Universität Bielefeld

Technische Fakultät
AG Praktische Informatik

RNA Folding via Algebraic Dynamic Programming

Dissertation

Dirk J. Evers

RNA Folding via Algebraic Dynamic Programming

Zur Erlangung des akademischen Grades eines
Doktors der Naturwissenschaften
der Universität Bielefeld
vorgelegte
Dissertation

von
Dirk J. Evers

Bielefeld, im März 2003

Practical Informatics
Faculty of Technology
Bielefeld University
D-33594 Bielefeld
Germany

dirk@TechFak.Uni-Bielefeld.DE

Dedicated to the Memory of Rebecca Wagner

Contents

List of Figures

1 Motivation and Overview

1.1 Problem Statement

The aim of this thesis is to apply the framework of Algebraic Dynamic Programming (short ADP) to a well known problem with established significance in Bioinformatics, to implement the current 'state of the art', and finally to go one step further and solve one of the open problems. Ab initio RNA secondary structure folding of a single sequence was chosen because it perfectly fit the bill. First, because of the compactness of the field, showing a clear path from the first description of the Nearest Neighbor model by Tinoco and others in a Nature paper from 1971, via the base pair maximization algorithm by Nussinov and others in 1978, to the first efficient and complete solution to the free energy minimization problem by Zuker and Stiegler in 1981, and then on to a number of further refinements to date (Tinoco et al., 1971; Nussinov et al., 1978; Zuker and Stiegler, 1981; Wuchty et al., 1999; Lyngsø et al., 1999). Second, there is a clear description of an open problem in a paper by Zuker and Sankoff in 1984, that to our knowledge has not been solved yet (Zuker and Sankoff, 1984). It is the problem of reducing the structure space of a given RNA to saturated secondary structures whose helices can not be extended any further by legal base pairs.

1.2 Related Work

Related work for this thesis naturally falls into three areas: RNA thermodynamics, RNA folding algorithms, and Dynamic Programming systems.

- The Nearest Neighbor RNA secondary structure thermodynamics model was conceived by Tinoco and refined by Freier, Turner, Ninio, and others (Tinoco et al., 1971; Borer et al., 1974; Papanicolaou et al., 1984; Turner et al., 1988; Walter et al., 1994; Xia et al., 1998). A review on the current knowledge of how RNA folds can be found in (Tinoco and Bustamante, 1999).

- Ab initio single sequence RNA secondary structure folding algorithms based on Dynamic Programming, Simulated Annealing, or Genetic Algorithms where published by Sankoff, Zuker, Nussinov, Hofacker, Steger, Lyngsø, Gultyaev, van Batenburg, Gouy, and others (Zuker and Stiegler, 1981; Zuker and Sankoff, 1984; Gouy et al., 1985; Zuker, 1989; Hofacker et al., 1994; Gultyaev et al., 1995; Wuchty et al., 1999; Tahi et al., 2002). The current review on relevant algorithms for RNA folding plus various utilities, for instance visualization programs, is (Zuker, 2000).

- Relevant papers on Dynamic Programming for this thesis are by Bellmann, Birney, Durbin, and Giegerich, (Bellman, 1957; Birney and Durbin, 1997; Giegerich, 1998; Evers et al., 1999; Giegerich et al., 1999; Giegerich, 1999).

1.3 Summary of Results

This thesis shows the applicability of Algebraic Dynamic Programming (ADP) to significant scientific problems in Bioinformatics as well as demonstrating the practical use of ADP as a rapid prototyping language for Dynamic Programming algorithms. Particular emphasis was laid on the foundations of ADP and its application to RNA secondary structure prediction by choosing simple example applications, such as the Base Pair Maximization algorithm by Nussinov (Nussinov et al., 1978). All known algorithms for ab initio, single sequence RNA secondary structure folding via Dynamic Programming – to the best knowledge of the author – are implemented rigorously in this thesis. To solve new and more complex problems the ADP method is extended by so called top-down and bottom-up attribute combinators. Finally, a new algorithm for 'saturated RNA secondary structures' is presented and evaluated concerning its effectiveness in structure space reduction. The recurrences for this algorithm were published in advance and presented at the German Conference on Bioinformatics in 2001 (Evers and Giegerich, 2001). The number of Dynamic Programming tables *times* their dimension needed per algorithm resembles the complexity of the problems solved (see Fig. 1.1).

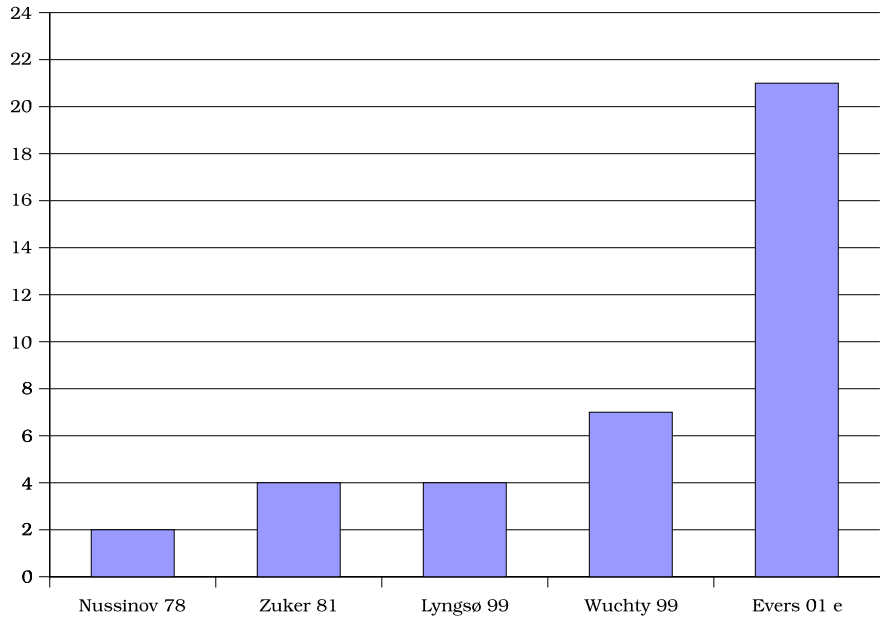


Figure 1.1: Sum of DP tables × table dimension needed per algorithm

The saturated folding algorithm utilizes 3 one-dimensional and 9 two-dimensional tables of size n to solve the problem in $O(n^3)$ time, where n is the length of the RNA sequence.

1.4 Structure of the Thesis

The thesis first presents the relevant details of RNA biochemistry and the Nearest Neighbor model of RNA secondary structure formation together with structural representations found in the literature. The following chapter explains the inner workings of Algebraic Dynamic Programming using Ruth Nussinovs base pair maximization algorithm as the

running example by first presenting the standard imperative implementation based directly on the recurrences taken from the literature, then translating them into an ADP parser and corresponding evaluation functions, and finally deriving the recurrences from the ADP implementation and comparing them to the original version.

From this point onwards various aspects of Dynamic Programming are elucidated by analyzing the major RNA folding algorithms in the order of publication. The basis of all RNA DP algorithms following the Nearest Neighbor model is Zukers algorithm published in 1981 (Zuker and Stiegler, 1981). Here, we discuss the ambiguity of the search space as a trade-off for table reduction. The next chapter has the canonical solution with unique structures as proposed by Wuchty and others in 1999 (Wuchty et al., 1999). The solution to the internal loop efficiency problem by Lyngsø and others is demonstrated in chapter 7; the various aspects of the decomposition of the search space and corresponding evaluation functions are discussed here (Lyngsø et al., 1999). In chapter 8 the Nearest Neighbor model is refined by adding dangling ends. The thesis culminates in demonstrating the solution to the 'no lonely pairs' and the saturated folding problem.

1.5 Future Research

In the future the saturated RNA folding algorithm could be extended to allow for dangling ends and co-axial stacking. The combination with a DP algorithm for pseudoknot prediction could lead to improved performance and selectivity of pseudoknot finding in large RNAs such as ribosomal RNAs. Algorithms of this complexity would have to be implemented as regular DP algorithms with a fill and traceback stage, preferably in the C or C++ languages to ensure a maximum of efficiency.

The ADP technology is under permanent development and refinement. Automatic generation of standard recurrences and an ADP compiler as well as the generalization of input streams to multiple dimensions are well under way. With these enhancements in place it will be possible to combine the sequence alignment or tree alignment of secondary structures with RNA folding, making it possible to perform complex approximate searches on RNA sequences and structures.

1.6 Acknowledgments

I would like to thank my supervisor Robert Giegerich for his enthusiasm concerning RNA secondary structure prediction and his inspiring work on Algebraic Dynamic Programming. I am indebted to Andreas Dress for showing me what "all the others" have to offer to Computational Biology and Bioinformatics. All that I know about RNA I learned from Gerhard Steger. Special thanks go to Stefan Kurtz who has patiently endured many questions and given much of his time to improve my understanding of computer science in general, and how to write a thesis in particular. I want to thank Jens Stoye for letting me work on the *Rose* project. The algorithms presented in this thesis would have many errors left in them without the invaluable help of Carsten Meyer and Ellen Latz. My fellow graduates Ute Bauer, Sebastian Böcker, Christian Büschking, Chris Schleiermacher, Alexander Sczyrba, and many others at Bielefeld University supported me in many more ways than I can possibly express in this short space.

Special thanks go out to my colleagues at Exelixis Deutschland GmbH and Exelixis Inc. Jochen Scheel, Torsten Will, Axel Küchler, Andreas Vogel, Gordon Stott, Yisheng Yin, and Angela Law for enduring and encouraging me while writing this thesis.

And finally, to Stephanie for supporting me and for never giving up believing that the day would finally come . . .

The work presented in this thesis was carried out while being a doctoral fellow at the Center for Interdisciplinary Research on Structure Formation (FSPM) and the Research Group for Practical Informatics at Bielefeld University. As a member of the “Graduate College on Structure Formation Processes” funding was generously provided by the German Research Council. The thesis was written while working for Exelixis Deutschland GmbH.

2 Conventions & Definitions

Some basic definitions and notational conventions used throughout this thesis are given in this chapter. Typographic conventions were introduced as needed to fit different circumstances as the thesis was written. Therefore, they are described in arbitrary order. The definitions are in logical order.

2.1 Typographic Conventions

Definitions are numbered and may have a titel enclosed in parentheses in **bold sans serif font**. The definition ends with the symbol □. Defined or indexed words may also be printed in **bold sans serif font**. Species are printed in *italics*. Haskell code fragments and examples are in `constant width font`. Comments and explanations of Haskell code are in small roman font. Formulae are in *math italics font* with data values and symbols represented in `constant width font`. Diagrams are printed in sans serif font enclosed by a frame.

2.2 Basic Definitions & Notations

Definition 1 «alphabet» An alphabet A is a finite, non-empty set of symbols.

RNA alphabet The RNA alphabet is defined as $A_{RNA} = \{\mathsf{A}, \mathsf{C}, \mathsf{G}, \mathsf{U}\}$. We will call the elements bases or nucleotides.

□

Definition 2 «sequence» A sequence is the concatenation of symbols from an alphabet A, where ε denotes the empty sequence.

Concatenation Sequences are concatenated by juxtaposition, as in $s = uv$.

Sequence partitions In any partitioning of $s = uvw$, into (possibly empty) subsequences u, v, and w, u is called a prefix, v a subsequence, and w a suffix of s.

Powerset The set of all finite sequences of symbols over an alphabet A is defined as $A^* = \bigcup_{i \geq 0} A^i$ with $A^0 = \{\varepsilon\}$ and $A^{i+1} = \{vw \mid v \in A,\ w \in A^i\}$.

Slice notation Let s_i denote the ith element in the sequence s, and $s_{k,l} = s_{k+1} \cdots s_l$ the subsequence between k and l of length $l - k$. For instance, given the sequence $s = \mathsf{GAGA}$ (for clarity: $s = {}_0\mathsf{G}_1\mathsf{A}_2\mathsf{G}_3\mathsf{A}_4$), then $s_1 = \mathsf{G}$, and $s_{1,3} = \mathsf{AG}$.

□

2.3 Programming Languages

The programming language used throughout this thesis is Haskell98, a pure functional programming language (Peyton Jones, 2003) (see also http://www.haskell.org). Every piece of written text in this thesis can either be interpreted as LaTeX code resulting in the print you are now reading, or interpreted as Haskell code resulting in executables of the programs presented and discussed herein. Therefore, the complete body of work that produced this thesis is reproducibly contained within, with no possibility for transription errors in programs. The downside of this is, that any errors you might find in the programs or the text are truly intellectual failures on my part.

3 Concepts & Models in Biology & Biochemistry

This chapter starts with a general introduction to RNA, giving examples of RNA functions and including the chemical forces governing RNA structure formation as well as the experimental methods of RNA structure determination. The choice of modeling the secondary structure level is motivated and followed by a formal definition and discussion of its properties. The substructures inherent in the model are then discussed. Different representations of RNA secondary structures are introduced and a general representation is chosen. The chapter ends with a mapping of the free energy terms in the Nearest Neighbor model to the secondary structure elements.

3.1 Introduction

Ribonucleic acid (RNA) has many known and probably many still unknown functions. In proteinbiosynthesis messenger (mRNA), transfer (tRNA), and ribosomal (rRNA) RNA work together. Ribonucleoproteins process pre-mRNA, and in some viruses RNA also fulfils the role as hereditary molecule generally assigned to DNA. Moreover, RNA acts in multiple ways from regulation of transcription to catalysis in the spliceosome. Still, RNA is commonly associated with its elementary function, as passive carrier of genomic information that finds its realization in proteins through biosynthesis.

The discovery of RNA autocatalytic activity in the 1980s brought about the renaissance of the "RNA World Hypothesis" which suggests the existence of a pre-biotic purely RNA based system of replication and catalysis, enabling evolution toward the complex system of DNA replication and protein catalytic activity we see today (Cech, 1986). In this light, catalytic RNAs probably form some of the oldest systems still functioning in living cells. The recent discovery, that transesterification – polypeptide formation from the aminoacylated tRNA to the peptide chain – in the ribosome is mediated by its rRNA, is further prominent supporting evidence to this hypothesis (Nissen et al., 2000).

3.1.1 RNA Functions

This section presents a short overview of the different classes and subclassifications of RNA with their typical properties and a short description of their function.

Messenger RNA (mRNA) is the intermediate message passing molecule in what is known as the central dogma of molecular biology (Crick, 1970).

$$\text{DNA} \rightarrow \text{(m)RNA} \rightarrow \text{Protein}$$

Accounting for about 1% to 3% of the total amount of RNA in a cell, mRNAs serve as intermediate template in protein biosynthesis.

Ribosomal RNA (rRNA) forms the largest RNA/protein complex in the cell – the ribosome – making up approximatly 85% of all RNA. It translates the genomic information encoded in the mRNA into proteins. With the discovery that no protein is near enough to the catalytic site of transesterification the ribosome turns out to be a ribozyme (see below) (Nissen et al., 2000).

Transfer RNA (tRNA) is the biomolecular manifestation of the genetic code — the decoder of protein sequence information in the genome. It has a characteristic L-shape as tertiary (see figure 3.1) and a cloverleaf as secondary structure (see 3.9). In its amino-acylated form (loaded with its amino acid) it associates with the ribosome and the mRNA to decode a codon on the mRNA via its anticodon. It has a share of 10% to 15% of the RNA.

Transfer messenger RNA (tmRNA) releases ribosomes stuck to mRNA without a stop codon. It combines two different functions in one molecule. One part acts as a tRNA continuing the halted process and releasing the mRNA and the remaining tRNA. Then, the second part translocates to the mRNAs position to be translated, suchthat a special tag is appended to the incomplete polypeptide chain marking it for immediate degradation (Zwieb et al., 1999).

Small nuclear RNA (snRNA) for example are components in the spliceosome that catalyse excision of introns in pre-mRNAs. Small nucleolar RNA (snoRNA) recognize specific sites in pre-ribosomal (immature) RNAs thereby invoking modification, for instance methylation (Eliceiri, 1999).

7 S RNA is a major part of the signal recognition particle (SRP) that initiates secretion of proteins through the endoplasmatic reticulum (Lütcke, 1995).

Ribozymes conduct self-cleavage and self-ligation of RNA, in fact acting as a true enzyme. The discovery of the catalytic properties of RNA by Sidney Altman and Thomas R. Cech was awarded with the Nobel prize in 1989 (Altmann, 1990; Cech, 1990).

3.1.2 RNA Structure Formation

The ability to perform the wide variety of tasks mentioned in the previous section is achieved by the plasticity of the RNA molecule.

RNA is a polymere of nucleotides made of a nucleoside and a phosphate building block. The nucleosides, in turn, consist of a ribose and one of the four bases, adenine (A), guanine (G), uracil (U), or cytosine (C). Note, that by convention any nucleic acid sequence is given in 5'-3' direction of the phosphate-ribose backbone, as in 5′-ACGU-3′, the so called **primary structure**.

With 10 degrees of freedom per nucleotide RNA is flexibel enough to bend back onto itself and engage in intramolecular interaction. The most frequent being base pairing via double or triple Hydrogen bonds, the so called Watson-Crick pairs A-U and G-C, and the wobble pair G-U. Other types of base pairs are possible, but less frequent. The pattern of paired and unpaired regions of an RNA make up its **secondary structure**.

In RNA, consecutive base pairs lead to the formation of a regular A-form helix with 11 base pairs per (right) turn and 30 Å height. The helix has a deep and narrow major groove and a wide and shallow minor groove. A B-form helix, as in DNA, is sterically impossible due to the 2'OH group of the ribose. The left turning Z-form ist possible, but has not been found in vivo.

Non-nested structures called pseudoknots (see figure 3.2), occur seldomly but regularly, for instance in rRNAs. Special triple- or quadruple-strand interactions may stabilize the relative positions of helices. As example, hydrogen bonds between the 2'OH group of the ribose and single stranded nucleotides are known to exist in some tRNAs. Contacts as these are termed tertiary structure elements.

Under physiological conditions an RNA molecule will fold into a compact tertiary structure presenting a direct hydration shell that is quite different from DNA. Protein-nucleic acid recognition is dependent on irregularities in helix formation caused by non-standard base pairing and loop formation. Thus, the form, i.e. the **tertiary structure** of a given molecule mediates its function. For a recent review on RNA folding see (Ferré-D'Amaré and Doudna, 1999).

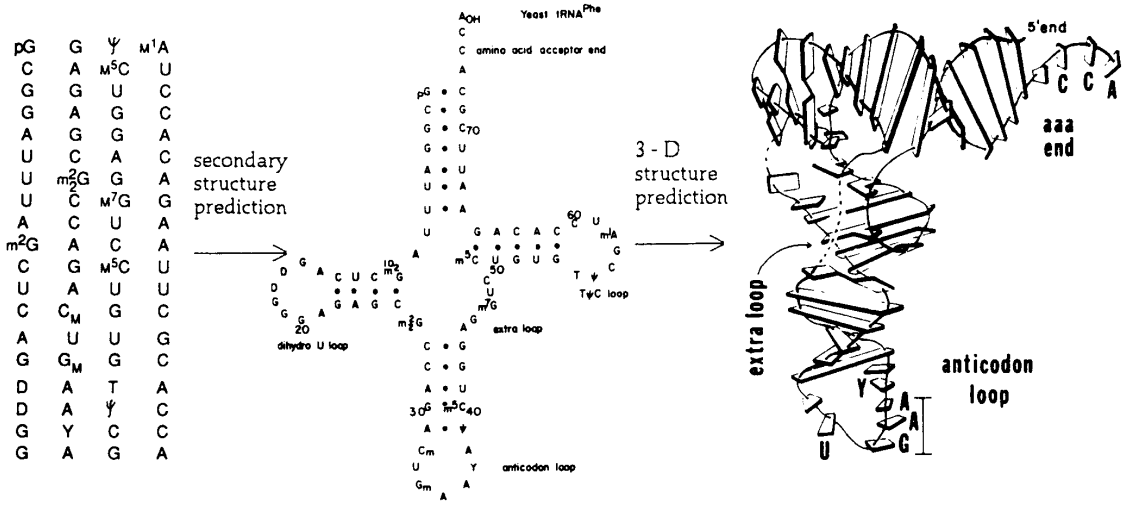


Figure 3.1: RNA structure prediction
from (Turner et al., 1988)

3.1.3 RNA Structure Determination

To fully understand the function of an RNA molecule one has to know its tertiary structure. Nuclear magnetic resonance spectroscopy (NMR) and X-ray diffraction can deliver this information, but only for relatively small structures of up to 100 nucleotides length. Getting even this kind of information is cumbersome at best. NMR does not deliver the resolution necessary to accurately place the atoms, and X-ray diffraction needs an RNA crystal, which is very hard to grow. That is why only few (about 10) RNA structures have been verified to date. Examples are $tRNA^{Phe}$ of *Saccharomyces cerevisae* (see figure 3.1) (Kim et al., 1974; Robertus et al., 1974), and the hammerhead ribozyme (see figure 3.2) (Pley et al., 1994; Scott et al., 1995).

To find other ways to elucidate the form of a given RNA molecule, one has to understand the way it folds. This process is governed by the speed in which the elements, making up a structure are able to form: the kinetics of RNA structure formation.

The structure formation processes work in parallel. Helices compete for dominance of their part of the nucleotide strand, eventually teaming up to form more complex structures and finally succumbing to the power of the equilibrium. For a thorough discussion see (Tinoco and Bustamante, 1999).

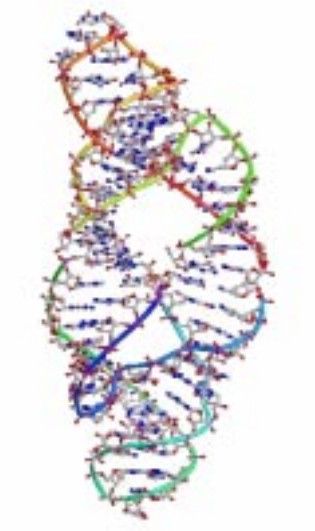


Figure 3.2: The Hammerhead Ribozyme
from (Berman, n.d.)

time: stacking < helices < simple loops < multiple loops < tertiary interactions

We will use this picture to explain the simplifications that lead to the formulation of the *RNA secondary structure folding problem*. The reason being that it lets us change from the parallel kinetic perspective to a hierarchical one.

stacking → helices → simple loops → multiple loops → tertiary interactions

Considering that secondary and tertiary elements are simply split in this ordering we may simplify even further without much loss of generality.

primary structure → secondary structure → tertiary structure

RNA Secondary Structure Folding Problem Given the primary structure of a single RNA molecule, infer the thermodynamically optimal secondary structure.

This simplified view discounts three factors.

1. The rate of synthesis the RNA polymerases achieve is far lower than the kinetics of RNA structure formation.
2. The degradation rate of the RNA in question may be higher than the refolding rate leading to the equilibrium distribution.
3. Tertiary structure influence is partially ignored.

There are models and algorithms that take sequential folding and metastable states into account (Mironov and Lebedev, 1993; Gultyaev et al., 1995; Schmitz and Steger, 1996). We will narrow the analyses in this thesis to the pure secondary structure problem. In the light of incomplete and error-prone thermodynamic data, the problem is hard to solve, requiring exploration of the secondary structure landscape.

3.2 RNA Secondary Structure Representation

We will now introduce the traditional formal terminology on RNA secondary structures. This will enable us to base the section on biochemical models (3.3) on this formal framework, and later-on demonstrate the reincarnation of these concepts in the ADP framework.

Definition 3 «primary structure» A primary structure p is given by a sequence over the **RNA alphabet** $A_{RNA} = \{\mathtt{A},\mathtt{C},\mathtt{G},\mathtt{U}\}$ denoting the bases of an RNA molecule in 5'-3' orientation. □

```
data Base = A | C | G | U
```

We need to access base positions in constant time, so we choose to define RNA sequences as integer indexed arrays of bases.

```
type RNA = Array Int Base
```

In the following algorithms, we will assume that the primary structure is stored in a global array `rna` of type `RNA`. The function definition for `rna` is given in Appendix A.8.

```
rna :: RNA
```

In the most general way nucleotide contacts would be defined as a set of base pairs without restrictions. $B \subseteq P \times P$ where P is the set of positions in the RNA sequence. Alas, this would include pseudoknots and triple strand interactions (see section 3.1.2), which have to be excluded for combinatorial reasons. We will discuss this thoroughly in chapter 4.

Definition 4 «secondary structure» A hydrogen bond of base p_i and p_j is denoted by $i \bullet j$ with $i < j$. A secondary structure s_p is a set of index pairs of p. We require that s_p satisfies the following requirements for all $i \bullet j \in s_p$ and $k \bullet l \in s_p$:

- $i = k \Leftrightarrow j = l$
  This is the uniqueness restriction stating that a base can only pair once.
- $i < k < j \Leftrightarrow i < l < j$
  The pseudoknot restriction ensures that pairs are nested properly.

□

The first part of the definition sets up the general set of base pair contacts. The constraints exclude triple-strand interactions and pseudoknots.

This enables us to define the type of a region of RNA and the position of a base, as well as the type of a base pair and a base pairing test. In section 3.4 on structural representations we will present a function to test for legal secondary structure.

```
type Region = (Int,Int) -- A region is an indexed subword in slice notation
                        -- (see definition "sequence").
type BasePos = Int

type BasePair = (BasePos,BasePos)

pair :: Base -> Base -> Bool
pair A U = True
pair U A = True
pair C G = True
pair G C = True
pair G U = True
pair U G = True
pair _ _ = False
```

The nearest neighbor model introduced in the next section is based on the loop decomposition illustrated in figure 3.3.

Definition 5 «loop» Given a pair $i \bullet j$ we call any position $x \in [i, j]$ *interior* of $i \bullet j$. If x is interior of $i \bullet j$ and there exists no pair $k \bullet l$ such that $i < k < x < l < j$ we call x *directly interior* to, or *accessible* from $i \bullet j$ and denote this relation as $x \triangleleft i \bullet j$. The loop $l_{i \bullet j}$ closed by $i \bullet j$ consists of all accessible nucleotides. All the nucleotides not enclosed in any loop form the *exterior* l_{ext}. The size of a loop, denoted by $|l_{i \bullet j}|$, is the number of unpaired accessible nucleotides. The degree of a loop is the number of accessible base pairs, and is written $l^{\circ}_{i \bullet j}$. □

```
type LoopSize   = Int
type LoopDegree = Int
```

Every secondary structure s_p has a unique decomposition into $|s_p| + 1$ loops.

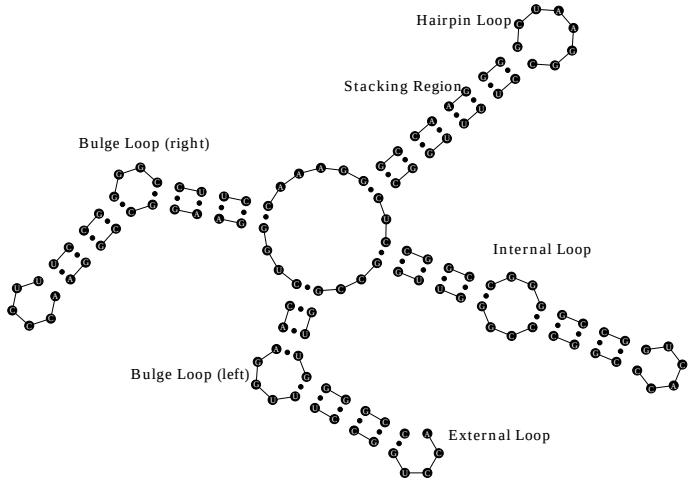


Figure 3.3: Loop decomposition of an imaginary RNA

3.3 Nearest Neighbor Model & Thermodynamics

The major thermodynamic effects involved in RNA structure formation will be introduced, together with the assumptions underlying the standard free energy minimization model.

```
module FreeEnergy where

 import RNA
 import EnergyTables
 import Array
 import Utils
```

3.3.1 Molecular Forces

The molecular forces giving rise to structure formation are the same for all biomolecules and of course RNA is no exception. The governing force in the RNA world is the conjugated π-electron system of parallely stacked planar bases in a helix. Other important forces are the formation of hydrogen bonds, the hydrophobic effect, and electrostatic and van der Waals interactions. It is important to note that a stable folded structure can only be achieved if the energy gain from folding is large enough to compensate for the loss of conformational entropy. We will see these two opposing forces in all the energy terms present in the nearest neighbor model.

3.3.2 Free Energy

The Gibbs Free Enthalpy – or short free energy – is the maximal amount of energy in a chemical reaction that can be utilized. We denote ΔG^0_X as the change in free enthalpy under standard conditions ($25°C$, every component $1Mol$) while forming a structure X from an unfolded molecule. By convention negative values of ΔG denote energy release, whereas positive ΔG-values indicate that the process needs to receive energy from the environment. The free energy is defined via the Gibbs-Helmholtz equation.

$$\Delta G^0 = \Delta H^0 - T \cdot \Delta S^0 \tag{3.1}$$

Where ΔH^0 is the reaction enthalpy, and $-T \cdot \Delta S^0$ the entropic term. The meaning of this equation lies in a combination of the 1. and 2. laws of Thermodynamics. This combination allows to formulate whether a process will be spontaneous ($\Delta G < 0$) or not, in a single physical measure. $\Delta G = 0$ implies thermodynamic equilibrium. It is important to note that the free energy makes no statement of the kinetics, i.e. speed, of a reaction.

3.3.3 The Nearest Neighbor Model

The basic assumption of all energy models is, that it is possible to sum over some energetic property of certain constituents of the chemical structure in question. There are some choices to be made in such a setting.

- What are the constituent substructures of the model?
- How is their energy contribution modelled?

Early RNA energy models counted the number of base pairs, later the model was extended to count the number of hydrogen bonds formed (Nussinov et al., 1978). However, it turned out that these models were too simple to accurately predict anything other than tightly structured molecules such as certain tRNAs. A more sophisticated model was needed, based on experimental data of small oligonucleotides. The previously ignored stacking interactions turned out to be a major factor in gaining accuracy in the energy computations. They necessitate a different model, based on loops rather than base pairs.

As a consequence, the free energy of an RNA secondary structure is assumed to be the sum over the free energies of its loops.

$$\Delta G(s_p) = \Delta G(l_{ext}) + \sum_{i \bullet j \in s_p} \Delta G(l_{i \bullet j}) \tag{3.2}$$

```
type Energy = Float -- All energies are given in kcal/mol.
```

In the following we present the model and data as implemented in *mfold* version 2.3 (Zuker, 2003).

Conformational Entropy of Loops

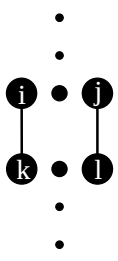
The entropic terms for a loop of type X are tabulated for sizes up to 30, beyond that loop energies are extrapolated using the Jacobson-Stockmayer rule which states that entropy loss should increase logarithmically (Jacobson and Stockmayer, 1950).

$$\Delta G_X(n) = \Delta G_X(30) + 1.75RT \ln \frac{n}{30} \tag{3.3}$$

We introduce the table independent part as a penalty function for all simple loops.

```
jacobson_stockmayer :: LoopSize -> Energy
jacobson_stockmayer size = 1.079 * logBase e ((fromIntegral size) / 30.0)
```

Stacking Region



Experimental measurements of the free energy contributions of specific stacking regions are crucial for the computation of the overall free energy of a structure. Therefore, these energies are the most carefully evaluated and complete data sets in the model (Borer et al., 1974; Xia et al., 1998).

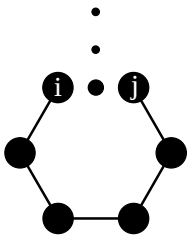
$$\Delta G_{SR} = \Delta G_{match}(i \bullet j, k \bullet l) \tag{3.4}$$

The free energy of a stacking region depends on the four nucleotides forming the loop. A simple 4 dimensional table lookup function `t_sr` suffices. The function `dg_sr` reduces the table lookup to loop parameters for $i \bullet j$.

```
dg_sr :: BasePair -> Energy
dg_sr (i,j) = t_sr (rna!i) (rna!(i+1)) (rna!(j-1)) (rna!j)

t_sr :: Base -> Base -> Base -> Base -> Energy

t_sr A A U U = -0.9
t_sr A C G U = -2.1

 ...

t_sr U U A G = -0.5
t_sr U U G G = -0.4

t_sr pi pk pl pj = error "t_sr: parameters not in table: (" ++
                         show pi ++ "," ++ show pj ++ ") (" ++
                         show pk ++ "," ++ show pl ++ ")"
```

Note that stacking interactions between non-canonical base pairs are possible in principle, but not accounted for in this table. The complete table lookup function can be found in section A.3.

Hairpin Loop

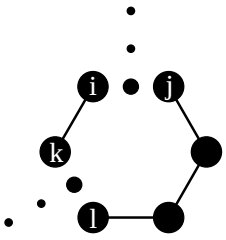


The free energy of a hairpin loop is composed of a destabilizing entropic term that is purely size dependent, a stabilizing term of stacking interactions between the closing base pair and the adjacent mismatching bases, as well as a bonus term for a number of specific tetra loops. Hairpins of size three are considered too tightly packed to be awarded stack-mismatch energies.

$$\Delta G_{HL} = \Delta G_{size}(|i \bullet j|) + \Delta G_{mismatch}(i \bullet j) + \Delta G_{bonus}(i, \ldots, k) \qquad (3.5)$$

```
dg_hl :: BasePair -> Energy

dg_hl (i,j) | len == 3 = size
            | len == 4 = size + mismatch + bonus
            | len  > 4 = size + mismatch
            | otherwise = error "dg_hl: size < 3"
  where
  len      = j - i - 1
  size     = ent_hl len
  bonus    = tetra_hl (slice (i-1,j))                    -- slice defined in Appendix
  mismatch = t_hl (rna!i) (rna!(i+1)) (rna!(j-1)) (rna!j)
```

Bulge Loop

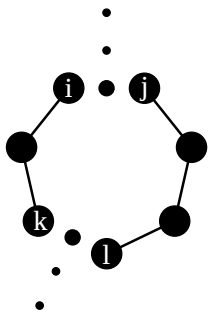


Bulge loops are considered to be mostly destabilizing, because the free energy contribution is mainly governed by the size of the bulge. In the special case of a single base bulge – where the base is assumed to be pushed into the stack – helix geometry is not distorted, so regular stacking region energies are assigned as well.

$$\Delta G_{BL} = \Delta G_{size}(|i \bullet j|) + \Delta G_{match}(i \bullet j, k \bullet l) \qquad (3.6)$$

```
dg_bl :: BasePair -> BasePair -> Energy

dg_bl (i,j) (k,l) | len == 1  = size + match
                  | len  > 1  = size
                  | otherwise = error "dg_bl size < 1"
  where
  size  = ent_bl len
  match = t_sr (rna!i) (rna!k) (rna!l) (rna!j)
  len   = (k - i) + (j - l) - 2
```

Internal Loop

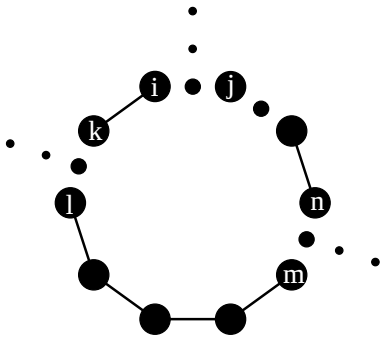


Interior loops have two main opposing influences. Stabilization is via stack-mismatch interaction of the closing base pairs. Destabilization of the loop is governed by the size of the bulges. Lopsidedness of the loop was discovered to be a further destabilizing influence by Ninio et al. and is accounted for in a separate term (Papanicolaou et al., 1984).

$$\Delta G_{IL} = \Delta G_{size}(|i \bullet j|) + \Delta G_{mismatch}(i \bullet j) + \Delta G_{mismatch}(k \bullet l) + \Delta G_{asym}(i \bullet j, k \bullet l) \tag{3.7}$$

```
dg_il :: BasePair -> BasePair -> Energy

dg_il (i,j) (k,l) = size + mismatch + asym
  where
  size     = ent_il (sl + sr)
  mismatch = t_il (rna!i) (rna!(i+1)) (rna!(j-1)) (rna!j)
           + t_il (rna!k) (rna!(k-1)) (rna!(l+1)) (rna!l)
  sl       = k - i - 1
  sr       = j - l - 1
  asym     = ninio sl sr
    where

    ninio :: LoopSize -> LoopSize -> Energy

    ninio sl sr = min 3.0 (lopside * 0.3)
      where
      lopside = fromIntegral (abs (sl - sr))
```

Multiple Loop



The energy of a multiple loop is governed by destabilization according to the Jacobson-Stockmayer formula, as in the simple loops. The stabilizing terms however, have to be represented differently.

In simple loops with $l^\circ = 2$, i.e. bulge and interior loops, which do not form a circular loop as the pictograms might suggest, but continue as a slightly "bloated" helix, the partners in the scenario are fixed. They are the two helices, which like to co-axially stack on top of each other for stability if the loop is small enough, and the unpaired neighboring bases, called dangles, which stack ontop of the helix in a frustrated attempt to continue its stabilizing influence.

These stabilizing effects also take place in multiple loops, but the helices present have a choice between stacking on their right and left neighbors, or none if they cannot find a stack in a loop with an odd number of helices, or the helices are more than one base apart. Moreover, a single unpaired base between two helices also has to decide which helix to stack onto. Therefore, dangling end energies have to be modeled for single bases, in contrast to mismatch stacking energies in simple loops. Going beyond that, co-axial stacking of helices has also been implemented in (Mathews et al., 1998) as a further model refinement.

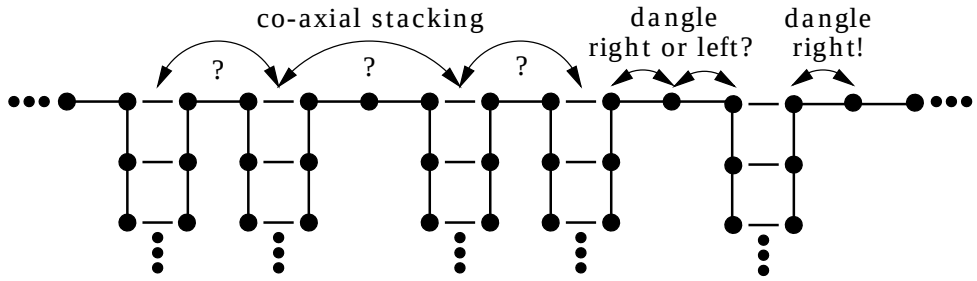


Figure 3.4: Co-axial stacking and dangling end contributions in loops with $l^{\circ}_{i \bullet j} > 2$

The general equation for the free energy of multiple loops consists of three different sub-terms.

$$\Delta G_{ML} = \Delta G_{size}(|i \bullet j|) + \min_{k,l,n,m \,\triangleleft\, i \bullet j} \{\Delta G_{dangle}(k \bullet l), \Delta G_{coaxial}(k \bullet l, m \bullet n)\} \tag{3.8}$$

Figure 3.4 illustrates the necessity to solve a localized minimization problem in the case of dangling ends, and a chained one in the co-axial stacking case. Without loss of generality and to retain compatibility with *mfold* 2.3, we will restrict our model to the dangling end case presented below.

```
dg_dangle :: BasePair -> [BasePair] -> Energy

-- (i,j) is the closing base pair.
-- p is the first helix in the loop.
-- ps are the rest.

dg_dangle (i,j) (p:ps) = share + (dangles p ps)
  where

-- Checking for dangling ends between the closing base pair and the first base pair in the loop.

  share :: Energy
  share | dist == 0 = 0.0                          -- no unpaired bases between helices
        | dist == 1 = min (dg_dl p) (dg_dli (i,j)) -- minimization case
        | otherwise = (dg_dl p) + (dg_dli (i,j))   -- the helices are far enough apart
    where                                          -- for two dangling ends
    dist :: LoopSize
    dist = (fst p) - i - 1

  dangles :: BasePair -> [BasePair] -> Energy

-- In this case the dangling end contributions of the last
-- helix in the loop and the closing base pair are computed.

  dangles x [] = share
    where
    share | dist == 0 = 0
          | dist == 1 = min (dg_dr x) (dg_dri (i,j))
          | otherwise = (dg_dr x) + (dg_dri (i,j))
      where
      dist = j - (snd x) - 1

-- The free energy of the dangling ends of two neighboring
-- helixes x and y.

  dangles x (y:ys) = share + (dangles y ys)
    where
    share | dist == 0 = 0
          | dist == 1 = min (dg_dr x) (dg_dl y)
          | otherwise = (dg_dr x) + (dg_dl y)
      where
      dist = (fst y) - (snd x) - 1
```

The Jacobson-Stockmayer formula is not computable for partial loops, and so the size dependent energy term for multiple loops is modeled as a simple affine cost model.

$$\Delta G_{size} = a + b \cdot n + c \cdot k \tag{3.9}$$

Here a is the cost for initiating a multiple loop, b is the cost for every unpaired base, and c the penalty for every start of a helix, i.e. the degree of the multiple loop. This linear function underestimates the entropic loss for small loops and overestimates it for large ones. We will discuss the reason for its use in the standard model in the next chapter.

```
dg_ml :: BasePair -> [BasePair] -> Energy

dg_ml (i,j) ps = a + b * n + c * k + (dg_dangle (i,j) ps)
  where
  a = 4.7
  b = 0.4
  c = 0.1
  k = fromIntegral (1 + length ps)
  n = fromIntegral (len (i,j) ps)
    where
    len (i,j) (p:ps) = (fst p) - i - 1 + count p ps
      where
      count x   []   = j - (snd x) - 1
      count x (y:ys) = (fst y) - (snd x) - 1 + count y ys

dg_dr :: BasePair -> Energy
dg_dr (i,j) = t_dr (rna!i,rna!j) (rna!(j+1))

dg_dli :: BasePair -> Energy
dg_dli (i,j) = t_dr (rna!j,rna!i) (rna!(i+1))

dg_dl :: BasePair -> Energy
dg_dl (i,j) = t_dl (rna!(i-1)) (rna!i,rna!j)

dg_dri :: BasePair -> Energy
dg_dri (i,j) = t_dl (rna!(j-1)) (rna!j,rna!i)
```

External Loop

Due to the fact that the external loop is not constrained by a closing base pair there are no entropic influences, i.e. the nucleotide chain is able to twist freely into any conformation. Thus, the destabilization term is zero. Local stabilizing influences are the same as in multiple loops.

$$\Delta G_{EL} = \min_{k,l,n,m \,\triangleleft\, i \bullet j} \{\Delta G_{dangle}(k \bullet l), \Delta G_{coaxial}(k \bullet l, m \bullet n)\} \tag{3.10}$$

The function `dg_el`, as in the multiple loop case above, implements the dangling end energy model.

```
dg_el :: [BasePair] -> Energy

dg_el [] = 0.0                          -- The unfolded chain
dg_el (p:ps) = share + (dangles p ps)
  where
  (i,j) = bounds rna
  share | dist == 0 = 0
        | otherwise = dg_dl p
    where
    dist = (fst p) - i

  dangles x [] = share
    where
    share | dist == 0 = 0
          | otherwise = dg_dr x
      where
      dist = j - (snd x)
```

```
dangles x (y:ys) = share + (dangles y ys)
  where
  share | dist == 0 = 0
        | dist == 1 = min (dg_dr x) (dg_dl y)
        | otherwise = (dg_dr x) + (dg_dl y)
    where
    dist = (fst y) - (snd x) - 1
```

3.4 Structural Representations

Having fully specified and implemented all elements of the nearest neighbor model, we want to proceed to evaluate the free energy of a complete structure. Before we can do that, we need some form of representation of a complete RNA secondary structure.

3.4.1 Data Representations

The definition given in 3.2 suffices formally, but has no direct reference to the substructures introduced by the energy model. We want to capture the structural semantics implicit in the nearest neighbor model by directly representing the structural elements of the model in a data representation.

There are various data representations for RNA secondary structures used in working programs, as well as in the literature. The most prominent are the helix notation, the CT notation, and the dot-bracket notation. We will use the structure shown in figure 3.5 to explain their differences.

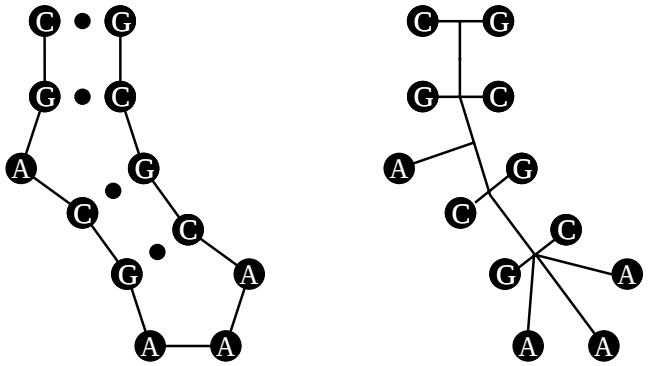


Figure 3.5: A simple hairpin structure and a hypothetical tree representation

Helix Notation is a compact representation of the base pair set as defined above. Only the outer base pair and the length of the helix are given. The helices are sorted by the 3'-base. A similar notation is used to constrain structures in the *mfold* program by Zuker (Zuker, 1989).

Example

```
> Energy = -0.8 Hairpin
GCGCAAAGCAGC
 1  12   2
 3   9   2
```

CT Notation is used as output format in the *mfold* package. The first line gives the length of the primary structure, the free energy, and its name. The following lines describe the structure in a column format sorted by sequence position.

1. sequence position
2. base type
3. index of predeceding nucleotide
4. index of successor
5. index of pairing base, 0 if unpaired
6. historical numbering

Example

```
 12 Energy = -0.8 Hairpin
  1  G   0   2  12   1
  2  C   1   3  11   2
  3  G   2   4   9   3
  4  C   3   5   8   4
  5  A   4   6   0   5
  6  A   5   7   0   6
  7  A   6   8   0   7
  8  G   7   9   4   8
  9  C   8  10   3   9
 10  A   9  11   0  10
 11  G  10  12   2  11
 12  C  11  13   1  12
```

Dot-bracket Notation is used in the RNAfold package by Hofacker (Hofacker et al., 1994), sometimes called Vienna notation in the literature, and used in enhanced form in the DCSE RNA alignment package by de Rijk (De Rijk and De Wachter, 1993).

Example

```
> Hairpin
GCGCAAAGCAGC
((((...)).)) (-0.8)
```

Loop Notation All formats lack a direct representation of structure elements in the nearest neighbor model and hide the tree structure inherent in the secondary structure. Therefore, we introduce a more concrete data type `Loop`.

```
data Loop =
    EL                [Loop]                | -- The External Loop
    HL  BasePos       Region       BasePos  | -- The Hairpin Loop
    SR  BasePos        Loop        BasePos  | -- The Stacking Region
    BL  BasePos Region  Loop        BasePos  | -- The Left Bulge
    BR  BasePos        Loop  Region BasePos  | -- The Right Bulge
    IL  BasePos Region  Loop  Region BasePos  | -- The Internal Loop
    ML  BasePos       [Loop]       BasePos  | -- The Multiple Loop
    SS                Region                | -- A Single Strand of unpaired bases
                                              --  in the external or multiple loop.
    NIL                                       -- The empty loop
                                              --  is used to represent structure fragments.
```

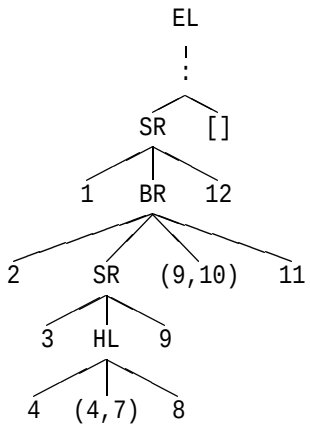


Figure 3.6: Tree representation of the simple hairpin in loop notation

`BasePos` and `Region` are defined in section 3.2.

Example

```
EL [SR 1 (BR 2 (SR 3 (HL 4 (4,7) 8) 9) (9,10) 11) 12]
```

The representation is complete when adding the corresponding primary structure and free energy value.

Dot-Bracket Notation Transformer The equivalence of the different representations is obvious. Nontheless, we present the mapping to dot-bracket notation for practical reasons. The transformer is needed to convert structures to a format used by other programs.

```
viennaFormat :: String -> String -> [(Loop,Energy)] -> IO ()
viennaFormat desc rnastr cs = do
  putStrLn ('>' : desc)
  putStrLn rnastr
  for [ db c ++ " (" ++ show e ++ ")" | (c,e) <- cs]
    putStrLn

    where
    db (NIL          ) = []
    db (SS     r     ) = dots r
    db (HL _   r   _ ) = "(" ++ dots r ++ ")"
    db (SR _   c   _ ) = "(" ++ db c ++ ")"
    db (BL _ r c   _ ) = "(" ++ dots r ++ db c ++ ")"
    db (BR _   c r _ ) = "(" ++ db c ++ dots r ++ ")"
    db (IL _ l c r _ ) = "(" ++ dots l ++ db c ++ dots r ++ ")"
    db (ML _   cs  _ ) = "(" ++ dotb cs ++ ")"
    db (EL     cs    ) = dotb cs

    dotb cs    = concat (map db cs)
    dots (i,j) = ['.' | k <- [1..j-i]]
```

Example

```
Structure> viennaFormat "Hairpin" "gcgcaaagcagc"
  [(EL [SR 1 (BR 2 (SR 3 (HL 4 (4,7) 8) 9) (9,10) 11) 12],-0.8)]

>Hairpin
gcgcaaagcagc
((((...)).)) (-0.8)
```

3.4.2 Secondary Structure Validation

The loop data type has a direct recursive structure and makes no provisions for illegal or unwanted combinations of loop elements. A more fine-grained representation could be utilized, but would be more tedious to handle without much benefit, as the parsers constructed in the following chapters will take care of semantic correctness. Nontheless, when reading structures from external sources checks for correctness are needed. In the following, the different constraints are explained and validation checks for them are presented.

- By definition, the external loop represents the root of the structure tree, but the data representation also allows it to be present within the structure.

```
checkEL :: Loop -> Bool

checkEL (EL cs) = cLoop cs
  where
  check (NIL          ) = True
  check (SS     _     ) = True
  check (HL _   _   _ ) = True
  check (SR _   c   _ ) = check c
  check (BL _ _ c   _ ) = check c
  check (BR _   c _ _ ) = check c
  check (IL _ _ c _ _ ) = check c
  check (ML _   cs  _ ) = cLoop cs
  check (EL     _     ) = False

  cLoop cs = and (map check cs)

checkEL    _    = False
```

- Substructure elements should not overlap. That is, index ranges from substructures should be completely contained in their parent structures. Furthermore, if the `NIL` element is not present in the structure the sequence of indices of the structure tree in left-to-right order should completely and uniquely cover the index range of the corresponding primary structure.

```
checkYield :: Loop -> Bool
```

  The invariant described above must hold.

```
checkYield c@(EL _) = [1..rnaLen] == (indexYield c)
  where
  indexYield (NIL           ) = error "NIL element encountered"
  indexYield (SS      r     ) = (iRegion r)
  indexYield (HL lb   r   rb) = lb : (iRegion r) ++ [rb]
  indexYield (SR lb   c   rb) = lb : (indexYield c) ++ [rb]
  indexYield (BL lb l c   rb) = lb : (iRegion l) ++ (indexYield c) ++ [rb]
  indexYield (BR lb   c r rb) = lb : (indexYield c) ++ (iRegion r) ++ [rb]
  indexYield (IL lb l c r rb) = lb : (iRegion l) ++ (indexYield c) ++
                                     (iRegion r) ++ [rb]
  indexYield (ML lb   cs  rb) = lb : iLoop cs ++ [rb]
  indexYield (EL      cs    ) = iLoop cs

  iRegion (i,j) = [(i+1)..j]
  iLoop cs      = concat (map indexYield cs)
```

- Given the primary structure, all base pairs indicated in the loop structure should be legal.

```
checkPair :: Loop -> Bool

checkPair (EL cs) = and (map check cs)
  where
```

```
check (NIL           ) = True
check (SS      _     ) = True
check (HL lb   _   rb) = isBasePair (lb,rb)
check (SR lb   _   rb) = isBasePair (lb,rb)
check (BL lb _ _   rb) = isBasePair (lb,rb)
check (BR lb   _ _ rb) = isBasePair (lb,rb)
check (IL lb _ _ _ rb) = isBasePair (lb,rb)
check (ML lb  cs   rb) = and ((isBasePair (lb,rb)) : (map check cs))
```

- Loops that can have a degree > 2, namely external and multiple loops, utilize a list to hold their elements. The unpaired nucleotides in these loops should be represented by a minimal number of single strand elements, i.e. two SS elements in succession should be forbidden. This ensures a canonical representation. Furthermore SS elements should only appear in said loop lists. Moreover, the lists are not allowed to be empty and multiple loops must contain at least two non-SS elements.

```
checkCanon :: Loop -> Bool

checkCanon (EL cs) = and ((noDoubleSS cs) : (map check cs))
  where
  noDoubleSS [] = False
  noDoubleSS (_:[]) = True
  noDoubleSS ((SS _):(SS _):_) = False
  noDoubleSS (_:b:cs) = noDoubleSS (b:cs)

  check (NIL          ) = True
  check (SS     _     ) = True
  check (HL _   _   _ ) = True
  check (SR _   c   _ ) = check c
  check (BL _ _ c   _ ) = check c
  check (BR _   c _ _ ) = check c
  check (IL _ _ c _ _ ) = check c
  check (ML _   cs  _ ) = and ((loopDegree cs > 2) : (noDoubleSS cs) : (map check cs))
```

The following functions are needed to convert to the (formal) base pair representation and to check properties such as accessibility and degree.

```
loopDegree :: [Loop] -> Int

loopDegree ls = 1 + (length (accessibleBPs ls))


accessibleBPs :: [Loop] -> [BasePair]

accessibleBPs ls = map toBasePair (filter (not . isSS) ls)
  where
  isSS :: Loop -> Bool
  isSS (SS _) = True
  isSS  _     = False


toBasePair :: Loop -> BasePair

toBasePair (HL lb   _   rb) = (lb,rb)
toBasePair (SR lb   _   rb) = (lb,rb)
toBasePair (BL lb _ _   rb) = (lb,rb)
toBasePair (BR lb   _ _ rb) = (lb,rb)
toBasePair (IL lb _ _ _ rb) = (lb,rb)
toBasePair (ML lb   _   rb) = (lb,rb)
toBasePair (EL _) = error "no closing base pair in EL"
toBasePair (SS _) = error "no closing base pair in SS"
toBasePair (NIL)  = error "no closing base pair in NIL"
```

A structure representation satisfying all constraints is called well-formed. There is a one-to-one correspondence to structures defined via the sets of pairings in 3.2. All structure representations in the sequel will be well-formed.

3.4.3 Graphical Representations

Different 2-D visualizations are used in the literature. Each emphasises a different aspect of RNA secondary structures.

The simplest graphic representation is the previously introduced dot-bracket notation. Comparisons between structures are feasible for the experienced user, but due to the sequential layout substructure identification is cumbersome for large structures. Nontheless, it is a very compact format and readily understood, especially for small structures.

Circle plots represent a structure by laying out the nucleotide sequence in a circle and drawing lines between pairing bases (see figure 3.7, algorithm A.7.1, and (Shapiro et al., 1984)). Sometimes curved lines are used to enhance the differences in small substructures.

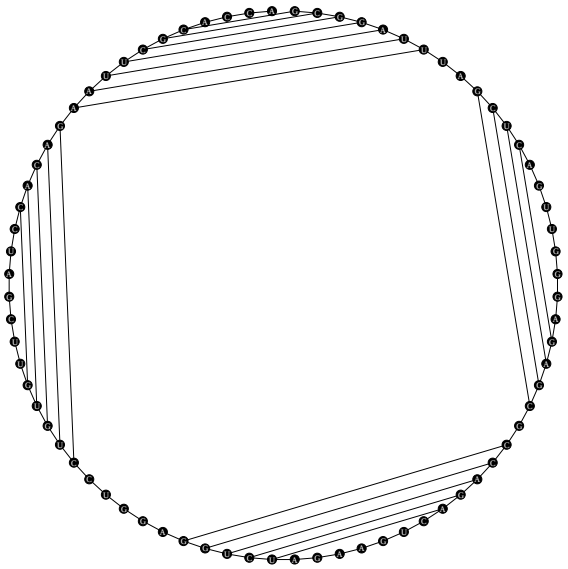


Figure 3.7: Circle plot of the $tRNA^{Phe}$ of *Saccharomyces cerevisae*

Substructure identification and comparison is easier than in dot-bracket notation, but only for substructures in the outer regions of the plot, i.e. for structures near the leaves of the RNA tree structure.

Mountain plots due to (Hogeweg and Hesper, 1984) (see picture 3.8) are a further elaboration on the dot-bracket notation. Here, the depth of a base pair in the RNA tree structure corresponds to the height of the plateau connecting the bases in the mountain plot. The algorithm used to produce figure 3.8 is presented in appendix A.7.2.

Polygon plots are the most often encountered visualizations in the literature (Shapiro et al., 1984; Bruccoleri and Heinrich, 1988). They are tree representations of the RNA secondary structure, usually with a fixed distance between the elements of a base pair. The simplest form, shown in figure 3.9, is the canonical polygon plot, with equal distances between neigboring elements and straight or right angle turns for substructures. These simple layout rules take no precautions to prevent overlapping of structure elements in the drawing.

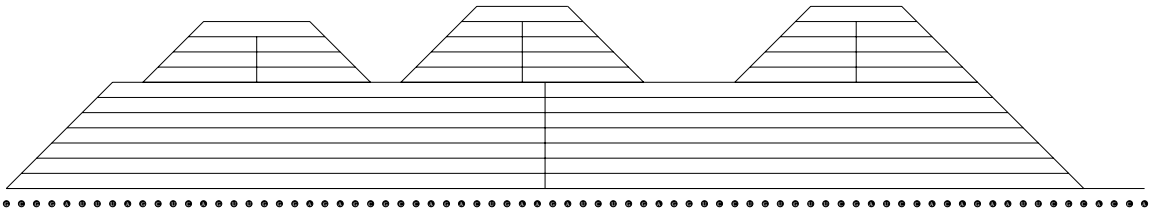


Figure 3.8: Mountain plot of the $tRNA^{Phe}$ of *Saccharomyces cerevisae*

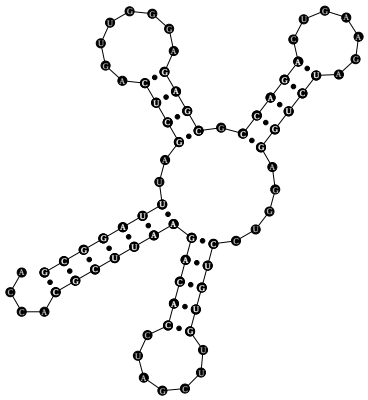


Figure 3.9: Polygon plot of the $tRNA^{Phe}$ of *Saccharomyces cerevisae*

The polygon plot algorithm was used throughout this thesis to construct all 2D-visualizations of RNA secondary structures. Note that the structural decomposition is identical to the loop decomposition of the nearest neighbor model. Figure 3.3 used to illustrate the loop decomposition in section 3.2 was created by introducing a spacer function in the algorithm above. The polygon plot algorithm with and without spacer can be found in appendix A.7.3.

3.5 Energy Yield

At the end of this chapter we are finally in the position to evaluate the free energy of a given RNA secondary structure according to the nearest neighbor model. All the needed energy functions were given in section 3.3 and the data representation of a secondary structure was chosen in 3.4.

```
module EnergyYield where

 import RNA
 import Structure
 import FreeEnergy
 import EnergyTables
 import Utils

 eYield :: Loop -> Energy

 eYield (SS r)                    = 0.0
 eYield (HL lb        _        rb) = dg_hl (lb,rb)
 eYield (SR lb        l        rb) = dg_sr (lb,rb)             + eYield l
 eYield (BL lb (_,y) l        rb) = dg_bl (lb,rb)  (y+1,rb-1) + eYield l
 eYield (BR lb        l (x,_) rb) = dg_bl (lb,rb) (lb+1,x)    + eYield l
 eYield (IL lb (_,y) l (x,_) rb) = dg_il (lb,rb)  (y+1,x)    + eYield l
 eYield (ML lb       ls       rb) = dg_ml (lb,rb) (accessibleBPs ls) + (sum (map eYield ls))
 eYield (EL          ls         ) = dg_el         (accessibleBPs ls) + (sum (map eYield ls))
```

This algorithm, like the validation algorithms in subsection 3.4.2, works via recursion over the structure tree. Therefore, its complexity in time is proportional to the number of nodes, i.e. loops, in the structure just as equation 3.2 suggests.

4 Algebraic Dynamic Programming

This chapter begins with a general introduction to Dynamic Programming including a short survey of its applications while emphasizing on Bioinformatics. Nussinov's RNA base pair maximization algorithm (Nussinov et al., 1978) is used as the running example to motivate various aspects of DP algorithm design. The separation of structure recognition and structure evaluation leads to the introduction of algebras and tree grammars. To support a declarative implementation of tree grammars, combinator parsing is used. The chapter ends by demonstrating the equivalence of the traditional recurrence relations to the ADP formulation. This is achieved by explicitely deriving the recurrences from the parser and algebra and comparing them to the original. The subsections 4.1.1, 4.1.4, and 4.1.5 closely follow (Evers and Giegerich, 2000), and the definitions in 4.2 follow (Giegerich, 2000). For a thorough introduction to ADP (including RNA secondary structure prediction and various alignment problems) please consult (Giegerich, 1998; Giegerich, 1999).

4.1 Dynamic Programming

4.1.1 The Role of DP in Bioinformatics

Dynamic Programming is a fundamental programming technique, applicable to great advantage where the input to a problem spawns an exponential search space in a structurally recursive fashion. If subproblems are shared and the principle of subproblem optimality holds, DP can evaluate such a search space in polynomial time. Classical application examples of DP are optimal matrix chain multiplication or the triangulation of a convex polygon (Cormen et al., 1990).

For very good reason, Dynamic Programming is the most popular paradigm in computational molecular biology. Sequence data — nucleic acids and proteins — are determined on an industrial scale today. The desire to give a meaning to these molecular data gives rise to an ever increasing number of sequence analysis tasks, which naturally fall into the class of problems outlined above. Dynamic Programming is used for assembling DNA sequence data from the fragments that are delivered by automated sequencing machines (Anson and Myers, 1997), and to determine the exon-intron structure of eucaroytic genes (Gelfand and Roytberg, 1993). It is used to infer function of proteins by homology to other proteins with known function (Needleman and Wunsch, 1970; Smith and Waterman, 1981), and it is used to predict the secondary structure of functional RNA genes or regulatory elements (Zuker, 1989). A recent textbook (Durbin et al., 1998) presents a dozen variants of DP algorithms in its introductory chapter on sequence comparison. In some areas of computational biology, Dynamic Programming problems arise in such variety that a specific code generation system for implementing such algorithms has been developed (Birney and Durbin, 1997).

4.1.2 DP Application Principles

When is Dynamic Programming the right strategy?

A basic property of the problems solved by DP algorithms is the recursive nature of the resulting solution search space. This enables an indexing scheme to be applied, to characterize subproblems via problem parameters in constant time. The usual indexing method is via input boundaries and tabulation, but hash algorithms are also employed whenever computed subsolutions are sparse in the index space.

«Search Space Recursion» The search space of a DP problem can be completely enumerated and indexed via recursion.

It may be argued, that the recursion property is very general and the indexing property holds for any function. This is certainly true, but nontheless these basic properties are presented here because they lay the foundation on which further constraining properties of DP algorithms are built, and as we will see later in this chapter, are also clearly represented in ADP.

The most often quoted feature of a DP problem is the principle of suboptimality (Bellman, 1957).

«Subproblem Optimality» An optimal solution to a DP problem is solely composed of optimal solutions to subproblems.

This property also includes problems that can be solved more efficiently by greedy algorithms. To discern from these a further property of the solution space is needed.

«Recursion Order Independence» There is no order of structural recursion that allows to make locally optimal decisions after partial evaluation.

This property ensures that the domain of subproblems partially overlap, making a greedy strategy impossible because it would have to prematurely commit to a specific subsolution.

The number of subproblems to be solved in a DP problem typically is polynomial in the length of the input, while the solution space is exponential. That is, the size of subproblem space is smaller than the solution space of the input. This means that there have to be multiple instances of the same subproblem that need to be solved during enumeration of the solution space. This is different from Divide-And-Conquer problems which generate unique subproblems during recursion. Furthermore, it enables to search the exponential search space in polynomial time by storing subsolutions in matrices that can be accessed in constant time.

«Subproblem Overlap» A DP problem exhibits recursion over identical subproblems.

The inherent properties of a DP problem are stated in a rather informal manner in this introduction. We will reinvestigate them in the ADP setting and give them a more formal (in our case: functional) representation.

4.1.3 An Example: Base Pair Maximization

The first application of a DP algorithm to an RNA folding problem was Nussinov's base pair maximization algorithm (Nussinov et al., 1978). We will use the version given in (Durbin et al., 1998) as our running example throughout this chapter.

```
module NussinovExample
where
import RNA
import Utils
import IOExts
```

Maximisation: The Fill Stage

Given a sequence rna return the maximum number of complementary base pairs of all legal secondary structures.

```
nussinov rna = do

  l     <- return (snd (bounds rna))
  table <- newIOArray ((1,1),(l,l)) 0

  -- Initialization
  for [2..l]
    (\i -> writeIOArray table (i,i-1) 0)
  for [1..l]
    (\i -> writeIOArray table (i,i) 0)

  -- Recursion
  for [ (j-x+1,j) | x <- [2..l], j <- [x..l] ]
    (fill rna table)

  -- Result
  n     <- readIOArray table (1,l)
  print n
    where

    -- Recurrence
    fill rna table (i,j) = do

      writeIOArray table (i,j) (
        maximum [
           access table (i+1,j),
           access table (i,j-1),
          (access table (i+1,j-1)) + (delta rna (i,j)),
           maximum (0:[
             (access table (i,k)) + (access table (k+1,j))
              | k <- [(i+1)..(j-1)]
           ])
         ]
       )
      where

      -- Base Pair Count
      delta rna (i,j) = if pair (rna!i) (rna!j) then 1 else 0
```

An execution example.

```
NussinovExample> nussinov "gggaaauuc"
3
```

Structure Retrieval: The Traceback Stage

To retrieve the secondary structure corresponding to the maximal base pair count a traceback algorithm can be employed on the filled matrix. This aspect of dynamic programming and its relation to ADP is *not* covered in this thesis, but is considered future work which is discussed in Section 1.5.

4.1.4 Understanding DP Recurrences

As demonstrated above, DP algorithms are hard to understand. The example given only utilizes a single table with a fairly simple recursive maximization formula. Moreover, thanks to Haskell's high level of abstraction this executable program is practically equivalent to the pseudo-code version it was derived from (see (Durbin et al., 1998)). Without further explanation, usually given beforehand, the rationale behind the recurrences is easily lost. For base pair maximization a short explanation is as follows.

There are four ways in which a optimal substructure can be extended. Add a base pair, add an unpaired base on the left or right, or combine two structures (see figures 4.1-4.3). A thorough discussion will be given shortly.

The development of a DP algorithm is a challenging task. Neither the bioinformatics literature (Waterman, 1995; Gusfield, 1997; Durbin et al., 1998), nor the computer science text books like (Cormen et al., 1990) give advice on how to systematically choose these matrices and construct the DP recurrences for a problem at hand. In all but the most simple cases, these recurrences are difficult to obtain and to validate. As demonstrated, even when given, they can be hard to understand. Their implementation is error-prone and time-consuming to debug, since a subtle subscript error may not make the program crash, but instead, quietly lead to a suboptimal answer every now and then.

4.1.5 Separation of Concerns

The intrinsic difficulty of DP recurrences can be explained by the following observation: For the sake of achieving polynomial efficiency, DP algorithms perform two subtasks in an interleaved fashion: The "first" phase is the construction of the search space, or in other words, the set of all possible answers. The "second" phase evaluates these answers and makes choices according to some optimality criterion. The interleaving of search space construction and evaluation is essential to prevent combinatorial explosion. On the other hand, describing both phases separately is a great intellectual relief. It is often used in instruction. To explain structure prediction, we may say: "First" we create all possible structures, "then" we score them all and choose the best. But this strategy lacks the amalgamation of both phases and suggests an algorithm of exponential runtime.

In the following subsection we will examine how search space construction and evaluation are interleaved.

4.1.6 Search Space

The search space recursion principle implies that the search space has the form of a rooted, node labeled tree. The subproblem overlap principle forces an exponential number of nodes to be identical, thereby reducing them to a polynomial number of distinct nodes and transforming the tree to a rooted, directed, acyclic graph by collapsing the identical nodes. The search space is represented by a matrix, where the different substructure extension cases are evaluated as follows.

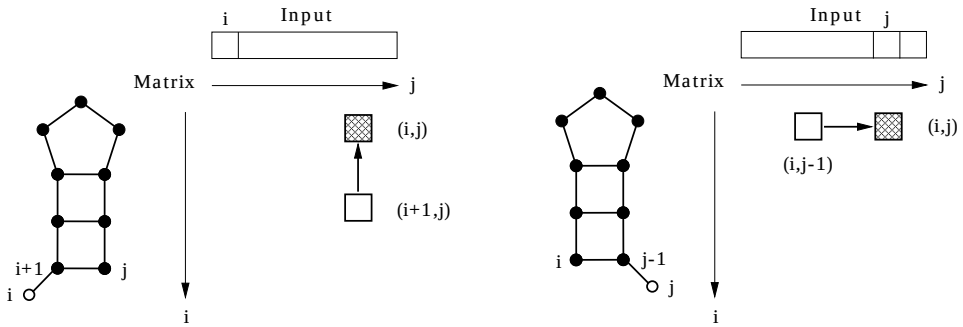
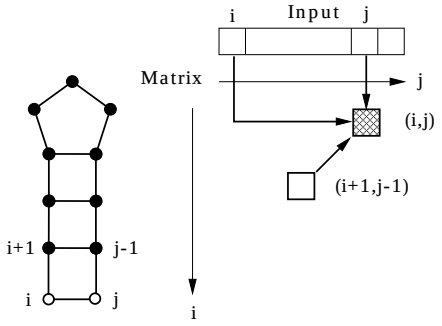


Figure 4.1: Adding a base

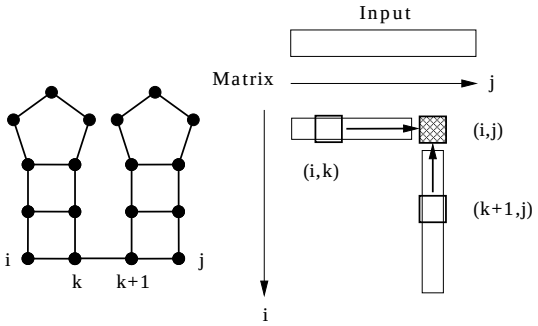
Adding a base We see that the base in question is actually not represented in the formula because it counts zero. This is one of the reasons why DP algorithms can be hard to understand.




```
access table (i+1,j-1)) + (delta rna (i,j)
```

Figure 4.2: Adding a base pair

Adding a base pair Here the matrix element representing the optimal substructure without p_i and p_j is accessed and the base pair count for p_i and p_j is added. Again, there are implicit assumptions in this formula. What if p_i and p_j do not pair?. The structure under evaluation is invalid, but because we are maximizing base pairs adding 0 will do no harm. In fact, we have evaluated the case of adding two single bases, which is already covered by case 1 applied twice.




```
maximum (0:[(access table (i,k)) + (access table (k+1,j)) | k <- [(i+1)..(j-1)]])
```

Figure 4.3: Combining two substructures

Combining two substructures Here, we find another example of substructure aggregation. In this case the structures in question are completely represented in the matrix, not consuming any of the input elements directly. Actually, this expression evaluates a range of structures of the same form. The maximization of these sub-expressions is superfluous, and could be misleading to the casual observer. In fact, the range of structures evaluated can be empty, which is why the functional implementation presented here concatenates a zero to the list. This may seem overly pedantic, but in optimization problems without simple (lower or upper) bounds – of which free energy minimization is an example – it can pose a problem.

Maximization Given that all considered substructures are optimal, we can select the optimum from the extended structures. This is where we reduce the space consumption of the algorithm from exponential to polynomial.

Recursion In our algorithm the search space recursion has been transformed into an iteration. The loop traversal over the search space, in form of the DP-matrix, has to ensure that all substructures needed to compute the structure under consideration have already been evaluated. This bottom-up approach needs to initialize elementary structures. In our case the smallest structures considered consist of a single base – corresponding to the main diagonal of the matrix – which by virtue is unpaired and therefore has a base pair count of zero.

The iteration then fills the diagonals from the main diagonal onwards according to the formulae discussed above.

```
for [ (j-x+1,j) | x <- [2..l], j <- [x..l] ] (fill rna table)
```

While evaluating a structure consisting of two bases a peculiar situation arises in the base pairing case: $(j-1) < (i+1)$. Which substructure is supposed to be represented by indices in descending order? It turns out, that the algorithm does not impose intervening unpaired nucleotides in a hairpin loop. Thus, an empty substructure is allowed here and the corresponding diagonal – only ever accessed in this special case – is initialized to zero. Optimizations like these, especially when accumulated, can be hard to understand.

Result The final result, that is the maximum number of base pairs possible on the input sequence, is now given in the matrix element representing the complete input sequence, namely element $(1,l)$.

4.1.7 Conclusion

Summing up what happens while evaluating a node, we have

1. inspection of input elements,
2. aggregation and validation of structures,
3. evaluation of structures, and
4. ordering of structures.

The aim of Algebraic Dynamic Programming is to separate these different tasks while retaining the effiency of a DP algorithm. The last two points will be dealt with in the following section on algebras, while the others are covered in the section on grammars and parsers.

4.2 Algebras

```
module Nussinov
where
import RNA
import Combinators
import Utils
```

In the previous section great care was taken to extract the different tasks performed in a DP algorithm in an amalgamated fashion. Figures 4.1-4.3 show that structures under evaluation (on the left) are represented as trees in the DP matrix (on the right). In fact, we can make this explicit by introducing an algebraic data type *str* to represent these tree patterns.

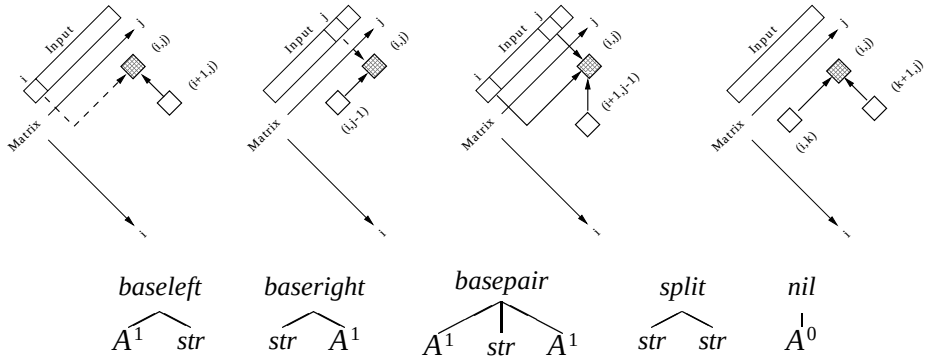


Figure 4.4: The operators in the term algebra *str*

4.2.1 Term Algebras

Definition 6 «T-algebra» An algebraic data type T is a type name and a family of typed function symbols, called the term algebra. We shall allow that these functions take additional arguments from A^*. An algebra that provides a function for each operator in T is a T-algebra. The interpretation t_I of a term t in a T-algebra I is obtained by substituting the corresponding function of the algebra for each operator. Thus, t_I evaluates to a value in the base set of I. □

Figure 4.4 shows the algebraic data type *str* for the base pair maximization algorithm and its relationship to the matrix fill stage. A straightforward interpretation of *str* is to introduce a direct representation as a fixed data type in Haskell.

```
data Str =
  BLeft  BasePos Str         |
  BRight         Str BasePos |
  BPair  BasePos Str BasePos |
  Split        Str Str       |
  Nil
  deriving Show
```

Note the similarity of the data type `Str` to the Nearest Neighbor data type `Loop` given in section 3.4. A different interpretation of *str* is as a family of functions counting the number of base pairs. Function `bpair` makes use of the sequence input `rna`, e.g. takes additional arguments from A^*.

```
bleft  _ x   = x
bright   x _ = x
bpair  i x j = if pair (rna!i) (rna!j) then x + 1 else x
split   x y  = x + y
nil          = 0
```

The number of structures can be computed as well. The *nil* operator has a different interpretation from the base pair algebra in this T-algebra although it has the same base set.

```
bleft  _ x   = x
bright   x _ = x
bpair  _ x _ = x
split   x y  = x * y
nil          = 1
```

In this way we can evaluate any given (well-formed) term, i.e. structure, in the algebra of choice as we did to compute the free energy in section 3.5. In the case of structure

counting this will trivially evaluate to 1. This kind of algebra only makes sense when we change focus from a single structure to the search space of a given input sequence – in this case counting the number of structures.

4.2.2 Evaluation Algebras

To compare structures we need to address the problem of how to define the structure space. The first step is to add a choice function to our algebra.

Definition 7 «evaluation algebra» An evaluation algebra is a T-algebra augmented by a choice function h. If l is a list of values of the algebra's value set, then $h(l)$ is a sublist thereof. We require h to be polynomial in $|l|$. Furthermore, h is called reductive if $|h(l)|$ is bounded by a constant. □

The evaluation algebras for our structure space now read as follows.

```
nussinovTermAlg = (BLeft,BRight,BPair,Split,Nil,id)
```

The term evaluation algebra uses the (non-reductive) identity as choice function.

```
nussinovBasePairAlg = (bleft,bright,bpair,split,nil,h)
  where
  bleft  _ x   = x
  bright   x _ = x
  bpair :: Int -> Int -> Int -> Int
  bpair  i x j = if pair (rna!i) (rna!j) then x + 1 else x
  split :: Int -> Int -> Int
  split   x y  = x + y
  nil          = 0::Int
  h :: [Int] -> [Int]
  h        []  = []
  h        xs  = [maximum xs]

nussinovBasePairEnum = (bleft,bright,bpair,split,nil,id)
  where
  bleft  _ x   = x
  bright   x _ = x
  bpair  i x j = if pair (rna!i) (rna!j) then x + 1 else x
  split   x y  = x + y
  nil          = 0
```

The base pairing evaluation algebra chooses the structure with the most base pairs.

```
nussinovCountAlg = (bleft,bright,bpair,split,nil,h)
  where
  bleft  _ x   = x
  bright   x _ = x
  bpair  _ x _ = x
  split   x y  = x * y
  nil          = 1
  h :: [Integer] -> [Integer]
  h        []  = []
  h        xs  = [sum xs]
```

Counting the number of structures is done by summing over the constituent substructure counts.

4.2.3 Conclusion

We have seen that it is possible to do various types of analyses on a given structure space. Now we need to address the problem of how to define the structure space in a declarative manner.

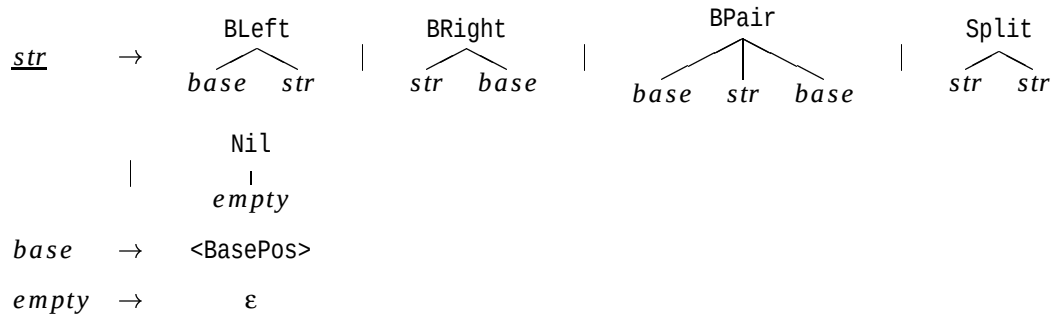
4.3 Grammars & Languages

The legal combination of substructures leads to the formation of a global structure covering the complete input. RNA secondary structures have been described by context free grammars (Sakakibara et al., 1994; Lefebvre, 1995; Searls, 1997) where the structures are represented by the derivation trees of a specific RNA parse. Unfortunately the language of derivation trees is implicit in this setup. Yet our aim is to capture the structure formation process in a DP problem and make it explicit in our algorithm. That is why we choose to represent our structures by a tree grammar.

4.3.1 Tree Grammars

Definition 8 «tree grammar» A tree grammar over an algebraic data type T has a set of nonterminal symbols, a designated axiom symbol, and a set of productions of the form $X \to t$ where X is a nonterminal symbol, and t is a tree pattern, i.e. a tree over T which may have nonterminals in leaf positions. Alternate tree patters in productions are interspersed by the symbol $\mid$ as in: $X \to t_1 \mid t_2$. The derivation relation on productions is denoted by $\to^*$. Therefore, the language of a nonterminal symbol X is given by $L(X) = \{t \in T \mid X \to^* t\}$. Consequently, the language of a tree grammar G is denoted by $L(G) = L(A)$ the language derived from the axiom symbol A. □

In our running example the resulting production is as follows.



Obviously *str* is the (underlined) axiom symbol. Thus, a legal RNA secondary structure in our model consists of exactly the alternative substructures discussed in Section 4.1.6. Note that basepairing has not been taken into account in `BPair`. This could be achieved here at the production level, but for reasons of similarity to the original algorithm and clarity will be implemented as a constraint in the resulting parser in Section 4.4.

4.3.2 Yield Grammars

An aspect we have not touched upon so far is the ordering of nodes in a tree pattern.

The graphical representation of patterns in *str* was chosen to suggest the obvious mapping on nonterminals in left-to-right order to an RNA sequence. Thus, resulting in a left-to-right ordering of terminal symbol leaves in the structure of a parse tree (see Figure 4.5). This is the convention we will use throughout the thesis, particularly in the implementation of the parser combinators given in Section 4.4. But in general this need not be the case.

As an aside, consider the problem of aligning two sequences (Needleman and Wunsch, 1970). One possible solution is to map the first seqence to left leaves in tree patterns, and the second sequence to right leaves. It is immediatly clear that the dual to this mapping is to reverse the second sequence and append it to the first, including a unique separating symbol, i.e. $s = x\$y^{-1}$. This retains the original left-to-right mapping of a single sequence. See Figure 4.6.

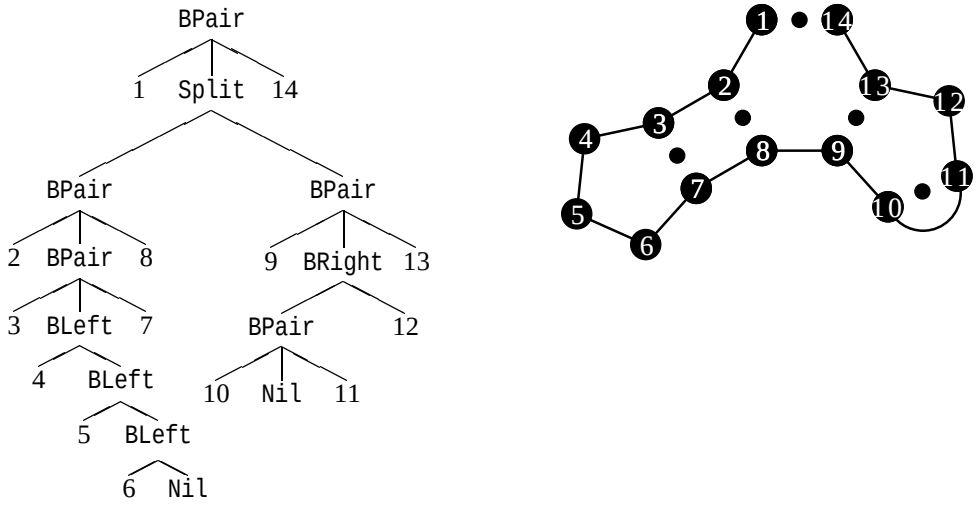


Figure 4.5: An example of a parse tree and its corresponding secondary structure

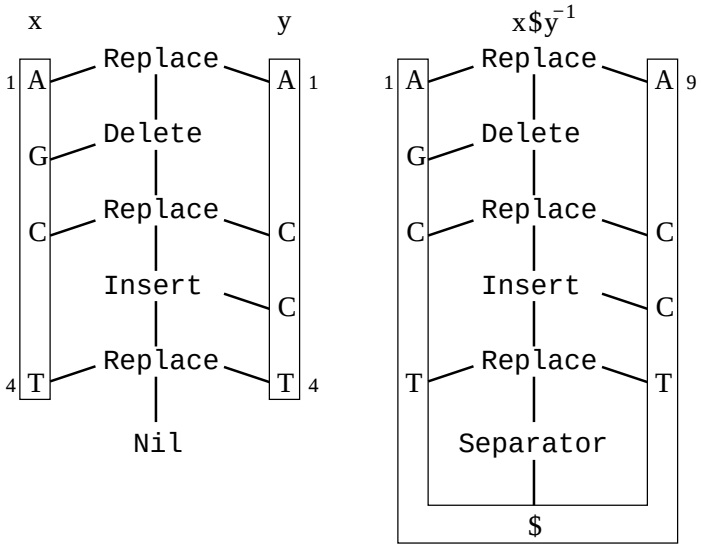


Figure 4.6: Two equivalent yield functions

Definition 9 «yield grammar» Given a tree grammar G over an algebraic data type T and an alphabet A, the yield function y is a homomorphism $y : L(G) \to A^*$. Then, the pair (G, y) is called a yield grammar. It defines the yield language $L(G, y) = y(L(G))$. □

4.3.3 DP Search Space

We can now define the search space of a Dynamic Programming problem as follows.

Definition 10 «yield parsing» Given a yield grammar (G, y) over A and $w \in A^*$, the search space of a Dynamic Programming problem is given by the the yield parsing problem: Find $P_G(w) := \{t \in L(G) \mid y(t) = w\}$. □

Structure recognition is computing the inverse of the yield function.

4.3.4 Blackboard Notation

We will now introduce a linear shorthand that is applicable in a parsing language context. Productions $X \to t$ are written as `X = t`, the alternate operator $t_1 \mid t_2$ is written as `t1 ||| t2`, the tree patterns are composed of the operator symbol $X \in T$ and the leaves $l_1, \ldots, l_n$ written as `X <<< l1 ~~~ l2 ~~~ l3` and pronounced "X applied to l1 and l2 and l3". The starting production is identified by the keyword `axiom`.

The grammar is now represented as:

```
nussinov = axiom str
  where
  str = bleft  <<< base ~~~ str          |||
        bright <<<          str ~~~ base |||
        bpair  <<< base ~~~ str ~~~ base |||
        split  <<<      str ~~~ str      |||
        nil    <<< empty
```

4.3.5 Conclusion

We have shown how to present DP recurrences as tree grammars and given a suitable linear notation. Note that index ranges are not given. The consequence of this has to be a specialization of operators presented in the following section.

4.4 Combinator Parsing

In this section we will turn the grammar into a recognizer by a technique termed 'combinator parsing' (Hutton, 1992). Parser combinators are higher order functions used to compose complex recognizers from simpler ones. Each parser consumes a certain part of the input. Combinators (our operators) ensure that juxtaposed sub-parsers are applied to adjacent parts of the input and alternative sub-parsers receive the same input sequence.

In the following definitions the RNA input sequence is named `rna` and defined as in Section 3.2 and the length of the sequence is stored in `rnaLen`. The exact function definitions are given in Appendix A.8.

```
module Combinators
 where
 import RNA
 import Utils
```

4.4.1 Parsers

Definition 11 «parser» A parser is a function that given an index i, j of the input s returns a list of all its derivations: $P_X(i,j) := [t \leftarrow L(X) \mid y(t) = s_{i,j}]$. If no legal parse exists, the empty list is returned. □

```
type Input = Region

type Parser parse = Input -> [parse]
```

Parsers for terminal symbols are simple because they return singleton lists on success. The `base` parser need only check if the subword consists of a single base, i.e. is of length 1 in which case its index is returned. The `region` parser accepts a non-empty stretch of bases, whereas the `stretch` parser accepts a stretch of given length. Finally the `empty` parser succeeds if the subword is empty. By convention this is the case if $i = j$; it then returns the empty expression `()`.

```
base :: Parser BasePos
base (i,j) = [ j | (i+1) == j ]


region :: Parser Region
region (i,j) = [ (i,j) | i < j ]


stretch :: Int -> Parser Region
stretch l (i,j) = [ (i,j) | (i - j) == l ]


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

These are the only terminal parsers needed for RNA secondary structure prediction throughout this thesis.

4.4.2 Combinators

Definition 12 «axiom» The axiom function applies the parser representing the axiom symbol A to the complete input: $P_A := [t \leftarrow L(X) \mid y(t) = s]$. □

The keyword "axiom" of the grammar turns into a function that returns all parses for the startsymbol `a` over the complete input.

```
axiom :: Parser b -> [b]
axiom a = a (0,rnaLen)
```

Definition 13 «alternate combinator» The alternate combinator combines the derivations of two alternative parsers: $P_{X|Y}(i,j) := P_X(i,j) + P_Y(i,j)$. □

Alternative parsers are combined by the `|||` operator. So, given two parsers `r` and `q` (`r ||| q`) itself is a parser that returns the concatenated result lists of `r` and `q` on the same input.

```
infixr 6 |||
(|||) :: Parser b -> Parser b -> Parser b
(r ||| q) inp = r inp ++ q inp
```

Definition 14 «interpretation combinator» The interpretation combinator, applies the algebras evaluation function I_X to the derivations of P_X: $P_{I_X}(i,j) := [I_X(t) \mid t \leftarrow P_X(i,j)]$. □

In other words: Interpreting a term t by an evaluation function f of a T-algebra I amounts to mapping the evaluation function `f` to all elements of the derivation list of parser `q`.

```
type Interpretation b c = b -> c

infix  8 <<<
(<<<) :: Interpretation b c -> Parser b -> Parser c
(f <<< q) inp =  map f (q inp)
```

The interpretation function for an empty parse is a nullary function. To suppress the empty parse parameter, a special variant of the application function is needed.

```
infix  8 ><<
(><<) :: c -> Parser b -> Parser c
(c ><< q) inp = [ c | s <- q inp ]
```

Definition 15 «choice combinator» The choice combinator, applies the algebras choice function h to P_{I_X}: $h_{I_X}(i,j) := [h(i) \mid i \leftarrow P_{I_X}(i,j)]$. □

A choice functions prunes a list of interpretations, suchthat, the choice combinator need only apply the choice function `h` to the results of the parser `p`.

```
type Choice i = [i] -> [i]

infix  5 ...
(...) :: Parser b -> (Choice b) -> Parser b
p ... h = h . p
```

Typical choice functions include the following:

```
minimize :: [Energy] -> [Energy]
minimize [] = []
minimize xs = [minimum xs]

maximize :: Ord a => [a] -> [a]
maximize [] = []
maximize xs = [maximum xs]

addup :: [Integer] -> [Integer]
addup [] = []
addup xs = [sum xs]
```

Definition 16 «juxtaposition combinator» The juxtaposition combinator folds two adjacent parsers into one. It combines all parses of X and Y split at k ranging from i to j. $P_{XY}(i,j) := [P_X(i,k)\ P_Y(k,j) \mid k \leftarrow (i,j)]$. □

```
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)]
```

Now the Nussinov Grammar turns into a combinator parser.

```
nussinov1 algebra = axiom str
  where
  (bleft,bright,bpair,split,nil,h) = algebra

  str = bleft  <<< base ~~~ str          |||
        bright <<<          str ~~~ base |||
        bpair  <<< base ~~~ str ~~~ base |||
        split  <<<      str ~~~ str      |||
        nil    ><< empty                 ... h
```

Various specializations exist for the juxtaposition operator. An often encountered case in traditional DP recurrences is this: The two composite parsers are not allowed to be empty. More precisely they each have to consume at least one input element. This is achieved by constraining k to range from $i+1$ to $j-1$, suchthat the parsers are never challenged with the empty input. This has the nice side effect of allowing direct recursion in a single production.

```
infixl 7 ~+~
(~+~) :: Parser (b -> c) -> Parser b -> Parser c
(r ~+~ q) (i,j) = [f y | k <- [i+1..j-1], f <- r (i,k), y <- q (k,j)]
```

This is the case in the split sub-production in our running example. So the ~~~ combinator should be replaced by the new ~+~ combinator to ensure a correct interpretation. Note also, that this is just a shorthand for an indirect recursion with a terminal production. Consider our running example with the new ~+~ combinator:

```
nussinov2 algebra = axiom str
  where
  (bleft,bright,bpair,split,nil,h) = algebra

  str = bleft  <<< base ~~~ str          |||
        bright <<<          str ~~~ base |||
        bpair  <<< base ~~~ str ~~~ base |||
        split  <<<      str ~+~ str      |||
        nil    >>< empty                 ... h
```

It is obvious that it may be transformed into the equivalent productions:

```
nussinov3 algebra = axiom str
  where
  (bleft,bright,bpair,split,nil,h) = algebra

  str =           nonil           |||
        split <<< nonil ~~~ nonil |||
        nil   >>< empty           ... h

  nonil = bleft  <<< base ~~~ str          |||
          bright <<<          str ~~~ base |||
          bpair  <<< base ~~~ str ~~~ base ... h
```

Here the recursion is split into the empty and the non-empty case. Hence, the ~~~ combinator may be applied without ending in a non-teminating loop.

The following specialized juxtaposition combinators exist for reasons of efficiency. Consider the case of a production of variable length in the left context of our previously introduced base parser:

```
x <<< region ~~~ base
```

Let us follow the reduction process of the combinators.

```
((x <<< region) ~~~ base) (0,rnaLen)
```

The juxtaposition combinator is applied to region and base.

```
((x <<< region) ~~~ base) (0,rnaLen) = [f y | k <- [0..rnaLen], f <-(x <<< region) (0,k),
                                        y <- base (k,rnaLen)]
```

Let us assume that rnaLen $\gg 1$. Initially k is 0, therefore base (0,rnaLen) fails. Then k is 1, and consequently base (0,rnaLen) fails again. It will only succeed once, in its second last test, when $k =$ rnaLen $- 1$. We took rnaLen tests to find out what we already knew,

namely that base only accepts a single base. To avoid this inefficiency of $O(n)$ time where $n \approx$ rnaLen, we restrict the ranges of the split.

The most versatile version restricts the parsing effort to subwords of an explixit length range on the left and right side.

```
infixl 7 ~~
(~~) :: (Int,Int) -> (Int,Int) -> Parser (b -> c) -> Parser b -> Parser c
(~~) (l,u) (l',u') r q (i,j) = [x y | k <- [max (i+l) (j-u') .. min (i+u) (j-l')],
                                      x <- r (i,k), y <- q (k,j)]
```

Whereas the following two versions only restrict one side.

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

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

The most often encountered versions in this thesis restrict the lefthand (respectively righthand) parser to single symbols.

```
infixl 7 +~~
(+~~) :: Parser (b -> c) -> Parser b -> Parser c
(r +~~ p) (i,j) = [x y | i < j, x <- r (i,i+1), y <- p (i+1,j)]

infixl 7 ~~+
(~~+) :: Parser (b -> c) -> Parser b -> Parser c
(p ~~+ r) (i,j) = [x y | i < j, x <- p (i,j-1), y <- r (j-1,j)]
```

Restrict the lefthand parser to two symbols.

```
infixl 7 ++~
(++~) :: Parser (b -> c) -> Parser b -> Parser c
(r ++~ p) (i,j) = [x y | i < j, x <- r (i,i+2), y <- p (i+2,j)]
```

Restrict the lefthand parser to three symbols.

```
infixl 7 +++
(+++) :: Parser (b -> c) -> Parser b -> Parser c
(r +++ p) (i,j) = [x y | i < j, x <- r (i,i+3), y <- p (i+3,j)]
```

Additional versions for the lefthand parser are needed, because the right-fold decomposition of the parsers makes it necessary to pass-through the exact amount of symbols needed for parsers on the left. This is exemplified in the following code which parses exactly four bases.

```
Combinators> ((((  (,,,) <<< base) +~~ base) ++~ base) +++ base) (0,4)
[(1,2,3,4)]
```

The corresponding righthand version does not need different versions.

```
Combinators> ((((  (,,,) <<< base) ~~+ base) ~~+ base) ~~+ base) (0,4)
[(1,2,3,4)]
```

Our recognizer now reads:

```
nussinov4 algebra = axiom str
  where
  (bleft,bright,bpair,split,nil,h) = algebra

  str = bleft  <<< base +~~ str          |||
        bright <<<          str ~~+ base |||
        bpair  <<< base +~~ str ~~+ base |||
        split  <<<      str ~+~ str      |||
        nil    >>< empty                 ... h
```

4.4.3 Filters

Now the base pairing constraint mentioned in Section 4.3.1 needs to be applied to the bpair pattern. This is achieved by introducing a new keyword with that applies a filter to the input.

```
type Filter = Input -> Bool

with :: Parser b -> Filter -> Parser b
(p `with` f) inp = if f inp then p inp else []
```

This ensures that the parser will only evaluate the input in case the filter succeeds.

```
basepairing :: Filter
basepairing (i,j) = (i+1 < j) && pair (rna!(i+1)) (rna!j)
```

Together with the filter function basepairing given above we can finally write a correct recognizer for the base pair maximization algorithm.

```
nussinov5 algebra = axiom str
  where
  (bleft,bright,bpair,split,nil,h) = algebra

  str = bleft  <<< base +~~ str                              |||
        bright <<<          str ~~+ base                     |||
        (bpair <<< base +~~ str ~~+ base) `with` basepairing |||
        split  <<<      str ~+~ str                        |||
        nil    >>< empty                                   ... h
```

Various specialized filter combinators exist to apply filters to specific parts of the input or output of a parser. Below we present those that occur in this thesis.

The within combinator applies a filter to the symbols to the left and right of the parser input.

```
within :: Parser b -> Filter -> Parser b
(p `within` f) (i,j) = [x | i == 0 || j == rnaLen || f (i-1,j+1), x <- p (i,j)]
```

The suchthat combinator filters the output of the parser via f. It is immediately obvious that a-posteriori filtering cannot enhance the efficieny of a parser.

```
suchthat :: Parser b -> (b -> Bool) -> Parser b
(p `suchthat` f) inp = filter f (p inp)
```

4.4.4 Memoization

The recognizer nussinov5 works, but is extremely inefficent. The toy example with rnaStr = "gggaaauuc" uses a lot of space and time, because we have not yet implemented memoization of intermediate results. Without storing the results in tables our recognizer works as a recursive-decent parser on a highly ambigous grammar. Combinatorial explosion sets in as every recurring result is computed over and over again.

```
Nussinov> nussinov5 nussinovBasePairAlg
[3]
(69861430 reductions, 105502896 cells, 466 garbage collections)
```

To memoize the parser functions we need to store the results in an array indexed by the input region.

```
type Parsetable b = Array Region [b]
```

The function tabulated stores the results of a parser p applied to input range (i,j) at table position (i,j).

```
tabulated :: Parser b -> Parsetable b
tabulated p = array ((0,0),(rnaLen,rnaLen))
              [ ((i,j),p (i,j)) | i<- [0..rnaLen], j<- [i..rnaLen] ]
```

The results can be retrieved via the table lookup function p which is the inverse to the tabulation function.

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

One-dimensional tabulation works analogous to tabulation of regions and is needed in chapter 6.

```
type Parselist a = Array Int [a]

listed :: Parser b -> Parselist b
listed p = array (0,rnaLen)
              [ (i, p (i,rnaLen)) | i<- [0..rnaLen] ]

q :: Parselist b -> Parser b
q table (i,_) = table!i
```

The final version of the example recognizer now applies memoization to its single production str.

```
nussinov6 algebra = axiom (p str)
  where
  (bleft,bright,bpair,split,nil,h) = algebra

  str = tabulated (
        bleft  <<< base +~~ p str                              |||
        bright <<<          p str ~~+ base                     |||
        (bpair <<< base +~~ p str ~~+ base) `with` basepairing |||
        split  <<<     p str ~+~ p str                         |||
        nil    ><< empty                                       ... h )
```

```
Nussinov> nussinov6 nussinovBasePairAlg
[3]
(61256 reductions, 89035 cells, 1 garbage collection)
```

The efficiency is now $O(n^3)$ in time and $O(n^2)$ in space with $n \approx$ rnaLen. This version is the ADP implementation of the imperative version given in Section 4.1.3 of this thesis. In the final section of this chapter the recurrences will be reconstructed from the ADP code, to demonstrate this equivalence.

4.5 Recurrence Derivation

The DP recurrences of an ADP grammar can be systematically derived by substituting the terminal parsers, the definitions of parser combinators, and the functions of the evaluation algebra in question.

Given our example:

```
nussinov algebra = axiom (p str)
  where
  (bleft,bright,bpair,split,nil,h) = algebra

  str = tabulated (
        bleft  <<< base +~~ p str                              |||
        bright <<<          p str ~~+ base                     |||
        (bpair <<< base +~~ p str ~~+ base) `with` basepairing |||
        split  <<<     p str ~+~ p str                         |||
        nil    ><< empty                                       ... h )
```

and the base pair maximization algebra:

```
nussinovBasePairAlg = (bleft,bright,bpair,split,nil,h)
  where
  bleft  _ x   = x
  bright   x _ = x
  bpair :: Int -> Int -> Int -> Int
  bpair  _ x _ = x + 1
  split   x y  = x + y
  nil          = 0::Int
  h :: [Int] -> [Int]
  h        []  = []
  h        xs  = [maximum xs]
```

First we derive the code to be inlined for the elementary parsers `base` and `empty`.

$$\begin{array}{lcl} \texttt{base (i,j)} & = & \texttt{[j | (i+1) == j]} \\ \text{Base}(i,j) & = & \text{if } i+1=j \text{ then } [j] \text{ else } [] \end{array}$$

$$\begin{array}{lcl} \texttt{empty (i,j)} & = & \texttt{[() | i == j]} \\ \text{Empty}(i,j) & = & \text{if } i=j \text{ then } [\varepsilon] \text{ else } [] \end{array}$$

The grammar `nussinov` requires one matrix `str` with an index range of $i \in [0,n], j \in [0,n]$ and an element type of `Int`. For clarity the productions are replaced by the symbol *Parser*.

$$\begin{array}{lcl} \texttt{str (i,j)} & = & \texttt{tabulated (} \textit{ Parser } \texttt{... h) (i,j)} \\ \text{Str!}(i,j) & = & \texttt{(} \textit{ Parser } \texttt{... h)}(i,j) \end{array}$$

With the definition of choice combinator `...` and the choice function `h` we obtain:

$$\begin{array}{lcl} \text{Str!}(i,j) & = & \texttt{h (} \textit{ Parser } \texttt{)}(i,j) \\ \text{Str!}(i,j) & = & \max\{ \textit{ Parser}(i,j) \} \end{array}$$

We will now reduce the sub-productions of the grammar in their order of appearance.

$$\begin{array}{lcl} \text{Bleft}(i,j) & = & \texttt{(bleft <<< base +\~{}\~{} p str)}(i,j) \\ & = & \texttt{[x y | i < j, x } \leftarrow \texttt{(bleft <<< base) (i,i+1), y } \leftarrow \texttt{ (p str) (i+1,j)]} \end{array}$$

The parser `y` reduces to $\texttt{y} \leftarrow \text{Str!}(i+1,j)$ and the `x`-parser reduces to:

$$\begin{array}{lcl} \texttt{x} & \leftarrow & \texttt{[bleft q | q } \leftarrow \texttt{ base (i,i+1)]} \\ & \leftarrow & \texttt{[bleft q | q } \leftarrow \texttt{(} \text{ if } i+1=i+1 \text{ then } [i+1] \text{ else } [] \texttt{)]} \\ & \leftarrow & \texttt{[bleft } i+1 \texttt{]} \end{array}$$

Applying `x` to `y` via `bleft _ x = x` results in:

$$\begin{array}{lcl} \text{Bleft}(i,j) & = & \texttt{[bleft \_ Str!}(i+1,j) \mid i<j \texttt{]} \\ & = & \texttt{[Str!}(i+1,j) \mid i<j \texttt{]} \end{array}$$

The derivation of sub-parser `bright` is analogous to that of `bleft` resulting in:

$$\text{Bright}(i,j) \quad = \quad \texttt{[Str!}(i,j-1) \mid i<j \texttt{]}$$

The base pairing parser `bpair` reduces according to the following rationale:
The constraint `basepairing` is applied to the input of the parser (again, for clarity *Parser* replaces the actual production).

$$\begin{array}{lcl} \text{Bpair}(i,j) & = & \texttt{(bpair <<< base +\~{}\~{} p str \~{}\~{}+ base 'with' basepairing)}(i,j) \\ & = & \texttt{(} \textit{ Parser } \texttt{'with' basepairing)}(i,j) \\ & = & \texttt{(} \text{ if basepairing}(i,j) \text{ then } \textit{Parser}(i,j) \text{ else } [] \texttt{)}(i,j) \end{array}$$

As already seen in `bright` the following partial production reduces to:

$$\begin{aligned}
\mathit{Parser}(i,j) &= \texttt{(} \mathit{PartialParser} \texttt{ ~~+ base)}(i,j) \\
&= \texttt{[x y | i < j, x} \leftarrow \mathit{PartialParser}(i,j-1)\texttt{, y} \leftarrow \texttt{base (j-1,j)]} \\
&= \texttt{[} \mathit{PartialParser}(i,j-1) \texttt{ base (j-1,j)} \mid i<j \texttt{]} \\
&= \{\ \mathit{PartialParser}(i,j-1)\ j \mid i<j\ \}
\end{aligned}$$

Likewise $\mathit{PartialParser}(i,j-1)$ reduces analogous to `bleft`.

$$\begin{aligned}
\mathit{PartialParser}(i,j-1) &= \texttt{(bpair <<< base +~~ p str)}(i,j-1) \\
&= \texttt{[x y | i < j-1, x} \leftarrow \texttt{(bpair <<< base) (i,i+1),} \\
&\quad \texttt{y} \leftarrow \texttt{(p str) (i+1,j-1)]} \\
&= \texttt{[bpair <<< base) (i,i+1)}\ \mathrm{Str!}(i+1,j-1) \mid i<j-1 \texttt{]} \\
&= \texttt{[bpair <<<}\ i+1\ \mathrm{Str!}(i+1,j-1) \mid i<j-1 \texttt{]}
\end{aligned}$$

Joining the partial reductions results in:

$$\begin{aligned}
\mathit{Parser}(i,j) &= \texttt{[bpair}\ i+1\ \mathrm{Str!}(i+1,j-1)\ j \mid i<j \texttt{]} \\
&= \texttt{[bpair _}\ \mathrm{Str!}(i+1,j-1)\ \texttt{_} \mid i<j \texttt{]} \\
&= \texttt{[}\ \mathrm{Str!}(i+1,j-1)+1 \mid i<j \texttt{]}
\end{aligned}$$

Finally resulting in:

$$\mathrm{bpair}(i,j) = \{\ \mathrm{Str!}(i+1,j-1)+1 \mid i<j,\ \mathrm{pair}(i,j)\ \}$$

The only sub-production to introduce a non-trivial reduction resulting in more than one successful parse is `split`.

$$\begin{aligned}
\mathrm{split}(i,j) &= \texttt{(split <<< p str ~+~ p str)}(i,j) \\
&= \texttt{((split <<< p str) ~+~ p str)}(i,j) \\
&= \texttt{[f x | k} \leftarrow \texttt{[i+1..j-1], f} \leftarrow \texttt{(split <<< p str) (i,k),} \\
&\quad \texttt{y} \leftarrow \texttt{(p str) (k,j)]} \\
&= \texttt{[(split <<< p str) (i,k)}\ \mathrm{Str!}(k,j) \texttt{ | k} \leftarrow \texttt{[i+1..j-1]]} \\
&= \texttt{[split}\ \mathrm{Str!}(i,k)\ \mathrm{Str!}(k,j) \texttt{ | k} \leftarrow \texttt{[i+1..j-1]]} \\
&= \texttt{[}\ \mathrm{Str!}(i,k)+\mathrm{Str!}(k,j) \texttt{ | k} \leftarrow \texttt{[i+1..j-1]]} \\
&= \{\ \mathrm{Str!}(i,k)+\mathrm{Str!}(k,j) \mid k \in \{i+1,\ldots,j-1\}\ \}
\end{aligned}$$

The last sub-production is `nil`.

$$\begin{aligned}
\mathrm{nil}(i,j) &= \texttt{(nil >>< empty)}(i,j) \\
&= \texttt{[nil | s} \leftarrow \texttt{empty (i,j)]} \\
&= \texttt{[nil | s} \leftarrow \texttt{(}\ \text{if } i=j \text{ then } [\varepsilon] \text{ else } [] \texttt{)}(i,j)] \\
&= \{\ 0 \mid i=j\ \}
\end{aligned}$$

Thus, the complete recurrence relation for the base pair maximization problem is given by:

$$\begin{aligned}
\mathrm{Str!}(i,j) = \max\{ & \\
& \{\ \mathrm{Str!}(i+1,j) \mid i<j\ \} \cup \\
& \{\ \mathrm{Str!}(i,j-1) \mid i<j\ \} \cup \\
& \{\ 1+\mathrm{Str!}(i+1,j-1) \mid \mathrm{pair}(i,j),\ i<j\ \} \cup \\
& \{\ \mathrm{Str!}(i,k)+\mathrm{Str!}(k,j) \mid k \in \{i+1,\ldots,j-1\},\ i<j\ \} \cup \\
& \{\ 0 \mid i=j\ \} \\
& \}
\end{aligned}$$

The 0 may be omitted if the table Str! is initialized with 0. The overall result is found by the function $\mathit{BasePairMax}(\mathit{rna}) = \mathrm{Str!}(0,n)$.

4.6 Comparison of Recurrences

We will now make a direct comparison of the pseudocode at the beginning of this chapter with our ADP derivation.

```
writeIOArray table (i,j) (
  maximum [
     access table (i+1,j),
     access table (i,j-1),
    (access table (i+1,j-1)) + (delta rna (i,j)),
     maximum (0:[
       (access table (i,k)) + (access table (k+1,j))
        | k <- [(i+1)..(j-1)] ]) ])
     where delta rna (i,j) = if pair (rna!i) (rna!j) then 1 else 0
```

We find that all sub-terms are equivalent. The function `delta` has a different decomposition in the ADP formulation. Looking at the pseudocode it is clear why this is the case. The `delta` function combines the structural check for a legal base pair with the base pair counting. In the ADP world the objective function and the structure formation are separated, thus the recurrence is formulated as a structural constraint $\text{pair}(i, j)$. If it succeeds then the objective function is the constant 1.

Another difference in appearance – but not in effect – are the ubiquitous structural constraints. They stem from the fact that ADP does not formulate a traceback stage, but guides the structure formation process via constraints formulated in the grammar and the combinators.

Finally, the maximization function is applied to the 'split case' sub-term in the pseudocode, for reasons of data type arbitration, whereas ADP recurrences always are the union of the subsets of the sub-terms. Therefore a single application of the choice function suffices. Still, these different forms are trivially equivalent in an optimization scenario. Note, that the ADP form is more general if sampling or thresholding come into play.

Thus, the original recurrences are equivalent to the recurrences derived from the ADP implementation.

5 Ambiguity (Zuker's Algorithm)

The first rigorous treatment of the RNA secondary structure folding problem according to the model of Tinoco (Tinoco et al., 1971) was presented by Zuker in 1981. The aim of this chapter is to reformulate this algorithm as closely as possible in ADP. The resulting implementation will be discussed with special attention towards efficiency and ambiguity. From this we will conclude desirable properties for a base implementation of RNA secondary structure prediction in ADP that will then be implemented in the next chapter (6). The following quote describing the algorithm is taken from the original paper (Zuker and Stiegler, 1981). Note, that all terms used here (admissible, etc.) are defined as in this thesis.

5.1 Zukers Description

> We shall now describe in detail how the minimal free energy of a scondary structure is obtained when bifurcation loops are given zero energy. The algorithm is simple yet extremely powerful. No compromises are made. Not a single possibility is overlooked, and yet the algorithm selects a structure of minimum energy out of a number of structures that can be immense even for a molecule that is as small as a 5S ribosomal RNA. The main mathematical technique is to compute two possible different energies for each subsequence S_{ij} of a given RNA sequence. For all pairs i, j satisfying $1 \leq i < j \leq N$, let $W(i,j)$ be the minimum free energy of all possible admissible structures formed from the subsequence S_{ij}. In addition, let $V(i,j)$ be the minimum free energy of all possible admissible structures formed from S_{ij} in which S_i and S_j base pair with each other. If S_i and S_j cannot base pair, the $V(i,j) = \text{inf}$. The numbers $V(i,j)$ and $W(i,j)$ are computed recursively, first for all pentanucleotide subsequences, and then for all successively larger and larger subsequences of S. Pentanucleotide sequences are very easy to deal with. They form no stable structures, so $W(i,j) = 0$ if $j - i = 4$. If S_i and S_j are a G-C or A-U pair in this case of a 3 nucleotide hairpin loop, $V(i,j) = +8.4$ or $+8.0$ kcal/mole respectively. If $j - i = d > 4$, $V(i,j)$ and $W(i,j)$ can be computed in terms of $V(i',j')$ and $W(i',j')$ for various pairs i', j' satisfying $j' - i' > d$. These numbers will already have been computed. Imagine an admissible structure on S_{ij} with energy $V(i,j)$, assuming that S_i and S_j can base pair. We denote by $FH(i,j)$ the hairpin loop containing the interior edge between S_i and S_j, and by $FL(i,j,i',j')$ the face containing exactly two interior edges, one between S_i and S_j, and the other bewteen $S_{i'}$ and $S_{j'}$ (assuming $i < i' < j' < j$). The faces denoted by FL are either stacking regions or else bulge or interior loops. The face adjacent to the edge between S_i and S_j is one of three possible types, as illustrated in Figure 5.1A. It has either one, two or more than two interior edges. In the first case, $V(i,j) = E(FH(i,j))$. In the second case, $V(i,j) = E(FL(i,j,i',j')) + V(i',j')$ for some pair i', j' satisfying $i < i' < j' < j$. In the last case, $V(i,j) = W(i+1,i') + W(i'+1,j-1)$ for some i' satisfying $i+1 < i' < j-2$. Here the energy splits into the sum of the energies of two substructures; hence the word 'bifurcation'. Thus $V(i,j)$ is the minimum of the energies

which can be obtained in these three ways, so that we can write $V(i,j) = \min\{E1, E2, E3\}$, where $E1 = E(FH(i,j))$, $E2 = \min_{i<i'<j'<j}\{E(FL(i,j,i',j')) + V(i',j')\}$, and $E3 = \min_{i+1<i'<j-2}\{W(i+1,i') + W(i'+1,j-1)\}$. Now imagine an admissible structure on S_{ij} with energy $W(i,j)$. Again there are three possibilities. As illustrated in figure 5.1B, either S_i or S_j (or both) do not participate in the structure, or they base pair with each other, or else they both base pair, but not with each other. The first case is trivial. The structure has at least one dangling end, and $W(i,j) = W(i+1,j)$ or $W(i,j-1)$. In the second case, $W(i,j) = V(i,j)$, which has already been computed. The last case is referred to as an open bifurcation because the structure splits into two separate parts with no connection between S_i and S_j. If S_i base pairs with $S_{i'}$ and S_j base pairs with $S_{j'}$, where $i < i' < j' < j$, then $W(i,j) = W(i,i') + W(i'+1,j) = W(i,j'-1) + W(j',j)$. Thus $W(i,j)$, the minimum energy obtainable from these three cases, is given by: $W(i,j) = \min\{W(i+1,j), W(i,j-1), V(i,j), E4\}$, where $E4 = \min_{i<i'<j-1}\{W(i,i') + W(i'+1,j)\}$. Heuristically, this recursive algorithm works by adding one nucleotide at a time to a sequence, and observing what the best structure is at each step. The last number to be computed, $W(1,n)$, is the desired answer. It is the minimum energy of an admissible structure on S. However, the labour expended to compute $W(1,n)$ has in fact produced much more, for the minimum energy of an admissible structre on every subsequence of S is also known. All that remains is the construction of the structure, which is equivalent to identifying the interior edges of the associated graph. This is achieved by a traceback through the matrices W and V and is straightforward.

From this example we may conclude, that, obviously, it is no simple task to put in words a complex Dynamic Programming algorithm. Even so, the description depends heavily on the graphical representation of Figure 5.1. Furthermore, the complexity of the algorithm in space and time is not immediately clear through its description.

5.2 Zukers Algorithm in ADP

```
module Zuker
where
import RNA
import Combinators
import Utils
import List
```

Formulated in ADP Zukers algorithm takes the following shape. The admissible substructures are made up of the following components: The hairpin loop `HL`, the stacking region `SR`, the bulge loop (left and right) `BL` and `BL`, the interior loop `IL`, and the bifurcation loop `BI`, as well as the open loop (left and right) `OL` and `OR`, and finally the open bifurcation `OB`.

```
data  ZC = HL  BasePos      Region      BasePos | -- A.1
           SR  BasePos        ZC        BasePos | -- A.2a
           BL  BasePos Region ZC        BasePos | -- A.2b
           BR  BasePos        ZC Region BasePos | -- A.2b
           IL  BasePos Region ZC Region BasePos | -- A.2c
           BI  BasePos      ZC  ZC      BasePos | -- A.3
           OL  BasePos        ZC                | -- B.1
           OR                 ZC        BasePos | -- B.2
           OB               ZC  ZC                -- B.3
           deriving (Eq,Ord,Show)
```

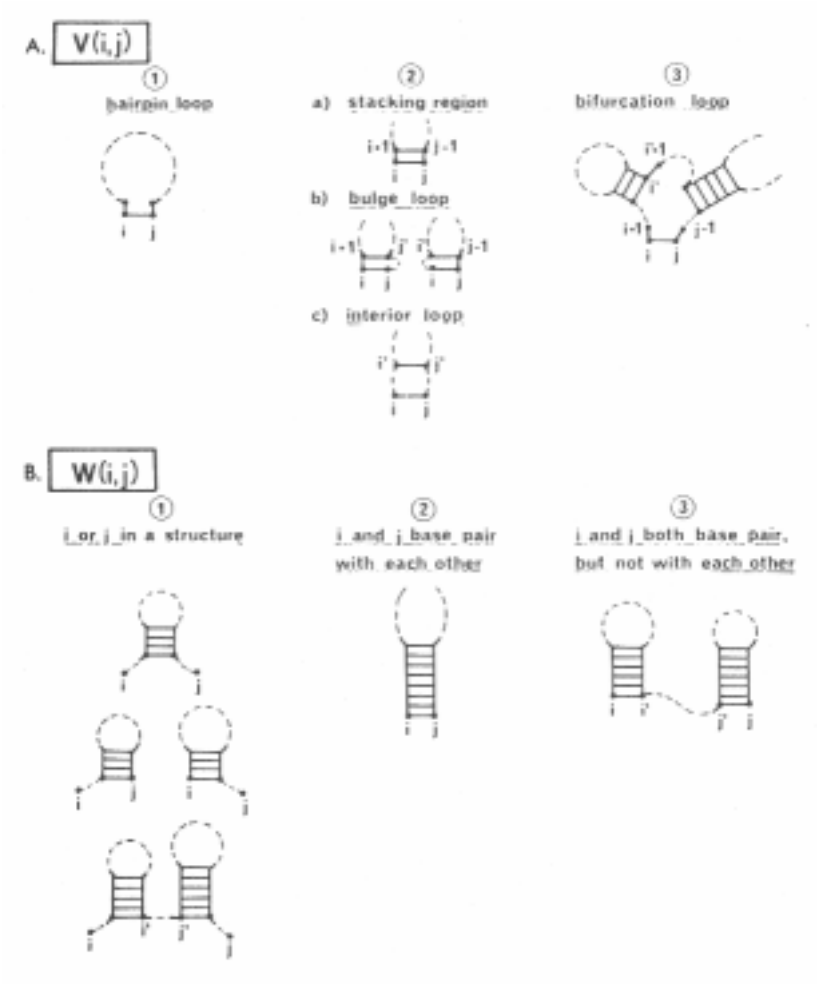


Figure 5.1: A: $V(i,j)$ is the minimum free energy of an admissible structure on the subsequence S_{ij} where S_i and S_j base pair with each other. B: $W(i,j)$ is the minimum free energy of an admissible structure on the subsequence S_{ij}.
(Figure taken from (Zuker and Stiegler, 1981).)

In conjunction with their graphical counterparts in Figure 5.1, the meaning of the structure constituents and their recursive nature is immediately clear. An algebra is then trivially made up of its structural components and the identity choice function.

```
zukerStructureAlg = (HL,BI,SR,BL,BR,IL,OL,OR,OB,id)
```

The ADP grammar given here, was made to very closely resemble the description in the quoted text in section 5.1.

```
zuker81 algebra = axiom (p w) where

  (hl,bi,sr,bl,br,il,ol,ox,ob,h) = algebra

  v = tabulated (
      ((hairpin ||| twoedged ||| bifurcation) `with` basepairing) ... h)  -- V(i,j)
    where
    hairpin     = hl  <<< base +~~ (region `with` minLoopSize 3) ~~+ base       -- E1
    bifurcation = bi  <<< base +~~          p w ~+~ p w          ~~+ base ... h -- E3
```

```
    twoedged    = stack ||| bulgeleft ||| bulgeright ||| interior         ... h -- E2
      where
      stack      = sr <<< base +~~              p v              ~~+ base
      bulgeleft  = bl <<< base +~~   region ~+~ p v              ~~+ base
      bulgeright = br <<< base +~~              p v ~+~ region   ~~+ base
      interior   = il <<< base +~~   region ~+~ p v ~+~ region   ~~+ base

  w = tabulated (
      openleft ||| openright ||| p v ||| openbifurcation ... h) -- W(i,j)
        where
        openleft        = ol <<< base +~~ p w
        openright       = ox <<<          p w ~~+ base
        openbifurcation = ob <<<      p w ~+~ p w      ... h -- E4
```

The decomposition of structures is easily seen. The two tables V and W and their interactions are clearly represented as `v`, and `w`. Moreover, the different cases are cleanly separated according to their classification in Figure 5.1. The 5 places where selection, i.e. minimization takes place are pointed out by the application of the choice function `h`

The algebra to count the search space size has the following form.

```
zukerCountAlg = (hl,bi,sr,bl,br,il,ol,ox,ob,addup) where
  hl _  _  _  = 1
  bi _ lc rc _  = lc * rc
  sr _   zc   _ = zc
  bl _ _ zc   _ = zc
  br _   zc _ _ = zc
  il _ _ zc _ _ = zc
  ol _   zc     = zc
  ox     zc _   = zc
  ob   lc rc    = lc * rc
```

Calling the grammar `zuker81` with the counting algebra for the small example of length 9 from the previous chapter gives the following result.

```
Zuker> zuker81 zukerCountAlg
[37]
```

The structure algebra displays the complete list of structures evaluated.

```
Zuker> zuker81 zukerStructureAlg
[
OL 1 (OL 2 (OL 3 (OR (HL 4 (4,7) 8) 9))),
OL 1 (OL 2 (OR (OL 3 (HL 4 (4,7) 8)) 9)),
OL 1 (OL 2 (OR (OR (HL 3 (3,6) 7) 8) 9)),
OL 1 (OL 2 (OR (HL 3 (3,7) 8) 9)),
OL 1 (OL 2 (HL 3 (3,8) 9)),
OL 1 (OL 2 (SR 3 (HL 4 (4,7) 8) 9)),
OL 1 (OR (OL 2 (OL 3 (HL 4 (4,7) 8))) 9),
OL 1 (OR (OL 2 (OR (HL 3 (3,6) 7) 8)) 9),
OL 1 (OR (OL 2 (HL 3 (3,7) 8)) 9),
OL 1 (OR (OR (OL 2 (HL 3 (3,6) 7)) 8) 9),
OL 1 (OR (OR (HL 2 (2,6) 7) 8) 9),
OL 1 (OR (HL 2 (2,7) 8) 9),
OL 1 (OR (SR 2 (HL 3 (3,6) 7) 8) 9),
OL 1 (HL 2 (2,8) 9),
OL 1 (SR 2 (HL 3 (3,7) 8) 9),
OL 1 (BL 2 (2,3) (HL 4 (4,7) 8) 9),
OL 1 (BR 2 (HL 3 (3,6) 7) (7,8) 9),
OR (OL 1 (OL 2 (OL 3 (HL 4 (4,7) 8)))) 9,
OR (OL 1 (OL 2 (OR (HL 3 (3,6) 7) 8))) 9,
OR (OL 1 (OL 2 (HL 3 (3,7) 8))) 9,
OR (OL 1 (OR (OL 2 (HL 3 (3,6) 7)) 8)) 9,
OR (OL 1 (OR (HL 2 (2,6) 7) 8)) 9,
OR (OL 1 (HL 2 (2,7) 8)) 9,
OR (OL 1 (SR 2 (HL 3 (3,6) 7) 8)) 9,
OR (OR (OL 1 (OL 2 (HL 3 (3,6) 7))) 8) 9,
OR (OR (OL 1 (HL 2 (2,6) 7)) 8) 9,
OR (OR (HL 1 (1,6) 7) 8) 9,
OR (HL 1 (1,7) 8) 9,
OR (SR 1 (HL 2 (2,6) 7) 8) 9,
```

```
OR (BL 1 (1,2) (HL 3 (3,6) 7) 8) 9,
HL 1 (1,8) 9,
SR 1 (HL 2 (2,7) 8) 9,
SR 1 (SR 2 (HL 3 (3,6) 7) 8) 9,
BL 1 (1,2) (HL 3 (3,7) 8) 9,
BL 1 (1,3) (HL 4 (4,7) 8) 9,
BR 1 (HL 2 (2,6) 7) (7,8) 9,
IL 1 (1,2) (HL 3 (3,6) 7) (7,8) 9
]
Zuker>
```

There are several things to note in this example.

- The completely single stranded sequence, the simplest structure possible, is not present.
- Bifurcation elements are not present. They need a minimum input length of 10 to appear in a term. The rationale behind this is simple: The smallest possible sub-structure is the hairpin loop, which has a minimum length of 5 while the bifurcation needs to harbour 2 closed structures resulting in a minimum length of 10.
- The first and second term represent identical structures. A closer look reveals that the terms, 1,2,7, and 18 are different representations of the same structure.

Obviously some structures are duplicated in the output whereas others are missing. The missing ones are due to overly simplified input examples or ommissions in the grammar. The point I want to make here is two-fold. Recurrences can be wrong, and finding the errors can be very difficult, partly because finding the right input to expose the error can be hard. Note, that Zuker claimed not to have overlooked any single possibility in his algorithm, yet the albeit simple case of the single stranded RNA is overlooked. The second point is, that DP recurrences suited to solve optimization problems need not heed duplicate solutions. Recurrences can be simplified by overlapping sub-structure solutions and stay perfectly viable if one has only a single (optimal) solution in mind.

The next section will discuss the different types of ambiguity, their effects, and measures to avoid them.

5.3 Ambiguity

Why is ambiguity a problem at all? Zuker's recurrences are correct and have proven their value in numerous experiments over the years (Jacobson and Zuker, 1993; Zuker, 2000). A naive search for the keyword 'mfold' in the PubMed database reveals 27 hits alone. But what if we would want to reuse the structure of the recurrences? To be usable to compute different salient features of a given problem they would have to be canonical. Canonicity implies that all solutions are represented in the structural search space exactly once. This is important for all measures that utilize counting or enumeration of structures.

Definition 17 «Yield Grammar Ambiguity» A tree grammar G is ambiguous if there are different leftmost derivations for some tree $t \in L(G)$. A yield grammar (G, y) is ambiguous, if G is ambiguous, otherwise it is unambiguous. □

To check if some derivations are identical we introduce the algebra `zukerDotBracketAlg` which represents the structures in dot bracket notation (see Section 3.4). In this way, we can easily compare structures regardless of the underlying derivations.

```
zukerDotBracketAlg = (hl,bi,sr,bl,br,il,ol,ox,ob,id) where
  hl _    r    _  = "(" ++ dots r ++ ")"
  bi _ lc rc _  = "(" ++ lc ++ rc ++ ")"
```

```
   sr _   zc   _ = "(" ++ zc ++ ")"
   bl _ r zc   _ = "(" ++ dots r ++ zc ++ ")"
   br _   zc r _ = "(" ++ zc ++ dots r ++ ")"
   il _ lr zc rr _ = "(" ++ dots lr ++ zc ++ dots rr ++ ")"
   ol _   zc     = "." ++ zc
   ox     zc _   = zc ++ "."
   ob   lc rc    = lc ++ rc

dots :: Region -> [Char]
dots (i,j) = ['.' | k <- [1..j-i]]
```

```
Zuker> zuker81 zukerDotBracketAlg
[
"...(...).",
"...(...).",
"..(...)..",
"..(....).",
"..(.....)",
"..((...))",
"...(...).",
"..(...)..",
"..(....).",
"..(...)..",
".(....)..",
".(.....).",
".((...)).",
".(......)",
".((....))",
".(.(...))",
".((...).)",
"...(...).",
"..(...)..",
"..(....).",
"..(...)..",
".(....)..",
".(.....).",
".((...)).",
"..(...)..",
".(....)..",
"(.....)..",
"(......).",
"((...)).",
"(.(...)).",
"(.......)",
"((.....))",
"((...))).",
"(.(....)).",
"(..(...)).",
"((....).)",
"(.(...).)"
]
Zuker> length $ zuker81 zukerDotBracketAlg
37

Zuker> length $ nub $ zuker81 zukerDotBracketAlg
23
```

We find 37 structures for our toy example and if we remove all duplicate structures the result is 23.

5.3.1 Types of Ambiguity

Zuker's recurrences harbour different types of ambiguous statements.

1. Branching recursion

   ```
   w = ob <<< p w ~+~ p w
   ```

2. Indirect recursion

```
v = bi <<< base +~~ p w ~+~ p w ~~+ base
w = p v
```

3. Ambiguous derivation

```
w = ol <<< base +~~ p w          |||
    ox <<<          p w ~~+ base |||
    ob <<<      p w ~+~ p w
```

Direct branching recursion is inherently ambiguous. Such a statement will always have to be rewritten to avoid ambiguity. Indirect branching is more subtle and not so easy to detect. Keep in mind that the recursion can occur after any number of intermediate steps. Furthermore, it need not necessarily be a cause of ambiguity. Observe that there are multiple evaluation functions (`ol`, `ox`) in our example. Ambiguity only occurs if these evaluation functions give final identical results. But we can only achieve canonicity for all possible evaluation functions if we avoid these types of statements.

5.3.2 Avoidance Measures

- Use only one direction in recursive structures.
  Especialy avoid direct branching recursion.
- Every structural entity should have a unique place of evaluation.
  This implies that only one evaluation function exists.

5.3.3 Efficiency

Ambiguous recurrences are specialized recurrences for one type of evaluation. Therefore, it is only logical to assume that Zuker allowed ambiguity to speed up the computation or to save space, because certain structural distinctions are irrelevant for the evaluation function in question. In our case this is the minimization of free energy. It does not matter wether we evaluate the energy for an internal loop in more than one place, and in more than one way, because it either is the structure with minimal free energy or it is not. Worse, the end result would still be correct if one of the alternative computations gave wrong results that always were higher in value. In the next section a canonical parser will be presented for the nearest neighbor model.

6 Canonicity (Wuchty's Algorithm)

The aim of a canonical parser for a given model is to produce all legal solutions for a given input (problem) without producing duplicates.

Definition 18 «Canonical Models and Canonical Yield Grammars» Let K be a set, the canonical model. Let k be a mapping from $L(G)$ to K. A yield grammar (G,y) is canonical w.r.t. K and k if it is unambiguous and the mapping k is bijective. A DP algorithm is canonical w.r.t. K and k, if the underlying yield grammar is canonical w.r.t. K and k. □

Wuchty and others (Wuchty et al., 1999) have presented DP recurrences in 1999 for the nearest neighbor model because they wanted to produce statistics of free energy folding landscapes. According to Zuker (Zuker, 2000) they reinvented the recurrences of Williams and Tinoco (Williams and Tinoco, 1986) in the process. To avoid this happening again we will base our ADP recognizer on the schema given in Wuchty's paper. Thus, gaining a more general implementation of the nearest neighbor model for RNA secondary structure formation not restricted to free energy calculation. This implementation will form the basis on which further improvements and enhancements will be presented in this thesis.

6.1 Wuchtys Algorithm in ADP

```
module Wuchty
where
import RNA
import Combinators
import FreeEnergy
import EnergyTables
import Structure
import Utils
import List

wuchty99 algebra = axiom external
  where
  (el,sadd,cons,sr,hl,bl,br,il,ml,
   concat,ul,addss,ssadd,nil,
   h,h_l,h_s) = algebra

  external      = el <<< q struct
```

The enclosing structure of a RNA secondary fold is the external loop, which consists of a chain of substructures.

```
                where
  struct        = listed (
                  sadd <<<   base   +~~ q struct |||
                  cons <<< p closed ~~~ q struct |||
                  nil  >>< empty                 ... h_s)
```

This chain may be empty or consist of any number of unpaired bases and closed substructures. The tabulation used here takes the form of a one-dimensional array (`listed` and `q`) because the boundary of all statements is fixed to n, the 3'-end of the

RNA sequence. Remember, that ADP combinator parsers always have to consume the complete input to succeed. We want the complete RNA sequence folded and not only a part of it. Canonicity is ensured, because every statement involved is right-recursive and enforces a unique structure element (base or closed) on its left side. Moreover, only the empty statement ends the recursion.

```
                where
    closed    = tabulated (
                ((stack ||| hairpin ||| leftB ||| rightB ||| iloop ||| multiloop)
                  'with' basepairing) ... h)
                where
      stack   = sr <<< base +~~ p closed ~~+ base
      hairpin = hl <<< base +~~  (region 'with' minLoopSize 3)  ~~+ base
      leftB   = bl <<< base +~~ region  ~~~ p closed            ~~+ base
      rightB  = br <<< base +~~             p closed ~~~ region ~~+ base
      iloop   = il <<< base +~~ region  ~~~ p closed ~~~ region ~~+ base
```

The statements stack, hairpin, leftB, rightB, and iloop, are old friends and are one to one chains of their constituent elements including the closing base pair (see Section 3.3). As such, there is no room for ambiguity.

```
      multiloop = ml <<< base +~~        p block ~~~ p comps      ~~+ base

      comps     = tabulated (
                  concat <<< p block ~~~ p comps |||
                             p block             |||
                  addss  <<< p block ~~~ region  ... h_l)

      block     = tabulated (
                  ul    <<<           p closed |||
                  ssadd <<< region ~~~ p closed ... h_l)
```

A multiple loop consists of two or more blocks and an optional trailing region of upaired bases at the 3'-end. A block in turn is a paired structure with an optional 5'-region of single stranded bases. Again, uniqueness of structures is ensured by strictly disjoint statements.

The free energy algebra adds up the energies of sub-structures and evaluates the free energy of given structures by applying the energy functions given in Section 3.3.

```
wuchtyFreeEnergyAlg = (id,sadd,cons,sr,hl,bl,br,il,ml,
                       concat,ul,addss,ssadd,nil,
                       minimize,minimize,minimize)
  where
  sadd       _  e          = e
  cons :: Energy -> Energy -> Energy
  cons     c   e          = e + c
  sr   i      e       j = e + dg_sr (i,j)
  hl   i      _       j =     dg_hl (i,j)
  bl   i (_,n) e      j = e + dg_bl (i,j) (n+1,j-1)
  br   i      e (m,_) j = e + dg_bl (i,j) (i+1,m)
  il   i (_,n) e (u,_) j = e + dg_il (i,j) (n+1,u)
  ml   _     b  c      _ = ml_init_penalty + b + c
  concat :: Energy -> Energy -> Energy
  concat      b  c        = b + c
  ul          c           = ml_helix_penalty + c
  addss       c r         = c + ml_unpaired r
  ssadd       r c         = c +  ml_helix_penalty + ml_unpaired r
  nil                     = 0.0::Energy
```

Counting works by multiplying sub-structure combinations and adding up structure alternatives.

```
wuchtyCountAlg = (id,sadd,cons,sr,hl,bl,br,il,ml,
                  concat,ul,addss,ssadd,nil,
                  addup,addup,addup)
  where
  sadd    _ e    = e
```

```
  cons    c e    = c * e
  sr   _   e   _ = e
  hl   _   _   _ = 1
  bl   _ _ e   _ = e
  br   _   e _ _ = e
  il   _ _ e _ _ = e
  ml   _ b c   _ = b * c
  concat  b c    = b * c
  ul       c     = c
  addss   c _    = c
  ssadd   _ c    = c
  nil            = 1::Integer
```

The structure algebra is the direct application of the corresponding elements in the data type. In some cases functions have to be applied to perform convolved list concatenations.

```
wuchtyStructureAlg = (EL,sadd,(:),SR,HL,BL,BR,IL,ml,
                      (++),(:[]),addss,ssadd,[],
                      id,id,id)
  where
  sadd  m      []       = SS (m-1,m) : []
  sadd  m (SS (i,j):c)  = SS (m-1,j) : c
  sadd  m       c       = SS (m-1,m) : c
  ml    i      b c    j = ML i (b ++ c) j
  addss        c r      = c ++ [SS r]
  ssadd        r c      =      [SS r, c]
```

6.2 Comparison to Zuker's Algorithm

By providing disjoint statements and identical directions of recursion we have provided a canonical parser for secondary structures. Some simple (and of course inconclusive) tests with our toy example gives the expected result of 24 structures without any duplicates.

```
Wuchty> wuchty99 wuchtyStructureAlg
[
EL [SS (0,9)],
EL [SS (0,3),HL 4 (4,7) 8,SS (8,9)],
EL [SS (0,2),HL 3 (3,6) 7,SS (7,9)],
EL [SS (0,2),HL 3 (3,7) 8,SS (8,9)],
EL [SS (0,2),SR 3 (HL 4 (4,7) 8) 9],
EL [SS (0,2),HL 3 (3,8) 9],
EL [SS (0,1),HL 2 (2,6) 7,SS (7,9)],
EL [SS (0,1),SR 2 (HL 3 (3,6) 7) 8,SS (8,9)],
EL [SS (0,1),HL 2 (2,7) 8,SS (8,9)],
EL [SS (0,1),SR 2 (HL 3 (3,7) 8) 9],
EL [SS (0,1),HL 2 (2,8) 9],
EL [SS (0,1),BL 2 (2,3) (HL 4 (4,7) 8) 9],
EL [SS (0,1),BR 2 (HL 3 (3,6) 7) (7,8) 9],
EL [HL 1 (1,6) 7,SS (7,9)],
EL [SR 1 (HL 2 (2,6) 7) 8,SS (8,9)],
EL [HL 1 (1,7) 8,SS (8,9)],
EL [BL 1 (1,2) (HL 3 (3,6) 7) 8,SS (8,9)],
EL [SR 1 (SR 2 (HL 3 (3,6) 7) 8) 9],
EL [SR 1 (HL 2 (2,7) 8) 9],
EL [HL 1 (1,8) 9],
EL [BL 1 (1,2) (HL 3 (3,7) 8) 9],
EL [BL 1 (1,3) (HL 4 (4,7) 8) 9],
```

```
EL [BR 1 (HL 2 (2,6) 7) (7,8) 9],
EL [IL 1 (1,2) (HL 3 (3,6) 7) (7,8) 9]
]
Wuchty> length $ wuchty99 wuchtyStructureAlg
24
Wuchty> length $ nub $ wuchty99 wuchtyStructureAlg
24
Wuchty> wuchty99 wuchtyCountAlg
[24]
Wuchty>
```

Compared to Zuker's algorithm the canonical version uses 3 two-dimensional and a single one-dimensional table instead of only 2 two-dimensional tables. Time consumption is in the order of $O(n^4)$ for both algorithms. The time complexity is dominated by the internal loop statement, which is identical in both algorithms. The next section will demonstrate how to reduce the time complexity by one order.

7 Decomposition (Lyngsø's Algorithm)

7.1 Introduction

The algorithms seen so far all incur a time complexity of $O(n^4)$. All substructures discerned in the nearest neighbor model are of complexity $O(n^3)$ except for the internal loop. Although the multiple loop seems more complex it is, in fact, trivialized to an affine cost model to avoid exponential complexity on the order of the degree k of the loop, i.e. $O(n^k)$. The linear dependency of the affine model can be decomposed by a single recursive split of the structure in question. Thus, resulting in a time complexity of $O(n \cdot n^2)$. The term n^2 stemming from the left and right boundaries (i, j) of the substructure positioned somewhere on the sequence of length n.

The question now is: Is there a way to reduce the complexity of the internal loop? The internal loop needs two splits (~~~) to cover all structure variants. These determine the lengths of the two single stranded regions of the internal loop, thereby fixing the position of the enclosed structure.

```
iloop = il <<< base +~~ region ~~~ p closed ~~~ region ~~+ base
```

Zuker and Hofacker reduce the complexity to $O(n^3)$ by setting the maximum length of the combined single stranded regions to 30 (Zuker, 1989; Hofacker et al., 1994). They reason, that longer internal loops are thermodynamically unfavored structures and that the probability of occurrence is very low in a biologically relevant sequence. The length bound effectively links both splits together, resulting in the desired reduction of complexity. It has to be said, that the – albeit constant – cost is still high in practice. The complexity reduction probably only pays of for sequences longer than 300 bases. Furthermore, this type of constraint cuts off the search space and could lead to the (highly unlikely) loss of the mfe-structure. For us, there is another reason to favor other solutions. This type of constraint is not elegantly captured in ADP. It would call for the introduction of a specialized combinator suitable only for this single application. The combinator is simple enough:

```
infixl 7 ~~~~

(~~~~) :: Parser (b -> c -> d) ->
          (Parser b, Parser c, (Int,Int,Int,Int) -> Bool) ->
          Parser d
```

The fold of two juxtaposition combinators together with a filter function.

```
(z ~~~~ (w,q,x)) (i,j) = [ a b | p <- [i..j],
                                 a <- [f y | k <- [i..p],
                                             f <- z (i,k),
                                             y <- w (k,p),
                                             x (i,p,k,j)],
                                 b <- q (p,j) ]
```

The size of the internal loop could then be tested as follows.

```
ilSize s (i,p,k,j) = if ((k-i) + (j-p)) <= s then True else False
```

Resulting in the parser statement:

```
iloop = il <<< base +~~ region ~~~~ (hairpin, region, ilSize 30) ~~+ base
```

We have seen enough of ADP to take this as a hint and to postulate: *"If it is not elegantly captured in ADP it is not a good solution to the problem."*

7.2 Possible Decompositions

Given the parser statement:

```
iloop = il <<< base +~~ region ~~~ p closed ~~~ region ~~+ base
```

which are the ways to produce the same structural elements regardless of evaluation functions and in time complexity $O(n^3)$ instead of $O(n^4)$? Every application of the full juxtaposition combinator (~~~) in a single statement exponentiates the order of complexity by one factor.

Definition 19 «Statement Width» The maximum number w of juxtaposition combinators (~~~) of the alternatives (separated by alternate combinators |||) in a given parser statement is called the *width* of the statement. The time complexity of the statement is $O(n^{d+w})$ where d is the dimension of the enclosing tabulation. □

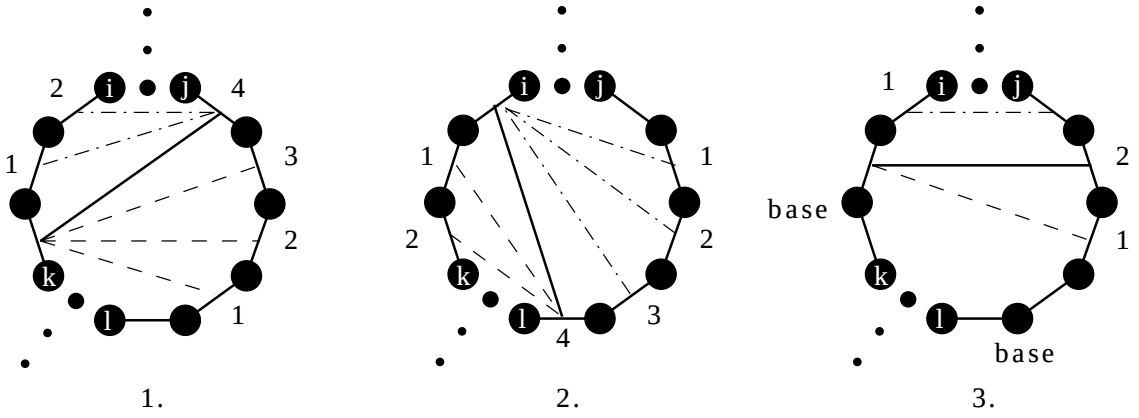


Figure 7.1: The three different decompositions of the internal loop statement.

By tabulating a partial term in a sub-statement we can avoid exponentiation. With the statement in question this can be achieved in three ways (see Fig. 7.1). The first two are the obvious cases, i.e. to choose either partial term containing one juxtaposition for tabulation.

```
1. iloop      = il <<< base +~~ region  ~~~ p rightpart ~~+ base
                where
    rightpart = tabulated (ilright <<< p closed ~~~ region ... h_i)

2. iloop     = il <<< base +~~ p leftpart ~~~ region ~~+ base
               where
    leftpart = tabulated (illeft <<< region ~~~ p closed ... h_i)
```

Whereas the third alternative is to recursively extend the loop with single bases on the left and right side. Then to end the recursion by exclusively adding an arbitrary amount of bases on the left or right or none in the case of a symmetric internal loop.

```
3. iloop     = iln <<< base +~~ p inloop ~~+ base
               where
    inloop   = tabulated (
                 ilx <<< base +~~ p inloop ~~+ base |||
                                  p loopend         ... h_i)
               where
     loopend = tabulated (
               ill <<< region ~~+ base ~~~ p closed ~~+ base            |||
               ilr <<<            base +~~ p closed ~~+ base ~~~ region |||
               ils <<<            base +~~ p closed ~~+ base            ... h_i)
```

All these alternatives produce all legal internal loop structures for a given input in different decompositions. They all have a time complexity of $O(n^3)$. The remaining problem is to analyze the internal loop energy function and see if it can be made to fit any of the three variants presented here.

7.3 Analysis of the Evaluation Function

The internal loop free energy evaluation function is the sum of four terms that are governed by the size of the loop, the mismatch at both paired ends, and the lopsidedness of the loop (see Section 3.3).

$$\Delta G_{IL} = \Delta G_{size}(|i \bullet j|) + \Delta G_{mismatch}(i \bullet j) + \Delta G_{mismatch}(k \bullet l) + \Delta G_{asym}(i \bullet j, k \bullet l) \tag{7.1}$$

- The mismatch terms are not dependent on the decomposition of the search space by the parser statements. They depend solely on the base pairs and their mismatched neighbors.

- The size term is additive. If we store the length of the unpaired region, i.e. loop size, together with the free energy of the partial internal loop we can subtract the energy for the old loop size and add the new entropic term. There is one caveat here. In the first two versions of decompositions this will break the application of a choice function.

- The asymmetry term depends on the difference in size of the two unpaired regions. This too, would break the choice function in the two first versions. Only the identity function could be applied, giving rise to partial results in the order of n and bringing us back to $O(n^4)$.

The third decomposition version is in accordance with the requirements of the free energy evaluation function for internal loops. We achieve the evaluation in $O(n^3)$ time at the cost of introducing two new tables of size $O(n^2)$. The solution presented here was published by Lyngsø and others (Lyngsø et al., 1999) for the mfe case only. They achieved this result without the use of further tables by storing the intermediate results in table space that was later overwritten with the end result for closed structures (personal communication). This approach is not applicable to ADP.

7.4 Invariant Formulation

If our analysis was correct the following invariant should hold true. The old decomposition together with its free energy evaluation functions should yield the exact same results as the new decomposition together with its corresponding evaluation functions for any input.

```
module LyngsoInvariant
where
import RNA
import Combinators
import FreeEnergy
import EnergyTables
import Utils
import List
```

The two decompositions are called `il1` and `il2`. Because the order of evaluation is not identical the result lists have to be sorted before being compared.

```
ilInvariant = (sort il1) == (sort il2)
              where
```

The enclosed structure is trivialized to a hairpin in both variants.

```
  hairpin = hl <<< base +~~  (region 'with' minLoopSize 3)  ~~+ base
            where
    hl i _ j = dg_hl (i,j)
```

The classic internal loop formulation in a single statement with the energy evaluation function.

```
  il1 = axiom iloop
        where
    iloop = il <<< base +~~ region  ~~~ hairpin ~~~ region ~~+ base
            where
      il   i (_,n) e (u,_) j = e + dg_il (i,j) (n+1,u)
```

The new formulation of the internal loop computation.

```
  il2 = axiom iloop
        where
    iloop     = iln <<< base +~~ p inloop ~~+ base
                where
      iln i (e,k) j = e + top_stack (i,j)

      inloop   = tabulated (
                   ilx <<< base +~~ p inloop ~~+ base |||
                                    p loopend           )
                 where
       ilx lb (e,k) rb = (e + (ent_il (k+2)) - (ent_il k), k+2)

       loopend = tabulated (
                   ill <<< region ~~+ base ~~~ hairpin ~~+ base            |||
                   ilr <<<            base +~~ hairpin ~~+ base ~~~ region |||
                   ils <<<            base +~~ hairpin ~~+ base               )
                 where
         ill (i,j) lb e rb = (e + (asym (j-i)) + (bot_stack (lb,rb)) +
                              (ent_il (j-i+2)), j-i+2)
         ilr lb e rb (i,j) = (e + (asym (j-i)) + (bot_stack (lb,rb)) +
                              (ent_il (j-i+2)), j-i+2)
         ils lb e rb       = (e + (asym 0) + (bot_stack (lb,rb)) +
                              (ent_il 2), 2::Int)
```

Regression tests show that the new formulation indeed yields identical results.

7.5 Lyngsø's Algorithm in ADP

Starting with the canonical parser presented in Section 6 we introduce a general canonical parser for RNA secondary structure evaluation with a complexity of $O(n^3)$ implemented in ADP.

```
module Lyngso
where
import RNA
import Combinators
import FreeEnergy
import EnergyTables
import Utils

lyngsø99 algebra = axiom external
  where
  (el,sadd,cons,sr,hl,bl,br,
   iln,ilx,ill,ilr,ils,
   ml,concat,ul,addss,ssadd,nil,
   h,h_l,h_s,h_i) = algebra

  external      = el <<< q struct
                  where
  struct        = listed (
                  sadd <<<   base   +~~ q struct |||
                  cons <<< p closed ~~~ q struct |||
                  nil  >>< empty                 ... h_s)
                  where
    closed      = tabulated (
                  ((stack ||| hairpin ||| leftB ||| rightB ||| iloop ||| multiloop)
                    'with' basepairing) ... h)
                  where
      stack     = sr <<< base +~~ p closed ~~+ base
      hairpin   = hl <<< base +~~  (region 'with' minLoopSize 3)  ~~+ base
      leftB     = bl <<< base +~~ region  ~~~ p closed            ~~+ base
      rightB    = br <<< base +~~             p closed ~~~ region ~~+ base
```

The statement:

```
      iloop     = il <<< base +~~ region  ~~~ p closed ~~~ region ~~+ base
```

is replaced by the following equivalent decomposition:

```
      iloop     = iln <<< base +~~ p inloop ~~+ base
                  where
       inloop   = tabulated (
                    ilx <<< base +~~ p inloop ~~+ base |||
                                     p loopend         ... h_i)
                  where
        loopend = tabulated (
                  ill <<< region ~~+ base ~~~ p closed ~~+ base            |||
                  ilr <<<            base +~~ p closed ~~+ base ~~~ region |||
                  ils <<<            base +~~ p closed ~~+ base            ... h_i)
```

The rest of the grammar remains unchanged.

```
      multiloop = ml <<< base +~~        p block ~~~ p comps      ~~+ base

      comps     = tabulated (
                  concat <<< p block ~~~ p comps |||
                             p block             |||
                  addss  <<< p block ~~~ region  ... h_l)

      block     = tabulated (
                  ul    <<<            p closed |||
                  ssadd <<< region ~~~ p closed ... h_l)
```

The free energy algebra is augmented by the new internal loop evaluation functions.

```
lyngsøFreeEnergyAlg = (id,sadd,cons,sr,hl,bl,br,
                       iln,ilx,ill,ilr,ils,
                       ml,concat,ul,addss,ssadd,nil,
                       minimize,minimize,minimize,h_i)
  where
  sadd     _   e          = e
  cons :: Energy -> Energy -> Energy
  cons     c   e          = e + c
  sr   i       e       j = e + dg_sr (i,j)
  hl   i       _       j =     dg_hl (i,j)
  bl   i (_,n) e       j = e + dg_bl (i,j) (n+1,j-1)
  br   i       e (m,_) j = e + dg_bl (i,j) (i+1,m)
```

The internal loop tables store the free energy `e` of the partial internal loop and the intermediate size `k` of the loop.

```
  iln :: Int -> (Energy,Int) -> Int -> Energy
  iln i (e,k) j = e + top_stack (i,j)
```

The free energy of the internal loop consists of the free energy of the partial loop `e` and the free energy of the mismatch of the pair $i \bullet j$ and its unpaired neighbors.

```
  ilx :: Int -> (Energy,Int) -> Int -> (Energy,Int)
  ilx lb (e,k) rb = (e + (ent_il (k+2)) - (ent_il k), k+2)
```

The intermediate recursion adds a base to each side of the bulge and adjusts the entropic term accordingly.

```
  ill :: Region -> Int -> Energy -> Int -> (Energy,Int)
  ill (i,j) lb e rb = (e + (asym (j-i)) + (bot_stack (lb,rb)) + (ent_il (j-i+2)), j-i+2)

  ilr :: Int -> Energy -> Int -> Region -> (Energy,Int)
  ilr lb e rb (i,j) = (e + (asym (j-i)) + (bot_stack (lb,rb)) + (ent_il (j-i+2)), j-i+2)

  ils :: Int -> Energy -> Int -> (Energy,Int)
  ils lb e rb = (e + (asym 0) + (bot_stack (lb,rb)) + (ent_il 2), 2::Int)
```

The end of the loop adds the free energy of the following closed structure `e` to the asymmetry term `asym`, the mismatch term `bot_stack` and the entropic term for the minimum loop size of 2.

```
  ml     _     b  c       _ = ml_init_penalty + b + c
  concat :: Energy -> Energy -> Energy
  concat      b  c         = b + c
  ul           c           = ml_helix_penalty + c
  addss       c r          = c + ml_unpaired r
  ssadd       r c          = c +  ml_helix_penalty + ml_unpaired r
  nil                      = 0.0::Energy

  h_i :: [(Energy,Int)] -> [(Energy,Int)]
  h_i [] = []
  h_i es =  [minimum es]
```

The counting algebra has additional terms for the internal loop evaluation functions.

```
lyngsøCountAlg = (id,sadd,cons,sr,hl,bl,br,
                  iln,ilx,ill,ilr,ils,
                  ml,concat,ul,addss,ssadd,nil,
                  addup,addup,addup,addup)
  where
  sadd    _ e    = e
  cons    c e    = c * e
  sr   _    e   _ = e
  hl   _    _   _ = 1
  bl   _ _  e   _ = e
  br   _    e _ _ = e
```

```
  iln  _   e   _ = e
  ilx  _   e   _ = e
  ilr  _ e _ _   = e
  ill    _ _ e _ = e
  ils  _   e   _ = e

  ml    _  b c  _ = b * c
  concat   b c    = b * c
  ul       c      = c
  addss    c _    = c
  ssadd    _ c    = c
  nil             = 1::Integer
```

The structure evaluation algebra calls for an additional data structure to store the intermediate internal loop structures.

```
data LLoop =
    EL                [LLoop]                | -- The External Loop
    HL  BasePos        Region        BasePos | -- The Hairpin Loop
    SR  BasePos        LLoop         BasePos | -- The Stacking Region
    BL  BasePos Region  LLoop        BasePos | -- The Left Bulge
    BR  BasePos        LLoop  Region BasePos | -- The Right Bulge
    IL  BasePos Region  LLoop  Region BasePos | -- The Internal Loop
    OIL         Region  LLoop  Region         | -- The Incomplete (i.e. Open) Internal Loop
    ML  BasePos       [LLoop]        BasePos | -- The Multiple Loop
    SS                 Region                | -- A Single Strand of unpaired bases
                                               -- in the external or multiple loop.
    NIL                                        -- The empty loop
                                               -- is used to represent structure fragments.
  deriving (Eq,Read,Show)

lyngsøStructureAlg = (EL,sadd,(:),SR,HL,BL,BR,
                      iln,ilx,ill,ilr,ils,
                      ml,(++),(:[]),addss,ssadd,[],
                      id,id,id,id)
  where
  sadd  m      []              = SS (m-1,m) : []
  sadd  m (SS (i,j):c)         = SS (m-1,j) : c
  sadd  m       c              = SS (m-1,m) : c
  ml    i      b c    j        = ML i (b ++ c) j
  addss        c r             = c ++ [SS r]
  ssadd        r c             =      [SS r, c]
  iln   m (OIL   l   c   r  ) n = IL m l c r n
  ilx   m (OIL (i,j) c (k,l)) n = OIL (m-1,j) c (k,n)
  ilr         m c n (i,j)       = OIL (m-1,m) c (n-1,j)
  ill   (i,j) m c n             = OIL   (i,m) c (n-1,n)
  ils         m c n             = OIL (m-1,m) c (n-1,n)
```

7.6 Conclusion

We have seen how to analyze an evaluation function. Then, how to find different decompositions to enhance efficiency. Moreover, we verified that our conclusions were true by finding an invariant formulation and using it to perform regression tests. Finally, we provided a general canonical parser for RNA secondary structures with a time complexity of $O(n^3)$ and a space complexity of $O(n^2)$ where n is the length of the RNA sequence. The next section is dedicated to enhancing the nearest neighbor model by refining the energy model for multiple loops.

8 Refining the Multiple Loop Energy Model

8.1 Introduction

In the previous three sections, the free energy of the multiple loop has always been computed as the cost of initiating the loop, plus a fixed penalty for every unpaired base, plus a cost for every helix in the loop (see Section 3.3). This crude affine cost model is used, because a more realistic logarithmic cost function governed by the Jacobson-Stockmayer formula would incur exponential time complexity. Furthermore, little is known about the global thermodynamics of large RNA secondary structure elements such as multiple loops. All the data used was derived by fitting model parameters to give optimal overall results for RNA where the resulting secondary structures were proven by experimental methods.

There are ways, however, to stay with an additive cost function and still arrive at more realistic results. Local effects in the area of helix-ends are well understood today (Mathews et al., 1999). So called dangling ends – single unpaired bases – like to stack ontop of helix-ends. The same goes for neighboring helices, that combine to form, what is known as a co-axial compound (see Fig. 8.1). These compounds may form if two helices are separated by, at most, one single base, which incorporates into the global helical structure (Walter et al., 1994). The effects just described are easily recognized as the governing force of RNA structure formation. They are the stacking interactions, which are now applied across sub-structure boundaries in the nearest neighbor model.

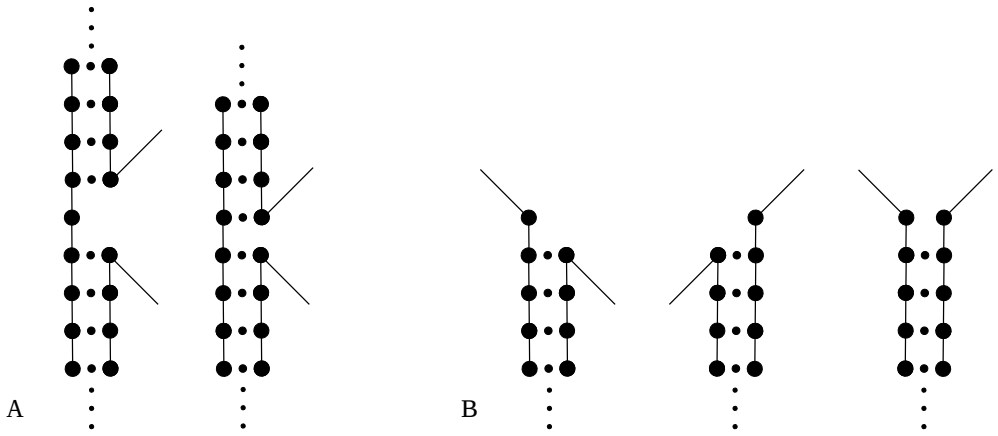


Figure 8.1: Co-axial compounds (A) and dangling end compounds (B).

8.2 The Modelling Choice

The choice has to be made wether to leave the recursion schema – the recognizer – untouched and hide the new computations within the evaluation functions, or to make

the new compounds explicit in the parsing rules.

The arguments in favor of hiding them are:

1. The recognizer is more general.
2. The count of structures would otherwise increase to include the new alternative structures.
3. All the current prediction programs known to the author use this approach.

The arguments in favor of making them explicit are:

1. All structure influencing forces can be seen in the output.
2. The common structure decomposition is completely retained as one variant in the new structural schema. Therefore, it is easy to formulate algebras to ignore the new elements, thus giving identical results to the common version.
3. Hiding the variant compound computation in the evaluation functions amounts to solving a separate DP optimization problem.

Conclusion We will make dangling ends and co-axial stacking first class citizens of our model.

The most decisive argument for this choice was phrased by Ivo Hofacker in a conversation we had, as having to: “solve a DP optimization problem within a DP optimization problem”. Especially in ADP this would seem like an unnatural and unelegant thing to do. I can only guess that the reason for doing so, in the standard RNA prediction programs, was not having to change the basic recurrence relations. A thing that is easily done and checked in ADP.

8.3 The Dangling-Ends Algorithm in ADP

The `wuchty99` recognizer is used as the basis for the dangling end aware version. The productions for the external loop and multiple loop have been changed to take dangling ends into account. The `closed` statement representing helical structures was replaced by new untabulated productions (`edangle` and `dangle`) representing a helical structure together with dangling end alternatives. Note, that the resulting set of structures does not contain the same helical region. Optimization via the choice function works in this case, because we compare affine costs at a given position, i.e. split, in the multiple loop.

```
module Dangles
where
import RNA
import Combinators
import FreeEnergy
import EnergyTables
import Utils


evers01a algebra = axiom external
                   where
```

The algebra is augmented by the alternative dangling end evaluation functions. A new choice function `h_d` for the dangling end alternatives is necessary too.

```
(el,sadd,cons,edl,edr,edlr,drem,sr,hl,bl,br,
 iln,ilx,ill,ilr,ils,
 ml,mldl,mldr,mldlr,concate,ul,addss,ssadd,dl,dr,dlr,nil,
 h,h_l,h_s,h_i,h_d) = algebra

external      = el <<< q struct
                where
struct        = listed (
                sadd <<< base    +~~ q struct |||
                cons <<< edangle ~~~ q struct |||
                nil  >>< empty               ... h_s)
                where
```

Parser edangle compares all different dangling end alternatives.

```
 edangle      = edl  <<< base +~~ p closed          |||
                edr  <<<          p closed ~~+ base |||
                edlr <<< base +~~ p closed ~~+ base |||
                drem <<<          p closed          ... h_d
                where
  closed      = tabulated (
                ((hairpin ||| stack ||| leftB ||| rightB ||| iloop ||| multiloop)
                  'with' basepairing) ... h)
                where
    stack     = sr <<< base +~~ p closed ~~+ base
    hairpin   = hl <<< base +~~  (region 'with' minLoopSize 3)  ~~+ base
    leftB     = bl <<< base +~~ region  ~~~ p closed            ~~+ base
    rightB    = br <<< base +~~             p closed ~~~ region ~~+ base

    iloop     = iln <<< base +~~ p inloop ~~+ base
                where
     inloop   = tabulated (
                  ilx <<< base +~~ p inloop ~~+ base |||
                                   p loopend         ... h_i)
                where
      loopend = tabulated (
                ill <<< region ~~+ base ~~~ p closed ~~+ base            |||
                ilr <<<            base +~~ p closed ~~+ base ~~~ region |||
                ils <<<            base +~~ p closed ~~+ base            ... h_i)
```

The start of the multiple loop also has to check for dangling ends on the inside of the helix that starts with the outer bases of this production.

```
    multiloop = ml    <<< base +~~          ml_comps          ~~+ base |||
                mldl  <<< base +~~ base ++~ ml_comps          ~~+ base |||
                mldr  <<< base +~~          ml_comps ~~+ base ~~+ base |||
                mldlr <<< base +~~ base ++~ ml_comps ~~+ base ~~+ base
                where
     ml_comps = combine <<< p block ~~~ p comps

    comps     = tabulated (
                concate <<< p block ~~~ p comps |||
                            p block             |||
                addss   <<< p block ~~~ region  ... h_l)
```

Again, the parser closed is replaced by the new parser dangle.

```
    block     = tabulated (
                ul    <<<           dangle |||
                ssadd <<< region ~~~ dangle ... h_l)
                where
```

The parser dangle is structurally identical to the parser edangle. The evaluation functions have to be different, because the additional multiple loop penalties have to be considered here.

```
      dangle  = dl   <<< base +~~ p closed          |||
                dr   <<<          p closed ~~+ base |||
                dlr  <<< base +~~ p closed ~~+ base |||
                drem <<<          p closed          ... h_d
```

The free energy algebra is identical to the algebra `wuchtyFreeEnergyAlg` with added dangling end choice and evaluation functions. To make dangling end evaluation work, the evaluation functions for the `closed` parser have to make the base pair explicit in their output. Therefore all functions involved not only return their free energy but the closing base pair tuple `(i,j)`. Regression tests indicate that the algebra `danglesFreeEnergyAlg` together with the parser `evers01a` yield identical results to mfold version 2.3 (Zuker, 2003).

```
danglesFreeEnergyAlg = (id,sadd,cons,
                        edl,edr,edlr,drem,sr,hl,bl,br,
                        iln,ilx,ill,ilr,ils,
                        ml,mldl,mldr,mldlr,
                        concat,ul,addss,ssadd,
                        dl,dr,dlr,nil,
                        h,h_l,h_s,h_i,h_d)
                       where
  sadd      _       e           =  e
  cons :: Energy -> Energy -> Energy
  cons      c  e           =  e + c
  sr   i       (e,_)       j = (e + dg_sr (i,j), (i,j))
  hl   i         _         j =     (dg_hl (i,j), (i,j))
  bl   i (_,n) (e,_)       j = (e + dg_bl (i,j) (n+1,j-1), (i,j))
  br   i       (e,_) (m,_) j = (e + dg_bl (i,j) (i+1,m), (i,j))

  iln i (e,k) j = (e + top_stack (i,j), (i,j))
  ilx lb (e,k) rb = (e + (ent_il (k+2)) - (ent_il k), k+2)
  ill (i,j) lb (e,_) rb = (e + (asym (j-i)) + (bot_stack (lb,rb)) +
                           (ent_il (j-i+2)), j-i+2)
  ilr lb (e,_) rb (i,j) = (e + (asym (j-i)) + (bot_stack (lb,rb)) +
                           (ent_il (j-i+2)), j-i+2)
  ils lb (e,_) rb = (e + (asym 0) + (bot_stack (lb,rb)) +
                    (ent_il 2), 2::Int)

  concat :: Energy -> Energy -> Energy
  concat      b  c       = b + c
  ul          c          = ml_helix_penalty + c
  addss       c r        = c + ml_unpaired r
  ssadd       r c        = c +  ml_helix_penalty + ml_unpaired r
  nil                    = 0.0::Energy
```

The dangling end bonus is given dependent on the base pair of the helical region (ij) and the neighboring dangling base.

```
  dl   _ (e,ij)   = e + ml_helix_penalty + ml_unpaired (0,1) + dg_dl ij
  dr     (e,ij) _ = e + ml_helix_penalty + ml_unpaired (0,1) + dg_dr ij
  dlr  _ (e,ij) _ = e + ml_helix_penalty + ml_unpaired (0,2) +
                    dg_dl ij + dg_dr ij
  drem            = fst
  edl  _ (e,ij)   = e + dg_dl ij
  edr    (e,ij) _ = e + dg_dr ij
  edlr _ (e,ij) _ = e + dg_dl ij + dg_dr ij
  ml    i   (b,c)   j = (ml_init_penalty + b + c, (i,j))
  mldl  i _ (b,c)   j = (ml_init_penalty + b + c +
                         dg_dli (i,j) + ml_unpaired (0,1), (i,j))
  mldr  i   (b,c) _ j = (ml_init_penalty + b + c +
                         dg_dri (i,j) + ml_unpaired (0,1), (i,j))
  mldlr i _ (b,c) _ j = (ml_init_penalty + b + c +
                         dg_dli (i,j) + dg_dri (i,j) + ml_unpaired (0,2), (i,j))

  h_d :: [Energy] -> [Energy]
  h_d [] = []
  h_d es =  [minimum es]

  h_i :: [(Energy,Int)] -> [(Energy,Int)]
  h_i [] = []
  h_i es =  [minimum es]

  h_l :: [Energy] -> [Energy]
  h_l [] = []
  h_l es =  [minimum es]

  h_s :: [Energy] -> [Energy]
  h_s [] = []
  h_s es =  [minimum es]
```

```
  h :: [(Energy,(Int,Int))] -> [(Energy,(Int,Int))]
  h [] = []
  h es =  [minimum es]

  el = id
```

Counting allways follows the same schema. Also in this case, where the additional evaluation functions where added to the algebra `wuchtyCountAlg`.

```
danglesCountAlg = (id,sadd,cons,
                   edl,edr,edlr,drem,sr,hl,bl,br,
                   iln,ilx,ill,ilr,ils,
                   ml,mldl,mldr,mldlr,
                   concat,ul,addss,ssadd,
                   dl,dr,dlr,nil,
                   addup,addup,addup,addup,addup)
  where
  sadd    _ e    = e
  cons    c e    = c * e
  sr   _    e   _ = e
  hl   _    _   _ = 1
  bl   _ _  e   _ = e
  br   _    e _ _ = e

  iln  _    e   _ = e
  ilx  _    e   _ = e
  ilr  _ e  _ _   = e
  ill    r  _ e _ = e
  ils  _    e   _ = e

  concat  b c    = b * c
  ul        c    = c
  addss   c _    = c
  ssadd   _ c    = c
  nil            = 1::Integer

  ml    _    (b,c)   _ = b * c
  mldl  _ _  (b,c)   _ = b * c
  mldr  _    (b,c) _ _ = b * c
  mldlr _ _  (b,c) _ _ = b * c
  dl    _ c    = c
  dr      c _  = c
  dlr   _ c _  = c
  drem    c    = c
  edl   _ c    = c
  edr     c _  = c
  edlr  _ c _  = c
```

This counting algebra generates the numbers for the standard RNA secondary structures. Thus, all additional dangling end functions evaluate to 0. Moreover, `wuchty99 wuchtyCountAlg == evers01a danglesStdCountAlg` should always yield `True`.

```
danglesStdCountAlg = (id,sadd,cons,
                      edl,edr,edlr,drem,sr,hl,bl,br,
                      iln,ilx,ill,ilr,ils,
                      ml,mldl,mldr,mldlr,
                      concat,ul,addss,ssadd,
                      dl,dr,dlr,nil,
                      addup,addup,addup,addup,addup)
  where
  sadd    _ e    = e
  cons    c e    = c * e
  sr   _    e   _ = e
  hl   _    _   _ = 1
  bl   _ _  e   _ = e
  br   _    e _ _ = e

  iln  _    e   _ = e
  ilx  _    e   _ = e
  ilr  _ e  _ _   = e
  ill    r  _ e _ = e
  ils  _    e   _ = e
```

```
concat  b c    = b * c
ul       c     = c
addss   c _    = c
ssadd   _ c    = c
nil            = 1::Integer

ml    _   (b,c)   _ = b * c
mldl  _ _   _     _ = 0
mldr  _     _   _ _ = 0
mldlr _ _   _   _ _ = 0
dl    _ _    = 0
dr      _ _ = 0
dlr   _ _ _ = 0
drem   c    = c
edl   _ _    = 0
edr     _ _ = 0
edlr  _ _ _ = 0
```

To represent the new structures, a new data type DLoop containing all known structural elements plus the new dangling end structures has to be derived.

```
data DLoop =
    EL                  [DLoop]                     | -- The External Loop
    HL  BasePos          Region        BasePos      | -- The Hairpin Loop
    SR  BasePos          DLoop         BasePos      | -- The Stacking Region
    BL  BasePos Region   DLoop         BasePos      | -- The Left Bulge
    BR  BasePos          DLoop  Region BasePos      | -- The Right Bulge
    IL  BasePos Region   DLoop  Region BasePos      | -- The Internal Loop
    OIL         Region   DLoop  Region              | -- The Incomplete (i.e. Open) Internal Loop
    ML  BasePos         [DLoop]        BasePos      | -- The Multiple Loop
                                                      -- The Closing Helix of the Multiple Loop:
    MLDL  BasePos BasePos [DLoop]          BasePos  | -- with Left Dangle
    MLDR  BasePos         [DLoop] BasePos BasePos  | -- with Right Dangle
    MLDLR BasePos BasePos [DLoop] BasePos BasePos  | -- with Both Dangles
    SS                     Region                  | -- A Single Strand of unpaired bases
                                                      -- in the external or multiple loop.
    DL              BasePos DLoop                  | -- A Helix with Left Dangle
    DR                      DLoop BasePos          | -- A Helix With Right Dangle
    DLR             BasePos DLoop BasePos          | -- A Helix With Both Dangles
    NIL                                              -- The empty loop is used to
                                                     -- construct partial structures.
  deriving (Eq,Read,Show)
```

```
danglesStructureAlg = (EL,sadd,(:),DL,DR,DLR,id,SR,HL,BL,BR,
                       iln,ilx,ill,ilr,ils,
                       ml,mldl,mldr,mldlr,
                       (++),(:[]),addss,ssadd,
                       DL,DR,DLR,[],
                       id,id,id,id,id)
                      where
  sadd  m      []              = SS (m-1,m) : []
  sadd  m (SS (i,j):c)         = SS (m-1,j) : c
  sadd  m       c              = SS (m-1,m) : c

  ml    i      (b,c)    j      = ML i (b ++ c) j
  mldl  i m (b,c)   j = MLDL  i m (b ++ c) j
  mldr  i   (b,c) n j = MLDR  i   (b ++ c) n j
  mldlr i m (b,c) n j = MLDLR i m (b ++ c) n j

  iln   m (OIL   l   c   r  ) n = IL m l c r n
  ilx   m (OIL (i,j) c (k,l)) n = OIL (m-1,j) c (k,n)
  ilr         m c n (i,j)       = OIL (m-1,m) c (n-1,j)
  ill   (i,j) m c n             = OIL   (i,m) c (n-1,n)
  ils         m c n             = OIL (m-1,m) c (n-1,n)

  addss        c r              = c ++ [SS r]
  ssadd        r c              =      [SS r, c]
```

8.4 Conclusion

We have succeeded in building an ADP recognizer and a free energy algebra that yield identical results compared to Michael Zukers mfold 2.3. As this parser is canonical, it is capable of generating the complete structure space without duplicates. Where Wuchty et al. leave out dangling ends of single unpaired bases, our algorithm is capable of traversing the complete structure space and thus also capable of producing all alternatives (Wuchty et al., 1999). However, we can not optimize over the dangling end alternatives for a given helix using this ADP recognizer.

9 Reducing the Conformation Space by Structural Constraints

9.1 Introduction

In section 8 of this thesis we presented an ADP recognizer capable of enumerating the complete conformation space of RNA secondary structures according to the current nearest neighbor model. This also entails finding the mfe-structure. However, finding the mfe-structure alone is often not sufficient. Near-optimal solutions must be investigated for a number of reasons:

1. The mfe-structure might be ill-defined (Zuker, 1989).
2. Empiric thermodynamic data are incomplete and erroneous, rendering all structures within approx. 10% of the mfe possible candidates (Mathews et al., 1999).
3. Competing structures, such as RNA switches in translation regulation are known to occur (Fayat et al., 1983).
4. Non-planar structures are excluded in the nearest neighbor model, yet they exist; the free energy of their constituent secondary structures would have to be sufficiently near the mfe to be feasible. Solving the so-called *Pseudoknot Problem* is possible with a DP approach in $O(n^6)$ time and $O(n^4)$ time complexity (Rivas and Eddy, 2000; Rivas and Eddy, 2001). As such it is also within reach of an ADP formulation, but out of scope of this thesis. For further information see (Giegerich, 2003).

As we have seen, computing sub-optimal structures is not difficult. However, the number of potentially interesting sub-optimal structures grows exponentially with the length of the nucleotide sequence, making all global analyses of such structures costly. Even in the non-redundant conformation space near-optimal structures exhibit high similarity in exponential numbers. A single representative of each such group would suffice for preliminary analyses.

This problem was originally observed by Zuker and Sankoff in 1984 (Zuker and Sankoff, 1984). A common approach in existing RNA secondary prediction programs is to eliminate so-called *lonely pairs* (see Fig 9.1). Zuker and Sankoff went beyond this simple constraint and its resulting conformation space reduction by suggesting to restrict secondary structure folding to structures whose stacking regions extend maximally in both directions. The resulting problem is termed the *saturated structure folding problem*.

The solution to this problem was developed for this thesis and first presented at GCB'01 in the form of DP recurrences (Evers and Giegerich, 2001). In the course of this thesis we have seen, that investigating fewer structures leads to more complex recognizers and thus, recurrences. A first attempt at saturated folding neither solved the problem completely, nor did it yield a practical algorithm (Giegerich, 1999). The ADP recognizer was excessively complex in the form of 23 tabulation statements. We will see, that the exact solution can be achieved by summarizing equivalent structural constraints in 2 `tabulated`, 3 `listed`, and 3 `attributed` statements. To do this requires the extension of

the ADP framework with parser attributes which are subsequently used in the second parser of this chapter.

9.2 Lonely Pairs

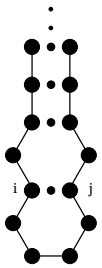


Figure 9.1: An RNA secondary structure with a lonely pair $i \bullet j$.

Lonely pairs sometimes also called isolated pairs are helices consisting of exactly one base pair (see Figure 9.1). The rationale behind removing this class of sub-structures is, that the energy of the base pair binding force is in the same order as the binding energy of the surrounding water and additional stabilizing stacking forces are not present in an isolated base pair. Therefore, a lonely pair is very unlikely to exist in a stable conformation. We will start by demonstrating the application of this structural constraint in our ADP recognizer.

9.3 The 'No Lonely Pairs' Algorithm in ADP

```
module NoLonelyPairs
where
import RNA
import Combinators
import FreeEnergy
import EnergyTables
import Utils
```

In the parsers presented so far, lonely pairs only arise in the context of the `closed` production, simply because this is the only place where a base pair is formed in the grammar. If we ensure that the initialization of a base pair stack always results in at least two base pairs our lonley pairs problem is solved. Stacks are extended by the recursive invocation of the `sr` alternative in `closed`. By pulling out this recursion into a separate production and ensuring that it is invoked at least once we can ensure the 'no lonely pairs' property. The necessary changes to parser `evers01b` are shown here.

```
...

closed      = tabulated (
              ((sr <<< base +~~ ( p closed ||| p weak) ~~+ base)
                'with' basepairing) ... h)

weak      = tabulated (
              ((hairpin ||| leftB ||| rightB ||| iloop ||| multiloop)
                'with' basepairing) ... h)
              where

              hairpin   = ...
```

The new parser is called closed, because it is the direct replacement of the old closed parser. This way, the changes to the grammar stay local. The rest of the old closed statement containing all structures with a single closing base pair now comprise the parser weak. These changes come at the price of a new table for the parser weak. It is possible to avoid this space impediment by introducing an indirect recursion in the stack extension case and ensuring that all weak elements are enclosed by a base pair. This is the final 'no lonely pairs' parser.

```
evers01d algebra = axiom external
                   where
  (el,sadd,cons,edl,edr,edlr,drem,sr,hl,bl,br,
   iln,ilx,ill,ilr,ils,
   ml,mldl,mldr,mldlr,concate,ul,addss,ssadd,dl,dr,dlr,nil,
   h,h_l,h_s,h_i,h_d) = algebra

  external      = el <<< q struct
                  where
  struct        = listed (
                    sadd <<< base    +~~ q struct |||
                    cons <<< edangle ~~~ q struct |||
                    nil  >>< empty               ... h_s)
                  where
   edangle      = edl  <<< base +~~ p closed          |||
                    edr  <<<          p closed ~~+ base |||
                    edlr <<< base +~~ p closed ~~+ base |||
                    drem <<<          p closed          ... h_d
                    where
```

Basepairing is checked in the closed parser to avoid unnecessary recursion.

```
    closed      = tabulated (
                  ((stack ||| strong)
                   'with' basepairing) ... h)
                  where
```

The parser stack checks helices of arbitrary length. A choice function is not needed in this production because it only produces a single alternative.

```
      stack     = sr <<< base +~~ p closed ~~+ base
```

The strong parser checks basepairing of the closed structure alternatives and adds a further base pair, thus ensuring the 'no lonely pairs' property.

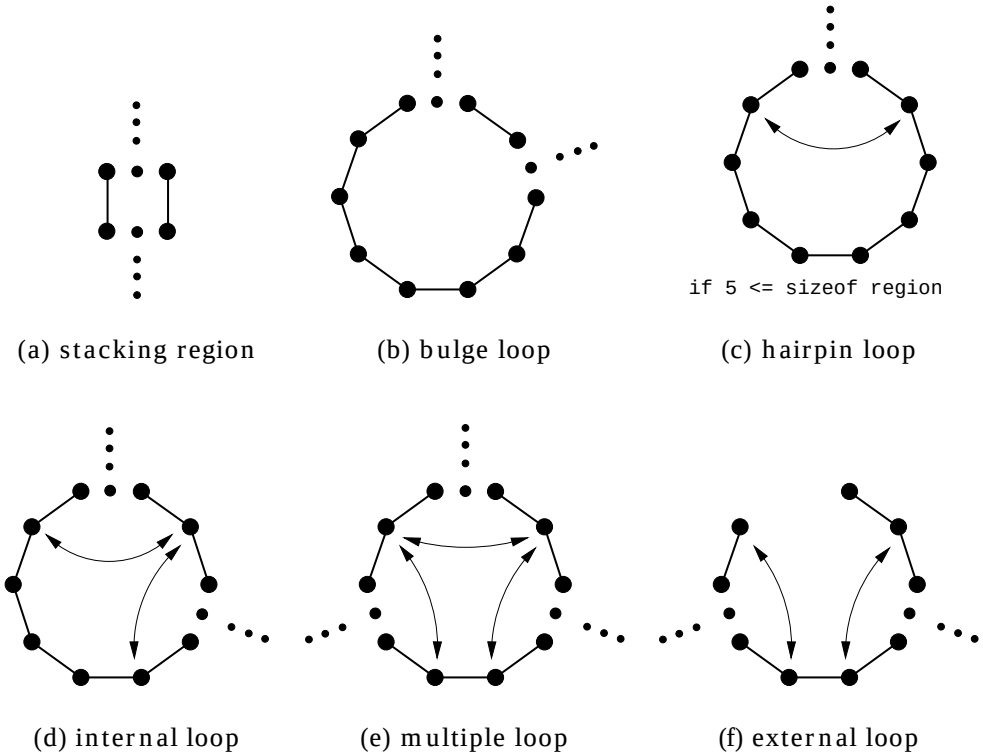
```
      strong    = sr <<< base +~~ (
                            (hairpin ||| leftB ||| rightB ||| iloop ||| multiloop)
                            'with' basepairing)
                          ~~+ base ... h
                  where

     hairpin    = hl <<< base +~~  (region 'with' minLoopSize 3)  ~~+ base
     leftB      = bl <<< base +~~ region  ~~~ p closed            ~~+ base
     rightB     = br <<< base +~~             p closed ~~~ region ~~+ base

     iloop      = iln <<< base +~~ p inloop ~~+ base
                  where
      inloop    = tabulated (
                    ilx <<< base +~~ p inloop ~~+ base |||
                                     p loopend         ... h_i)
                  where
       loopend  = tabulated (
                  ill <<< region ~~+ base ~~~ p closed ~~+ base            |||
                  ilr <<<            base +~~ p closed ~~+ base ~~~ region |||
                  ils <<<            base +~~ p closed ~~+ base            ... h_i)

     multiloop  = ml    <<< base +~~          ml_comps          ~~+ base |||
                  mldl  <<< base +~~ base ++~ ml_comps          ~~+ base |||
                  mldr  <<< base +~~          ml_comps ~~+ base ~~+ base |||
```

```
            mldlr <<< base +~~ base ++~ ml_comps ~~+ base ~~+ base
            where
 ml_comps = combine <<< p block ~~~ p comps

comps     = tabulated (
            concate <<< p block ~~~ p comps |||
                        p block             |||
            addss   <<< p block ~~~ region  ... h_l)

block     = tabulated (
            ul    <<<           dangle |||
            ssadd <<< region ~~~ dangle ... h_l)
            where
  dangle  = dl   <<< base +~~ p closed          |||
            dr   <<<          p closed ~~+ base |||
            dlr  <<< base +~~ p closed ~~+ base |||
            drem <<<          p closed          ... h_d
```

The algebras for the 'no lonely pairs' parser are the same as for the 'dangling end' parser in Section 8.3 and are not shown here.

9.4 Saturated Structures

Pruning the search space to represent only saturated structures entails checking every secondary structure element for single bases in the direct neighborhood of base pairs, rejecting those that could form a legal base pair and still represent a legal secondary structure. In the following we will discuss all secondary structure elements and the checks necessary to ensure saturation (see also Fig. 9.2).



(a) stacking region (b) bulge loop (c) hairpin loop

(d) internal loop (e) multiple loop (f) external loop

Figure 9.2: Examples of secondary structure elements with saturation checks indicated by arrows.

Stacking Region The stacking region element contains no single stranded bases. It therefore has no need of saturation checks (see Fig. 9.2 (a)).

Bulge Loop The same holds true for bulge loops, because there are only unpaired bases on one side of the adjacent helices (see Fig. 9.2 (b)).

Hairpin Loop For sterical reasons the single strand of a hairpin loop has to have a minimum length of three bases. So, saturation is only tested if the alternative structure also fulfills the minimum length restriction, i.e. when the overall length of the single strand is greater or equals five (see Fig. 9.2 (c)).

Internal Loop In the case of the internal loop the saturation check has to be performed unconditionally, because single stranded bases on both sides of the helices are guaranteed and all alternative structures, i.e. bulge loops, or stacking regions, are legal secondary structures (see Fig. 9.2 (d)).

Multiple or External Loop In multiple as well as external loops, all single stranded regions interspersed by a closed structure potentially violate the saturation constraint and have to be checked at the end neighboring the helical region. Again, all alternative structures are legal (see Fig. 9.2 (e,f)).

9.4.1 Discussion

It is immediatly obvious that the saturation check for simple structure elements like the hairpin or the internal loop are simple filters that can be added to the existing productions without complex changes in the structural decomposition of the recognizers.

The multiple loop decomposition, however, now needs to ensure not only a minimum of three closed sub-structure elements in the loop but also has to single out the situation when a closed structure is embedded between two single stranded regions where the saturation check has to be performed. One solution is to spell out all the patterns involving the first two closed structures in the loop and then enter in to a recursive structure as shown in the following ADP fragment.

```
multiloop    = ml <<< p_base +~~ mlcomps ~~+ p_base

mlcomps      = p   sccu ||| p  ccu   |||
               p  csmsr ||| p  csccu ||| p  csc |||
               p smsmsr ||| p smsccu ||| p smsc
```

Mnemonic used

s: singlestrand

c: stack

m: saturated stack

r: starts with saturated stack (restricted)

u: starts with stack or single strand (unrestricted)

```
sat          = p closed 'within' nomatch inp
sr           = tabulated (cons <<< region   ~~~ p rcomps)
cu           = tabulated (cons <<< p closed ~~~ p ucomps)
ccu          = tabulated (cons <<< p closed ~~~ p cu    )
sccu         = tabulated (cons <<< region   ~~~ p ccu   )
```

```
  csc          = tabulated (cons <<< p closed ~~~ p sc    )
  sc           = tabulated (cons <<< region   ~~~ (ul <<< p closed))
  csccu        = tabulated (cons <<< p closed ~~~ p sccu  )
  csmsr        = tabulated (cons <<< p closed ~~~ p smsr  )
  smsr         = tabulated (cons <<< region   ~~~ p msr   )
  msr          = tabulated (cons <<< sat      ~~~ p sr    )
  smsc         = tabulated (cons <<< region   ~~~ p msc   )
  msc          = tabulated (cons <<< sat      ~~~ p sc    )
  smsccu       = tabulated (cons <<< region   ~~~ p msccu )
  msccu        = tabulated (cons <<< sat      ~~~ p sccu  )
  smsmsr       = tabulated (cons <<< region   ~~~ p msmsr )
  msmsr        = tabulated (cons <<< sat      ~~~ p smsr  )
```

Moreover, saturation of the closing base pair has to be ensured, leading to even more productions and tables, because the cases where the loop recursion ends in a closed structure versus a single stranded region have to be separated out. In the latter case a saturation check is necessary if the loop recursion starts with a single stranded region.

The other, more compact solution involves an extension of the ADP framework by so called attribute parsers introduced in the next sub-section. This will make it possible to simply count the number of closed structure elements encountered in the structural recursion and to conditionally allow alternatives in the productions to be executed, or to annotate solutions with information about the productions involved.

9.5 Attribute Combinators

```
module AttributeCombinators
 where
 import RNA
 import Utils
```

As demonstrated in sub-section 4.1.6 the search space of a DP problem is represented by a rooted, directed, acyclic graph. Normaly we think of information flowing upward from the evaluation of the leaf nodes toward the root of the graph to give us the answer to our problem. It is interesting to note, however, that information also travels downward towards the leaves in the form of the input boundaries.

Clearly, adding further parameters, which we will call *attributes*, does not give us more power. There is no gain in space or time complexity, because, as is immediatly obvious, every attribute can be replaced by separate productions with their respective tables. What it gives us is a more compact representation and expressiveness in our productions.

What happens if we add further data? In top-down direction this amounts to adding a further parameter, i.e. a further dimension to the input.

```
 type Input a = (Int,Int,a)

 type TopDownAttributeParser tdAttribute parse = Input tdAttribute -> [parse]
```

In bottom-up direction we can simply structure the output of the parsers to carry suplemental information by forming a tuple.

```
 type BottomUpAttributeParser tdAttribute  parse buAttribute =
      TopDownAttributeParser  tdAttribute (parse,buAttribute)
```

9.5.1 Bottom-Up Attributes

A bottom-up attribute parser is a regular (top-down) parser whose return values are augmented by a constant.

Definition 20 «attach combinator» The attach combinator adds the symbol y to every element of the derivation set of P_X: $P_{X\uparrow y}(i,j,a) := [(t,y) \mid t \leftarrow P_X(i,j,a)]$. □

```
(<==>) :: TopDownAttributeParser a b -> c -> BottomUpAttributeParser a b c
(p <==> bu) inp = [(x,bu) | x <- p inp]
```

So, in the bottom-up direction the additional data are stored with the evaluation data and should be of constant size, otherwise the tables needed to store the intermediate results would in fact gain a dimension.

The interpretation of bottom-up attributes happens in parallel to the interpretation of the parse results.

Definition 21 «attribute interpretation combinator» The attribute interpretation combinator applies the algebras evaluation function I_X to the derivations of P_X and the attribute evaluation function A_X to the corresponding attributes:
$P_{I,A_X}(i,j,a) := [(I_X,A_X)(t) \mid t \leftarrow P_{X\uparrow y}(i,j,a)]$. □

In Haskell we need to provide a separate attribute interpretation combinator for every type signature. We do so for all patterns involving two or three sub-parsers. A plus indicates a bottom-up attribute in the corresponding position of the production. An equals sign indicates no attribute. Thus, <=+> stands for a production where the first sub-parser contains no attribute and the second does.

```
(e <++> f) (x,ax) (y,ay) = (e x y, f ax ay)
(e <+=> f) (x,ax)  y     = (e x y, f ax)
(e <=+> f)  x     (y,ay) = (e x y, f ay)

(e <+++> f) (x,ax) (y,ay) (z,az) = (e x y z, f ax ay az)
(e <++=> f) (x,ax) (y,ay)  z     = (e x y z, f ax ay)
(e <+==> f) (x,ax)  y      z     = (e x y z, f ax)
(e <=++> f)  x     (y,ay) (z,az) = (e x y z, f ay az)
(e <==+> f)  x      y     (z,az) = (e x y z, f az)
(e <+=+> f) (x,ax)  y     (z,az) = (e x y z, f ax az)
(e <=+=> f)  x     (y,ay)  z     = (e x y z, f ay)
```

Some primitive attribute evaluation functions are in order here. `flagTrue` and `flagFalse` test a boolean attribute.

```
flagTrue :: (r,Bool) -> Bool
flagTrue = snd

flagFalse :: (r,Bool) -> Bool
flagFalse = not . snd
```

While `strip` removes the attribute from the derivation.

```
strip :: (a,b) -> a
strip = fst
```

9.5.2 Bottom-Up Attribute Example

The following example demonstrates the use of bottom-up attributes to ensure saturation of the closing base pair of the multiple loop.

```
multiloop  = ml <<< base +~~ mlblocks ~~+ base
             where
  mlblocks = tabulated 0 (
        concat <<< (strip <<<  p block) ~~~
                   (strip <<< (p comps 'suchthat' flagFalse)) |||

        concat <<< (strip <<< (p block 'suchthat' flagFalse )) ~~~
                   (strip <<< (p comps 'suchthat' flagTrue )) |||
```

The saturation case: The loop starts and ends with a single strand.

```
concat <<< (strip <<< (p block ‘suchthat‘ flagTrue )) ~~~
           (strip <<< (p comps ‘suchthat‘ flagTrue ))
            ‘with‘ (not . basepairing) ... h)
```

When the block starts with a single stranded region the attribute `True` is attached.

```
block   = tabulated 0 (
          (ul    <<<           p closed) <==> False |||
          (ssadd <<< region ~~~ p closed) <==> True  ... h_l)
```

The attribute from the `block` parser is not needed in `comps`, so it is discarded. `True` is attached whenever the loop ends in a single stranded region. In the recursive alternative the attribute from the `comps` parser is passed upward.

```
comps   = tabulated 0 (
          (concat <=+> id) <<< (strip <<< p block) ~~~ p comps           |||
                               (strip <<< p block)           <==> False |||
          addss            <<< (strip <<< p block) ~~~ region <==> True  ... h_l)
```

9.5.3 Top-Down Attribute Example

Consider the case where a structural element has to appear in succession for a constant number of times. The cloverleaf structure of tRNAs are such a case (see Figure 3.1). The central multiple loop has to contain exactly three helices. One way to achieve this is by chaining the productions the right number of times as in the following fragment:

```
multiloop   = tabulated (
              ml <<< base +~~ p firstBlock ~~~ p secondBlock ~~+ base ... h)

firstBlock  = tabulated (
              ul    <<<           p closed |||
              ssadd <<< region ~~~ p closed ... h)

secondBlock = tabulated (
                        p block            |||
              addss  <<< p block ~~~ region  ... h)
```

Using top-down attributes we can simply count how many times a production containing a helix was called in a direct recursion.

```
multiloop   = setAttr 0 >>> (
              ml <<< base +~~ ap mlcomps ~~+ base)

mlcomps     = attributed 0 1 ( setAttr 1 >>> (
                badd  <<< p block ~+~ ap mlcomps |||
              (         p block                |||
               addss <<< p block ~~~ region ) ‘with‘ attrEquals 1) ... h)

block       = tabulated 0 (
              ul    <<<           p closed |||
              ssadd <<< region ~~~ p closed ... h)
```

The table used in `mlcomps` contains the space of two regular tables, because its attribute can hold two values (0 or 1). Thus, the exact same tabulation space is used and therefore the space and time complexity of the solution using top-down attributes stays the same.

Another aspect of using top-down attributes is flexibility. What if we were to decide that we needed exactly four helices in a multiple loop? In the solution using top-down attributes we simply need to change the attribute numbers and the attribute table size, whereas we would have to come up with new productions in the regular case.

Note that we purposely used new combinators in this example without giving their definition first. This was done to illustrate that having read the thesis this far, the reader

should have an idea of the semantics of the new combinators due to the declarative nature of ADP. Of course the most important top-down attribute combinators will be introduced next.

9.5.4 Top-Down Attribute Combinators

Two new combinators are added to facilitate the manipulation of top-down attributes.

Definition 22 «attribute transformation combinator» The attribute transformation combinator applies the function f to the attribute parameter a of the input. $P_{X\downarrow F}(i,j,a) := P_X(i,j,F(a))$. □

The general attribute transformation combinator >>> takes the funktion f as first input and applies it to the attribute portion of the input before calling the parser p.

```
infixl 9 >>>
(>>>) :: (a -> a) -> TopDownAttributeParser a b -> TopDownAttributeParser a b
(f >>> p) (i,j,a) =  p (i,j, f a)
```

Next, is the set attribute combinator, that replaces the attribute in the input with its first argument a.

```
infixl 9 >><
(>><) :: a -> TopDownAttributeParser a b -> TopDownAttributeParser a b
(a >>< p) (i,j,_) =  p (i,j,a)
```

The alternative, interpretation, and choice combinators are identical to the non attributed versions. The juxtaposition combinator also remains fundamentally unchanged. It now has to pass on the attribute argument to the sub-parsers. The same holds for all its variants as well.

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

The filter combinators with, within, and suchthat are unaffected by the signature change of the attribute parsers as in the case of the juxtaposition combinators. They now fulfill the additional function as filters for attributes.

Interesting changes occur in the tabulation functions, in the sense that they make an appearance in the syntax of the resulting ADP recognizer. Shown here once again, but already demonstrated in the introductory examples:

```
mlblocks = tabulated 0 ( ... )
```

The un-attributed tables now need a further parameter to pass on as a constant to the underlying parser. The same holds for the axiom parser combinator not shown here.

```
type Parsetable b = Array Region [b]

tabulated :: a -> TopDownAttributeParser a b -> Parsetable b
tabulated a p = array ((0,0),(rnaLen,rnaLen))
              [ ((i,j),p (i,j,a)) | i<- [0..rnaLen], j<- [i..rnaLen] ]

p :: Parsetable b -> TopDownAttributeParser a b
p table (i,j,_) = if i <= j then table!(i,j) else []

type Parselist b = Array Int [b]

listed :: a -> TopDownAttributeParser a b -> Parselist b
listed a p = array (0,rnaLen)
              [ (i, p (i,rnaLen,a)) | i<- [0..rnaLen] ]
```

```
q :: Parselist b -> TopDownAttributeParser a b
q table (i,_,_) = table!i
```

The attributed parse tables use new parameters to give the bounded range of the extra dimension needed to accomodate the derivations of the top-down attribute parsers.

```
type AttributeParsetable a b = Array (Int,Int,a) [b]

attributed :: (Ix a, Enum a) => a -> a -> TopDownAttributeParser a b
           -> AttributeParsetable a b
attributed n m p = array ((0,0,n),(rnaLen,rnaLen,m))
  [ ((i,j,a), p (i,j,a)) | i<- [0..rnaLen], j<- [i..rnaLen], a <- [n..m] ]
```

Likewise, the new ap attribute table lookup function needs to pass on the additional attribute argument.

```
ap :: (Ix a) => AttributeParsetable a b -> TopDownAttributeParser a b
ap table (i,j,a) = if i <= j then table!(i,j,a) else []
```

The one-dimensional case is analogous.

```
type AttributeParselist a b = Array (Int,a) [b]

alisted :: (Ix a, Enum a) => a -> a -> TopDownAttributeParser a b
        -> AttributeParselist a b
alisted n m p = array ((0,n),(rnaLen,m))
              [ ((i,a), p (i,rnaLen,a)) | i<- [0..rnaLen], a <- [n..m] ]

aq :: (Ix a) => AttributeParselist a b -> TopDownAttributeParser a b
aq table (i,_,a) = table!(i,a)
```

The complete set of combinators and parsers can be found in Appendix A.6.

9.6 Saturated Structures in ADP

The recognizer for saturated RNA secondary structures follows the same layout as the canonical recognizers from the chapters 6 and 7. The productions for the external and the multiple loop use a different decomposition to isolate the case where a closed structure is enclosed by single stranded regions on both sides. This is where the saturation check is performed.

Instead of combining a region `~~~` closed and closed to build chains of blocks containing a helix and to ensure that a minimum of two are used to build a loop, the strategy now is to separate the case of a closed structure living in the left context of a single stranded region from the case where it follows after a closed structure. In the first case, it is clear that a saturation check has to be performed whenever the sub-production for chaining a single stranded region is chosen.

```
module Saturated
where
import RNA
import AttributeCombinators
import FreeEnergy
import EnergyTables
import Utils

evers01e algebra = axiom 0 external
  where
  (el,sr,hl,bl,br,
   iln,ilx,ill,ilr,ils,
   ml,ul,addss,ssadd,cadd,
   eul,ecadd,esadd,
   nil,
   h,h_l,h_s,h_i,h_k) = algebra

  external = el <<< (q ec ||| q es ||| nil ><< empty) ... h_s
             where
```

The external loop productions of the saturated recognizer will make this clearer. ec is the production that is used whenever a closed structure is in the left context of another closed structure or the beginning of the loop. There are three alternatives. Either the loop ends with this closed structure, or it continues with another closed structure which in turn will be in the left context of our current helix, or it will be followed by a single strand. In all three cases no saturation check has to be performed.

```
    ec = listed 0 (
    eul   <<< p closed          |||
  ecadd <<< p closed ~~~ q ec |||
  ecadd <<< p closed ~~~ q es ... h_k)
```

The single strand can only be followed by a closed structure. In this case the closed structure will be in the left context of the current unpaired region.

```
    es = listed 0 (
         esadd <<< region ~~~ q ex )
```

These are the productions for the closed structure living in the left context of a single strand. The first alternative is the termination of a loop ending in a closed structure. The second case is the ending in a single strand. The following case continues with a closed structure. In all these cases there is no saturation check because, either there is no closed structure, or there is no downstream single strand. The last case continues with a single strand. This is the case living in the left and right context of an unpaired region, which is why the production satclosed for a saturated closed structure is used.

```
    ex = listed 0 (
         eul   <<< p closed           |||
         nil   >>< empty              |||
         ecadd <<< p closed  ~~~ q ec |||
         ecadd <<< satclosed ~~~ q es ... h_k)
```

The production satclosed ensures that the production closed lives within unpaired bases.

```
    satclosed   = p closed `within` (not . basepairing)
```

We use the 'no lonely pairs' version of the canonical recognizer here.

```
    closed      = tabulated 0 (
                  ((stack ||| strong)
                    `with` basepairing) ... h)
                  where
     stack      = sr <<<  base +~~               p closed              ~~+ base
     strong     = sr <<<  base +~~ (
                            (hairpin ||| leftB ||| rightB ||| iloop ||| multiloop)
                            `with` basepairing)
                          ~~+ base ... h
                  where
```

Remember that the hairpin has to be checked for saturation only if it is large enough. This is taken care of in saturatedHairpin (shown farther down).

```
      hairpin   = hl <<<  base +~~   region `with` saturatedHairpin    ~~+ base
      leftB     = bl <<<  base +~~   region  ~~~ p closed              ~~+ base
      rightB    = br <<<  base +~~               p closed ~~~ region   ~~+ base
```

The internal loop needs to be checked for saturation at the beginning and end of the loop.

```
iloop     = iln <<< base +~~ p inloop ‘with‘ (not . basepairing) ~~+ base
            where
 inloop   = tabulated 0 (
              ilx <<< base +~~ p inloop ~~+ base |||
                               p loopend         ... h_i)
            where
  loopend = tabulated 0 (
            ill <<< region ~~+ base ~~~ satclosed ~~+ base            |||
            ilr <<<            base +~~ satclosed ~~+ base ~~~ region |||
            ils <<<            base +~~ satclosed ~~+ base            ... h_i)

multiloop = ml <<<  base +~~               mlcomps             ~~+ base
            where
```

The multiple loop uses the same decomposition as the external loop at the beginning of the recognizer. In addition the minimum number of helices in the loop has to be ensured. This is achieved using top-down attributes to count how often a production containing a `closed` or `satclosed` parser was applied. Entering the loop in `mlcomps` the top-down attribute is set to zero.

```
 mlcomps = 0 >>< (
           strip <<<  ap mc |||
```

Bottom-up attributes are used to discern two variants of loops. Those ending in a closed structure and those ending in a single strand. Only in the (following) case where the loop starts with a single strand `ms` and also ends in a single strand ‘`suchthat`‘ `flagTrue` a saturation check for the closing base pair in `multiloop` (above) has to be performed.

```
           strip <<< (ap ms ‘suchthat‘ flagTrue) ‘with‘ (not . basepairing) |||
           strip <<<  ap ms ‘suchthat‘ flagFalse) ... h_k
           where
```

The clauses containing closed structures in the left context of a closed structure set the top-down attribute to one. The results of the terminating clauses are attached with a boolean bottom-up attribute. The terminating clauses are only activated if the top-down attribute is one. See Figure 9.3 for an example decomposition.

```
   mc = attributed 0 1 (
         (cadd <=+> id) <<< p closed ~~~ (1 >>< ap mc)      |||
         (cadd <=+> id) <<< p closed ~~~ (1 >>< ap ms)      |||
        ((ul            <<< p closed )          <==> False |||
         (addss         <<< p closed ~~~ region) <==> True)
        ‘with‘ attrEquals 1 ... h_l)

   ms = attributed 0 1 (
        (ssadd <=+> id) <<< region ~~~ ap mx ... h_l)

   mx = attributed 0 1 (
         (cadd <=+> id) <<< p closed  ~~~ (1 >>< ap mc)      |||
         (cadd <=+> id) <<< satclosed ~~~ (1 >>< ap ms)      |||
        ((ul            <<< p closed)            <==> False |||
         (addss         <<< satclosed ~~~ region) <==> True)
        ‘with‘ attrEquals 1 ... h_l)
```

The filter for saturated hairpins only checks for saturation if the unpaired loop is at least 5 bases long.

```
saturatedHairpin :: Input a -> Bool
saturatedHairpin inp = if minLoopSizeA 5 inp
                       then (not . basepairing) inp
                       else minLoopSizeA 3 inp
```

The algebras use additional functions to accomodate the new external and multiple loop decompositions.

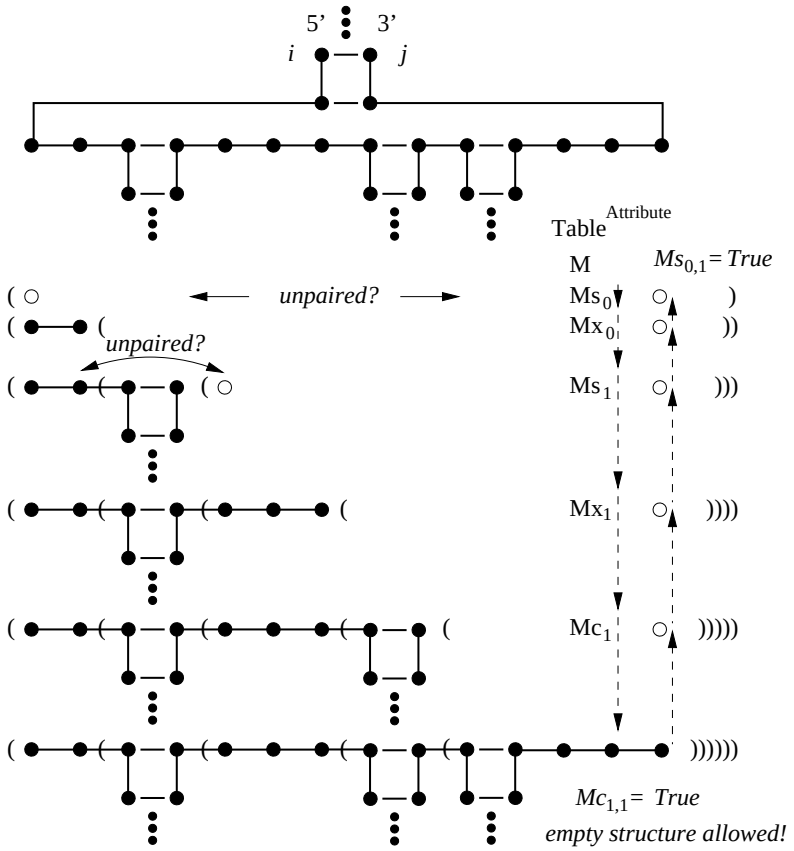


Figure 9.3: An Example Decomposition of a Multiple Loop

```
saturatedCountAlg = (id,sr,hl,bl,br,
                     iln,ilx,ill,ilr,ils,
                     ml,id,addss,ssadd,cadd,
                     id,ecadd,esadd,
                     nil,
                     addup,h_l,addup,addup,addup)
  where
  sr    _    e    _ = e
  hl    _    _    _ = 1::Integer
  bl    _ _  e    _ = e
  br    _    e  _ _ = e
  iln   _    e    _ = e
  ilx   _    e    _ = e
  ilr   _ e  _ _    = e
  ill     _ _ e  _ = e
  ils   _    e    _ = e
  ml    _  e  _    = e
  addss   e  _    = e
  ssadd   _  e    = e
  cadd    e  c    = e * c
  ecadd   e  c    = e * c
  esadd   _  e    = e
  nil             = 1::Integer

  h_l [] = []
  h_l xs = [(a,True),(b,False)] where
           a = sum (map fst (filter snd xs))
           b = sum (map fst (filter (not . snd) xs))

saturatedFreeEnergyAlg = (id,sr,hl,bl,br,
                          iln,ilx,ill,ilr,ils,
```

```
                        ml,ul,addss,ssadd,cadd,
                        id,ecadd,esadd,
                        nil,
                        minimize,h_l,minimize,h_i,minimize)
  where
  sr   i       e       j = e + dg_sr (i,j)
  hl   i       _       j =     dg_hl (i,j)
  bl   i (_,n) e       j = e + dg_bl (i,j) (n+1,j-1)
  br   i       e (m,_) j = e + dg_bl (i,j) (i+1,m)
  iln  i     (e,k)     j = e + top_stack (i,j)
  ilx  _     (e,k)     _ = (e + (ent_il (k+2)) - (ent_il k),                                  k+2)
  ill  (i,j) m e n     = (e + (asym (j-i))   + (bot_stack (m,n)) + (ent_il (j-i+2)), j-i+2)
  ilr        m e n (i,j) = (e + (asym (j-i))   + (bot_stack (m,n)) + (ent_il (j-i+2)), j-i+2)
  ils  i       e       j = (e + (asym 0)       + (bot_stack (i,j)) + (ent_il 2),      2::Int)
  ml   _       e       _ = e + ml_init_penalty
  ul           e         = e + ml_helix_penalty
  addss       e r        = e + ml_helix_penalty + ml_unpaired r
  ssadd       r e        = e + ml_unpaired r
  cadd        e c        = e + c + ml_helix_penalty
  ecadd :: Energy -> Energy -> Energy
  ecadd       e c        = e + c
  esadd       _ e        = e
  nil                    = 0.0::Energy

  h_i :: [(Energy,Int)] -> [(Energy,Int)]
  h_i [] = []
  h_i es =  [minimum es]
```

The free energy choice function for the multiple loop has to separate the two boolean bottom-up attribute cases. This is the proof that we save no space, but instead pack two DP tables into one, by combining the contents of the corresponding cells.

```
h_l :: [(Energy,Bool)] -> [(Energy,Bool)]
h_l [] = []
h_l xs =  c ++ d where
        c = if a == [] then [] else [(minimum a, True)] where
            a = map fst (filter snd xs)
        d = if b == [] then [] else [(minimum b, False)] where
            b = map fst (filter (not . snd) xs)
```

9.7 Discussion

The following discussion of the results obtained with the recognizer for the *saturated RNA secondary structure problem* is taken from the paper presented at GCB'01 (Evers and Giegerich, 2001). The standard recurrences can also be found in this paper and are not repeated here.

RNA	*n*	saturated			no lonely pairs			regular		
% mfe:		10%	20%	all	10%	20%	all	10%	20%	all
(a)	56	4	19	2,890,038	9	46	25,599,618	9	49	6,431,846,133,769
(b)	69	6	22	115,864,112	10	168	2,203,497,483	10	196	22,497,763,081,040,577
(c)	40	1	5	1,857	1	5	4,222	1	6	55,611,266

Table 9.1: Structure Space Reduction in Three RNA Examples
(a) Spliced Leader RNA from Leptomonas collosoma, (b) Tetrahymena thermophila Group I Ribozyme Domain, (c) Neurospora crassa 5S ribosomal RNA. Percentages denote range from the minimum free energy, n denotes RNA sequence length.

Our data (see table above) show the reduction of the conformation space for three RNA examples. We see the expected dramatic reduction in the overall number of saturated versus regular structures. More importantly, consider the structures within 10% and 20% of the mfe optimum. Naturally, the reduction is smaller here, since low energy structures tend to be saturated. But the reduction is still significant in the 10-20%

range. Specifically in the case of *Leptomonas collosoma*, which is known to act as a conformational switch, the alternative conformation shows up at the 11% level. To evaluate the benefit that can result from such a search space reduction, consider, for example, the paRNAss approach (Giegerich et al., 1998) for the prediction of conformational switches in RNA. This heuristic algorithm is quadratic in the size of the structure space sample which needs to be collected. Based on the traditional recurrences, it requires 49 feasible structures and 2401 comparisons of structures to predict the conformational switch. With the saturated approach, 19 structures and hence only 361 structure comparisons should suffice to obtain the same result.

9.7.1 Local Minima

It is important to note that saturated structures are not local minima in the free energy landscape of an RNA. Ellen Latz has provided an example in her Diploma thesis (Latz, 2000) on pages 28 and 29. In hairpins the energy of a lonely pair in the vicinity of an extra stable tetra loop can be lower than a slightly larger hairpin loop without an internal loop. This also implies, that the saturated mfe structure can differ from the regular mfe structure of an RNA!

A Functions

A.1 Energy Functions

```
module EnergyTables
where
import RNA
import Utils
```

Some constants and utilities

```
e :: Float
e = 2.718281828459
```

Zero Degrees Celsius in Kelvin.

```
t :: Float
t = 273.15
```

Universal Gas Constant

```
r :: Float
r = 8.3143
```

Convert Celsius degrees to Kelvin degrees.

```
kelvin :: Float -> Float
kelvin cels = t + cels
```

The free energy is measured in $\frac{kcal}{mol}$.

```
type Energy = Float -- kcal/mol
```

The Jacobson-Stockmayer term for loop interpolation.

```
jacobson_stockmayer :: LoopSize -> Energy
jacobson_stockmayer size = 1.079 * logBase e ((fromIntegral size) / 30.0)
```

A.2 Stacking Region Energies

A.2.1 Stacking Energies

```
t_sr :: Base -> Base -> Base -> Base -> Energy
```

```
t_sr A A U U = -0.9
t_sr A C G U = -2.1
t_sr A G C U = -1.7
t_sr A G U U = -0.5
```

```
t_sr A U A U = -0.9
t_sr A U G U = -1.0
t_sr C A U G = -1.8
t_sr C C G G = -2.9
t_sr C G C G = -2.0
t_sr C G U G = -1.2
t_sr C U A G = -1.7
t_sr C U G G = -1.9
t_sr G A U C = -2.3
t_sr G A U U = -1.1
t_sr G C G C = -3.4
t_sr G C G U = -2.1
t_sr G G C C = -2.9
t_sr G G U C = -1.4
t_sr G G C U = -1.9
t_sr G G U U = -0.4
t_sr G U A C = -2.1
t_sr G U G C = -2.1
t_sr G U A U = -1.0
t_sr G U G U =  1.5
t_sr U A U A = -1.1
t_sr U A U G = -0.8
t_sr U C G A = -2.3
t_sr U C G G = -1.4
t_sr U G C A = -1.8
t_sr U G U A = -0.8
t_sr U G C G = -1.2
t_sr U G U G = -0.2
t_sr U U A A = -0.9
t_sr U U G A = -1.1
t_sr U U A G = -0.5
t_sr U U G G = -0.4

t_sr _ _ _ _ = error "t_sr: not in table"
```

A.3 Hairpin Loop Energies

A.3.1 Entropic Term

Destabilizing energies by size of hairpin loop.

```
ent_hl :: LoopSize -> Energy

ent_hl  3 = 4.1
ent_hl  4 = 4.9
ent_hl  5 = 4.4
ent_hl  6 = 4.7
ent_hl  7 = 5.0
ent_hl  8 = 5.1
ent_hl  9 = 5.2
ent_hl 10 = 5.3
ent_hl 11 = 5.4
ent_hl 12 = 5.5
ent_hl 13 = 5.6
ent_hl 14 = 5.7
ent_hl 15 = 5.8
ent_hl 16 = 5.8
ent_hl 17 = 5.9
ent_hl 18 = 5.9
ent_hl 19 = 6.0
ent_hl 20 = 6.1
ent_hl 21 = 6.1
ent_hl 22 = 6.2
ent_hl 23 = 6.2
ent_hl 24 = 6.3
ent_hl 25 = 6.3
ent_hl 26 = 6.3
ent_hl 27 = 6.4
ent_hl 28 = 6.4
ent_hl 29 = 6.5
ent_hl 30 = 6.5

ent_hl loopSize = if loopSize < 3
```

```
                then error "ent_hl: loop size < 3"
                else ent_hl 30 + jacobson_stockmayer loopSize
```

A.3.2 Tetraloop Bonus Energies

```
tetra_hl :: [Base] -> Energy

tetra_hl [A,G,A,A,A,U] =  -2.0
tetra_hl [A,G,C,A,A,U] =  -2.0
tetra_hl [A,G,A,G,A,U] =  -2.0
tetra_hl [A,G,U,G,A,U] =  -2.0
tetra_hl [A,G,G,A,A,U] =  -2.0
tetra_hl [A,U,U,C,G,U] =  -2.0
tetra_hl [A,U,A,C,G,U] =  -2.0
tetra_hl [A,G,C,G,A,U] =  -2.0
tetra_hl [A,U,C,C,G,U] =  -2.0
tetra_hl [A,G,U,A,A,U] =  -2.0
tetra_hl [A,C,U,U,G,U] =  -2.0
tetra_hl [A,A,U,U,U,U] =  -2.0
tetra_hl [A,U,U,U,A,U] =  -2.0
tetra_hl [C,G,A,A,A,G] =  -2.0
tetra_hl [C,G,C,A,A,G] =  -2.0
tetra_hl [C,G,A,G,A,G] =  -2.0
tetra_hl [C,G,U,G,A,G] =  -2.0
tetra_hl [C,G,G,A,A,G] =  -2.0
tetra_hl [C,U,U,C,G,G] =  -2.0
tetra_hl [C,U,A,C,G,G] =  -2.0
tetra_hl [C,G,C,G,A,G] =  -2.0
tetra_hl [C,U,C,C,G,G] =  -2.0
tetra_hl [C,G,U,A,A,G] =  -2.0
tetra_hl [C,C,U,U,G,G] =  -2.0
tetra_hl [C,A,U,U,U,G] =  -2.0
tetra_hl [C,U,U,U,A,G] =  -2.0
tetra_hl [G,G,A,A,A,C] =  -2.0
tetra_hl [G,G,C,A,A,C] =  -2.0
tetra_hl [G,G,A,G,A,C] =  -2.0
tetra_hl [G,G,U,G,A,C] =  -2.0
tetra_hl [G,G,G,A,A,C] =  -2.0
tetra_hl [G,U,U,C,G,C] =  -2.0
tetra_hl [G,U,A,C,G,C] =  -2.0
tetra_hl [G,G,C,G,A,C] =  -2.0
tetra_hl [G,U,C,C,G,C] =  -2.0
tetra_hl [G,G,U,A,A,C] =  -2.0
tetra_hl [G,C,U,U,G,C] =  -2.0
tetra_hl [G,A,U,U,U,C] =  -2.0
tetra_hl [G,U,U,U,A,C] =  -2.0
tetra_hl [U,G,A,A,A,A] =  -2.0
tetra_hl [U,G,C,A,A,A] =  -2.0
tetra_hl [U,G,A,G,A,A] =  -2.0
tetra_hl [U,G,U,G,A,A] =  -2.0
tetra_hl [U,G,G,A,A,A] =  -2.0
tetra_hl [U,U,U,C,G,A] =  -2.0
tetra_hl [U,U,A,C,G,A] =  -2.0
tetra_hl [U,G,C,G,A,A] =  -2.0
tetra_hl [U,U,C,C,G,A] =  -2.0
tetra_hl [U,G,U,A,A,A] =  -2.0
tetra_hl [U,C,U,U,G,A] =  -2.0
tetra_hl [U,A,U,U,U,A] =  -2.0
tetra_hl [U,U,U,U,A,A] =  -2.0
tetra_hl [G,G,A,A,A,U] =  -2.0
tetra_hl [G,G,C,A,A,U] =  -2.0
tetra_hl [G,G,A,G,A,U] =  -2.0
tetra_hl [G,G,U,G,A,U] =  -2.0
tetra_hl [G,G,G,A,A,U] =  -2.0
tetra_hl [G,U,U,C,G,U] =  -2.0
tetra_hl [G,U,A,C,G,U] =  -2.0
tetra_hl [G,G,C,G,A,U] =  -2.0
tetra_hl [G,U,C,C,G,U] =  -2.0
tetra_hl [G,G,U,A,A,U] =  -2.0
tetra_hl [G,C,U,U,G,U] =  -2.0
tetra_hl [G,A,U,U,U,U] =  -2.0
tetra_hl [G,U,U,U,A,U] =  -2.0
tetra_hl [U,G,A,A,A,G] =  -2.0
tetra_hl [U,G,C,A,A,G] =  -2.0
tetra_hl [U,G,A,G,A,G] =  -2.0
tetra_hl [U,G,U,G,A,G] =  -2.0
```

```
tetra_hl [U,G,G,A,A,G] =  -2.0
tetra_hl [U,U,U,C,G,G] =  -2.0
tetra_hl [U,U,A,C,G,G] =  -2.0
tetra_hl [U,G,C,G,A,G] =  -2.0
tetra_hl [U,U,C,C,G,G] =  -2.0
tetra_hl [U,G,U,A,A,G] =  -2.0
tetra_hl [U,C,U,U,G,G] =  -2.0
tetra_hl [U,A,U,U,U,G] =  -2.0
tetra_hl [U,U,U,U,A,G] =  -2.0

tetra_hl _ = 0.0
```

A.3.3 Mismatch Stacking Energies

```
t_hl :: Base -> Base -> Base -> Base -> Energy

t_hl A A U A = -1.0
t_hl A A U C = -0.7
t_hl A A U G = -1.8
t_hl A A A U = -0.8
t_hl A A C U = -1.0
t_hl A A G U = -1.7
t_hl A A U U = -1.0
t_hl A C G A = -1.1
t_hl A C G C = -1.1
t_hl A C G G = -2.3
t_hl A C A U = -0.7
t_hl A C C U = -0.7
t_hl A C G U = -0.7
t_hl A C U U = -0.7
t_hl A G C A = -1.4
t_hl A G U A = -1.2
t_hl A G C C = -1.0
t_hl A G U C = -0.9
t_hl A G C G = -2.1
t_hl A G U G = -2.0
t_hl A G A U = -1.5
t_hl A G C U = -1.0
t_hl A G G U = -1.0
t_hl A G U U = -1.0
t_hl A U A A = -0.8
t_hl A U G A = -0.8
t_hl A U A C = -0.7
t_hl A U G C = -0.7
t_hl A U A G = -1.5
t_hl A U G G = -1.5
t_hl A U A U = -0.8
t_hl A U C U = -0.8
t_hl A U G U = -0.8
t_hl A U U U = -0.8
t_hl C A U A = -0.8
t_hl C A U C = -0.6
t_hl C A A G = -1.4
t_hl C A C G = -2.0
t_hl C A G G = -2.1
t_hl C A U G = -1.9
t_hl C A U U = -0.6
t_hl C C G A = -1.3
t_hl C C G C = -0.6
t_hl C C A G = -1.0
t_hl C C C G = -1.1
t_hl C C G G = -1.0
t_hl C C U G = -0.8
t_hl C C G U = -0.8
t_hl C G C A = -2.0
t_hl C G U A = -1.4
t_hl C G C C = -1.1
t_hl C G U C = -0.9
t_hl C G A G = -2.1
t_hl C G C G = -1.9
t_hl C G G G = -1.4
t_hl C G U G = -1.9
t_hl C G C U = -1.5
t_hl C G U U = -1.1
t_hl C U A A = -1.0
t_hl C U G A = -1.0
```

```
t_hl C U A C = -0.7
t_hl C U G C = -0.7
t_hl C U A G = -1.4
t_hl C U C G = -1.5
t_hl C U G G = -1.4
t_hl C U U G = -1.2
t_hl C U A U = -0.8
t_hl C U G U = -0.8
t_hl G A U A = -1.8
t_hl G A A C = -1.1
t_hl G A C C = -1.3
t_hl G A G C = -2.0
t_hl G A U C = -1.3
t_hl G A U G = -1.2
t_hl G A A U = -0.8
t_hl G A C U = -1.0
t_hl G A G U = -1.7
t_hl G A U U = -1.0
t_hl G C G A = -2.0
t_hl G C A C = -1.1
t_hl G C C C = -0.6
t_hl G C G C = -0.6
t_hl G C U C = -0.5
t_hl G C G G = -1.4
t_hl G C A U = -0.7
t_hl G C C U = -0.7
t_hl G C G U = -0.7
t_hl G C U U = -0.7
t_hl G G C A = -2.1
t_hl G G U A = -2.0
t_hl G G A C = -2.3
t_hl G G C C = -1.5
t_hl G G G C = -1.4
t_hl G G U C = -1.5
t_hl G G C G = -1.4
t_hl G G U G = -1.3
t_hl G G A U = -1.5
t_hl G G C U = -1.0
t_hl G G G U = -1.0
t_hl G G U U = -1.0
t_hl G U A A = -1.7
t_hl G U G A = -1.7
t_hl G U A C = -0.8
t_hl G U C C = -0.8
t_hl G U G C = -0.8
t_hl G U U C = -0.7
t_hl G U A G = -1.0
t_hl G U G G = -1.0
t_hl G U A U = -0.8
t_hl G U C U = -0.8
t_hl G U G U = -0.8
t_hl G U U U = -0.8
t_hl U A A A = -1.0
t_hl U A C A = -0.8
t_hl U A G A = -1.8
t_hl U A U A = -0.9
t_hl U A U C = -0.5
t_hl U A A G = -1.2
t_hl U A C G = -1.4
t_hl U A G G = -2.0
t_hl U A U G = -1.4
t_hl U A U U = -0.5
t_hl U C A A = -0.7
t_hl U C C A = -0.6
t_hl U C G A = -0.3
t_hl U C U A = -0.5
t_hl U C G C = -0.5
t_hl U C A G = -0.9
t_hl U C C G = -0.9
t_hl U C G G = -0.7
t_hl U C U G = -0.7
t_hl U C G U = -0.7
t_hl U G A A = -1.8
t_hl U G C A = -0.9
t_hl U G G A = -1.2
t_hl U G U A = -0.9
t_hl U G C C = -0.8
t_hl U G U C = -0.7
```

```
t_hl U G A G = -2.0
t_hl U G C G = -1.4
t_hl U G G G = -1.3
t_hl U G U G = -1.4
t_hl U G C U = -1.2
t_hl U G U U = -0.9
t_hl U U A A = -0.3
t_hl U U C A = -0.6
t_hl U U G A = -0.3
t_hl U U U A = -0.5
t_hl U U A C = -0.7
t_hl U U G C = -0.7
t_hl U U A G = -0.9
t_hl U U C G = -1.1
t_hl U U G G = -0.9
t_hl U U U G = -0.9
t_hl U U A U = -0.8
t_hl U U G U = -0.8

t_hl _ _ _ _ = error "t_hl: not in table"
```

A.3.4 Entropic Term

Destabilizing energies by size of bulge loop.

```
ent_bl :: LoopSize -> Energy

ent_bl  1 = 3.9
ent_bl  2 = 3.1
ent_bl  3 = 3.5
ent_bl  4 = 4.2
ent_bl  5 = 4.8
ent_bl  6 = 5.0
ent_bl  7 = 5.2
ent_bl  8 = 5.3
ent_bl  9 = 5.4
ent_bl 10 = 5.5
ent_bl 11 = 5.7
ent_bl 12 = 5.7
ent_bl 13 = 5.8
ent_bl 14 = 5.9
ent_bl 15 = 6.0
ent_bl 16 = 6.1
ent_bl 17 = 6.1
ent_bl 18 = 6.2
ent_bl 19 = 6.2
ent_bl 20 = 6.3
ent_bl 21 = 6.3
ent_bl 22 = 6.4
ent_bl 23 = 6.4
ent_bl 24 = 6.5
ent_bl 25 = 6.5
ent_bl 26 = 6.5
ent_bl 27 = 6.6
ent_bl 28 = 6.7
ent_bl 29 = 6.7
ent_bl 30 = 6.7

ent_bl size = if size < 1
              then error "ent_bl: size < 1"
              else ent_bl 30 + jacobson_stockmayer size
```

A.4 Bulge Loop Energies

A.4.1 Entropic Term

Destabilizing energies by size of internal loop.

```
ent_il :: LoopSize -> Energy

ent_il  2 = 4.1
```

```
ent_il  3 = 5.1
ent_il  4 = 4.9
ent_il  5 = 5.3
ent_il  6 = 5.7
ent_il  7 = 5.9
ent_il  8 = 6.0
ent_il  9 = 6.1
ent_il 10 = 6.3
ent_il 11 = 6.4
ent_il 12 = 6.4
ent_il 13 = 6.5
ent_il 14 = 6.6
ent_il 15 = 6.7
ent_il 16 = 6.8
ent_il 17 = 6.8
ent_il 18 = 6.9
ent_il 19 = 6.9
ent_il 20 = 7.0
ent_il 21 = 7.1
ent_il 22 = 7.1
ent_il 23 = 7.1
ent_il 24 = 7.2
ent_il 25 = 7.2
ent_il 26 = 7.3
ent_il 27 = 7.3
ent_il 28 = 7.4
ent_il 29 = 7.4
ent_il 30 = 7.4

ent_il size = if size < 2
              then error "ent_il: size < 2"
              else ent_il 30 + jacobson_stockmayer size
```

A.4.2 Mismatch Stacking Energies

```
t_il :: Base -> Base -> Base -> Base -> Energy

t_il A A U A = -1.0
t_il A A U C = -1.0
t_il A A U G = -2.2
t_il A A A U = -1.0
t_il A A C U = -1.0
t_il A A G U = -2.2
t_il A A U U = -0.5
t_il A C G A = -1.5
t_il A C G C = -1.5
t_il A C G G = -2.7
t_il A C A U = -1.0
t_il A C C U = -1.0
t_il A C G U = -0.2
t_il A C U U = -1.0
t_il A G C A = -1.5
t_il A G U A = -1.0
t_il A G C C = -1.5
t_il A G U C = -1.0
t_il A G C G = -2.7
t_il A G U G = -2.2
t_il A G A U = -2.2
t_il A G C U = -0.5
t_il A G G U = -1.0
t_il A G U U = -0.5
t_il A U A A = -1.0
t_il A U G A = -1.5
t_il A U A C = -1.0
t_il A U G C = -1.5
t_il A U A G = -2.2
t_il A U G G = -2.7
t_il A U A U = -0.3
t_il A U C U = -1.0
t_il A U G U = -0.3
t_il A U U U = -2.0
t_il C A U A = -1.0
t_il C A U C = -1.0
t_il C A A G = -1.5
t_il C A C G = -1.5
t_il C A G G = -2.7
```

```
t_il C A U G = -1.9
t_il C A U U = -1.0
t_il C C G A = -1.5
t_il C C G C = -1.5
t_il C C A G = -1.5
t_il C C C G = -1.5
t_il C C G G = -1.0
t_il C C U G = -1.5
t_il C C G U = -1.5
t_il C G C A = -1.5
t_il C G U A = -1.0
t_il C G C C = -1.5
t_il C G U C = -1.0
t_il C G A G = -2.7
t_il C G C G = -1.9
t_il C G G G = -1.5
t_il C G U G = -1.9
t_il C G C U = -1.5
t_il C G U U = -1.0
t_il C U A A = -1.0
t_il C U G A = -1.5
t_il C U A C = -1.0
t_il C U G C = -1.5
t_il C U A G = -1.4
t_il C U C G = -1.5
t_il C U G G = -1.4
t_il C U U G = -2.5
t_il C U A U = -1.0
t_il C U G U = -1.5
t_il G A U A = -2.2
t_il G A A C = -1.5
t_il G A C C = -1.5
t_il G A G C = -2.7
t_il G A U C = -1.3
t_il G A U G = -1.0
t_il G A A U = -1.5
t_il G A C U = -1.5
t_il G A G U = -2.7
t_il G A U U = -1.3
t_il G C G A = -2.7
t_il G C A C = -1.5
t_il G C C C = -1.5
t_il G C G C = -0.6
t_il G C U C = -1.5
t_il G C G G = -1.5
t_il G C A U = -1.5
t_il G C C U = -1.5
t_il G C G U = -0.6
t_il G C U U = -1.5
t_il G G C A = -2.7
t_il G G U A = -2.2
t_il G G A C = -2.7
t_il G G C C = -1.5
t_il G G G C = -1.5
t_il G G U C = -1.5
t_il G G C G = -1.5
t_il G G U G = -1.0
t_il G G A U = -2.7
t_il G G C U = -1.5
t_il G G G U = -1.5
t_il G G U U = -1.5
t_il G U A A = -2.2
t_il G U G A = -2.7
t_il G U A C = -0.8
t_il G U C C = -1.5
t_il G U G C = -0.8
t_il G U U C = -2.5
t_il G U A G = -1.0
t_il G U G G = -1.5
t_il G U A U = -0.8
t_il G U C U = -1.5
t_il G U G U = -0.8
t_il G U U U = -2.5
t_il U A A A = -1.0
t_il U A C A = -1.0
t_il U A G A = -2.2
t_il U A U A = -0.4
t_il U A U C = -1.0
```

```
t_il U A A G = -1.0
t_il U A C G = -1.0
t_il U A G G = -2.2
t_il U A U G = -0.4
t_il U A U U = -2.0
t_il U C A A = -1.0
t_il U C C A = -1.0
t_il U C G A =  0.2
t_il U C U A = -1.0
t_il U C G C = -1.5
t_il U C A G = -1.0
t_il U C C G = -1.0
t_il U C G G =  0.2
t_il U C U G = -1.0
t_il U C G U = -2.5
t_il U G A A = -2.2
t_il U G C A = -0.4
t_il U G G A = -1.0
t_il U G U A = -0.4
t_il U G C C = -1.5
t_il U G U C = -1.0
t_il U G A G = -2.2
t_il U G C G = -0.4
t_il U G G G = -1.0
t_il U G U G = -0.4
t_il U G C U = -2.5
t_il U G U U = -2.0
t_il U U A A =  0.2
t_il U U C A = -1.0
t_il U U G A =  0.2
t_il U U U A = -2.0
t_il U U A C = -1.0
t_il U U G C = -1.5
t_il U U A G =  0.2
t_il U U C G = -1.0
t_il U U G G =  0.2
t_il U U U G = -2.0
t_il U U A U = -2.0
t_il U U G U = -2.5

t_il _ _ _ _ = error "t_il: not in table"
```

A.4.3 Lyngsø's decomposition

```
top_stack :: BasePair -> Energy
top_stack (i,j) = t_il (rna!i) (rna!(i+1)) (rna!(j-1)) (rna!j)


bot_stack :: BasePair -> Energy
bot_stack (i,j) = t_il (rna!(i+1)) (rna!i) (rna!j) (rna!(j-1))


asym :: Int -> Energy
asym a = min 3.0 ((fromIntegral a) * 0.3)
```

A.5 Multiple Loop Energies

A.5.1 Affine Costs

```
ml_init_penalty :: Energy
ml_init_penalty = 4.7


ml_unpaired_penalty :: Energy
ml_unpaired_penalty = 0.4


ml_unpaired :: Region -> Energy
ml_unpaired r = ml_unpaired_penalty * fromIntegral (regionSize r)


ml_helix_penalty :: Energy
ml_helix_penalty = 0.1
```

A.5.2 Dangling End Energies

```
t_dr :: (Base,Base) -> Base -> Energy

t_dr (U,A) A = -0.8
t_dr (U,A) C = -0.5
t_dr (U,A) G = -0.8
t_dr (U,A) U = -0.6

t_dr (G,C) A = -1.7
t_dr (G,C) C = -0.8
t_dr (G,C) G = -1.7
t_dr (G,C) U = -1.2

t_dr (C,G) A = -1.1
t_dr (C,G) C = -0.4
t_dr (C,G) G = -1.3
t_dr (C,G) U = -0.6

t_dr (U,G) A = -0.8
t_dr (U,G) C = -0.5
t_dr (U,G) G = -0.8
t_dr (U,G) U = -0.6

t_dr (A,U) A = -0.7
t_dr (A,U) C = -0.1
t_dr (A,U) G = -0.7
t_dr (A,U) U = -0.1

t_dr (G,U) A = -1.2
t_dr (G,U) C = -0.5
t_dr (G,U) G = -1.2
t_dr (G,U) U = -0.7

t_dr _ _ = error "t_dr: not in table"

t_dl :: Base -> (Base,Base) -> Energy

t_dl A (U,A) = -0.3
t_dl C (U,A) = -0.1
t_dl G (U,A) = -0.2
t_dl U (U,A) = -0.2

t_dl A (G,C) = -0.2
t_dl C (G,C) = -0.3
t_dl G (G,C) =  0.0
t_dl U (G,C) =  0.0

t_dl A (C,G) = -0.5
t_dl C (C,G) = -0.3
t_dl G (C,G) = -0.2
t_dl U (C,G) = -0.1

t_dl A (U,G) = -0.2
t_dl C (U,G) = -0.2
t_dl G (U,G) = -0.2
t_dl U (U,G) = -0.2

t_dl A (A,U) = -0.3
t_dl C (A,U) = -0.3
t_dl G (A,U) = -0.4
t_dl U (A,U) = -0.2

t_dl A (G,U) = -0.2
t_dl C (G,U) = -0.2
t_dl G (G,U) = -0.2
t_dl U (G,U) = -0.2

t_dl _ _ = error "t_dl: not in table"
```

A.6 Combinator Parsing with Attributes

```
module AttributeCombinators
```

```
where
import RNA
import Utils
```

A.6.1 Top-Down Attribute Combinators

The input to a parser is an indexed subword of the input and it's attributes.

```
type Input a = (Int,Int,a)
```

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

```
type Parser a parse = Input a -> [parse]
```

The primitive parsers.

```
base :: Parser a BasePos
base (i,j,_) = [ j | (i+1) == j ]

region :: Parser a Region
region (i,j,_) = [ (i,j) | i < j ]

stretch :: Int -> Parser a Region
stretch l (i,j,_) = [ (i,j) | (i - j) == l ]

empty :: Parser a ()
empty (i,j,_) = [ () | i == j ]
```

The keyword `axiom` of the grammar turns into a function that returns all parses for the startsymbol `p` over the complete input.

```
axiom :: a -> Parser a b -> [b]
axiom a p = p (0,rnaLen,a)
```

`>>>` applies function `f` to the attribute before calling parser `p`.

```
infixl 9 >>>
(>>>) :: (a -> a) -> Parser a b -> Parser a b
(f >>> p) (i,j,a) =  p (i,j, f a)
```

Set an attribute directly.

```
infixl 9 >><
(>><) :: a -> Parser a b -> Parser a b
(a >>< p) (i,j,_) =  p (i,j,a)
```

Concatenation of result lists of alternative parses.

```
infixr 6 |||
(|||) :: Parser a b -> Parser a b -> Parser a b
(r ||| q) inp = r inp ++ q inp
```

The `<<<` combinator applies an interpretation function to the results of subsequent parsers.

```
type Interpretation b c = b -> c

infix  8 <<<
(<<<) :: Interpretation b c -> Parser a b -> Parser a c
(f <<< q) inp =  map f (q inp)
```

Set the interpretation to `c`.

```
infix  8 ><<
(><<) :: c -> Parser a b -> Parser a c
(c ><< q) inp = [ c | s <- q inp ]
```

Or alternatively use setAttr together with >>> .

```
setAttr y _ = y
```

Test an attribute using with.

```
attrEquals x (_,_,a) | a == x    = True
                     | otherwise = False
```

The combinator for the choice function.

```
type Choice i = [i] -> [i]

infix  5 ...
(...) :: Parser a b -> (Choice b) -> Parser a b
p ... h = h . p
```

Some standard choice functions.

```
minimize :: [Energy] -> [Energy]
minimize [] = []
minimize xs = [minimum xs]

maximize :: Ord a => [a] -> [a]
maximize [] = []
maximize xs = [maximum xs]

addup :: [Integer] -> [Integer]
addup [] = []
addup xs = [sum xs]
```

The juxtaposition combinator combines all parses of r and q split at k ranging from i to j.

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

Restrict the parsing effort to non-empty subwords.

```
infixl 7 ~+~
(~+~) :: Parser a (b -> c) -> Parser a b -> Parser a c
(r ~+~ q) (i,j,a) = [f y | k <- [i+1..j-1], f <- r (i,k,a), y <- q (k,j,a)]
```

Restrict the parsing effort to subwords of an appropriate length range.

```
infixl 7 ~~
(~~) :: (Int,Int) -> (Int,Int) -> Parser a (b -> c) -> Parser a b -> Parser a c
(~~) (l,u) (l',u') r q (i,j,a) = [x y | k <- [max (i+l) (j-u') .. min (i+u) (j-l')],
                                        x <- r (i,k,a), y <- q (k,j,a)]

infixl 7 |~~
(|~~) :: Int -> Parser a (b -> c) -> Parser a b -> Parser a c
(|~~) l r q (i,j,a) = [x y | k <- [(i+l) .. j], x <- r (i,k,a), y <- q (k,j,a)]

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

Restrict the lefthand (respecitvely righthand) parser to a single symbols.

```
infixl 7 +~~
(+~~) :: Parser a (b -> c) -> Parser a b -> Parser a c
(r +~~ p) (i,j,a) = [x y | i < j, x <- r (i,i+1,a), y <- p (i+1,j,a)]

infixl 7 ~~+
(~~+) :: Parser a (b -> c) -> Parser a b -> Parser a c
(p ~~+ r) (i,j,a) = [x y | i < j, x <- p (i,j-1,a), y <- r (j-1,j,a)]
```

Restrict the lefthand parser to two symbols.

```
infixl 7 ++~
(++~) :: Parser a (b -> c) -> Parser a b -> Parser a c
(r ++~ p) (i,j,a) = [x y | i < j, x <- r (i,i+2,a), y <- p (i+2,j,a)]
```

Restrict the lefthand parser to three symbols.

```
infixl 7 +++
(+++) :: Parser a (b -> c) -> Parser a b -> Parser a c
(r +++ p) (i,j,a) = [x y | i < j, x <- r (i,i+3,a), y <- p (i+3,j,a)]
```

`with` applies a filter to the input boundaries and the attribute.

```
type Filter a = Input a -> Bool

with :: Parser a b -> Filter a -> Parser a b
(p `with` f) inp = if f inp then p inp else []
```

The basepairing filter.

```
basepairing :: Filter a
basepairing (i,j,_) = (i+1 < j) && pair (rna!(i+1)) (rna!j)
```

`within` applies a filter to the input left and right of the parser.

```
within :: Parser a b -> Filter a -> Parser a b
(p `within` f) (i,j,a) = [x | i == 0 || j == rnaLen || f (i-1,j+1,a), x <- p (i,j,a)]
```

`suchthat` filters the output of the parser `p` via function `f`.

```
suchthat :: Parser a b -> (b -> Bool) -> Parser a b
(p `suchthat` f) inp = filter f (p inp)
```

`tabulated` records the results of parser `p` for all subwords of the input in a two-dimensional array.

```
type Parsetable b = Array Region [b]

tabulated :: a -> Parser a b -> Parsetable b
tabulated a p = array ((0,0),(rnaLen,rnaLen))
              [ ((i,j),p (i,j,a)) | i<- [0..rnaLen], j<- [i..rnaLen] ]
```

`p` is the table lookup function.

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

The functions `listed` and `q` cover the one-dimensional case.

```
type Parselist b = Array Int [b]

listed :: a -> Parser a b -> Parselist b
listed a p = array (0,rnaLen)
              [ (i, p (i,rnaLen,a)) | i<- [0..rnaLen] ]
```

```
q :: Parselist b -> Parser a b
q table (i,_,_) = table!i
```

attributed records the results of parser p for all attributes and subwords of the input in a three-dimensional array. The range of attributes for the third dimension have to be specified in n and m, thus resulting in a table of size $O(n^2)$ where n is the input length.

```
type AttributeParsetable a b = Array (Int,Int,a) [b]

attributed :: (Ix a, Enum a) => a -> a -> Parser a b -> AttributeParsetable a b
attributed n m p = array ((0,0,n),(rnaLen,rnaLen,m))
  [ ((i,j,a), p (i,j,a)) | i<- [0..rnaLen], j<- [i..rnaLen], a <- [n..m] ]
```

ap is the attribute table lookup function.

```
ap :: (Ix a) => AttributeParsetable a b -> Parser a b
ap table (i,j,a) = if i <= j then table!(i,j,a) else []
```

Again, the one-dimensional case.

```
type AttributeParselist a b = Array (Int,a) [b]

alisted :: (Ix a, Enum a) => a -> a -> Parser a b -> AttributeParselist a b
alisted n m p = array ((0,n),(rnaLen,m))
              [ ((i,a), p (i,rnaLen,a)) | i<- [0..rnaLen], a <- [n..m] ]

aq :: (Ix a) => AttributeParselist a b -> Parser a b
aq table (i,_,a) = table!(i,a)
```

A.6.2 Bottom-Up Attributes

Attach a bottom-up attribute.

```
type BottomUpAttributeParser td b bu = Parser td (b,bu)

(<==>) :: Parser a b -> c -> BottomUpAttributeParser a b c
(p <==> bu) inp = [(x,bu) | x <- p inp]
```

Evaluate parses and bottom-up attributes in parallel.

```
(e <++> f) (x,ax) (y,ay) = (e x y, f ax ay)
(e <+=> f) (x,ax)  y     = (e x y, f ax)
(e <=+> f)  x     (y,ay) = (e x y, f ay)

(e <+++> f) (x,ax) (y,ay) (z,az) = (e x y z, f ax ay az)
(e <++=> f) (x,ax) (y,ay)  z     = (e x y z, f ax ay)
(e <+==> f) (x,ax)  y      z     = (e x y z, f ax)
(e <=++> f)  x     (y,ay) (z,az) = (e x y z, f ay az)
(e <==+> f)  x      y     (z,az) = (e x y z, f az)
(e <+=+> f) (x,ax)  y     (z,az) = (e x y z, f ax az)
(e <=+=> f)  x     (y,ay)  z     = (e x y z, f ay)
```

Test a boolean bottom-up attribute.

```
flagTrue :: (r,Bool) -> Bool
flagTrue = snd

flagFalse :: (r,Bool) -> Bool
flagFalse = not . snd
```

Remove a bottom-up attribute.

```
strip :: (a,b) -> a
strip = fst
```

A.7 2D Graphics Functions

```
module Graphics where

 import RNA
 import Structure
 import Utils
```

Basic 2D data types.

```
 type Point  = (Double,Double)
 type Vector = Point
 type Angle  = Double
 type Radius = Double

 data Gobj = Gline   [Point]      |
             Gcircle Radius Point |
             Gtext   String Point |
             GwhiteT String Point
             deriving Show
```

Rotate a vector by an angle.

```
 rotate :: Angle -> Vector -> Vector
 rotate a (x,y) = (x*c - y*s, x*s + y*c)
   where
     c = cos a
     s = sin a
```

Add two vectors.

```
 add :: Vector -> Vector -> Vector
 add (x,y) (x',y') = (x+x',y+y')
```

Subtract two vectors.

```
 sub :: Vector -> Vector -> Vector
 sub (x,y) (x',y') = (x-x', y-y')
```

Scale a point or vector.

```
 scale :: Double -> Vector -> Vector
 scale s (x,y) = (s * x, s * y)
```

Convert a point into string format (show)

```
 showPoint :: Point -> String
 showPoint (x,y) = (show $ floor x) ++ " " ++ (show $ floor y)
```

Generate a polygon from p1 to p2 with size vertices where edge length is distance from p1 to p2

```
 loop :: Int -> Point -> Point -> [Point]
 loop size p1 p2 = points size p2
   where
   points 0 _ = []
   points n p = p : points (n-1)
                (add p (rotate (rho * fromIntegral (n-1)) delta))
     where
     delta = sub p2 p1
     rho = -(2*pi / fromIntegral size)

 spacer :: Point -> Point -> (Point,Point)
 spacer p1 p2 = (add p1 i, add p2 i) where
   delta = scale 1.0 (sub p2 p1)
   i = rotate (pi/2) delta
```

Generate an xfig compatible header.

```
figHeader =
 ["#FIG 3.2",
  "Landscape",
  "Center",
  "Metric",
  "A4",
  "100.00",
  "Single",
  "-2",
  "1200 2"]
```

The xfig primitive line prefixes.

```
polylineHeader  = "2 1 0 2 0 7 100 0 -1 0.000 0 0 -1 0 0"
circleHeader    = "1 3 0 1 0 0 100 0 20 0.000 1 0.0000"
textHeader      = "4 0 0 100 0 8 11 0.0000 4 105 0"
whiteTextHeader = "4 0 7 100 0 8 11 0.0000 4 105 0"
```

The scale we will use is centimetres.

```
figScale        = 450.0 -- 450 = 1cm
```

Compute loop sizes.

```
loopsize :: Loop -> Int
loopsize (SS x) = regionSize x
loopsize (SR _ _ _) = 4
loopsize (HL _ x _) = 2 + regionSize x
loopsize (BL _ x _ _) = 4 + regionSize x
loopsize (BR _ _ x _) = 4 + regionSize x
loopsize (IL _ x _ y _) = 4 + (regionSize x) + (regionSize y)
loopsize (ML _ cs _) = 2 + extloopsize cs
loopsize (EL   cs  ) = extloopsize cs

extloopsize :: [Loop] -> Int
extloopsize cs = sum (map size cs) where
  size (SS x) = regionSize x
  size _      = 2
```

Generate the list of base indexes for a given sub-structure element.

```
loopindex :: Loop -> [Int]
loopindex (SS (i,j)) = [i+1..j]
loopindex (SR lb _ rb) = [lb, lb+1, rb-1, rb]
loopindex (HL lb x rb) = [lb..rb]
loopindex (BL lb (_,j) _ rb) = [lb..j+1] ++ [rb-1,rb]
loopindex (BR lb _ (i,_) rb) = lb : lb+1 : [i..rb]
loopindex (IL lb (i,j) _ (k,l) rb) = [lb..j+1] ++ [k..rb]

loopindex (ML lb cs rb) = [lb..(fst (head bps))]
                          ++ accessible rb bps
  where
  bps = accessibleBPs cs
  accessible rb ((i,j):[]) = [j..rb]
  accessible rb ((_,j):b@(k,_):bs) = [j..k]
                                     ++ accessible rb (b:bs)


loopindex (EL cs) = [1..(fst (head bps))]
                    ++ accessible bps
  where
  bps = accessibleBPs cs
  accessible ((i,j):[]) = [j..]
  accessible ((_,j):b@(k,_):bs) = [j..k]
                                  ++ accessible (b:bs)
```

Write an xfig format file containing the graphical objects of a secondary structure.

```
writeFIG :: String -> Double -> [Gobj] -> IO ()

writeFIG filename s dat = writeFile filename contents
  where
  contents = unlines (figHeader ++ (map (convert s) dat))
    where
    convert s (Gline ps) = polylineHeader
                         ++ " " ++ l ++ "\n\t" ++ convPs
      where
      l = (show (length ps))
      convPs = unwords (map convP ps)
        where
        convP = showPoint . (scale s)

    convert s (Gcircle r p) = unwords [circleHeader, cp, cr, cp, cq]
      where
      cp = showPoint (scale s p)
      cr = showPoint (scale s (r,r))
      cq = showPoint (scale s (add p (r,r)))

    convert s (Gtext t p) = unwords [textHeader, cp, ct]
      where
      cp = showPoint (scale s p)
      ct = t ++ "\\001"

    convert s (GwhiteT t p) = unwords [whiteTextHeader, cp, ct]
      where
      cp = showPoint (scale s p)
      ct = t ++ "\\001"
```

Print a black circle with a white character in the middle.

```
circChar :: String -> Point -> [Gobj]

circChar t p@(i,j) = [Gcircle 0.2 p , GwhiteT t (i-0.13,j+0.13)]
```

A.7.1 Circle Plot

```
circlePlot :: RNA -> Loop -> [Gobj]

circlePlot p sp = backbone : (lines sp) ++ bases
  where
  backbone = Gline points

  lines (NIL           ) = []
  lines (SS        _   ) = []
  lines (HL lb   l   rb) = [line lb rb]
  lines (SR lb   l   rb) = line lb rb : lines l
  lines (BL lb _ l   rb) = line lb rb : lines l
  lines (BR lb   l _ rb) = line lb rb : lines l
  lines (IL lb _ l _ rb) = line lb rb : lines l
  lines (ML lb _ ls _ rb) = line lb rb : concat (map lines ls)
  lines (EL     ls     ) = concat (map lines ls)

  bases = concat [circChar (show (p!x)) (circle!x) | x <- [1..len]]

  line i j = Gline [circle!i,circle!j]
  len      = snd (bounds p)
  circle   = array (1,len) (zip [1..len] points)
  points   = loop len (0,0) (1,0)
```

A.7.2 Mountain Plot

The mountain plot function uses plotting commands (up, down, hor) to draw the plot.

```
mountainPlot :: RNA -> Loop -> [Gobj]

mountainPlot p sp = (lines 0 sp) ++ (bases p)
  where
  lines x (NIL)        = []
  lines x (SS (i,j))   = []
```

```
lines x (HL lb l rb) = [up x lb,
                        hor (x-1) l,
                        down x rb,
                        hor x (lb-1,rb)]
lines x (SR lb l rb) = [up x lb,
                        down x rb,
                        hor x (lb-1,rb),
                        ver x lb rb] ++ lines (x-1) l
lines x (BL lb (i,j) l rb) = [up x lb,
                        down x rb,
                        hor (x-1) (i,j+1),
                        hor x (lb-1,rb),
                        ver x lb rb] ++ lines (x-1) l
lines x (BR lb l (i,j) rb) = [up x lb,
                        down x rb,
                        hor (x-1) (i-1,j),
                        hor x (lb-1,rb),
                        ver x lb rb] ++ lines (x-1) l
lines x (IL lb (i,j) c (k,l) rb) = [up x lb,
                        down x rb,
                        hor (x-1) (i,j+1),
                        hor (x-1) (k-1,l),
                        hor x (lb-1,rb),
                        ver x lb rb] ++ lines (x-1) c
lines x (ML lb ls rb) = [up x lb,
                        down x rb,
                        hor (x-1) (lb,rb-1),
                        hor x (lb-1,rb),
                        ver x lb rb] ++ concat (map (lines (x-1)) ls)
lines x (EL ls) = (hor x (0,len)) : concat (map (lines x) ls)

bases p  = concat [circChar (show (p!x)) (fromIntegral x,1) | x <- [1..len]]

hor  x (i,j) = Gline [(fromIntegral i+1,x),(fromIntegral j,x)]
ver  x  i j  = Gline [(w,x),(w,x-1)]
  where
  u = fromIntegral i
  v = fromIntegral j
  w = u + (v - u)/2
up   x   i   = Gline [(fromIntegral i,x),(fromIntegral i+1,x-1)]
down x   i   = Gline [(fromIntegral i-1,x-1),(fromIntegral i,x)]

len = snd (bounds p)
```

A.7.3 Polygon Plot

The polygonPLot function draws circular polygons corresponding to the structural elements of the RNA. It traverses the RNA structure and draws polygons relative to the parent structure.

```
polygonPlot :: RNA -> Loop -> [Gobj]

polygonPlot rna sp = d sp ++ plot (0,0) (1,0) sp
  where
  d   (EL _) = []
  d c@(SS _) = plotLoop c ps ++ plotBases rna c ps
               ++ dots 2 (0,-1.5) (1,-1.5)
    where
    ps = loop (loopsize c) (0,0) (1,0)
  d       _  = plotBasePair (0,0) (1,0)
               : dots 2 (0,-1.5) (1,-1.5)


  plot p q   (NIL     )   = dots 2 p q
  plot _ _   (SS _    )   = []


  plot p q c@(HL lb l rb) = plotLoop c ps
                            ++ plotBases rna c ps
    where
    ps = loop (loopsize c) p q

  plot p q c@(SR lb l rb) = plotLoop c ps
                            ++ plotBases rna c ps
                            ++ plot (ps!!2) (ps!!1) l
```

```
    where
    ps = loop (loopsize c) p q

  plot p q c@(BL lb (i,j) l rb) = plotLoop c ps
                                ++ plotBases rna c ps
                                ++ plot (ps!!(s-2)) (ps!!(s-3)) l
    where
    ps = loop s p q
    s = loopsize c

  plot p q c@(BR lb l (i,j) rb) = plotLoop c ps
                                ++ plotBases rna c ps
                                ++ plot (ps!!2) (ps!!1) l
    where
    ps = loop s p q
    s = loopsize c

  plot p q c@(IL lb lr l rr rb) = plotLoop c ps
                                ++ plotBases rna c ps
                                ++ plot (ps!!(r+2)) (ps!!(r+1)) l
    where
    ps = loop (loopsize c) p q
    r = regionSize lr

  plot p q c@(ML lb ls rb) = plotLoop c ps
                           ++ plotBases rna c ps
                           ++ draw_comps 1 ps ls
    where
    ps = loop (loopsize c) p q

  plot p q c@(EL ls)       =  plotLoop c ps
                           ++ plotBases rna c ps
                           ++ draw_comps 0 ps ls
    where
    ps = loop (loopsize c) p q

  draw_comps :: Int -> [Point] -> [Loop] -> [Gobj]
  draw_comps _ _ [] = []
  draw_comps i ps ((SS r):cs) = draw_comps (i + (regionSize r)) ps cs
  draw_comps i ps (c:cs)      = (plot (ps!!(i+1)) (ps!!(i)) c)
                              ++ draw_comps (i+2) ps cs
```

Polygon Plot with Spacer The polygonPlot Spacer function is equivalent to the polygonPlot function. It inserts space between the structural elements to enhance their decomposition.

```
polygonPlotS :: RNA -> Loop -> [Gobj]

polygonPlotS rna sp = d sp
                    ++ plot (0,0) (1,0) sp
  where
  d   (EL _) = []
  d c@(SS _) = plotLoop c ps
             ++ plotBases rna c ps
             ++ dots 2 (0,-1.5) (1,-1.5)
    where
    ps = loop (loopsize c) (0,0) (1,0)
  d       _  = plotBasePair (0,0) (1,0)
             : dots 2 (0,-1.5) (1,-1.5)

  plot p q   (NIL     )   = dots 2 p q
  plot _ _   (SS _    )   = []

  plot p q c@(HL lb l rb) = plotLoop c ps
                          ++ plotBases rna c ps
    where
    ps = loop (loopsize c) p q

  plot p q c@(SR lb l rb) = plotBasePair p q
                          : plotLoop c ps
                          ++ plotBases rna c ps
                          ++ plot i j l
```

```
    where
    ps    = loop (loopsize c) p q
    (i,j) = spacer (ps!!2) (ps!!1)

  plot p q c@(BL lb (i,j) l rb) = plotBasePair p q
                                : plotLoop c ps
                                ++ plotBases rna c ps
                                ++ plot i j l
    where
    ps = loop s p q
    s = loopsize c
    (i,j) = spacer (ps!!(s-2)) (ps!!(s-3))

  plot p q c@(BR lb l (i,j) rb) = plotBasePair p q
                                : plotLoop c ps
                                ++ plotBases rna c ps
                                ++ plot i j l
    where
    ps = loop s p q
    s = loopsize c
    (i,j) = spacer (ps!!2) (ps!!1)

  plot p q c@(IL lb lr l rr rb) = plotBasePair p q
                                : plotLoop c ps
                                ++ plotBases rna c ps
                                ++ plot i j l
    where
    ps = loop (loopsize c) p q
    r = regionSize lr
    (i,j) = spacer (ps!!(r+2)) (ps!!(r+1))

  plot p q c@(ML lb ls rb) = plotBasePair p q
                           : plotLoop c ps
                           ++ plotBases rna c ps
                           ++ draw_comps 1 ps ls
    where
    ps = loop (loopsize c) p q

  plot p q c@(EL ls)       =  plotLoop c ps
                              ++ plotBases rna c ps
                           ++ draw_comps 0 ps ls
    where
    ps = loop (loopsize c) p q

  draw_comps :: Int -> [Point] -> [Loop] -> [Gobj]
  draw_comps _ _ [] = []
  draw_comps i ps ((SS r):cs) = draw_comps (i + (regionSize r)) ps cs
  draw_comps i ps (c:cs)      = (plot p q c)
                                ++ draw_comps (i+2) ps cs
    where
    (p,q) = spacer (ps!!(i+1)) (ps!!(i))

half :: Point ->Point -> Point

half p1 p2 = add p1 (scale 0.5 (sub p2 p1))

dots n p1 p2 = map (Gcircle 0.05) (take n ps)
  where
  delta = scale 0.5 (sub p2 p1)
  i     = rotate (pi/2) delta
  p0    = add i (add p1 delta)
  ps    = iterate (add i) p0

plotBases rna l ps = concat [circChar t p| (t,p) <- els]
  where
  els = zip [(show (rna!i)) | i <- loopindex l] ps

plotLoop l ps = dotslines els
  where
  dotslines [] = []
  dotslines ((i,p):[]) = []
  dotslines ((i1,p1):e@(i2,p2):els) | i2 - i1 == 1 = Gline [p1,p2]
                                                   : dotslines (e:els)
                                    | otherwise    = plotBasePair p1 p2
                                                   : dotslines (e:els)
  els = zip (loopindex l) ps
```

```
plotBasePair p q = Gcircle 0.1 (half p q)

loopPoints :: Point -> Point -> Loop -> Array Int Point

loopPoints p q l = array (1,s) (zip [1..s] (loop s p q))
  where
  s = loopsize l
```

A.7.4 Examples

$tRNA^{Phe}$ of S. cerevisae

```
rnaStr =    "gcggauuuagcucaguugggagagcgccagacugaaga"
         ++ "ucuggagguccuguguucgauccacagaauucgcacca"
```

Secondary Structure of $tRNA^{Phe}$ of S. cerevisae

```
tRNA = (EL [SR 1 (SR  2 (SR 3 (SR 4 (SR 5 (SR 6 (
           ML 7 [
                 SS (7,9),
                 SR 10 (SR 11 (SR 12 (HL 13 (13,21) 22) 23) 24) 25,
                 SS (25,26),
                 SR 27 (SR 28 (SR 29 (SR 30 (HL 31 (31,38) 39) 40) 41) 42) 43,
                 SS (43,48),
                 SR 49 (SR 50 (SR 51 (SR 52 (HL 53 (53,60) 61) 62) 63) 64) 65
                ]
          66) 67) 68) 69) 70) 71) 72,
            SS (72,76)])

rnaExample = (EL
               [ML 1
                [SR 2
                 (HL 3 (3,6) 7) 8,
                 BR 9
                 (HL 10 (10,10) 11)
                 (12,13)
                 14]
                15])

rnaEx1 = (EL [BR 1 (HL 2 (2,2) 3) (4,10) 11])
```

Circle Plot

```
writeFIG "circle.fig" figScale $ circlePlot rna tRNA
```

Mountain Plot

```
writeFIG "mountain.fig" figScale $ mountainPlot rna tRNA
```

Polygon Plot

```
writeFIG "polygon.fig" figScale $ polygonPlot rna tRNA
```

A.8 Utilities

```
module Utils (
 module Array,
 slice, regionSize, minLoopSize, strToRNA, charToBase, toArray, combine,
 combine3, combine4, rnaStr, rna, rnaLen, for, isBasePair,
 toRNA, access, Energy
 ) where
 import System
 import IOExts
 import Array
 import RNA
```

A slice of RNA.

```
slice :: Region -> [Base]
slice (i,j) = [ rna!k | k <- [i+1 .. j]]
```

Return the length of a region.

```
regionSize :: Region -> Int
regionSize (i,j) = j - i

minLoopSize :: Int -> Region -> Bool
minLoopSize s reg = (regionSize reg) >= s
```

The input to a parser is a slice of the input and it's attributes.

```
type Input attr = (Region,attr)
```

Convert a string of characters to a list of bases.

```
strToRNA :: String -> [Base]
strToRNA = map charToBase
```

Convert a character to a base.

```
charToBase :: Char -> Base
charToBase 'A' = A
charToBase 'C' = C
charToBase 'G' = G
charToBase 'U' = U
charToBase 'a' = A
charToBase 'c' = C
charToBase 'g' = G
charToBase 'u' = U
charToBase  _  = error "malformed base"
```

Create an array and fill it with a list.

```
toArray :: [b] -> Array Int b
toArray l = listArray (1,length l) l
```

Regroup function arguments into a tuple.

```
combine :: a -> b -> (a,b)
combine  a b     = (a,b)
combine3 :: a -> b -> c -> (a,b,c)
combine3 a b c   = (a,b,c)
combine4 :: a -> b -> c -> d -> (a,b,c,d)
combine4 a b c d = (a,b,c,d)
```

The free energy is measured in $\frac{kcal}{mol}$.

```
type Energy = Float -- kcal/mol
```

Process the command line arguments.

```
argV :: [String]
argV = unsafePerformIO getArgs
```

The mini example from the book "Biological sequence analysis" (Durbin et al., 1998).

```
rnaStr = "gggaaauuc"
```

Convert the command line arguments to the global rna array used in all recognizers.

```
rnaStr = argV!!((length argV) - 1)
rna = toArray $ strToRNA (argV!!((length argV) - 1))

toRNA s = toArray $ strToRNA s
rna = toRNA rnaStr
rnaLen = snd (bounds rna)
```

Utility functions for the imperative DP example

Sequentially apply the function `f` to a list of elements `xs`.

```
for xs f = sequence_ (map f xs)
```

The base pair predicate.

```
isBasePair :: BasePair -> Bool
isBasePair (i,j) = pair (rna!i) (rna!j)
```

Access a table entry.

```
access a (i,j) = unsafePerformIO $ readIOArray a (i,j)
```

Bibliography

Altmann, S. (1990). Enzymatic cleavage of RNA by RNA, *Biosci. Rep.* **10**: 317–337. Nobel lecture. 3.1.1

Anson, E. L. and Myers, G. W. (1997). ReAligner: A program for refining DNA sequence multi-alignments, *1st Conference on Computational Molecular Biology*, pp. 9–16. 4.1.1

Bellman, R. E. (1957). *Dynamic Programming*, Princeton University Press. 1.2, 4.1.2

Berman, H. M. e. a. (n.d.). The RCSB protein databank, `http://www.pdb.org/`. PDB ID: 1MME. 3.2

Birney, E. and Durbin, R. (1997). Dynamite: A flexible code generating language for dynamic programming methods, *Proc. Intelligent Systems for Molecular Biology*, AAAI Press, Menlo Park, CA, USA, pp. 56–64. 1.2, 4.1.1

Borer, P. N., Dengler, B., Tinoco Jr., I. and Uhlenbeck, O. C. (1974). Stability of ribonucleic acid doublestranded helices, *J. Mol. Biol.* **86**: 843–853. 1.2, 3.3.3

Bruccoleri, R. E. and Heinrich, G. (1988). An improved algorithm for nucleic acid secondary structure display, *Comput. Appl. Biosci.* **4**(1): 167–173. 3.4.3

Cech, T. R. (1986). RNA as an enzyme, *Sci. Am.* **255**(5): 64–75. 3.1

Cech, T. R. (1990). Self-splicing and enzymatic activity of an intervening sequence RNA from *Tetrahymena*, *Biosci. Rep.* **10**: 239–261. 3.1.1

Cormen, T. H., Leiserson, C. E. and Rivest, R. L. (1990). *Introduction to Algorithms*, MIT Press, Cambridge, MA. 4.1.1, 4.1.4

Crick, F. (1970). Central dogma of molecular biology, *Nature* **227**: 561–563. 3.1.1

De Rijk, P. and De Wachter, R. (1993). DCSE, an interactive tool for sequence alignment and secondary structure research, *Comput. Appl. Biosci.* **9**(6): 735–740. 3.4.1

Durbin, R., Eddy, S. R., Krogh, A. and Mitchison, G. (1998). *Biological sequence analysis*, Cambridge University Press. 4.1.1, 4.1.3, 4.1.4, A.8

Eliceiri, G. L. (1999). Small nucleolar RNAs, *Cell. Mol. Life Sci.* **56**(1-2): 22–31. 3.1.1

Evers, D. and Giegerich, R. (2000). Systematic dynamic programming in bioinformatics, *Intelligent Systems for Molecular Biology (Tutorial Notes)*, AAAI Press, Menlo Park, CA, USA. 4

Evers, D. and Giegerich, R. (2001). Reducing the conformation space in RNA structure prediction, *in* E. Wingender, R. Hofestädt and I. Liebich (eds), *Proceedings of the German Conference on Bioinformatics*, German Research Center for Biotechnology. ISBN: 3-00-008114-3. 1.3, 9.1, 9.7

Evers, D., Giegerich, R. and Kurtz, S. (1999). A general pattern matching language for specific motifs in RNA secondary structure, *Proc. of the 4th European Conference on Theory and Mathematics in Biology and Medicine*. 1.2

Fayat, G., Mayaux, J. F., Sacerdot, C., Fromant, M., Springer, M., Grunberg-Manago, M. and Blanquet, S. (1983). Escherichia coli phenylalanyl-tRNA synthetase operon region. evidence for

an attenuation mechanism. identification of the gene for the ribosomal protein, *J. Mol. Biol.* **171**(3): 239–261. 3

Ferré-D'Amaré, A. R. and Doudna, J. A. (1999). RNA FOLDS: Insights from recent crystal structures, *Annu. Rev. Biophys. Biomol. Struct.* **28**: 57–73. 3.1.2

Gelfand, M. S. and Roytberg, M. A. (1993). A dynamic programming approach for predicting the exon-intron structure, *Biosystems* **30**: 173–182. 4.1.1

Giegerich, R. (1998). A declarative approach to the development of dynamic programming algorithms, applied to RNA folding, *Report 98-02*, Faculty of Technology, Bielefeld University. ISSN 0946-7831. 1.2, 4

Giegerich, R. (1999). A systematic approach to dynamic programming in bioinformatics. Part 1 and 2: Sequence comparison and RNA folding, *Report 99-05*, Faculty of Technology, Bielefeld University. ISSN 0946-7831. 1.2, 4, 9.1

Giegerich, R. (2000). Explaining and controlling ambiguity in dynamic programming, *in* R. Giancarlo and D. Sankoff (eds), *Proceedings of the 11th Annual Symposium on Combinatorial Pattern Matching, CPM 2000*, Vol. 1848 of *Lecture Notes in Computer Science*, Springer Verlag, Berlin, Heidelberg, pp. 46–59. 4

Giegerich, R. (2003). Pseudoknots, to appear. 4

Giegerich, R., Haase, D. and Rehmsmeier, M. (1998). Prediction and visualization of structural switches in RNA, *Biocomputing '99, Proceedings of the Pacific Symposium*, World Scientific Press, pp. 126–137. 9.7

Giegerich, R., Kurtz, S. and Weiller, G. F. (1999). An algebraic dynamic programming approach to the analysis of recombinant DNA sequences, *Proceedings of the First Workshop on Algorithmic Aspects of Advanced Programming Languages*, pp. 77–88. 1.2

Gouy, M., Marliere, P., Papanicolaou, C. and Ninio, J. (1985). Prediction of secondary structures of nucleic acids: algorithmic and physical aspects, *Biochimie.* **67**(5): 523–531. French. 1.2

Gultyaev, A. P., van Batenburg, F. H. and Pleij, C. W. (1995). The computer simulation of RNA folding pathways using a genetic algorithm, *J. Mol. Biol.* **250**(1): 37–51. 1.2, 3.1.3

Gusfield, D. (1997). *Algorithms on Strings, Trees, and Sequences*, Cambridge University Press. 4.1.4

Hofacker, I. L., Fontana, W., Stadler, P. F., Bonhoeffer, S., Tacker, M. and Schuster, P. (1994). Fast folding and comparison of RNA structures, *Monatsh. Chem.* **125**: 167–188. 1.2, 3.4.1, 7.1

Hogeweg, P. and Hesper, B. (1984). Energy directed folding of RNA sequences, *Nucleic Acids Res.* **12**: 67–74. 3.4.3

Hutton, G. (1992). Higher order functions for parsing, *Journal of Functional Programming* **3**(2): 323–343. 4.4

Jacobson, A. B. and Zuker, M. (1993). Structural analysis by energy dot plot of a large mRNA, *J. Mol. Biol.* **233**(2): 261–269. 5.3

Jacobson, H. and Stockmayer, W. H. (1950). Intramolecular reaction in polycondensations, *J. Chem. Phys.* **18**: 1600–1606. 3.3.3

Kim, S. H., Suddath, F. L., Quigley, G. J., McPherson, A. and Sussman, J. L. e. a. (1974). Three-dimensional tertiary structure of yeast phenylalanine transfer RNA, *Science* **185**: 435–440. 3.1.3

Latz, E. (2000). *Erkennung lokaler Minima im Faltungsraum einer RNA*, Master's thesis, Faculty of Technology, Bielefeld University. 9.7.1

Lefebvre, F. (1995). An optimized parsing algorithm well suited to RNA folding, *Proc. of the Third Conference on Intelligent Systems for Molecular Biology ISMB 95*, AAAI Press, pp. 222–230. 4.3

Lütcke, H. (1995). Signal recognition particle (SRP), a ubiquitous initiator of protein translocation, *Eur. J. Biochem.* **228**(3): 531–550. 3.1.1

Lyngsø, R. B., Zuker, M. and Pedersen, C. N. (1999). Fast evaluation of internal loops in RNA secondary structure prediction, *Bioinformatics* **15**(6): 440–445. 1.1, 1.4, 7.3

Mathews, D. H., Andre, T. C., Kim, J., Turner, D. H. and Zuker, M. (1998). An updated recursive algorithm for RNA secondary structure prediction with improved free energy parameters, *American Chemical Society Symposium Series* **682**: 246–257. 3.3.3

Mathews, D. H., Sabina, J., Zuker, M. and Turner, D. H. (1999). Expanded sequence dependence of thermodynamic parameters improves prediction of RNA secondary structure, *J. Mol. Biol* **288**(5): 911–940. 8.1, 2

Mironov, A. A. and Lebedev, V. F. (1993). A kinetic model of RNA folding, *Biosystems* **30**: 49–56. 3.1.3

Needleman, S. B. and Wunsch, C. D. (1970). A general method applicable to the search for similarities in the amino acid sequence of two proteins, *J. Mol. Biol.* **48**(3): 443–453. 4.1.1, 4.3.2

Nissen, P., Hansen, J., Ban, N., Moore, P. B. and Steitz, T. A. (2000). The structural basis of ribosome activity in peptide bond synthesis, *Science* **289**(5481): 920–930. 3.1, 3.1.1

Nussinov, R., Pieczenik, G., Griggs, J. R. and Kleitman, D. J. (1978). Algorithms for loop matchings, *SIAM J. Appl. Math.* **35**: 68–82. 1.1, 1.3, 3.3.3, 4, 4.1.3

Papanicolaou, C., Gouy, M. and Ninio, J. (1984). An energy model that predicts the correct folding of both the tRNA and the 5S RNA molecules, *Nucleic Acids Res.* **12**: 31–44. 1.2, 3.3.3

Peyton Jones, S. (2003). *Haskell 98 Language and Libraries*, Cambridge University Press. ISBN: 0521826144. 2.3

Pley, H. W., Flaherty, K. M. and McKay, D. B. (1994). Three-dimensional structure of a hammerhead ribozyme, *Nature* **372**(6501): 68–74. 3.1.3

Rivas, E. and Eddy, S. (2000). Pseudoknot dp program, *xxx*. 4

Rivas, E. and Eddy, S. (2001). Pseudoknot grammar, *Bioinformatics*. 4

Robertus, S. P., Ladner, J. E., Finch, J. T., Rhodes, D. and Brown, R. S. e. a. (1974). Structure of yest phenylalanine tRNA at 3 Å resolution, *Nature* **250**: 546–551. 3.1.3

Sakakibara, Y., Brown, M., Hughey, R., Mian, I. S., Sjölander, K., Underwood, R. C. and Haussler, D. (1994). Recent methods for RNA modeling using stochastic context-free grammars, *in* D. Gusfield (ed.), *Proc. of the Fifth Annual Symposium on Combinatorial Pattern Matching*, Vol. 807 of *Lecture Notes in Computer Science*, Springer Verlag, pp. 289–306. 4.3

Schmitz, M. and Steger, G. (1996). Description of RNA folding by "simulated annealing", *J. Mol. Biol.* **255**(1): 254–266. 3.1.3

Scott, W. G., Finch, J. T. and Klug, A. (1995). The crystal structure of an all-RNA hammerhead ribozyme: a proposed mechanism for RNA catalytic cleavage, *Cell* **81**(7): 991–1002. 3.1.3

Searls, D. B. (1997). Linguistic approaches to biological sequences, *CABIOS* **13**(4): 333–344. 4.3

Shapiro, B., Maizel, J., Lipkin, L. E., Currey, K. and Whitney, C. (1984). Generating non-overlapping displays of nucleic acid secondary structure, *Nucleic Acids Res.* **12**(1): 75–99. 3.4.3, 3.4.3

Smith, T. F. and Waterman, M. S. (1981). Comparison of biosequences, *Adv. Appl. Math.* **2**: 482–489. 4.1.1

Tahi, F., Gouy, M. and Regnier, M. (2002). Automatic rna secondary structure prediction with a comparative approach, *Comput Chem. 2002*. 1.2

Tinoco, I. J. and Bustamante, C. (1999). How RNA folds, *J. Mol. Biol.* **293**(2): 271–281. 1.2, 3.1.3

Tinoco, I. J., Uhlenbeck, O. C. and D., L. M. (1971). Estimation of secondary structure in ribonucleic acids, *Nature* **230**(5293): 362–367. 1.1, 1.2, 5

Turner, D. H., Sugimoto, N. and Freier, S. M. (1988). RNA structure prediction, *Ann. Rev. Biophys. Biophys. Chem.* **17**: 167–192. 1.2, 3.1

Walter, A. E., Turner, D. H., Kim, J., Lyttle, M. H., Muller, P., Mathews, D. H. and Zuker, M. (1994). Coaxial stacking of helixes enhances binding of oligoribonucleotides and improves predictions of RNA folding, *Proc Natl Acad Sci USA* **91**(20): 9218–9222. 1.2, 8.1

Waterman, M. S. (1995). *Introduction to Computational Biology. Maps, Sequences and Genomes*, Chapman & Hall, London, UK. 4.1.4

Williams, A. L. J. and Tinoco, I. J. (1986). A dynamic programming algorithm for finding alternative RNA secondary structures, *Nucleic Acids Res.* **14**(1): 299–315. 6

Wuchty, S., Fontana, W., Hofacker, I. L. and Schuster, P. (1999). Complete suboptimal folding of RNA and the stability of secondary structures, *Biopolymers* **49**(2): 145–165. 1.1, 1.2, 1.4, 6, 8.4

Xia, T., SantaLucia Jr., J., Burkard, M. E., Kierzek, R., Schroeder, S. J., Jiao, X., Cox, C. and Turner, D. H. (1998). Parameters for an expanded nearest-neighbor model for formation of RNA duplexes with Watson-Crick pairs, *Biochemistry* **37**: 14719–14735. 1.2, 3.3.3

Zuker, M. (1989). On finding all suboptimal foldings of an RNA molecule, *Science* **244**: 48–52. 1.2, 3.4.1, 4.1.1, 7.1, 1

Zuker, M. (2000). Calculating nucleic acid secondary structure, *Curr Opin Struct Biol.* **10**(3): 303–310. 1.2, 5.3, 6

Zuker, M. (2003). Michael Zuker's Home Page , `http://www.bioinfo.rpi.edu/~zukerm/` . 3.3.3, 8.3

Zuker, M. and Sankoff, D. (1984). RNA secondary structures and their prediction, *Bull. Math. Biol.* **46**: 591–621. 1.1, 1.2, 9.1

Zuker, M. and Stiegler, P. (1981). Optimal computer folding of large RNA sequences using thermodynamics and auxiliary information, *Nucleic Acids Res.* **9**(1): 133–148. 1.1, 1.2, 1.4, 5, 5.1

Zwieb, C., Wower, I. and Wower, J. (1999). Comparative sequence analysis of tmRNA, *Nucleic Acids Res.* **27**(10): 2063–2071. 3.1.1