we shall seek a perfect separation of concerns, which will make our programs easier to understand, and usable in a modular fashion.

3 Foundations of Algebraic Dynamic Programming

ADP is based on the notions of yield grammars, evaluation algebras, and a formalization of Bellman's Principle. They are introduced in this section, while the program development method based on them is formulated at the beginning of Section 4.

3.1 Basic terminology

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

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 $o_\mathcal{I}$ for each operator o. Each $o_\mathcal{I}$ has type $o_\mathcal{I} : (s_1)_\mathcal{I}...(s_{k_o})_\mathcal{I} \to S_\mathcal{I}$ where $\mathcal{A}_\mathcal{I} = \mathcal{A}$.

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 $V \subset T_\Sigma(V)$. By convention, operator names are capitalized in the term algebra.

Tree grammars. Terms will be viewed as rooted, ordered, node-labeled trees in the obvious way. All inner nodes carry (non-nullary) operators from Σ, while leaf nodes carry nullary operators or 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 $C[...t...]$.

A tree language over Σ is a subset of T_Σ. Tree languages are described by tree grammars, which can be defined by analogy to the Chomsky hierarchy of string grammars. Here we use regular tree grammars, originally studied in [Bra69]. In [GS88] they were redefined to specify term languages over some signature. This is the form of tree grammars we use here; our further specialization in Definition 1 lies solely in the distinguished role of $\mathcal{A}$.

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

- a set V of nonterminal symbols,
- a designated nonterminal symbol Z, called the axiom, and
- 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 $\mathcal{L}(\mathcal{G}) = \mathcal{L}(Z)$.□

Figure 3 shows a tree grammar for the global similarity example of Section 2.3.

For convenience, we add a lexical level to the grammar concept, allowing strings from $\mathcal{A}^*$ in place of single symbols. By convention, `achar` denotes an arbitrary character, `char c` a specific character `c`, `string` an arbitrary string and `empty` the empty string.

Also for brevity, we allow syntactic conditions associated with righthand sides.

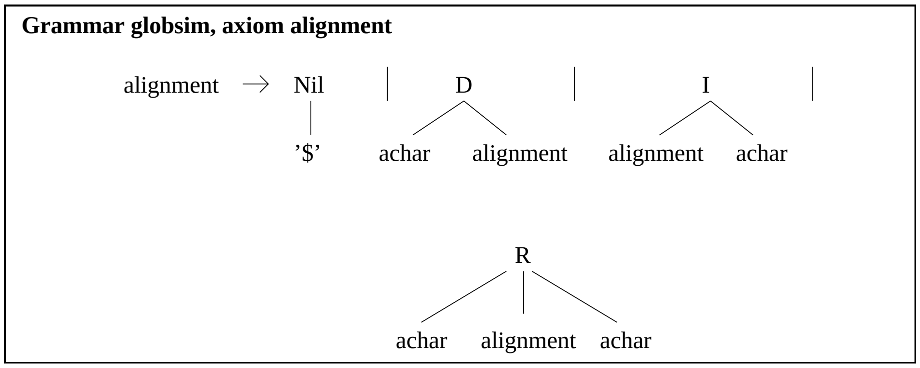

Fig. 3. The tree grammar `globsim` for global similarity (see Section 2.3)

The yield of a tree is normally defined as its sequence of leaf symbols. Here we are only interested in the symbols from $\mathcal{A}^*$; nullary constructors by definition have yield ε. A formal definition follows in Section 3.4.

Figure 4 shows a tree derived by grammar `globsim`, together with its yield string.

```
candidate1 = D('d',R('a',I(R('r',R('l',R('i',R('n',R('g',Nil('$'),
'e'),'n'),'i'),'l'),'r'),'i'),'a'))
yield(candidate1) = "darling$enilria"
```

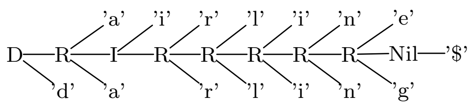

Fig. 4. The term representation of a global similarity candidate `candidate1` for `darling` and `airline`, and the tree representation of this term. Only the rightmost `R` is a proper replacement, the others are matches.

3.2 Conceptual separation of recognition and evaluation

Any dynamic programming algorithm implicitly constructs a search space from its input. The elements of this search space have been given different names: *policy* in [Bel57], *solution* in [Mor82], *subject under evaluation* in [Gie00a]. Since the former two terms have been used ambiguously, and the latter is rather technical, we shall use the term *candidate* for elements of the search space. Each candidate will be evaluated, yielding a *final state*, a *cost*, or a *score*, depending whether one follows [Bel57], [Mor82] or [DEKM98]. We shall use the term *answer* for the result of evaluating a candidate.

Typically, there is an ordering defined on the answer data type. The DP algorithm returns a maximal or minimal answer, and if so desired, also one or all the candidates that evaluate(s) to this answer. Often, the optimal answer is determined first, and a candidate that led to it is reconstructed by backtracing. The candidates themselves do not have an explicit representation during the DP computation. Our goal to separate recognition and evaluation requires an explicit representation of candidates.

This is our general idea of how to obtain such an explicit representation of candidates in any DP algorithm: Imagine that during computing an answer, we did not actually call those functions that perform the evaluation. Instead, we would apply them symbolically, building up a formula that – once evaluated – would yield this answer value. This formula itself is a perfect choice for the candidate representation, because

- the formula represents everything we need to know about the candidate to eventually evaluate it,
- the complete ensemble of such formulas, constructed for a specific problem instance, is a precise description of the search space.

Apparently, the above idea works for any DP algorithm over any data domain. After all, whenever we compute an answer value, we can as well compute it symbolically and record the formula. The subsequent treatment, however, only considers the case of sequential input. Our running example of global string similarity actually requires two string inputs. Figure 4 shows a global similarity candidate for the strings `"darling"` and `"airline"`. To represent them as a single string, the latter is reversed and appended to the former, separated by a separator symbol \$ not occurring elsewhere[2].

To design a DP algorithm, we therefore need to specify three aspects: the language of candidate formulas, the search space of candidates spawned by a particular input, and their eventual evaluation. We start with the latter aspect.

3.3 Evaluation algebras

Definition 2 (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. $\Box$

[2] In Section 5.1 the relation between single sequence analysis and pairwise sequence comparison is discussed.

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. Aside from minimization or maximization, 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. This is why in general, the objective function will return a list of answers. Each answer is the evaluation of a candidate; if the enumeration of candidates is our objective, it is the candidate itself. If maximization is our objective, this list holds the maximal answer as its only element. If the search space should be empty for a particular input, this list will be empty.

We formulate a signature Π for the global similarity example:

$$\begin{array}{lllll}
Nil: & \mathcal{A} & & & \rightarrow Ans \\
D: & \mathcal{A} & \times Ans & & \rightarrow Ans \\
I: & Ans & \times \; \mathcal{A} & & \rightarrow Ans \\
R: & \mathcal{A} & \times Ans & \times \mathcal{A} & \rightarrow Ans \\
h: & [Ans] & & & \rightarrow [Ans]
\end{array}$$

We formulate two evaluation algebras for signature Π. The algebra `unit` (Figure 5 right) scores each matching character by $+1$, and each character mismatch, deletion or insertion by -1. The algebra `wgap` (Figure 5 left) is a minor generalization of `unit`. It uses two parameter functions `w` and `gap`, that may score (mis)matches and deletions or insertions depending on the concrete characters involved. For example, a phoneticist would choose `w('v','b')` as a (positive) similarity rather than a (negative) mismatch.

Ans_{wgap} = $\mathbb{N}$

```
wgap = (nil,d,i,r,h)
 where
nil(a)   = 0
d(x,s)   = s + gap(x)
i(s,y)   = s + gap(y)
r(a,s,b) = s + w(a,b)
h([])    = []
h (l)    = [maximum(l)]
```

Ans_{unit} = $\mathbb{N}$

```
unit = (nil,d,i,r,h)
 where
nil(a)   = 0
d(x,s)   = s - 1
i(s,y)   = s - 1
r(a,s,b) = if a==b then s + 1 else s - 1
h([])    = []
h (l)    = [maximum(l)]
```

Fig. 5. Algebras `wgap` (left) and `unit` (right)

For term `candidate1` of Figure 4, we obtain:

$\texttt{candidate1}_{unit}$ = 2

$\texttt{candidate1}_{wgap}$ = `gap('d') + w('a','a') + gap('i')+ w('r','r') + w('l','l') + w('i','i')+ w('n','n') + w('e','g') + 0`

3.4 Yield grammars

To describe the search space, we have to solve two problems: Not all elements of T_Σ are legal candidates in general, and for a given input, only the legal candidates are to be considered which have this input as their yield. In our formalism, the yield function y has type $T_\Sigma \to \mathcal{A}^*$ and is defined by $y(a) = a$ for $a \in \mathcal{A}$, and $y(C(x_1, ..., x_n)) = y(x_1)...y(x_n)$ for $n \geq 0$ and each n-ary operator C of Σ.

Definition 3 (Yield grammars and yield languages.) Let $\mathcal{G}$ be a tree grammar over Σ and $\mathcal{A}$, and y the yield function. The pair $(\mathcal{G}, y)$ is called a yield grammar. It defines the yield language $\mathcal{L}(\mathcal{G}, y) = y(\mathcal{L}(\mathcal{G}))$. □

The grammar of Figure 3 can now be interpreted in two ways: as a regular tree grammar defining the tree language $\mathcal{L}(\mathcal{G})$, or as a yield grammar, defining the string language $y(\mathcal{L}(\mathcal{G}))$.

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

Yield parsing is the computational engine underlying ADP. The search space spawned by input w is $P_\mathcal{G}(w)$. A yield parser constructs the terms $t \in T_\Sigma$ in a bottom up fashion, identically to the way in which an interpretation $\mathcal{I}$ evaluates the term $t_\mathcal{I}$. Thus, if we substitute each term constructor function f in the yield parser by the function $f_\mathcal{I}$ of the evaluation algebra, the "parser" will produce answers of the desired kind rather than terms representing candidates.

3.5 Algebraic dynamic programming and Bellman's Principle

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

Definition 5 (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 $\mathcal{I}$ with objective function $h_\mathcal{I}$.
- 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

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

in polynomial time and space with respect to $|w|$.

□

So far, there is one essential ingredient missing: efficiency. Since the size of the search space may be exponential in terms of the input size, an ADP problem can be solved in polynomial time and space by the yield parser only under the condition well known as Bellman's Principle of Optimality. In Bellman's 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." [Bel57]

We formalize this principle:

Definition 6 (Algebraic version of Bellman's Principle.) An evaluation algebra satisfies Bellman's Principle, if for each k-ary operator f in Σ and all answer lists $z_1, \ldots, z_k$, the objective function h satisfies

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

□

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 have formulated Bellman's Principle such that it is correct to reduce any intermediate answer list, computed by any function f, by an application of the choice function h.

In practice, we can allow ourselves to cheat a little bit: We may omit the application of h for answers computed by certain "bad" functions f not satisfying the principle. This means we have to deal with longer intermediate answer lists, until a "good" function is applied to them. In practice, we shall annotate the tree grammar to indicate the cases where h is to be applied.

The above formulation of Bellman's Principle assumes that *all* answers from different alternatives of a production are computed before the choice function is applied to the answer list. This is normally sufficient, and we know of only one exception, where a choice from the answers of alternatives 1 and 2 is required before alternative 3 is considered. This exception, however, is quite an interesting one - a technique that enables the production of an arbitrary number of answers in order of optimality, by a merge-sorting of the intermediate answer lists [GM02]. In such a case, Bellman's Principle must be strengthened by another requirement:

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

Comparing our version of Bellman's Principle to others given in the literature, we find that it is quite general, because nothing is assumed about the choice function except that it maps lists to lists. The standard case, considered by Bellman and in most programming textbooks, arises when h is optimization with respect to some total ordering; in this case our condition requires monotonicity of each f. In the words of [BM97], f distributes over the ordering relation. Morin elaborates the case when h is to return *all* smallest (or largest) answers [Mor82]. In this case, our condition implies strict monotonicity, consistent with Morin's analysis. Our condition also applies in settings where no optimization is involved. In the *counting algebras* introduced in the next section, h is list summation while f is $*$, and the counting algebras are correct because $*$ distributes over $+$. A similar situation arises with the probabilistic algebras used for significance evaluation of structural motifs in RNA [MG02].

In their relational "Algebra of Programming" [BM97], Bird and de Moor cast optimization problems in the form $\min R \cdot \Lambda(([S]) \cdot ([Q])^{\circ})$. Here, $([Q])^{\circ}$ is the inverse of an homomorphism from some intermediate data type to the input domain. $([S])$ is evaluation. Λ is needed to convert relations to set-valued functions, and finally, $\min R$ denotes minimization with respect to relation R. Thus, our concepts T_{Σ}, $\mathcal{I}$ and h are explicitly present in the relational approach as Q, S and R, while $\mathcal{G}$ remains implicit. The need for $\mathcal{G}$ is not felt in [BM97], since the grammars implicit in the examples studied there only use a single nonterminal symbol, and structural recursion over T_{Σ} is sufficient in such a case. We shall later see examples where all the craftsmanship of DP lies in the design of nontrivial grammars (Section 4.5).

The relational formalism is more general than ours, as programs can be relations and the input domain is not restricted to sequential data. Another, minor difference is that our choice function h is not restricted to optimization – this is motivated by the development methodology we shall propose, and its usefulness will emerge in Section 4. Aside from this aspect, the ADP approach can be seen as a specialization of the relational formalism, which operates on an intermediate level of abstraction, giving us more control over the actual implementation while retaining conceptual clarity.

This completes the basic theoretical framework of ADP. We can now practice ADP with pencil and paper – but we want more.

3.6 ADP notation

For practical algorithm development in the ADP framework, we need a computer readable notation, which will be introduced in this section. The declarative semantics of ADP notation is that it allows one to describe signatures, evaluation algebras and yield grammars. The operational semantics of ADP notation is that of DP recurrences – ADP notation actually constitutes executable code. How this is achieved is an aspect completely deferred to Section 5.

The signature Σ is written as an algebraic data type definition in the style of the functional programming language Haskell [PJ03]. This is signature Π in ADP notation:

```
data Alignment = Nil Char           |
                 D  Char Alignment |
                 I  Alignment Char |
                R  Char Alignment Char
```

As in 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. Parentheses are used as required for larger trees. (Later, this notation will also be embedded in Haskell (Section 5); for the moment, just take it as our ASCII notation for yield grammars.) The axiom symbol is indicated by the keyword `axiom`, and syntactic conditions may be attached to productions via the keyword `with`. Finally, the evaluation algebra is a tuple of functions, provided as a single parameter `alg` with the grammar, which is split up into its components by a pattern matching clause.

This is the grammar `globsim` in ADP notation:

```
globsim alg = axiom alignment where
  (nil, d, i, r, h) = alg

  alignment  = nil <<< char '$'                          |||
               d   <<< achar  ~~~ alignment              |||
               i   <<<            alignment ~~~ achar    |||
               r   <<< achar  ~~~ alignment ~~~ achar
```

3.7 Parsing, tabulation and choice

Given a yield grammar and an evaluation algebra, a tabulating yield parser will solve a problem instance as declared in Definition 5. Implementation of yield parsing is explained in detail in Section 5.2. For programming with ADP, we do not really have to know how yield parsing works. Think of it as a family of recursive functions, one for each nonterminal of the grammar. However, the yield parser needs two pieces of information not yet expressed in the grammar: tabulation and choice.

If nothing was said about tabulation, the yield parser would compute partial results many times, quite like our original Fibonacci function. By adding the keyword `"tabulated"`, we indicate that the parser for a particular nonterminal symbol shall make use of tabulation. When a tabulated symbol `v` is used in a righthand side, this means a table lookup rather than a recursive call.

In the small examples we have shown so far, there is only a single nonterminal symbol, and naturally, it is tabulated. In larger grammars, typically only a subset of all nonterminal symbols needs to be tabulated. This is why we provide an explicit keyword to indicate tabulation. It is in fact a major advantage of ADP that we can write the grammar first and afterwards decide about tabulation. One might even think of automating the task of table assignment, but trying to optimize both time and space requirements makes this a difficult problem.

If nothing was said about choice, the parser would not apply the objective function and hence return a list of all answers. By adding `"... h"` to the righthand side of a production, we indicate that whenever a list of alternative answers has been constructed according to this production, the objective function `h` is to be applied to it.

With these two kinds of annotation, our yield grammar example `globsim` looks like this:

```
globsim alg = axiom alignment where
  (nil, d, i, r, h) = alg

  alignment  = tabulated(
              nil <<< char '$'                          |||
              d   <<< achar  ~~~ alignment              |||
              i   <<<            alignment ~~~ achar    |||
              r   <<< achar  ~~~ alignment ~~~ achar    ... h)
```

3.8 Efficiency analysis of ADP programs

From the viewpoint of programming methodology, it is important that asymptotic efficiency can be analyzed and controlled on the abstract level. This property is a major virtue of ADP – it allows one to formulate efficiency tuning as grammar and algebra transformations. Such techniques are described in [GM02]. Here we give only the definition and the theorem essential for determining the efficiency of an ADP algorithm.

Definition 7 (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 by a constant. We define $width(t) = k - 1$. Let π be a production $v \rightarrow t_1|\dots|t_r$. We define $width(\pi) = max\{width(t_1, \dots, t_r)\}$, and $width(\mathcal{G}) = max\{width(\pi) \mid \pi$ is a production in $\mathcal{G}\}$. □

Theorem 8 Assuming the number of answers selected by each application of h is bounded by a constant, the execution time of an ADP algorithm described by yield grammar $(\mathcal{G}, y)$ on input w of length n is $O(n^{2+width(\mathcal{G})})$.

Proof: See [GS02] □

In a standard application using minimization or maximization, the objective function h always reduces an answer list to a single (minimal or maximal) element. Asking for the k best answers, where k is a constant, does not affect asymptotic efficiency. However, in some applications one might ask for all answers, or for all answers within a certain threshold of optimality. In such a case, the algorithm will become output sensitive, as even a fraction of all answers may have a size exponential in n.

Considering the efficiency of `globsim`, we find that all productions have width 0. Hence, Theorem 8 says that the global similarity problem is solved in $O(n^2)$ space and time, consistent with the explicit recurrences given in Equations 6 - 10 (Section 2.3).

3.9 A summary of the ADP framework

By means of an evaluation algebra and a yield grammar we can completely specify a dynamic programming algorithm. We can analyze its efficiency using Theorem 8. We suggest a specific notation for writing grammars and algebras, which will be converted into executable yield parsers in Section 5.

This completes our framework. Let us summarize the key ideas of Algebraic Dynamic Programming:

Phase separation: We conceptually distinguish recognition and evaluation phases.
Term representation: Individual candidates are represented as elements of a term algebra T_Σ; the set of all candidates is described by a tree grammar.
Recognition: The recognition phase constructs the set of candidates arising from a given input string, using a tabulating yield parser.
Evaluation: The evaluation phase interprets these candidates in a concrete Σ-algebra, and applies the objective function to the resulting answers.
Phase amalgamation: To retain efficiency, both phases are amalgamated in a fashion transparent to the programmer. The term algebra in the yield parser is substituted by an evaluation algebra.

When running the algorithm, the candidate representation effectively cancels out. This is why it is not seen in traditional DP recurrences. For algorithm design, however, it plays a crucial role. It allows one to resolve the conglomeration of issues criticised initially. While the traditional recurrences deal with search space construction, evaluation and efficiency concerns in a non-separable way, ADP has separated them: Evaluation is in the algebra, the search space is in the grammar, and efficiency concerns are treated by the grammar annotation.

3.10 Yield grammars versus context free grammars

Before proceeding, let us comment on the most frequent question asked by computer scientists when first confronted with the ADP approach. It is easy to show that yield languages are simply context free languages (cf. [GS02]). Could not the same be achieved by sticking with the familiar context free grammars and their parse trees? The answer is no. Yield grammars are a two level concept, first generating a tree by the tree grammar and then deriving its yield. Shortcutting the two-level scheme by a context free grammar eliminates the generated tree – the candidate – which is our cornerstone for the separation of the search space and its evaluation.

We discuss three situations where the difference between using CFGs or yield grammars matters. The first case is a matter of convenience, the second a matter of efficiency, and the third a matter of ambiguity control.

Imagine we are using CFGs and an Earley or CYK-type parser that calls a semantic routine with each production. Consider three tree grammar productions:

```
a = f <<< b ~~~ (g <<< c ~~~ d)
b = f <<< b ~~~ c      |||
    g <<< b ~~~ c
```

The first production uses the fact that we can write trees of any size on the righthand side of a tree grammar. Slightly less convenient, in a CFG we would introduce a extra nonterminal and split the production in two:

$$a \to b\ a' \text{ (call } f\text{)}$$
$$a' \to c\ d \text{ (call } g\text{)}$$

For nonterminal b, we obtain two identical string productions,

$$b \to b\ c \quad | \quad b\ c$$

The standard definition of CFGs speaks of a *set* of productions, which cannot contain multiples. A solution would be to allow labelled productions, one labelled f and one labelled g to make them distinct – the signature returns through the back door.

Next, we turn to efficiency concerns and are confronted with a phenomenon called the *yield parsing paradox* [GS02]: Parsing of ambiguous CFGs can be done in $O(n^3)$ time. On the other hand, there are DP algorithms that require $O(n^4)$ or higher, described by yield grammars of width ≥ 2.

A context free production can be brought into Chomsky normal form, thereby reducing its width to 1. For example,

$$a \to b\ c\ d$$

is transformed to

$$a \to b\ a'$$
$$a' \to c\ d$$

Why can we not apply this transformation to the yield grammar production

```
a  = f  <<< b ~~~ c ~~~ d ... h
```

by writing

```
a  = f1 <<< b ~~~ a' ... h
a' = f2 <<< c ~~~ d  ... h'  ?
```

There are always functions $f1, f2$ such that

$$f(x, y, z) = f1(x, f2(y, z))$$

but a choice function h' that satisfies Bellman's Principle may not exist. In other words, we cannot make an optimal choice based on seeing c and d alone.

We may drop the application of h' altogether – but then, a' will return a list of answers of length proportional to n – bringing the parser back to efficiency $O(n^4)$. This explains how the efficiency of DP algorithms depends not only on the grammar, but also on the scoring scheme. Such explanation cannot be given in the terminology of CFGs.

Finally, let us look at ambiguity. This is an issue of great practical importance, and will reoccur when we discuss "real world" applications of ADP in Section 4.5. The CFG corresponding to a yield grammar is always ambiguous (except for trivial cases). After all, if an input string did not have several parses, we would not have a problem of optimization. This ambiguity, however, can have a good or a bad reason[3]. These cannot be distinguished in CFG terminology, whereas in yield grammars, they are separated: Many different candidates have the same yield string: This is good, they constitute our search space. The same candidate has two derivations in the tree grammar: This is bad, as the algorithm will yield redundant answers when asking for more than one, and all counting and probabilistic scoring will be meaningless. We remark without proof that for tree grammars, ambiguity is decidable, while for CFGs, it is not.

Altogether, while part of our treatment could also be formulated in the terminology of CFGs, we feel that yield grammars have a lot to offer. And after all, they represent a small hurdle for a trained computer scientist.

4 The ADP program development method

In this chapter, we first formulate the method we advocate for developing DP algorithms, based on the concepts introduced so far. We then apply this method to three of our four introductory problems. (We shall not re-address the Fibonacci numbers problem, since it is a rather untypical problem. Several analyses performed on Fibonacci numbers via ADP, however, can be studied on the ADP web site.) We shall emphasize the systematics of programming with ADP. Finally, in Section 4.5, we report on applications of ADP in our work on RNA structure prediction and analysis.

Except for minor details omitted for the sake of the presentation, all the ADP algorithms developed here are executable code. In the examples, we use the notation `P ===> R` to indicate that program call `P` delivers result `R`. Readers are invited to run these programs themselves by visiting the ADP website at `http://bibiserv.techfak.uni-bielefeld.de/adp`.

4.1 Systematic program development

As an ADP algorithm is completely specified by the alphabet $\mathcal{A}$, the signature Σ, the yield grammar $(\mathcal{G}, y)$ and an evaluation algebra $\mathcal{E}$, we must mainly decide on the appropriate order of designing these constituents.

[3] There is actually a third reason which we ignore here – see [Gie00b] for a deeper discussion.

Step $\mathcal{A}$ We fix the input alphabet. This is normally clear from the problem statement.

Step Σ We design the signature, introducing one operator for each situation that may be evaluated differently from others.

Step $\mathcal{E}$ We design one or more evaluation algebras:
- a scoring algebra, solving our optimization problem,
- the enumeration algebra, implementing the enumeration of all candidates of a particular problem instance,
- the counting algebra, computing the size of the search space, in a more efficient way than by explicit enumeration,
- a prettyprinting algebra, useful when the application domain knows a user friendly representation of candidates, typically different from our term algebra.

While only the scoring algebra is strictly required to solve the optimization problem at hand, we consider it part of our method also to provide the enumeration and counting algebras, as they open up systematic testing methods.

Step $\mathcal{G}$ We specify the yield grammar $\mathcal{G}$ in ADP notation, describing how candidates are composed from the different constructs represented by the operators of Σ.

After solving each of our introductory problems, we formulate some problem variants, in order to demonstrate how parts of the ADP design can be re-used on related problems.

4.2 The oldest DP problem in the world

The alphabet Input to our problem is a formula written in ASCII characters, consisting of digits and operator symbols. We shall also allow multi-digit numbers.

The signature Rather than adding parentheses, our signature `Bill` introduces operators `Add` and `Mult` to make explicit the different possible internal structures of El Mamun's bill. The operators `Val` and `Ext` represent the conversion of digit strings to integer values.

```
data Bill = Mult  Bill Char Bill |
            Add   Bill Char Bill |
            Ext   Bill Char      |
            Val   Char
```

In the sequel, we consider three different readings of El Mamun's bill:

bill1: $(1+2)*((3*4)+5)$
bill2: $((1+2)*3)*(4+5)$
bill3: $1+((2*(3*4))+5)$

Figure 6 shows the representations of these candidates, both as terms and as trees.

```
bill1 = Mult (Add (Val '1') '+' (Val '2')) '*' (Add (Mult (Val '3') '*'
(Val '4')) '+' (Val '5'))

bill2 = Mult (Mult (Add (Val '1') '+' (Val '2')) '*' (Val '3')) '*'
(Add (Val '4') '+' (Val '5'))

bill3 = Add (Val '1') '+' (Add (Mult (Val '2') '*' (Mult (Val '3') '*'
(Val '4'))) '+' (Val '5'))
```

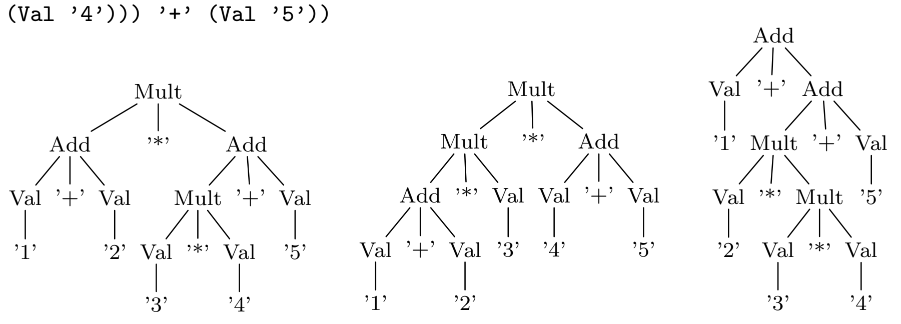

Fig. 6. The term representations of the three candidates for El Mamun's bill and their tree visualizations.

The evaluation algebras

The enumeration and the counting algebra. Enumeration and counting algebras are two standard algebras that come with our method. They are used for systematic testing. For any given signature Σ, the enumeration algebra is just the term algebra T_Σ, augmented with the identity as an objective function. Using the enumeration algebra, the yield parser strictly runs as a parser – it does no evaluation, but produces a list of all candidates it has recognized. Mathematically speaking, all other algebras are homomorphic images of the enumeration algebra.

Ans_{enum} = T_{Bill}

```
enum = (val,ext,add,mult,h) where
val      = Val
ext      = Ext
add      = Add
mult     = Mult
h        = id
```

Ans_{count} = $\mathbb{N}$

```
count = (val,ext,add,mult,h) where
val(c)          = 1
ext(n,c)        = 1
add(x,t,y)      = x * y
mult(x,t,y)     = x * y
h([])           = []
h([x1,...,xr])  = [x1 + ··· + xr]
```

For any given signature Σ, the counting algebra evaluates each individual candidate to 1. However, this is not done by simply interpreting each operator in Σ by a constant function equal to 1. Instead, an operator taking (say) two constituents, multiplies their counts. If, for a given section of the input, a candidate of form `Mult a b` can be built from c_a candidates for a and c_b candidates for b, then itself has a count of $c_a * c_b$. Choosing summation for the choice function in the counting algebra, we can compute the number of candidates much more efficiently than by enumerating all candidates and then counting them.

We will return to the use of the enumeration and the counting algebra in the section on testing.

The buyer's and the seller's algebra. The following two scoring algebras solve El Mamun's problem. They use the function `decode` to convert a digit to an integer value. Each candidate evaluates to its value, according to the parenthesization implicit in the candidate's structure. On the buying side, of course, El Mamun seeks to minimize the bill. On the selling side, he seeks to maximize it.

$Ans_{buyer} = \mathbb{N}$

```
buyer = (val,ext,add,mult,h) where
val(c)       = decode(c)
ext(n,c)     = 10 * n + decode(c)
add(x,t,y)   = x + y
mult(x,t,y)  = x * y
h([])        = []
h (l)        = [minimum(l)]
```

$Ans_{seller} = \mathbb{N}$

```
seller = (val,ext,add,mult,h) where
val(c)       = decode(c)
ext(n,c)     = 10 * n + decode(c)
add(x,t,y)   = x + y
mult(x,t,y)  = x * y
h([])        = []
h (l)        = [maximum(l)]
```

The prettyprinting algebra Candidates in term form are particularly hard to read. We provide a prettyprinting algebra that computes candidates as parenthesized strings.

$Ans_{pretty} = \mathcal{A}^*$

```
pretty = (val,ext,add,mult,h) where
val(c)       = [c]
ext(n,c)     = n ++ [c]
add(x,t,y)   = "(" ++ x ++ (t:y) ++ ")"
mult(x,t,y)  = "(" ++ x ++ (t:y) ++ ")"
h            = id
```

The yield grammar The yield grammar describes all possible internal readings of El Mamun's formula (and any other such formula).

```
bill alg  = axiom formula where
  (val, ext, add, mult, h) = alg

  formula = tabulated (
            number |||
            add  <<< formula ~~~ plus  ~~~ formula |||
            mult <<< formula ~~~ times ~~~ formula ... h)

  number = val <<< digit ||| ext <<< number ~~~ digit
```

```
  digit  = char ’0’ ||| char ’1’ ||| char ’2’ ||| char ’3’ |||
           char ’4’ ||| char ’5’ ||| char ’6’ ||| char ’7’ |||
           char ’8’ ||| char ’9’

  plus   = char ’+’
  times  = char ’*’
```

Testing Running the yield parser for grammar `bill`, using the five algebras in turn, and input `"1+2*3*4+5"`, we obtain:

```
bill enum  "1+2*3*4+5" ===> [
  Add (Val ’1’) ’+’ (Add (Mult (Val ’2’) ’*’ (Mult (Val ’3’) ’*’
      (Val ’4’))) ’+’ (Val ’5’)),
  Add (Val ’1’) ’+’ (Add (Mult (Mult (Val  ’2’) ’*’ (Val ’3’)) ’*’
      (Val ’4’)) ’+’ (Val ’5’)),
  Add (Val ’1’) ’+’ (Mult (Val ’2’) ’*’ (Add (Mult (Val’3’) ’*’
      (Val ’4’)) ’+’ (Val ’5’))),
  ...]

bill pretty  "1+2*3*4+5" ===> [
  "(1+((2*(3*4))+5))",
  "(1+(((2*3)*4)+5))",
  "(1+(2*((3*4)+5)))",
  ...]

bill count  "1+2*3*4+5" ===> [14]
bill buyer  "1+2*3*4+5" ===> [30]
bill seller "1+2*3*4+5" ===> [81]
```

The first call, using `enum`, yields a protocol of the complete search space for the given input. This is feasible only for small inputs, but is a most helpful testing aid. It helps us to verify that the candidates in the search space actually have the shape we expect, all relevant cases are in fact discovered, and so on. The same applies to the second call, using `pretty`, with the extension that this provides the candidates in a more user-friendly form.

The third call, using `count`, merely computes the size of the search space for the given input. We do this here to verify Al Chwarizmi’s discovery that there were 14 alternative readings of El Mamun’s formula.

While the enumeration algebra produces a result of potentially exponential size, the counting algebra merely computes a single number and hence is much more efficient. The invariance `[length(bill enum z)] = bill count z` must hold for all `z`.

Finally, we see that the buyer and the seller algebra solve El Mamun’s problems of minimizing and maximizing the value of the formula.

Problem variation: *A processor allocation problem*
Computation in the days of El Mamun was very slow. A good computing slave took about 2 minutes to perform an addition, and 5 minutes to perform a

multiplication. Even then, understanding the value of a number took practically no time. Fortunately, there were abundant slaves, and they could work in parallel as much as the formula permitted. On a busy day at the bazaar, it might be better to minimize the time consumed for each individual business contact. The following algebra selects for the candidate that has the shortest computation time:

```
Ans_time      = ℕ (computation time in minutes)

time = (val,ext,add,mult,h)  where
val(c)       = 0
ext(n,c)     = 0
add(x,t,y)   = max(x,y) + 2
mult(x,t,y)  = max(x,y) + 5
h([])        = []
h (l)        = [minimum(l)]
```

Evaluating the three candidates shown in Figure 6 we find computation times between 12 and 14 minutes

$$\texttt{h[bill1}_{time}\texttt{, bill2}_{time}\texttt{, bill3}_{time}\texttt{] =}$$
$$\texttt{minimum[12, 12, 14]} \quad = 12$$

and we find that 12 minutes is actually optimal:

```
bill time "1+2*3*4+5" ===> [12]
```

4.3 Optimal matrix chain multiplication

The alphabet Our problem input here is a sequence of matrix dimensions (r_i, c_i). Hence, $\mathcal{A} = \mathbb{N} \times \mathbb{N}$.

The signature Similar to the previous example, we introduce two operators to represent parenthesization of an expression. A matrix chain can be a single matrix or a product of two matrix chains.

```
data Matrixchain = Mult   Matrixchain Matrixchain |
                   Single (Int, Int)
```

Taking from Section 2.2 our example matrices, $A_1 : 10 \times 100, A_2 : 100 \times 5$ and $A_3 : 5 \times 50$, we get two candidates for this chain multiplication. Figure 7 shows the term representation of these candidates and their tree representation.

The evaluation algebras

The enumeration and the counting algebra

```
Ans_enum = T_Matrixchain

enum = (single,mult,h) where
single   = Single
mult     = Mult
h        = id
```

```
Ans_count          = ℕ

count = (single,mult,h) where
single((r,c)) = 1
mult(x,y)     = x * y
h([])         = []
h([x_1,...,x_r]) = [x_1 + ... + x_r]
```

```
candidate1 = Mult (Single (10,100)) (Mult (Single (100,5)) (Single
(5,50)))

candidate2 = Mult (Mult (Single (10,100)) (Single (100,5))) (Single
(5,50))
```

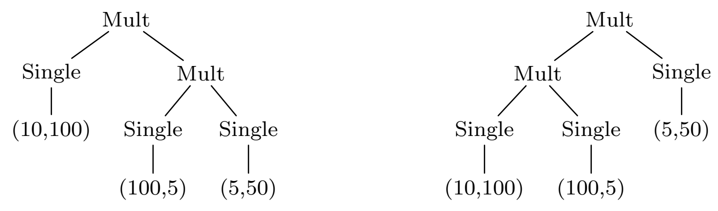

Fig. 7. The term representations of the two candidates for the example matrices and their tree representations.

The scoring algebra The algebra for determining the minimal number of scalar multiplications uses a triple (r, m, c) as answer type. (r, c) denotes the dimension of the resulting matrix and m the minimal number of operations needed to calculate it. With this answer type writing down the algebra is simple:

$Ans_{minmult}$ $= \mathbb{N} \times \mathbb{N} \times \mathbb{N}$

```
minmult = (single,mult,h) where
single((r,c))             = (r,0,c)
mult((r,m,c),(r',m',c')) = (r,m + m' + r*c*c',c')
h([])                     = []
h (l)                     = [minimum(l)]
```
[4]

The yield grammar The yield grammar describes all possible groupings of the matrix chain product.

```
matrixmult alg  = axiom matrices where
   (single, mult, h) = alg

   matrices = tabulated (
              single <<< achar                 |||
              mult   <<< matrices ~~~ matrices ... h )
```

Note that, by the definition of $\mathcal{A}$, `achar` here denotes a single "character" (r_i, c_i).

[4] The objective function considers all three triple elements for minimization. But since `r` and `c` are the same for all candidates of a fixed subchain, only `m` is relevant to this operation.

Testing For input `z = [(10,100),(100,5),(5,50)]` we obtain:

```
matrixmult enum    z ===> [Mult (Single (10,100)) (Mult (Single (100,5))
                                                   (Single (5,50))),
                           Mult (Mult (Single (10,100)) (Single (100,5)))
                                (Single (5,50))]
matrixmult count   z ===> [2]
matrixmult minmult z ===> [(10,7500,50)]

matrixmult count   (z++z++z) ===> [1430]
matrixmult minmult (z++z++z) ===> [(10,20375,50)]
```

Problem variation: *Minimizing intermediate storage*

Another interesting problem is to determine the optimal evaluation order for minimizing the memory usage needed for processing the matrix chain. This is motivated by the fact that memory allocated during calculation can be released in succeeding steps. Consider two matrix chains C_1 and C_2. For multiplying $C_1 * C_2$ we have two possible orders of calculation. When processing C_1 first we have to store the resulting matrix while processing C_2 and then store both results during this multiplication. As a second possibility, we can process C_2 first and store the resulting matrix while calculating C_1. Let *maxloc* C be the biggest memory block allocated during calculation of matrix chain C. Let *loc* C be the size of the resulting matrix. $loc\ A_i = 0$ for all input matrices. The minimal memory usage for processing $C_1 * C_2$ is given by

$$\begin{aligned} &maxloc\ C_1\ C_2 = \\ &\quad min\{max\{maxloc\ C_1, loc\ C_1 + maxloc\ C_2, loc\ C_1 + loc\ C_2 + loc\ C_1C_2\} \\ &\qquad max\{maxloc\ C_2, loc\ C_2 + maxloc\ C_1, loc\ C_1 + loc\ C_2 + loc\ C_1C_2\}\} \end{aligned} \tag{21}$$

This can be expressed by the following algebra:

```
Ans_minmem                   = ℕ × ℕ × ℕ

minmem = (single,mult,h) where
single((r,c))                = (r,0,c)
mult((r,m,c),(r',m',c'))     = (r, minimum
                               [maximum [m,r*c+ m',r*c + r'* c' + r*c'],
                               maximum [m',r'*c'+ m,r*c + r'* c' + r*c']],c')
h([])                        = []
h (l)                        = [minimum(l)]
```

4.4 Global and local similarity problems

This application has been used as a running example in Section 3. We only recollect the results, and then proceed to the problem variations.

The alphabet $\mathcal{A}$ is the ASCII alphabet.

The signature

```
data Alignment = Nil Char                |
                 D  Char Alignment       |
                 I  Alignment Char       |
                 R  Char Alignment Char
```

Figure 8 shows the term representation of a global similarity candidate and its tree representation.

```
candidate1 = D 'd' (R 'a' (I (R 'r' (R 'l' (R 'i' (R 'n' (R 'g' (Nil
'$') 'e') 'n') 'i') 'l') 'r') 'i') 'a')
```

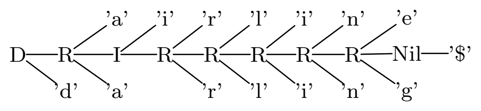

Fig. 8. The term representation of a global similarity candidate `candidate1` for `darling` and `airline` and the tree representation of this term (lying on its side).

The evaluation algebras

The enumeration and the counting algebra

$Ans_{enum} = T_{Alignment}$

```
enum = (nil,d,i,r,h) where
nil     = Nil
d       = D
i       = I
r       = R
h       = id
```

$Ans_{count} = \mathbb{N}$

```
count = (nil,d,i,r,h) where
nil(a)         = 1
d(x,s)         = s
i(s,y)         = s
r(a,s,b)       = s
h([])          = []
```
h($[x_1, \ldots, x_r]$) = $[x_1 + \cdots + x_r]$

The scoring algebras

$Ans_{wgap} = \mathbb{N}$

```
wgap = (nil,d,i,r,h) where
nil(a)   = 0
d(x,s)   = s + gap(x)
i(s,y)   = s + gap(y)
r(a,s,b) = s + w(a,b)
h([])    = []
h (l)    = [maximum(l)]
```

$Ans_{unit} = \mathbb{N}$

```
unit = (nil,d,i,r,h) where
nil(a)   = 0
d(x,s)   = s - 1
i(s,y)   = s - 1
r(a,s,b) = if a==b then s + 1 else s - 1
h([])    = []
h (l)    = [maximum(l)]
```

The prettyprinting algebra We provide a prettyprinting algebra that computes candidates in the familiar form of two strings padded with gap symbols.

```
Ans_pretty       = A* × A*

pretty = (nil,d,i,r,h) where
nil(a)          = ("","")
d(x,(l,r))      = (x:l, gap:r)
i((l,r),y)      = (gap:l, y:r)
r(x,(l,r),y)    = (x:l,y:r)
h               = id
gap             = '-'
```

The yield grammars

Global similarity The yield grammar describes all possible ways to transform one string into the other by character replacement, deletion and insertion.

```
globsim alg = axiom alignment where
  (nil, d, i, r, h) = alg

  alignment  = tabulated(
               nil <<< char '$'                       |||
               d   <<< achar  ~~~ alignment           |||
               i   <<<            alignment ~~~ achar |||
               r   <<< achar  ~~~ alignment ~~~ achar ... h)
```

Local similarity To formulate the yield grammar for local similarity, we modify the signature. We introduce two new operators `skip_left` and `skip_right` for skipping characters in the beginning of x and y. To allow skipping at the end of x and y, we modify the argument of `Nil` to be an arbitrary string, including the separator symbol.

Strictly speaking, the skip operators should become part of the algebra. But since they are useful for any case when going from a global to a local problem, we assume they are predefined as shown below, and part of any algebra when desired.

```
skip_right a b    = a
skip_left  a b    = b

locsim alg = axiom skipR where
  (nil, d, i, r, h) = alg

  skipR     = skip_right <<<           skipR ~~~ achar |||
              skipL                                    ... h

  skipL     = skip_left  <<< achar ~~~ skipL           |||
              alignment                                ... h
```

```
  alignment = tabulated(
              nil <<< string                        |||
              d   <<< achar ~~~ alignment           |||
              i   <<<           alignment ~~~ achar |||
              r   <<< achar ~~~ alignment ~~~ achar ... h)
```

Testing For inputs "`darling`" and "`airline`", concatenated to `z = "darling$enilria"` we obtain:

```
globsim enum z ===> [
  D'd'(D'a'(D'r'(D'l'(D'i'(D'n'(D'g'(I(I(I(I(I(I(I(Nil'$')
   'e')'n')'i')'l')'r')'i')'a'))))))),
  D'd'(D'a'(D'r'(D'l'(D'i'(D'n'(I(D'g'(I(I(I(I(I(I(Nil'$')
   'e')'n')'i')'l')'r')'i'))'a')))))),
  D'd'(D'a'(D'r'(D'l'(D'i'(D'n'(I(I(D'g'(I(I(I(I(I(Nil'$')
   'e')'n')'i')'l')'r'))'i')'a')))))),
  ...]

globsim pretty z ===> [
  ("darling-------","-------airline"),
  ("darlin-g------","------a-irline"),
  ("darlin--g-----","------ai-rline"),
  ...]

globsim count z ===> [48639]
globsim unit  z ===> [2]

locsim enum z ===> [
  Nil (7,8),
  Nil (6,8),
  D 'g' (Nil (7,8)),
  Nil (5,8),
  D 'n' (Nil (6,8)),
  D 'n' (D 'g' (Nil (7,8))),
  ...]

locsim pretty z ===> [
  ("",""),
  ("",""),
  ("g","-"),
  ("",""),
  ("n","-"),
  ...]

locsim count z ===> [365600]
locsim unit  z ===> [4]
```

Problem variation: *Affine gap scores*

In the algebras presented so far, consecutive insertions or deletions achieve the same score as the same number of single gaps (deletions and insertions). In order to analyze biological sequence data, it is more adequate to use an affine gap score model. This means to assign an opening cost (`open`) to each gap and an extension cost (`extend`) for each deleted (or inserted) character. Choosing `open` > `extend`, this results in a better model, favoring few long gaps over many short ones. The use of affine gap scores was introduced for biosequence analysis in [Got82].

The signature In order to distinguish the opening of a gap and the extension of a gap we have to extend the signature `Alignment`:

```
data Alignment = Nil Char                |
                 D  Char Alignment       |
                 I  Alignment Char       |
                 R  Char Alignment Char  |
                 Dx Char Alignment       |
                 Ix Alignment Char
```

The affine gap score algebra

Ans_{affine} = ℕ

```
affine = (nil,d,i,r,dx,ix,h) where
nil(a)    = 0
d(x,s)    = s + open + extend
i(s,y)    = s + open + extend
r(a,s,b)  = s + w(a,b)
dx(x,s)   = s + extend
ix(s,y)   = s + extend
h([])     = []
h (l)     = [maximum(l)]
```

The yield grammar In the modified yield grammar for global similarity, we have to distinguish the opening of a gap and the extension of a gap. Thus, our yield grammar requires three nonterminal symbols.

```
affineglobsim alg = axiom alignment where
  (nil, d, i, r, dx, ix, h) = alg

  alignment = tabulated (
               nil <<< char '$'                       |||
               d   <<< achar  ~~~ xDel                |||
               i   <<<            xIns      ~~~ achar |||
               r   <<< achar  ~~~ alignment ~~~ achar ... h)

  xDel      = tabulated (
              alignment             |||
```

```
              dx <<< achar  ~~~ xDel ... h )

  xIns      = tabulated (
              alignment              |||
              ix <<< xIns ~~~ achar  ... h )
```

To achieve the yield grammar for local alignments using the affine gap score model, the grammar for global alignments has to be modified in the same manner as shown for the simple gap score model.

4.5 Applications of ADP in RNA structure prediction and analysis

In this section we report on applications of ADP in our work on RNA structure prediction and analysis. We shall reformulate some early approaches in ADP, and then outline some recent, more refined techniques, where the convenience of the ADP approach has proved to be a major advantage. Table 4.5 gives a preview of the problems discussed in this section.

All genetic information in living organisms is encoded in long chain molecules. DNA is the storage form of genetic information. Its shape is always the double helix discovered by Watson and Crick, consisting of two chain molecules that are complementary to each other in the following sense: Each chain is a linear arrangement of the bases Adenine, Cytosine, Guanine, and Thymine, labeled A, C, G, T, connected by covalent bonds to a backbone built from sugar and phosphate. Hydrogen bonds can form between A–T and C–G, creating base pairs in the form of a densely stacked helix. The human genome consists of roughly 3×10^9 such base pairs. Mathematically it is a string of length 3×10^9 over the alphabet $\{A, C, G, T\}$.

grammar name	problem solved	grammar size # nonterminals	# tables	complexity time	space
nussinov78	base pair maximization	1	1	n^3	n^2
zuker81	mfe folding	13	2	n^4	n^2
wuchty98	non redundant folding	11	4	n^4	n^2
evers01	saturated structures	19	9	n^3	n^2
pknotsRE00	chained pseudoknots	–	–	n^6	n^4
pknotsRG03	pseudoknotted structures	47	17	n^4	n^2

Table 1. Application sizes (In practice, size restrictions are imposed that reduce the time complexity of `zuker81` and `wuchty98` to $O(n^3)$.)

RNA is the active form of genetic information. It is transcribed from DNA as a chain of A, C, G and U, where U denotes Uracil, the RNA representative of Thymine. Possible base pairs in RNA are G–C, A–U and also G–U. RNA is typically single stranded, and by folding back onto itself, it forms structure. One distinguishes three levels: The primary structure of an RNA molecule is simply the string of bases. Its tertiary structure is the spatial arrangement of its atoms. On an intermediate and more abstract level, we have the secondary structure: The set of base pairs that a molecule forms in attaining its 3D shape. Some RNAs are simply building plans for proteins, and their structure bears no significance. But there are many classes of RNA whose structure determines their function. Predicting structure given the sequence is therefore an important task in bioinformatics.

Structure formation is driven by the forces of hydrogen bonding between base pairs, and energetically favorable stacking of base pairs. While the prediction of RNA tertiary structure is inaccessible to computational methods, secondary structure can be predicted quite reliably. Figure 9 gives examples of typical elements found in RNA secondary structure, called stacking regions (or helices), bulge loops, internal loops, hairpin loops and multiple loops.

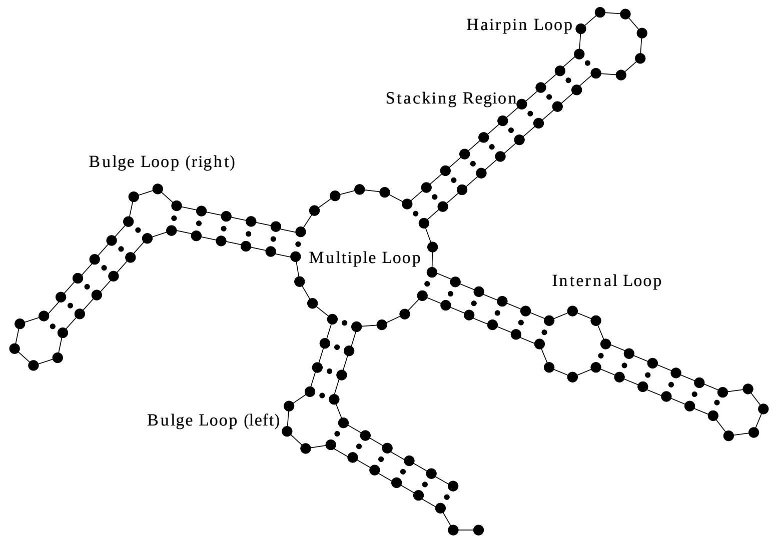

Fig. 9. Typical elements found in RNA secondary structure

The first approach to RNA structure prediction was based on the idea of maximizing the number of base pairs [NPGK78]. In the RNA domain, our input sequence is a string over $\{A, C, G, U\}$. The parser `achar` is renamed to `base`. The predicate `basepairing` (i, j) checks whether the bases at input positions $i + 1$ and j can actually form a base pair.

The grammar `nussinov78` implements the algorithm of [NPGK78], with the evaluation algebra designed for maximizing the number of base pairs.

```
nussinov78 alg = axiom s where
 (nil,left,right,pair,split,h) = alg

 s = tabulated (
     nil   <<< empty                |||
     left  <<< base ~~~ s           |||
     right <<<          s ~~~ base  |||
     pair  <<< base ~~~ s ~~~ base
                 `with` basepairing |||
     split <<<        s ~~~ s        ... h)
```

An enumeration algebra can be provided in the usual way, a counting algebra, however, would be useless. The case analysis in the Nussinov algorithm is redundant – even the base string "a" is assigned the two structures

`Left('a', Nil)` and `Right(Nil, 'a')`

which actually denote the same shape.

Base pair maximization ignores the favorable energy contributions from base pair stacking, as well as the unfavorable contributions from loops. A major advance was brought about by the algorithm of Zuker and Stiegler [ZS81], which uses experimentally determined energy parameters and solves structure prediction as a problem of minimizing free energy. The grammar `zuker81` implementing this algorithm uses two tables and an algebra distinguishing 10 cases. We do not show it here, as some details are intricate to explain. In particular, this grammar is also ambiguous. This approach can be used to enumerate some near-optimal structures after filtering out redundant answers.

A non-redundant algorithm was actually our first application of ADP [Gie98]. Simultaneously, the problem was also solved in the traditional way by Wuchty et al. [WFHS99]. We show the grammar `wuchty98`. Here the signature has 8 operators, each one modeling a particular structure element, plus the list constructors (`nil, ul, cons`) to collect sequences of components in a unique way. We use different choice functions (`h_s, h_l, h`) operating on them. An idea of [Gie98] was integrated in `wuchty98`: Nonterminal symbol `strong` is used to avoid structures with isolated (unstacked) base pairs. We know in beforehand from the energy model that unstacked base pairs are energetically unstable. Purging them from the search space decreases the number of candidates considerably. This grammar, because of its non-ambiguity, can also be used to study combinatorics, such as the expected number of feasible structures of a particular sequence of length n.

```
wuchty98 alg = axiom struct where

 (str,ss,hl,sr,bl,br,il,ml,nil,cons,ul,h,h_l,h_s) = alg
```

```
struct       =  str  <<< comps                        |||
                str  <<< (ul  <<< singlestrand)       |||
                str  <<< (nil <<< empty)              ... h_s
comps        = tabulated(
                cons <<< block  ~~~ comps             |||
                ul   <<< block                        |||
                cons <<< block  ~~~ (ul <<< singlestrand) ... h_l)
singlestrand =  ss   <<< region
block        = tabulated(
                         strong                       |||
                bl   <<< region ~~~ strong            ... h)
strong       = tabulated(
                (sr  <<< base ~~~ (strong ||| weak) ~~~ base)
                     `with` basepairing               ... h)
weak         = tabulated(
                (hairpin ||| leftB ||| rightB ||| iloop ||| multiloop)
                     `with` basepairing                ... h)
 where hairpin   = hl <<< base ~~~ (region `with` minsize 3)    ~~~ base
       leftB     = sr <<< base ~~~ (bl   <<< region ~~~ strong) ~~~ base
       rightB    = sr <<< base ~~~ (br   <<< strong ~~~ region) ~~~ base
       multiloop = ml <<< base ~~~ (cons <<< block  ~~~ comps ) ~~~ base
       iloop     = sr <<< base ~~~ (il   <<< region ~~~ strong
                                                ~~~ region) ~~~ base
```

The users of RNA folding programs are typically not satisfied with a single answer, a structure of minimal free energy. They would rather see an ensemble of near-optimal structures, sufficiently distinct to characterize what shapes a molecule can actually attain. If there is a single predominant structure, there is probably some biological function associated with it. If there is a large number of different structures, all close to minimal free energy, then the molecule's shape is probably undefined and meaningless. Finally, there are rare and interesting cases of exactly two near-optimal structures, where the molecule may act as a conformational switch. Computing exactly the characteristic ensemble of structures is still an unsolved problem.

Following the incentive to reduce the search space without losing interesting structures, D. Evers developed an ADP grammar describing the class of saturated structures – those whose helices have maximal extent under the rules of base pairing. This fairly sophisticated grammar requires 19 nonterminals and 9 tables; it requires some advanced techniques that nicely fit in the ADP framework. For example, evaluation functions need to compute auxiliary information in addition to energy values, hence we use different answer types and different choice functions operating on them. The grammar `evers01` can be found in [Eve03], the traditional recurrences are given in [EG01]. Although published first, they were actually derived from the ADP grammar. This problem of saturated folding was already posed by Sankoff in 1984 [ZS84]; a look at the recurrences in [EG01] explains why it required the help of a formal method to be solved.

Another problem currently in focus is the folding of structures containing pseudoknots. All the structures discussed previously can be seen as strings of properly nested parentheses, e.g. ...(((...)...))...(...)..., where matching parentheses denote base pairs, and dots denote unpaired bases. Pseudoknots contain base pairs that interact in a crosswise fashion, like in ...(((...[[[...)))...]]].... Finding the optimal, potentially pseudoknotted structure under the current energy model has been proven to be an NP-complete problem [Aku00]. Rivas and Eddy developed an algorithm `pknotsRE` [RE98] that solves this problem for a restricted class of pseudoknots, running in $O(n^6)$ time and $O(n^4)$ space. In spite of its high computational effort it is actually used in practice. However, there lies a touch of tragedy in this algorithm: In theory, it can recognize quite sophisticated structures containing chains of pseudoknots – however, such structures require a molecule of a certain minimal size, clearly longer than the algorithm can handle in practice. Hence, it will never fulfill its promise.

As a reaction to this situation, we have developed a folding algorithm for the more restricted, but still useful class of “canonical simple recursive pseudoknots”. This program `pknotsRG` is our largest ADP grammar so far, using 47 nonterminals, 140 alternatives, and 17 tables. Its complexity is $O(n^4)$ time and $O(n^2)$ space. Its implementation in compiled Haskell runs on the Bielefeld Bioinformatics Server, used by the bioinformatics community. It can handle sequences of up to 400 bases within 77 minutes, and is currently the only program to solve folding problems of this size. A description of this program is published in [SKM+03].

4.6 Summary of Section 4

We hope that, by the series of examples treated in this section, we have convinced the reader that using the ADP framework is an aid rather than a formal obstacle to designing dynamic programming algorithms. Often the grammar design is the most difficult part; by using the enumeration algebra, we can check this design before we apply any scoring algebras. The ease of replacing one scoring algebra by another is quite useful in a design phase where the adequate modeling of the problem domain is not yet totally clear. We also hope to have shown that this design method can be used without worrying about implementation details. These will be provided in the next section, while readers mainly interested in methodology may skip ahead to the conclusion.

5 Three ways to implement ADP

In this section we show three ways to make ADP algorithms actually run. The most convenient one is the embedding of ADP in Haskell. Here the ADP algorithm can be executed as is. Advocates of imperative programming are provided with translation schemes that lead from the yield grammar to the traditional DP recurrences, to be implemented in C or FORTRAN. This transition, when done by hand, is quite cumbersome. It is rewarded by better runtime (by a constant factor) and an imperative implementation that can be tested against the same algorithm executed via the Haskell embedding. Eventually, this transition should be automated, and to this end, we report on an ADP compiler project currently under way.

5.1 Unifying single sequence analysis and pairwise sequence comparison

We have been considering two kinds of problems: In El Mamun's and in the matrix chain problem, the task was to recover an internal structure of a single sequence x. A candidate t for x has $yield(t) = x$. In the similarity problem, we are comparing two sequences x and y. We saw that here a candidate t for inputs x, y has $yield(t) = xy^{-1}$. If we choose to include a separator symbol \$ between x and the reverse of y, we have $yield(t) = x\$y^{-1}$. This is a matter of convenience. To unify both cases, in the sequel we assume we have a single input z, where either $z = x$ or $z = xy^{-1}$, or $z = x\$y^{-1}$. We assume that $z, m = |x|, n = |y|, l = |z|$ are known and represented by global variables. Thus, in the pairwise case, even when we do not use the separator symbol, we know the boundary between x and y^{-1} in z.

This input convention is used for the sake of a uniform treatment here. In our practical work, we use a special version of the yield parser for the pairwise case, which explicitly reads from two inputs in forward direction. (See ADP website for details.)

Since z is global, a subword $z_{i+1}, ..., z_j$ of z is simply represented by the subscript pair `(i,j)`. Note that i marks the subscript position *before* the first character of subword (i, j). This convention allows one to use k as the common boundary of adjacent subwords when splitting (i, j) into (i, k) and (k, j).

5.2 Embedding ADP in Haskell

ADP has been designed as a domain specific language embedded in Haskell [PJ03]. An algorithm written in ADP notation can be directly executed as a Haskell program. Of course, this requires that the functions of the evaluation algebra are coded in Haskell. A smooth embedding is achieved by adapting the technique of parser combinators [Hut92], which literally turn the grammar into a parser. Hutton's technique applies to context free grammars and string parsing. We need to introduce suitable combinator definitions for yield parsing, and add tabulation.

Generally, a parser is a function that, given a subword of the input, returns a list of all its parses.

Lexical parsers The lexical parser `achar` recognizes any single character except the separator symbol. Parser `string` recognizes a (possibly empty) subword. Specific characters or symbols are recognized by `char` and `symbol`. Parser `empty` recognizes the empty subword.

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

> empty        :: Parser ()
> empty  (i,j) =  [() | i == j]

> achar        :: Parser Char
> achar  (i,j) =  [z!j | i+1 == j, z!j /= '$']

> char         :: Char -> Parser Char
> char c (i,j) =  [c | i+1 == j, z!j == c]

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

> symbol       :: String -> Parser Subword
> symbol s (i,j) = [(i,j)| and [z!(i+k) == s!!(k-1) | k <-[1..(j-i)]]
```

Nonterminal parsers The nonterminal symbols in the grammar are interpreted as parsers, with the productions serving as their mutually recursive definitions. Each righthand side is an expression that combines parsers using the parser combinators.

Parser combinators The operators introduced in the ADP notation are now defined as parser combinators: `|||` concatenates result lists of alternative parses, `<<<` grabs the results of subsequent parsers connected via `~~~` and successively "pipes" them into the algebra function on its left. 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 b -> ([b] -> [b]) -> Parser b
> (...) r h (i,j) = h (r (i,j))
```

Note that the operator priorities are defined such that an expression `f <<< a ~~~ b ~~~ c` is read as `((f <<< a) ~~~ b) ~~~ c`. This makes use of curried functions: the results of parser `f <<< a` are functions – i. e. calls to `f` with (only) the first argument bound.

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 specified nonterminal symbol and the complete input.

```
> type Filter    =  Subword -> Bool
> with           :: Parser b -> Filter -> Parser b
> with q c (i,j) =  if c (i,j) then q (i,j)  else []

> axiom          :: Parser b -> [b]
> axiom ax       =  ax (0,l)
```

When a parser is called with the enumeration algebra – i.e. the functions applied are actually tree constructors, and the objective function is the identity function – then it behaves like a proper yield parser and generates a list of trees according to Definition 4. However, called with some other evaluation algebra, it computes any desired type of answer.

Tabulation Adding tabulation is merely a change of data type, replacing a recursive function by a recursively defined table – just in the way we did this in Section 2.4. Now we need a general scheme for this purpose: The function `table` records the results of a parser `p` for all subwords of an input of size `n`, and returns as its result a function that does lookup in this table. Note the essential invariance `(table n f)!(i,j) = f(i,j)`. Therefore, `table n p` is a tabulated parser, that can be used in place of parser `p`. In contrast to the latter, it does not compute the results for the same subword `(i,j)` repeatedly – here we enjoy the blessings of lazy evaluation and avoid exponential explosion.

The keyword `tabulated` is now defined as `table` bound to the global variable l, the length of the input.

```
> table     :: Int -> Parser b -> Parser b
> table n p =  (!) (array ((0,0),(n,n))
>                  [((i,j), p (i,j)) | i<- [0..n], j<- [i..n]])

> tabulated = table l
```

Removing futile computations Consider the production `a = f <<< a ~~~ char`. Our definition of the `~~~` combinator splits subword (i, j) in all possible ways, including empty subwords on either side. Obviously, `achar`, which recognizes a single character, has a fixed yield size of 1, leaving the subword $(i, j-1)$ for the yield of nonterminal symbol `a`. In this case, iteration over all splits of (i, j) into (i, k) and (k, j) is mathematically correct, but a futile effort. The only successful split can be $(i, j-1)$ and $(j-1, j)$. What is worse, since the production is left-recursive, the last split considered without need is (i, j) and (j, j), which leads to infinite recursion.

Both situations are avoided by using specializations of the `~~~` combinator that are aware of bounded yield sizes and avoid unnecessary splits. For the case of splitting off a single character, we use `-~~` and `~~-`, while the fully general case of an arbitrary, but known yield size limit is treated by the `~~` combinator.

```
> infixl 7 ~~,-~~ , ~~-

> (-~~) 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)]


> (~~) :: (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)]
```

These combinators are used in asymptotic efficiency tuning via width reduction as described in [GM02]. Using these special cases, our global similarity grammar can be written in the form

```
> globsim alg = axiom alignment where
>   (nil, d, i, r, h) = alg

>   alignment  = tabulated(
>               nil <<< char '$'                      |||
>               d   <<< achar  -~~ alignment          |||
>               i   <<<            alignment ~~- achar  |||
>               r   <<< achar  -~~ alignment ~~- achar  ... h)
```

which now, as a functional program, has the appropriate efficiency of $O(mn)$.

5.3 Derivation of explicit recurrences

In the previous section we showed how to embed an ADP algorithm smoothly in a functional language. Although efficient implementations of Haskell exist, it still seems desirable to derive an imperative version of the algorithm. The sheer volume of data present in many dynamic programming domains and an easy integration in existing systems are two of the reasons. The classical approach in dynamic programming is to implement the imperative version of the algorithm starting from the matrix recurrences derived by experience or intuition. In this section we show how to derive the recurrences in a systematic way from an algorithm in ADP notation.

Each tabulated production will result in a matrix recurrence relation. The definitions of non tabulated productions can be inserted directly at the occurrences of the corresponding nonterminal symbols in the grammar. In the following, we assume that all productions are tabulated.

Translation schemes The matrix recurrences for a grammar $\mathcal{G}$ can be derived by the following translation patterns starting with $\mathcal{C}[\![\mathcal{G}]\!]$ in Pattern 22. We use list comprehension notation, in analogy to set notation: $[f(x,y)|x \in xs, y \in ys]$ denotes the list of all values $f(x,y)$ such that x is from list xs and y from list ys. To distinguish a parser call $q(i,j)$ from a semantically equivalent table lookup, we denote the latter by $q!(i,j)$. The function pair $(low(p), up(p))$ shall provide the yield size of a tree pattern p and is defined by $(low(p), up(p)) = (\inf_{q \in \mathcal{L}(p)} |q|, \sup_{q \in \mathcal{L}(p)} |q|)$ if $\mathcal{L}(p) \neq \emptyset$, and $(low(p), up(p)) = (\infty, 0)$ otherwise.

$$\mathcal{C}[\![\texttt{grammar alg} \;=\; \texttt{axiom p where } \texttt{v}_1 = \texttt{q}_1 \ldots \texttt{v}_\texttt{m} = \texttt{q}_\texttt{m}]\!] = \tag{22}$$

```
for $j = 0$ to $l$
    for $i = 0$ to $j$
        $v_1!(i,j) = \mathcal{C}[\![\texttt{q}_1]\!](i,j)$
        $\vdots$
        $v_m!(i,j) = \mathcal{C}[\![\texttt{q}_\texttt{m}]\!](i,j)$
return $p!(0,l)$
```

$$\mathcal{C}[\![\texttt{q ... h}]\!](i,j) = h(\mathcal{C}[\![\texttt{q}]\!](i,j)) \tag{23}$$

$$\mathcal{C}[\![\texttt{q}_1 \texttt{ ||| } \ldots \texttt{ ||| } \texttt{q}_\texttt{r}]\!](i,j) = \mathcal{C}[\![\texttt{q}_1]\!](i,j) \;{+\!+} \ldots {+\!+}\; \mathcal{C}[\![\texttt{q}_\texttt{r}]\!](i,j) \tag{24}$$

$$\mathcal{C}[\![\texttt{t <<< q}_1 \texttt{ \~{}\~{}\~{} } \ldots \texttt{ \~{}\~{}\~{} q}_\texttt{r}]\!](i,j) = \tag{25}$$

$$[t(p_1, \ldots, p_r)|p_1 \in \mathcal{C}[\![\texttt{q}_1]\!](i,k_1), \ldots, p_r \in \mathcal{C}[\![\texttt{q}_\texttt{r}]\!](k_{r-1},j)]$$

for $k_1, \ldots, k_{r-1}$, such that $k_0 = i,\ k_r = j,$

$$max(k_{l-1} + low(q_l), k_{l+1} - up(q_{l+1})) \leq k_l \leq$$
$$min(k_{l-1} + up(q_l), k_{l+1} - low(q_{l+1}))$$

$$\mathcal{C}[\![\texttt{q with c}]\!](i,j) = \text{if } c(i,j) \text{ then } \mathcal{C}[\![\texttt{q}]\!](i,j) \text{ else } [\,] \tag{26}$$

$$\mathcal{C}[\![\texttt{v}]\!](i,j) = v!(i,j) \qquad \text{for } v \in V \tag{27}$$

$$\mathcal{C}[\![\texttt{t}]\!](i,j) = \mathcal{T}[\![\texttt{t}]\!](i,j) \qquad \text{for } t \text{ terminal} \tag{28}$$

In Pattern 25, note the direct correspondence to the definition of the ~~~ combinator in Section 5.2.

Translation patterns $\mathcal{T}[\![\texttt{w}]\!]$ for terminal symbols must be chosen according to their respective semantics. We give three examples:

$$\mathcal{T}[\![\texttt{char c}]\!](i,j) = \text{if } i+1 \equiv j \wedge z_j \equiv c \text{ then } [c] \text{ else } [\,]$$
$$\mathcal{T}[\![\texttt{achar}]\!](i,j) = \text{if } i+1 \equiv j \wedge z_j \neq \text{'\$'} \text{ then } [z_j] \text{ else } [\,]$$
$$\mathcal{T}[\![\texttt{string}]\!](i,j) = \text{if } i \leq j \text{ then } [(i,j)] \text{ else } [\,]$$

Example We demonstrate the translation for the global similarity example of Section 4.4:

```
globsim alg = axiom alignment where
  (nil, d, i, r, h) = alg

  alignment  = nil <<< char '$'                         |||
```

```
d   <<< achar ~~~ alignment             |||
i   <<<           alignment ~~~ achar   |||
r   <<< achar ~~~ alignment ~~~ achar   ... h
```

Applying Pattern 22 to this grammar provides the framework of the control structure:

```
for $j = 0$ to $l$
    for $i = 0$ to $j$
        $alignment!(i, j) = \mathcal{C}[\![\texttt{nil <<< ...}]\!](i, j)$
return $alignment!(0, l)$
```

Starting with $alignment!(i, j) = \mathcal{C}[\![\texttt{nil <<< ...}]\!](i, j)$ we apply Patterns 23 and 24 to the righthand side of the production:

$$\begin{aligned}
alignment!(i, j) = h(&\\
&\mathcal{C}[\![\texttt{nil <<< char '\$'}]\!](i, j) +\!\!+ && (29)\\
&\mathcal{C}[\![\texttt{d <<< achar \~{}\~{}\~{} alignment}]\!](i, j) +\!\!+ && (30)\\
&\mathcal{C}[\![\texttt{i <<< alignment \~{}\~{}\~{} achar}]\!](i, j) +\!\!+ && (31)\\
&\mathcal{C}[\![\texttt{r <<< achar \~{}\~{}\~{} alignment \~{}\~{}\~{} achar}]\!](i, j)) && (32)
\end{aligned}$$

The resulting four expressions can be translated separately according to Pattern 25. Expression 29 translates to:

$$\begin{aligned}
&[nil(p_1) | p_1 \in \mathcal{T}[\![\texttt{char '\$'}]\!](i, j)]\\
&= \text{if } i + 1 \equiv j \wedge z_j \equiv '\$' \text{ then } [nil('\$')] \text{ else } [\,]
\end{aligned}$$

Translation of Expressions 30 − 32 makes use of the yield size functions *low* and *up*. Table 2 shows their values for the expressions needed in this example. The constant yield sizes of the terminal symbols can be taken directly from the corresponding parser definitions. For nonterminal symbols and arbitrary expressions this requires a deeper analysis of the grammar. This is detailed in [GS02]. Accepting the risk of ending up with suboptimal code for the resulting matrix recurrences, a yield size $(0, \infty)$ is always a safe approximation.

x	$(low(x), up(x))$
`char c`	$(1, 1)$
`achar`	$(1, 1)$
`alignment`	$(1, \infty)$

Table 2. Yield sizes needed for alignment example

Proceeding with Expression 30 leads to the following calculation:

$$\begin{aligned}
&\mathcal{C}[\![\texttt{d <<< achar \~{}\~{}\~{} alignment}]\!](i,j)\\
&= [d(p_1,p_2)|p_1 \in \mathcal{T}[\![\texttt{achar}]\!](i,k_1), p_2 \in \mathcal{C}[\![\texttt{alignment}]\!](k_1,j)]\\
&\qquad \text{for } k_1 \text{ such that}\\
&\qquad\qquad max(i + low(achar), j - up(alignment)) \leq k_1 \leq\\
&\qquad\qquad min(i + up(achar), j - low(alignment))
\end{aligned}$$

With yield sizes $(1,1)$ and $(1,\infty)$ for `achar` and `alignment` the loop variable k_1 simplifies to a constant $k_1 = i+1$ and the condition $i+2 \leq j$:

$$\begin{aligned}
&[d(p_1,p_2)|i+2 \leq j, p_1 \in \mathcal{T}[\![\texttt{achar}]\!](i,i+1), p_2 \in \mathcal{C}[\![\texttt{alignment}]\!](i+1,j)]\\
&= [d(p_1,p_2)|i+2 \leq j \wedge z_{i+1} \neq \text{'\$'}, p_1 \in [z_{i+1}], p_2 \in alignment!(i+1,j)]\\
&= [d(z_{i+1},p_2)|i+2 \leq j \wedge z_{i+1} \neq \text{'\$'}, p_2 \in alignment!(i+1,j)]
\end{aligned}$$

Translating Expressions 31 and 32 in the same way we arrive at the following recurrence relation for the matrix *alignment*:

$$\begin{aligned}
&alignment!(i,j) = h(\\
&\quad \text{if } i+1 \equiv j \wedge z_j \equiv \text{'\$'} \text{ then } [nil(\text{'\$'})] \text{ else } [\,] +\!\!+\\
&\quad [d(z_{i+1},p_2)|i+2 \leq j \wedge z_{i+1} \neq \text{'\$'}, p_2 \in alignment!(i+1,j)] +\!\!+\\
&\quad [i(p_1,z_j)|i+2 \leq j \wedge z_j \neq \text{'\$'}, p_1 \in alignment!(i,j-1)] +\!\!+\\
&\quad [r(z_{i+1},p_2,z_j)|i+3 \leq j \wedge z_{i+1} \neq \text{'\$'} \wedge z_j \neq \text{'\$'},\\
&\qquad\qquad\qquad p_2 \in alignment!(i+1,j-1)])
\end{aligned} \tag{33}$$

The explicit recurrences derived so far can be used together with code implementing the functions of an arbitrary evaluation algebra. If this code is simple, it can be inlined, which often allows further simplification of the recurrences.

Inlining evaluation algebras We demonstrate inlining by means of the count algebra and the unit cost algebra introduced in Section 4.4:

Ans_{count}	= $\mathbb{N}$	Ans_{unit}	= $\mathbb{N}$
`count = (nil,d,i,r,h) where`		`unit = (nil,d,i,r,h) where`	
`nil(x)`	`= 1`	`nil(x)`	`= 0`
`d(x,s)`	`= s`	`d(x,s)`	`= s - 1`
`i(s,y)`	`= s`	`i(s,y)`	`= s - 1`
`r(a,s,b)`	`= s`	`r(a,s,b)`	`= if a==b then s + 1 else s - 1`
`h([])`	`= []`	`h([])`	`= []`
`h(`$[x_1,\ldots,x_r]$`)`	`= [`$x_1+\cdots+x_r$`]`	`h (l)`	`= [maximum(l)]`

For the counting algebra this results in the following recurrence for the matrix *alignment*:

$$
\begin{aligned}
&\mathit{alignment}!(i,j) = \\
&\quad (\text{if } i+1 \equiv j \wedge z_j \equiv {}'\$' \text{ then } 1 \text{ else } 0) + \\
&\quad (\text{if } i+2 \leq j \wedge z_{i+1} \neq {}'\$' \text{ then } \mathit{alignment}!(i+1,j) \text{ else } 0) + \\
&\quad (\text{if } i+2 \leq j \wedge z_j \neq {}'\$' \text{ then } \mathit{alignment}!(i,j-1) \text{ else } 0) + \\
&\quad (\text{if } i+3 \leq j \wedge z_{i+1} \neq {}'\$' \wedge z_j \neq {}'\$' \text{ then } \mathit{alignment}!(i+1,j-1) \text{ else } 0)
\end{aligned}
$$

And for the unit cost algebra:

$$
\begin{aligned}
&\mathit{alignment}!(i,j) = \max(\\
&\quad (\text{if } i+1 \equiv j \wedge z_j \equiv {}'\$' \text{ then } [0] \text{ else } [\,]) \mathbin{++} \\
&\quad (\text{if } i+2 \leq j \wedge z_{i+1} \neq {}'\$' \text{ then } [\mathit{alignment}!(i+1,j) - 1] \text{ else } [\,]) \mathbin{++} \\
&\quad (\text{if } i+2 \leq j \wedge z_j \neq {}'\$' \text{ then } [\mathit{alignment}!(i,j-1) - 1] \text{ else } [\,]) \mathbin{++} \\
&\quad (\text{if } i+3 \leq j \wedge z_{i+1} \neq {}'\$' \wedge z_j \neq {}'\$' \text{ then } [\text{if } z_{i+1} \equiv z_j \\
&\qquad\qquad \text{then } \mathit{alignment}!(i+1,j-1) + 1 \\
&\qquad\qquad \text{else } \ \mathit{alignment}!(i+1,j-1) - 1] \text{ else } [\,]))
\end{aligned}
$$

Solving dependencies Consider the example of string similarity. By adding the production

```
loc_align = alignment                                 |||
            skip_right <<<           loc_align ~~~ achar  |||
            skip_left  <<< achar ~~~ loc_align            ... h
```

we can extend the algorithm for global similarity to an algorithm for local similarity. (Here we use a single production rather than two as in Section 4.4, to shorten the translation example.) Following the translation scheme of the previous paragraphs we derive the matrix recurrence for *loc_align*:

$$
\begin{aligned}
&\mathit{loc\_align}!(i,j) = \max(\\
&\quad [\mathit{alignment}!(i,j)] \mathbin{++} \\
&\quad (\text{if } i+2 \leq j \wedge z_j \neq {}'\$' \text{ then } [\mathit{loc\_align}!(i,j-1)] \text{ else } [\,]) \mathbin{++} \\
&\quad (\text{if } i+2 \leq j \wedge z_{i+1} \neq {}'\$' \text{ then } [\mathit{loc\_align}!(i+1,j)] \text{ else } [\,]))
\end{aligned}
$$

The dependency between *loc_align*$!(i,j)$ and *alignment*$!(i,j)$ leads us to a new issue not present in the functional version of the algorithm. Functional languages are data-driven, so in the functional implementation of the algorithm the computational model of the programming language guarantees that all computations are made on demand. Since we cannot assume this in an imperative setting, we have to find a suitable ordering of calculations, so that all dependencies are solved and all values are calculated before they are used. For the small example shown here this is an easy task. But consider an algorithm with about 20 productions and various dependencies between them. Finding a suitable order of calculation is a strenuous and error-prone venture. Solving this problem is one of the tasks of the compiler described in the next section.

5.4 Compiling ADP notation to C

In the previous section we showed how to derive the traditional recurrences in a systematic way, as an intermediate step towards an implementation of an ADP algorithm in an imperative programming language, such as C. The C program can be tested systematically against the Haskell prototype, a procedure that guarantees much higher reliability than ad-hoc testing. Still, the main difficulties with this approach are twofold: It proves to be time consuming to produce a C program by hand that is equivalent to the Haskell prototype. Furthermore, for sake of efficiency, developers are tempted to perform ad-hoc yield size analysis and use special combinators in the prototype. This introduces through the back door the possibility of subscript errors otherwise banned by the ADP approach. The ADP compiler currently under development eliminates both problems.

Aside from parsing the ADP program and producing C code, the core of the compiler implements yield size and dependency analysis, and performs the translation steps described in the previous section. 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 `globsim` is shown in Figure 10.

We also 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 program designer using the functional embedding is no longer committed to delicate tuning efforts.

The compiler, at the time of this writing, successfully handles the simple examples presented in this article. However, it is not yet complete, as there are some more ambitious goals to persue, such as automating the efficiency annotation.

6 Conclusion

6.1 Review of the ADP development method

The systematic method of program development in the ADP framework was specified rather rigidly in Section 4. It comprises the steps $\mathcal{A} - \Sigma - \mathcal{E} - \mathcal{G}$, named after the four main constituents of ADP. We discuss the experiences with the use of this method, some of which we hope to have already conveyed to the reader.

In the design of the signature Σ, we perform the fundamental case analysis of our problem domain. This records all the details that may be entering our

```
void calc_alignment(int i, int j)
{
   int v1, v2, v3, v4, v5, v6, v7;

   if ((j-i) == 1) {                      /* nil <<< (char '$') */
      if (z[j] == '$') {
         v1 = 0;
      } else { v1 = MINVAL; }
   } else { v1 = MINVAL; }

   if ((j-i) >= 2) {             /* d <<< achar ~~~ p alignment */
      if (z[i+1] != '$') {
         v2 = tbl_alignment[i+1][j-(i+1)-1] + gap(z[i+1]);
      } else { v2 = MINVAL; }
   } else { v2 = MINVAL; }

   if ((j-i) >= 2) {             /* i <<< p alignment ~~~ achar */
      if (z[j] != '$') {
         v3 = tbl_alignment[i][j-1-i-1] + gap(z[j]);
      } else { v3 = MINVAL; }
   } else { v3 = MINVAL; }

   if ((j-i) >= 3) {   /* r <<< achar ~~~ p alignment ~~~ achar */
      if ((z[i+1] != '$') && (z[j] != '$')) {
         v4 = tbl_alignment[i+1][j-1-(i+1)-1] + w(z[i+1], z[j]);
      } else { v4 = MINVAL; }
   } else { v4 = MINVAL; }

   v5 = v3 > v4 ? v3 : v4;
   v6 = v2 > v5 ? v2 : v5;
   v7 = v1 > v6 ? v1 : v6;

   if ((j-i) >= 1) { tbl_alignment[i][j-i-1] = v7; }
}

void mainloop()
{
   int i, j;

   for (j=0; j<=n; j++)
      for (i=j; i>=0; i--)
         calc_alignment(i, j);
   printf("%d", tbl_alignment[0][n-0-1]);
}
```

Fig. 10. C-Code produced by the ADP compiler for grammar `globsim` with algebra `wgap`

objective of optimization. Describing all evaluation algebras $\mathcal{E}$ as Σ-algebras makes sure that they all are homomorphic images of the enumeration algebra T_Σ. This has two major methodical benefits. First, Definition 6 formulates the proof obligation that must be met to show that a particular evaluation algebra satisfies Bellman's Principle. Second, it assures that various evaluation algebras work correctly with the same algorithm, as all aspects of evaluation are encapsulated within the algebras.

In the design of the grammar, we concentrate exclusively on the top-down problem decomposition. Different types of sub-problems are cast as different nonterminals, while their potential evaluation is restricted to the operators provided by Σ. There is no chance to include ad-hoc aspects of evaluation into the grammar.

Not even the most systematic development makes testing superfluous. In our case, testing is greatly aided by the enumeration, counting and prettyprinting algebras. The use of the enumeration algebra enables inspection of the search space actually traversed by the algorithm. The counting algebra sometimes indicates that the search space of the program is much larger than we would expect - in this case the grammar contains redundancies which we may not have chosen consciously, and whose avoidance is essential if we are also interested in near-optimal solutions.

The beneficial use of several algebras has led us to introducing a product operation on algebras, denoted `***`. The product algebra (`alg1 *** alg2`) simply operates on pairs of answer values. However, the product of the choice functions is defined in a nontrivial way, leading to a number of interesting properties that are outside the scope of the present paper. For example, if `opt` is an optimizing evaluation algebra, then (`opt *** count`) gives us the count of co-optimal solutions together with the optimal score, (`opt *** enum`) or (`opt *** pretty`) gives us the optimal candidate together with its score. Using a second table entry, we get backtracing for free. And there is more flexibility, as algebra products may be nested.

Altogether, our experience is that the ADP framework leads to a productive and rewarding reconciliation of mathematical rigor with the programmer's intuition about dynamic programming.

All the elements of methodical guidance available heretofore [CLR90,Sch01] re-appear in our framework, often in a more rigorous fashion. There is only one major point where ADP re-adjusts the classical view: Cormen et al. advocate, as the initial step, the study of how the structure of an optimal problem solution is composed from its subproblems. In the light of ADP, this is misleading: The *structure* of a solution guides its evaluation, but does not determine its optimality. Optimality is solely in the eye of the evaluation algebra – change the algebra, and a different structure will be found to be optimal. The structure of all solutions in the search space, optimal or not, is defined by the grammar. The classical view certainly implies this. However, it can not formulate it adequately, as the notion of an explicit representation of candidates is lacking.

6.2 Extensions in theory and practice

Dynamic programmers have discovered many tricks that improve the efficiency of a DP algorithm. In the ADP framework, many such tricks turn into techniques, which can be formalized as transformation schemes on grammars and algebras, can be taught and re-used. This aspect is elaborated in [GM02]. Interestingly, the transition from a global to a local optimization problem (as in our string comparison example) is such a transformation. Once the global problem has been solved, its local version comes (intellectually) for free.

We have occasionally touched upon the case where one is interested in more than a single answer. One may ask for all co-optimal answers, or all near-optimal answers within a certain threshold of optimality. In such a context, the redundancy or ambiguity of the grammar must be understood and controlled. This is a nontrivial issue studied in [Gie00b]. A general technique to enumerate solutions in order of optimality is described in [GM02].

In non-trivial examples, it often occurs that different types of subproblems have different types of answers. This leads to the use of many-sorted evaluation algebras, where a separate choice function is associated with each sort of answer. A case in point is the grammar `wuchty98`. There is no theoretical difficulty associated with this extension.

6.3 Future work

In our practical work, we have used several techniques that have not yet been studied theoretically, such as attributes associated with nonterminals of the tree grammar, or parsers that rely on precomputed information. While the Haskell embedding of ADP is a convenient test bed for such ideas, their adequate integration in the theoretical framework is not obvious. When teaching ADP to our students, we strictly discourage ad-hoc extensions borrowed from the Haskell background.

Moving on from sequences to more structured data such as trees or two dimensional images, the new technical problem is to provide a suitable tabulation method. To this end, we are currently studying local similarity problems on trees [HTGK03].

An interesting extension in the realm of sequential data is to consider language intersection and complement. We may allow productions with *and* and *butnot* operators, written `u = v &&& z` and `u = v --- z`. The former means that an input string can be reduced to u if it can be reduced to both v and z. The latter means that it can be reduced to u if it can be reduced to v, but not to z. While this clearly extends beyond the realm of context free yield languages, it is easy to implement in the parser, and quite useful in describing complicated pattern matching applications.

The most urgent task we are pursuing at the time of this writing is the formal definition of an ADP language version 1.0, and the completion of the compiler. The Haskell embedding is easy to use, and efficient enough for many realistic applications when using the Glasgow Haskell compiler GHC. Functional programming and Haskell are quite popular in the computer science community today, but not so in biology and bioinformatics. Understanding and even developing an algorithm written in ADP notation does not require any Haskell expertise – it can be based solely on the notion of yield grammars, evaluation algebras, and ADP notation. However, in the case of an – albeit trivial – error, the programmer is confronted with the error handling of the underlying language system, which is not aware of the ADP concepts. Hence, the current dependence on Haskell is a severe limitation in the application field for which ADP was originally developed.

6.4 Acknowledgements

Many people have contributed to the development of the ADP approach by profound discussion, critical comments, and application challenges. We acknowledge the contributions of Dirk Evers, Matthias Höchsmann, Stefan Kurtz, Enno Ohlebusch, Jens Reeder, and Marc Rehmsmeier. Morris Michael has compiled the ADP web site, where the reader is welcome to experiment with ADP in an interactive fashion. It can found in the education department of the Bielefeld Bioinformatics Webserver at `http://bibiserv.techfak.uni-bielefeld.de/`.

References

[AHU83] A. V. Aho, J. E. Hopcroft, and J. D. Ullman. *Data Structures and Algorithms.* Addison-Wesley, Reading, MA, USA, 1983.

[Aku00] T. Akutsu. Dynamic programming algorithms for RNA secondary structure prediction with pseudoknots. *Discr. Appl. Math.*, 104:45–62, 2000.

[BB88] G. Brassard and P. Bratley. *Algorithmics: Theory and Practice.* Prentice-Hall, 1988.

[BD62] R. Bellman and S.E. Dreyfus. *Applied Dynamic Programming.* Princeton University Press, 1962.

[Bel57] R. Bellman. *Dynamic Programming.* Princeton University Press, 1957.

[BM93] R. S. Bird and O. de Moor. From dynamic programming to greedy algorithms. In B. Möller, editor, *State-of-the-Art Seminar on Formal Program Development.* Springer LNCS 755, 1993.

[BM97] R. S. Bird and O. de Moor. *Algebra of Programming.* Prentice Hall, 1997.

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

[CLR90] T.H. Cormen, C.E. Leiserson, and R.L. Rivest. *Introduction to Algorithms.* MIT Press, Cambridge, MA, 1990.

[Cur97] S. Curtis. Dynamic programming: A different perspective. In R. Bird and L. Meertens, editors, *Algorithmic Languages and Calculi*, pages 1–23. Chapman & Hall, London, U.K., 1997.

[DEKM98] R. Durbin, S. Eddy, A. Krogh, and G. Mitchison. *Biological Sequence Analysis.* Cambridge University Press, 1998.

[EG01] D. Evers and R. Giegerich. Reducing the conformation space in RNA structure prediction. In *German Conference on Bioinformatics*, 2001.

[Eve03] D. Evers. *RNA Folding via Algebraic Dynamic Programming.* PhD thesis, Faculty of Technology, Bielefeld University, 2003.

[Gie98] R. Giegerich. A declarative approach to the development of dynamic programming algorithms, applied to RNA folding. Report 98–02, Technische Fakultät, Universität Bielefeld, 1998.

[Gie00a] R. Giegerich. A Systematic Approach to Dynamic Programming in Bioinformatics. *Bioinformatics*, 16:665–677, 2000.

[Gie00b] R. Giegerich. Explaining and controlling ambiguity in dynamic programming. In *Proc. Combinatorial Pattern Matching*, pages 46–59. Springer LNCS 1848, 2000.

[GKW99] 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.

[GM02] R. Giegerich and C. Meyer. Algebraic Dynamic Programming. In Hélène Kirchner and Christophe Ringeissen, editors, *Algebraic Methodology And Software Technology, 9th International Conference, AMAST 2002*, pages 349–364. Springer LNCS 2422, 2002.

[Got82] O. Gotoh. An improved algorithm for matching biological sequences. *J. Mol. Biol.*, 162:705–708, 1982.

[GS88] R. Giegerich and K. Schmal. Code selection techniques: Pattern matching, tree parsing and inversion of derivors. In *Proc. European Symposium on Programming 1988*, pages 247–268. Springer LNCS 300, 1988.

[GS02] R. Giegerich and P. Steffen. Implementing algebraic dynamic programming in the functional and the imperative programming paradigm. In E.A. Boiten and B. Möller, editors, *Mathematics of Program Construction*, pages 1–20. Springer LNCS 2386, 2002.

[Gus97] D. Gusfield. *Algorithms on Strings, Trees, and Sequences.* Cambridge University Press, 1997.

[HTGK03] M. Höchsmann, T. Töller, R. Giegerich, and S. Kurtz. Local similarity in RNA secondary structures. In *Proceedings of the IEEE Bioinformatics Conference (CSB)*, pages 159–168, 2003.

[Hut92] G. Hutton. Higher order functions for parsing. *Journal of Functional Programming*, 3(2):323–343, 1992.

[Meh84] K. Mehlhorn. *Data structures and algorithms.* Springer Verlag, 1984.

[MG02] C. Meyer and R. Giegerich. Matching and Significance Evaluation of Combined Sequence-Structure Motifs in RNA. *Z.Phys.Chem.*, 216:193–216, 2002.

[Mit64] L. Mitten. Composition principles for the synthesis of optimal multi-stage processes. *Operations Research*, 12:610–619, 1964.

[Moo99] O. de Moor. Dynamic Programming as a Software Component. In M. Mastorakis, editor, *Proceedings of CSCC, July 4-8, Athens.* WSES Press, 1999.

[Mor82] T. L. Morin. Monotonicity and the principle of optimality. *Journal of Mathematical Analysis and Applications*, 86:665–674, 1982.

[NPGK78] R. Nussinov, G. Pieczenik, J.R. Griggs, and D.J. Kleitman. Algorithms for loop matchings. *SIAM J. Appl. Math.*, 35:68–82, 1978.

[PJ03] S. Peyton Jones, editor. *Haskell 98 Language and Libraries – The Revised Report.* Cambridge University Press, April 2003.

[RE98] E. Rivas and S. Eddy. A dynamic programming algorithm for RNA pseudoknots. In *Poster Presentation, ISMB98*, 1998.

[Sch01] U. Schöning. *Algorithmik.* Spektrum Akad. Verl., Heidelberg, 2001.

[Sed89] R. Sedgewick. *Algorithms.* Addison-Wesley, 2nd edition, 1989.

[SKM$^+$03] A. Sczyrba, J. Krüger, H. Mersch, S. Kurtz, and R. Giegerich. RNA-related tools on the Bielefeld Bioinformatics Server. *Nucl. Acids. Res.*, 31(13):3767–3770, 2003.

[Sni92] M. Sniedovich. *Dynamic Programming.* Marcel Dekker, 1992.

[SW81] T. F. Smith and M. S. Waterman. The identification of common molecular subsequences. *J. Mol. Biol.*, 147:195–197, 1981.

[WFHS99] S. Wuchty, W. Fontana, I. L. Hofacker, and P. Schuster. Complete suboptimal folding of RNA and the stability of secondary structures. *Biopolymers*, 49:145–165, 1999.

[ZS81] M. Zuker and P. Stiegler. Optimal computer folding of large RNA sequences using thermodynamics and auxiliary information. *Nucleic Acids Res.*, 9(1):133–148, 1981.

[ZS84] M. Zuker and S. Sankoff. RNA secondary structures and their prediction. *Bull. Math. Biol.*, 46:591–621, 1984.