Locomotif - a Graphical Programming System for RNA Motif Search

Dissertation zur Erlangung des akademischen Grades eines Doktors der Naturwissenschaften (Dr. rer. nat.) der Technischen Fakultät der Universität Bielefeld

vorgelegt von

Janina Reeder

Bielefeld im Dezember 2006

Acknowledgments

I thank my supervisor, Robert Giegerich, for providing me with an interesting research topic and for his support throughout the years, both scientifically, as well as by offering me the chance to continue my work while taking care of my daughter. Thanks also to Jens Stoye for appraising this thesis.

I thank Peter Steffen who developed the ADP compiler that is an integral part of the Locomotif system and adapted it for my needs.

I thank Jan Krüger for help in installing the system on the webserver and for guidance in XML and Java questions. Nan Zhang gave me a headstart on the XML schema.

I appreciate the financial support from the DFG. I enjoyed being part of the GK Bioinformatik and value travel opportunities and scientific merit of the BREW workshops in Helsinki and Berlin.

I thank Jens Reeder for many ideas, discussions, hours spent debugging the Locomotif system and proofreading this thesis, but most of all for his enduring emotional support over all these years.

I am deeply grateful for my daughter Emma.

Finally, I thank my parents for all their support and patience.

Gedruckt auf alterungsbeständigem Papier °° ISO 9706

Contents

Chapter 1

Introduction

The field of computational biology is characterized by the need to combine substantial knowledge from computer science and molecular biology. Sophisticated algorithmics are required to handle the large data volume and the elaborated biological models e. g. in whole genome comparison, gene structure prediction or simulation of metabolic processes. Deep knowledge from domain experts is required to choose the right questions to ask and to interpret the results.

There has been a paradigm shift in molecular biology recently. Three types of chain molecules form the basis of life: deoxy-ribonucleic acid (DNA), ribonucleic acid (RNA), and proteins. The long-lasting "dogma" of molecular biology saw DNA as the prime source of genetic information, stored in the well-known double helix. RNA was mainly seen as an intermediate carrier, a single stranded molecule bringing genetic information from the DNA to the cell's translational machinery. There, the proteins are synthesized as chains of amino acids, according to the universal genetic code. Proteins were seen as the exclusive actors in both the cell metabolism and its regulation. Recent findings have overturned this view. It has become apparent that RNA is by no means only an intermediate channel of information. The complexity of higher organisms is due to a "hidden" level of regulation, where new classes of RNA genes, transcribed from DNA just like protein coding genes, become active players by themselves [Mat03]. They regulate processes such as cell differentiation, intensity of the translation of other genes, immune response, or sensing of environmental conditions. Close to a hundred functional classes of such RNA regulators have been identified [BFF$^+$05], and their precise number remains evasive.

These RNA regulators exert their function by means of their structure. An RNA chain molecule folds back onto itself, and by forming helical regions similar to double-stranded DNA, it creates the stable structure relevant for its particular function. While the computational prediction of the full 3D structure of an RNA molecule is still out of reach, there is an intermediate level of information, the 2D structure, which can be predicted with good success. This is achieved by dynamic programming algorithms implementing a well-established thermodynamic model [ZS81, HFS$^+$94, GVR04]. They evaluate the complete folding space of a given molecule, which holds a number of 2D structures that is exponential in the length of the molecule, and return the structure of minimal free energy. This approach is called MFE-folding. However, the MFE structure is often weakly defined, and relevant structures may be hidden in the near-optimal folding space.

To classify RNA molecules that have been obtained experimentally, or in order to predict functionally active RNA genes from genomic sequences, general MFE folding is inadequate. A large number of specialized structure prediction algorithms must be developed, one for each structural class. The elaborate thermodynamic model must be combined with detailed domain knowledge about the relevant structural features. While an experimental biologist can be expected to determine these features, he or she cannot be expected to develop, implement and debug the required dynamic programming algorithms.

In recent years, substantial work has been devoted to bridging this semantic gap. Robert Giegerich and coworkers have developed a systematic approach to dynamic programming over sequence data, which is useful in bioinformatics beyond the area of RNA structure analysis [GMS04]. This approach, called algebraic dynamic programming (ADP), provides a declarative language for specifying dynamic programming algorithms, and has been successfully used in the development of several recent bioinformatics tools [RG04, GVR04, RSHG04]. ADP is based on the concepts of tree grammars and algebras, and closely related to the well-studied field of algebraic path problems [Rot90, PS05]. It presents no difficulty to a person with a solid computer science background. But again, the domain experts in the field of regulatory RNA usually have no previous experience in declarative programming and cannot be expected to adopt this new method.

Hence, the goal of my thesis was to take this programming method one step further: Biologists usually communicate RNA structures graphically, drawing pictures of relevant structural features annotated by extra information. Therefore, I have developed an editor that allows to draw such graphics from pre-defined building blocks. The semantics of these blocks are such that they can be transformed into components of a declarative program that folds RNA sequences into the 2D structure described by the graphics. This program incorporates a substantial body of domain knowledge, in particular the thermodynamic model with its more than 1000 parameters. The new program is composed from components which faithfully implement this model, and are adapted from existing general RNA folding programs implemented in ADP. Since the resulting program is generated from the graphics, the user can produce a (biologically) inadequate motif description, but not a buggy program.

Organization of Work

In this thesis, I am presenting the results of my work in designing, implementing and installing a software environment for RNA motif searches: Locomotif. It includes a visual editor for motif definition, translation of the motif structure to XML code and client-server interactions, and further, translation of the XML code to ADP and compilation to C.

In the following Chapter, I introduce the biological background of the problem domain starting with a historic overview of the functions of RNA. I continue with modern noncoding RNAs and the thermodynamic properties of RNA structures. Then, I define the concept of RNA motifs as compared to RNA secondary structures and present a well-known RNA motif, the Iron Responsive Element, which serves as an example for the remainder of this thesis.

In Chapter 3, I focus on the computational background of modeling RNA structures. I introduce the concepts of context free languages and their use to model RNA structures using probabilistic production rules. I describe domain specific languages (DSLs) as a more general means to represent RNA structures and introduce algebraic dynamic programming as a context free grammar based DSL for writing dynamic programming algorithms. I provide the complete language structure for modeling RNA motifs using ADP and finish the chapter with a specification of the graphical language building blocks used within the Locomotif system.

In Chapter 4, I present current approaches to the problem of finding RNA motifs in DNA/RNA sequences. I classify these approaches in descriptor-based, homology-based and specific search programs. I provide a thorough review of the most widely used programs for either class, using RNAMotif [MEG$^+$01] as the main example for descriptor-based and Infernal [Edd02] for homology-based tools. I identify the strengths and weaknesses of the different approaches and draw conclusions to what is required for the Locomotif system.

In the main part of the thesis, Chapter 5, I introduce the Locomotif software environment. I present the composition of the system and its integration in the client-server architecture. I provide an in-depth report on the concepts of the visual editor, the techniques used for visualization and data storage and manipulation. I give insights into the handling of user interactions and the functions for traversing through RNA structures for the translation to XML. I introduce the concepts behind the translation from XML to declarative language and give several examples for RNA motif blocks in ADP code. I conclude with a short description of the ADP compiler used to obtain executable C code. The compiler itself was developed and adapted for the Locomotif system by Peter Steffen [GS06, Ste06].

In Chapter 6, I include a publication from a related area of my work which deals with checking for grammar ambiguity. I explain the relevance of this work to the Locomotif system and present cases of ambiguity issues within the generated motif grammars.

In the last Chapter, I identify aspects of the software that need further improvement and present ideas for future work based on the existing software environment.

I conclude with a review of the work I achieved.

Chapter 2

Biological Background

For many years the historic dogma by Francis Crick from 1958 held true in the world of molecular biology:

> *Biologists should not deceive themselves with the thought that some new class of biological molecules, of comparable importance to proteins, remains to be discovered. This seems highly unlikely.*

In recent years though, it has become apparent that another important group of molecular players are involved in a multitude of cellular processes. We came to realize that active RNA molecules are of at least equal importance to proteins. An ancient world of RNA, completely independent of protein-based functionality is even thinkable.

The earliest notion of the importance of RNA was grasped in the 1930s by Torbjörn Caspersson who suggested a functional part of RNA in the cytosol of eukaryotic cells based on microscopic studies. The first understanding of that kind of function was made in the 1950s when Francis Crick stated the adaptor hypothesis claiming that translation is mediated by transfer RNA adaptor molecules. Around the same time, experiments to discover the nowadays well-known tRNAs were undertaken. It was known back then, that proteins are assembled sequentially from amino acids on ribosomes. In 1960, Brenner and colleagues discovered mRNA as an unstable RNA molecule that serves as an information carrier from genes to ribosomes. The fact that RNA is involved in the synthesis of proteins was established by James Watson in 1963. Finally, in 1965 Crick deciphered the genetic code with RNA as the messenger of genetic information. Yet, any other kind of RNA was either considered to be an mRNA or to be of no importance, some kind of junk RNA.

The first catalytic activity of RNA was found in 1975 by Cech and coworkers. This discovery gave rise to the notion of a new world of RNA and the foundation for the changing belief system concerning the evolution of DNA, RNA and proteins. Now, it was evident that RNA could not only act as an information storage capacity (such as DNA), but also carry out catalytic functions (such as proteins). In the following years more examples for such catalytic functions were found and in the 1990s, RNA's active membership in the catalytic center of the ribosome was determined. The historical overview on the functions of RNA follows [Dan02].

The fact that RNA can serve both as storage for genetic information and as active molecules gave rise to the idea that in early days life depended only on RNA [Gil86, LG90]. Many new functions and types of RNA have been discovered and continue to be found every day. We have entered the new world of noncoding RNA also entitled the Modern RNA world. It is believed that higher organisms rely on active RNA molecules regulating processes such as cell differentiation, intensity of the translation of other genes, immune response, or sensing of environmental conditions [Mat03].

2.1 Noncoding RNAs

In the new days of RNA several functional classes have already been established. Examples for small nuclear RNAs (snRNA) are spliceosomal RNAs in RNP complexes. Small nucleolar RNAs (snoRNAs) are involved in rRNA modification. Other functional RNA classes are short interfering RNAs (siRNA) and micro RNAs of regulatory function.

In 2001, Erdmann et al. [EBH$^+$01] published a review on regulatory RNAs. According to him, ncRNAs can be divided in five groups:

- DNA markers involved in dosage compensation and imprinting
- Gene regulators based on silencing or RNA-RNA interaction
- Abiotic stress signals
- Biotic stress signals
- Others of various origins and functions, e.g. brain-specific RNAs, RNAs involved in meiosis, oogenesis and other specific processes.

Other review articles by Sean Eddy [Edd01] and Gisela Storz [Sto02] describe the evolvement of the new world of RNA and the diverse functions ncRNAs encompass. According to these reports, ncRNAs are involved in many cellular processes such as transcription, gene silencing, replication, RNA processing, modification and stability, mRNA translation and protein stability and translocation. A prominent example is the *Xist* RNA, which is produced by the inactive X chromosome and is involved in gene silencing. Another important class of ncRNAs are snoRNAs which exist in two different types. The

C/D-box snoRNAs direct methylation of RNAs, whereas the H/ACA-box type is involved in pseudouridylation of RNAs. They form basepairs with sequences near the sites that need to be modified. The class of microRNAs contains many family members and still is a hot research topic concerning the detection of new microRNAs as well as their targets.

Even though many ncRNAs of diverse functions have already been identified, this may be just the tip of the iceberg. In eukaryotes, the major part of the genome is classified as junk DNA offering space for many more potential ncRNAs genes.

2.2 Thermodynamics of RNA secondary structures

For all these types of RNA, it is important to note that their function does not only depend on their sequence, but much rather on their three - dimensional structure. Short sequence motifs are often involved in binding operations, but the overall structure is responsible for the form of the molecule. The 3D structure of an RNA is formed by the folding of the chain molecule, leading to basepairs arranged in helical regions similar to double-stranded DNA. Although one cannot yet compute the 3D structure of an RNA molecule, the prediction of 2D structures is possible using dynamic programming algorithms based on a thermodynamic model [ZS81, HFS$^+$94, GVR04]. These algorithms evaluate the complete folding space of a given molecule and return the structure of minimal free energy. This approach is called MFE-folding. In order to obtain the overall MFE, the structure is distinguished in stacked regions and loops. Elements of both kind are given free energies: For base pair stacks the hydrogen bonds as well as stacking effects are taken into account. Loops are usually less favorable since they interrupt stacking regions. Furthermore, as they are usually bounded, e.g. the loop region of the hairpin loop is constrained at both ends, these bounds restrict their desire too move freely, causing an overall negative effect by increasing the minimum free energy of the structure. The MFE structures for motif searches with Locomotif are based on the energy minimization rules described by Mathews et al. ([MSZT99]). Dangling bases, i.e. bases in single strands or loop regions adjacent to stacking regions, are not yet taken into account, but will be incorporated soon. It is important to note that the MFE structure is often weakly defined, and relevant structures may be hidden in the near-optimal folding space.

2.3 RNA structural motif definition and search

RNA secondary structures A typical graphical description of an RNA secondary structure can be seen in Figure 2.1.

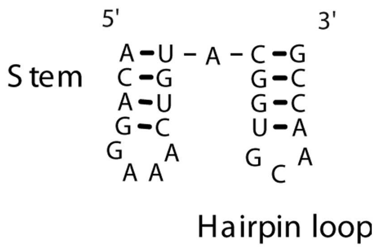


Figure 2.1: A simple RNA secondary structure consisting of two hairpins connected through a single strand. 5' refers to the start of the sequence "acaggaaacuguacggugcaaccg" constituting the backbone of the structure; 3' to the end.

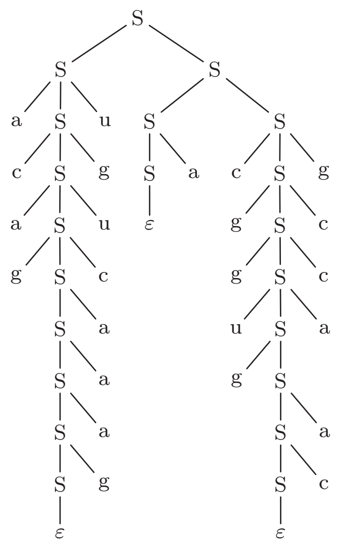
This shows a concrete structure for a particular sequence, including the concrete sequence of bases A, C, G, and U in the chain molecule, and the structural features such as loops and stems that are formed by thermodynamic forces.

For computer input and output, structures are often represented as annotated strings: There is the sequence of bases, mathematically a string over $\{A, C, G, U\}$, and its structural annotation, a string (of the same length) over $\{$`(`,`.`,`)`$\}$. Parentheses must be properly nested, and hence the possible annotation strings form a context free language. Each matching pair of parentheses denotes a pairing between the bases in the corresponding positions. For short molecules, the structural features such as hairpins etc. are easy to spot, but for longer ones, the graphical representation is clearly superior in communication between humans.

A parsing algorithm is used to recognize the structural features in the annotation string. This leads to a tree representation of structures, which is explicitly or implicitly used by many programs that deal with structure prediction or comparison. These two important alternative representations are shown in Figure 2.2.

RNA structural motifs An RNA structural motif is independent of a concrete RNA sequence. It can be seen as a parameterized structure that specifies a particular arrangement of structural features. This arrangement may be annotated with additional information to make the motif more specific. The two most important annotations are size limits on the overall sequence or on particular features, and specific base information at points where it is known to be critical for the molecule's biological function. (Such information can be drawn from studying sequence segments conserved in the course of evolution or from wet-lab experiments.)

There is no common code established in biology to describe structural motifs. The



```
((((....)))).((((...))))
acaggaaacuguacggugcaaccg
```

Figure 2.2: Tree representation (top) and annotation string (bottom) for the secondary structure shown in Figure 2.1.

most successful proposal in this respect is the motif description language defined by the tool RNAMotif [MEG$^+$01], where essentially a list of structural features is described in ASCII. While this notation allows to express quite complex motifs, the connection to the graphical notation is lost. More details on current approaches to model and search RNA motifs are given in Chapters 3 and 4.

Here, I will introduce a biologically important RNA motif that is used in examples in the remainder of this thesis.

Iron Responsive Element

The Iron Responsive Element (IRE) (Figure 2.3) is an essential element in the regulation of the iron metabolism of all vertebrates by serving as the binding site for the IRE-Binding Protein (IRE-BP). The IRE-BP represses the translation of ferritin mRNA whose protein (ferritin) is responsible for the storage of iron. Also, it inhibits the degradation of the transferin mRNA (cellular uptake of iron) which is otherwise rapidly degraded. Since iron is required for many cellular processes such as oxygen transport and storage, electron transport and storage or DNA synthesis, the IRE plays an important role in molecular

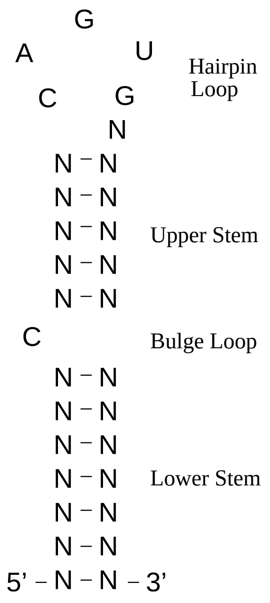


Figure 2.3: The Iron Responsive Element exists in slightly different variations, but a consensus form is composed of a long stem that is interspersed by a single-c bulge followed by a hairpin loop.

biology. It has been shown that the structure of the motif in combination with some specific sequence information and length constraints is responsible for the function of the motif [KEBM95].

Chapter 3

Modeling RNA Secondary Structures

In order to analyze RNA secondary structures computationally, they have to be modeled in a fashion that preserves the concept of a secondary structure derived from the folding of its primary sequence. The underlying thermodynamic rules must be included either implicitly, e.g. via the statistical properties contained in a covariance model, or explicitly using scoring functions. Choosing a useful data structure for RNA depends on the information that needs to be stored and accessed. From a strictly computer science point of view, RNA is first of all a sequence of letters over a small alphabet of size 4. A second dimension is the connection of two letters that do not occur next to each other, but at more distant locations within the string. For this work, tertiary connections (although important) are not considered, so we omit these in regarding modeling aspects of RNA structures. While the primary sequence is fixed, the secondary connections are variable and depend on intricate underlying data that must either be modeled or simulated.

A tree is often employed to represent a typical RNA structure. Usually, an RNA folding algorithm which calculates the secondary structure for a given RNA sequence, uses a tree representation for the structure. This is commonly achieved by modeling RNA structures with context free grammars (CFG). These are well-suited for the problem because they allow for production rules generating pairwise terminal symbols, i.e. basepairs. Depending on the choice of a particular production rule, a different tree representation for the RNA sequence is generated. The secondary structure is found as the parse tree of the CFG.

Alternatively, RNA structures can be described in language-like terms using a domain specific language for RNA secondary structures. While many approaches using domain specific languages will actually model the sequence structure relation in a CFG fashion, this approach allows for greater freedom. In CFG, only nested basepairs can be generated. Thus, no pseudoknots or other tertiary interactions can be incorporated. Using a DSL, one can include these features, but then, one's own formalism must be found. Here, I will first introduce (stochastic) context free grammars commonly used in programs such as Infernal (see Chapter 4). Then, I will talk about domain specific languages, focusing on algebraic dynamic programming grammars which I use to model and search RNA structures in the Locomotif approach.

3.1 Stochastic Context Free Grammars

3.1.1 Formal grammars

A formal grammar is composed of a set of rules that can be used to generate a set of strings comprising a formal language. A formal generative grammar has a set of nonterminal symbols including the start symbol, an alphabet of terminal symbols and a set of production rules. There are several kinds of grammars with associated languages that can be distinguished by the types of rules they employ. The Chomsky hierarchy [Cho56] defines four levels of grammar types according to their expressive power: regular grammars, context-free grammars, context-sensitive grammars and unrestricted grammars. For natural languages the later two are of importance, and general pseudoknotted structures can only be modeled with context-sensitive grammars. Regular grammars are not able to produce pairing terminals and are thus inappropriate for modeling RNA structures.

3.1.2 Context free grammars

A context free grammar has a set of terminal symbols (the alphabet), a set of nonterminal symbols including the start symbol and a set of production rules of the form $V \to w$ where V is a nonterminal symbol and w is a string of terminal and nonterminal symbols (including the empty string ε). It contains only productions rules that do not depend on the context of their occurrence. That means, there are no terminal symbols on the left-hand side of the production rules.

The context free language $L(G)$ is generated by the repetitive use of these production rules beginning with the start symbol. A set of terminal strings is obtained by successively replacing the nonterminals with the right-hand sides of the appropriate production rules. The order in which the rules are used to obtain a terminal string is captured in a syntax derivation tree. In case of modeling RNA structures, there is a strong correlation between the derivation tree and the RNA structure it represents. The sequence of the structure can be obtained by reading the leaves in the tree from left to right whereas the structure itself is visible in the shape of the tree.

A simple example for describing RNA structures is Dowell and Eddy's grammar $G1$ [DE04] to describe RNA secondary structures:

$$\begin{aligned} G1: \quad & S \to aSu \,|\, uSa \,|\, cSg \,|\, gSc \,|\, gSu \,|\, uSg \\ & S \to aS \,|\, cS \,|\, gS \,|\, uS \\ & S \to Sa \,|\, Sc \,|\, Sg \,|\, Su \\ & S \to SS \\ & S \to \varepsilon \end{aligned}$$

A shorthand notation is commonly used where a stands for any base A,C,G,U, while a and $\hat{a}$ occurring in the same rule stand for either one of the base pairs (A,U), (U,A), (C,G), (G,C), (G,U), or (U,G).

$$G1: \quad S \to aS\hat{a} \,|\, aS \,|\, Sa \,|\, SS \,|\, \varepsilon$$

3.1.3 Stochastic context free grammars

Stochastic context free grammars associate a (non-zero) probability with each production, such that the probabilities for all alternative productions emerging from the same non-terminal symbol add up to 1. As a string is derived, probabilities of the involved rules multiply, leading to more or less probable structures for a specific input sequence.

We extend the CFG $G1$ to a SCFG by the following example probabilities:

$$\begin{aligned} P_{S\to aS\hat{a}} &= 0.4 \\ P_{S\to aS} &= 0.2 \\ P_{S\to Sa} &= 0.2 \\ P_{S\to SS} &= 0.1 \\ P_{S\to \varepsilon} &= 0.1 \end{aligned}$$

3.1.4 CYK algorithm

All derivations for a string can be constructed by a CYK-type parser [AU73]. The algorithm checks whether the string can be produced by the given set of rules and how it is produced. It can compute the overall probability of a given string, summing up probabilities over all its derivations, in which case it is called the Inside algorithm. Alternatively, the parser can return the most likely derivation of the input string, in which case it is known as the Viterbi algorithm. For grammar $G1$, the corresponding CYK-based Viterbi algorithm is shown here:

```
Input: Sequence $x = x_1 \dots x_n$
Initialization: for $1 \leq i \leq n$
$S(i,i) = P_{S \to \varepsilon}$

Iteration: for $1 \leq i < j \leq n$
```

$$S(i,j) = \max \begin{cases} S(i+1, j-1) * P_{S \to x_i S x_j} \\ S(i+1, j) * P_{S \to x_i S} \\ S(i, j-1) * P_{S \to S x_j} \\ \max_{i \leq k < j} \{ S(i,k) * (S(k+1, j) * P_{S \to SS}) \} \end{cases}$$

Considering the application of SCFG to modeling RNA secondary structures, the CYK algorithm allows us to identify the best scoring manner in which the model (SCFG) can generate the target sequence. Its result is a parse tree. Thus, given an RNA sequence, it determines the optimal order of rules to produce the input sequence. Using the representation of this order in a derivation tree structure, one can obtain a possible secondary structure for the given sequence.

3.2 Domain Specific Languages

A domain specific language (DSL) is essentially a programming language tailored for a specific problem domain. Ideally, the syntax is easy to understand for an expert in the domain who is usually not a programming expert. Therefore, a computer scientist has to provide the infrastructure to deliver a running program based on the specifications in the domain specific language. There are two ways to go: either the domain specific language is a restricted programming language so that the domain experts write programs on their own. Or the language is a declarative one where the domain experts only define the problem and a computer scientist translates (usually as an automatic procedure) the DSL into an executable program.

In the realm of bioinformatics, the second approach is probably more successful as the domain experts have oftentimes limited computational skills and the problem domain is difficult to grasp for a non-expert. DSL are well-suited to model RNA structures as this is a clearly defined domain where the overall structure is made up of repetitive substructures. For each of these substructures, terms in the DSL can be found and operators describing their order in the overall structure must be specified. In fact, since CFG lend themselves so easily to modeling RNA structures, a DSL language for RNA structures will oftentimes also have an underlying context free grammar.

In our application domain, i.e. RNA motif searches, the DSL for modeling RNA motifs evolved naturally during our work in algorithms for bioinformatics. Following this evolution, I will first present the technique of Dynamic Programming which is widely used for many problems in bioinformatics. Robert Giegerich developed an approach termed Algebraic Dynamic Programming (ADP) [Gie00] that allows to separate the recognition phase (definition of the problem domain) from the evaluation phase (computational analysis of the defined problem domain). ADP is a DSL suited for all problems that can be solved via Dynamic Programming over sequence data.

The graphical programming system presented here generates declarative ADP code for RNA motif search. As such, it covers a subdomain of ADP for biosequence analysis. Thus, this technique can be considered as Graphical Algebraic Dynamic Programming for RNA motif search. As most aspects of the programming system are described in detail in Chapter 5, here, I will only focus on those topics relevant in the context of modeling RNA structures.

3.2.1 Dynamic Programming

Dynamic Programming is an algorithmic procedure to solve optimization problems. It can be used whenever the problem consists of several uniform partial problems and if the overall solution can be obtained by combining optimal solutions to the partial problems. (Bellman's optimization criteria). The partial problems are split recursively until a solution can be found directly. Then, the solutions are combined to find the optimal answer for a bigger partial problem. Solutions that have already been calculated are stored in tables that can be accessed during different stages of the algorithm.

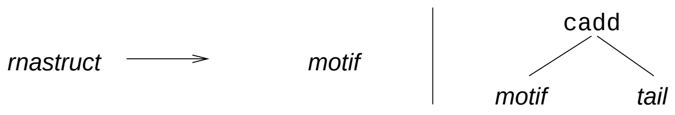
3.2.2 Algebraic Dynamic Programming

Even though a dynamic programming algorithm is an elegant way to solve optimization problems, it can be very difficult to develop the appropriate algorithm. The main task is the definition of recurrences used to fill the tables storing the results of the subproblems. There is no systematic approach to choose the necessary tables and construct the recurrences. The reason for this difficulty is the fact that a dynamic programming algorithm performs two tasks at the same time: the search space is constructed by defining all potential solutions and the potential solutions must be evaluated according to the optimization criterion. In the DP recurrences, these two parts are intertwined and cannot generally by separated. Simple separation of the generation and evaluation phases would lead to algorithms of exponential runtime as all possible solutions would first be enumerated and then be filtered.

Algebraic Dynamic Programming (ADP) offers a means for the programmer to separate the description of the search space from the functions needed to evaluate candidate solutions. It is a domain specific language implemented in Haskell and the syntax clearly reflects this origin. An ADP compiler that translates ADP syntax in C code was developed by Peter Steffen [GS06] making the DSL independent from its original Haskell background.

3.2.3 Modeling RNA motifs in ADP

RNA structures are seen as trees based on a set of tree operators. The start element for the language describing RNA structures is an `rnastruct` which is a `motif` or a `motif` next to a `tail`:

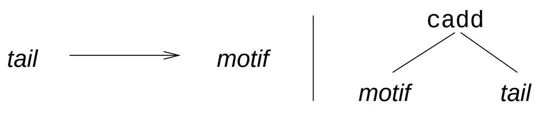


A `motif` is one of the structural elements of RNA secondary structures:

- single strands (`ss`)
- hairpin loops (`hl`)
- stems, i.e. stacking regions (`sr`)
- left and right bulge loops (`bl`,`br`)
- internal loops (`il`)
- multiloops (`ml`)

Each of the building blocks represents a tree structure as shown in Figure 3.1. The tree operators are the evaluation functions applied to the subtrees rooted at the specific structural element. Some of these trees end in nonterminals that refer to the root node of the next subtree of the RNA structure. Others, i.e. single strands and hairpin loops contain only leave nodes.

A `tail` is either just a `motif` or a `motif` next to a `tail`.



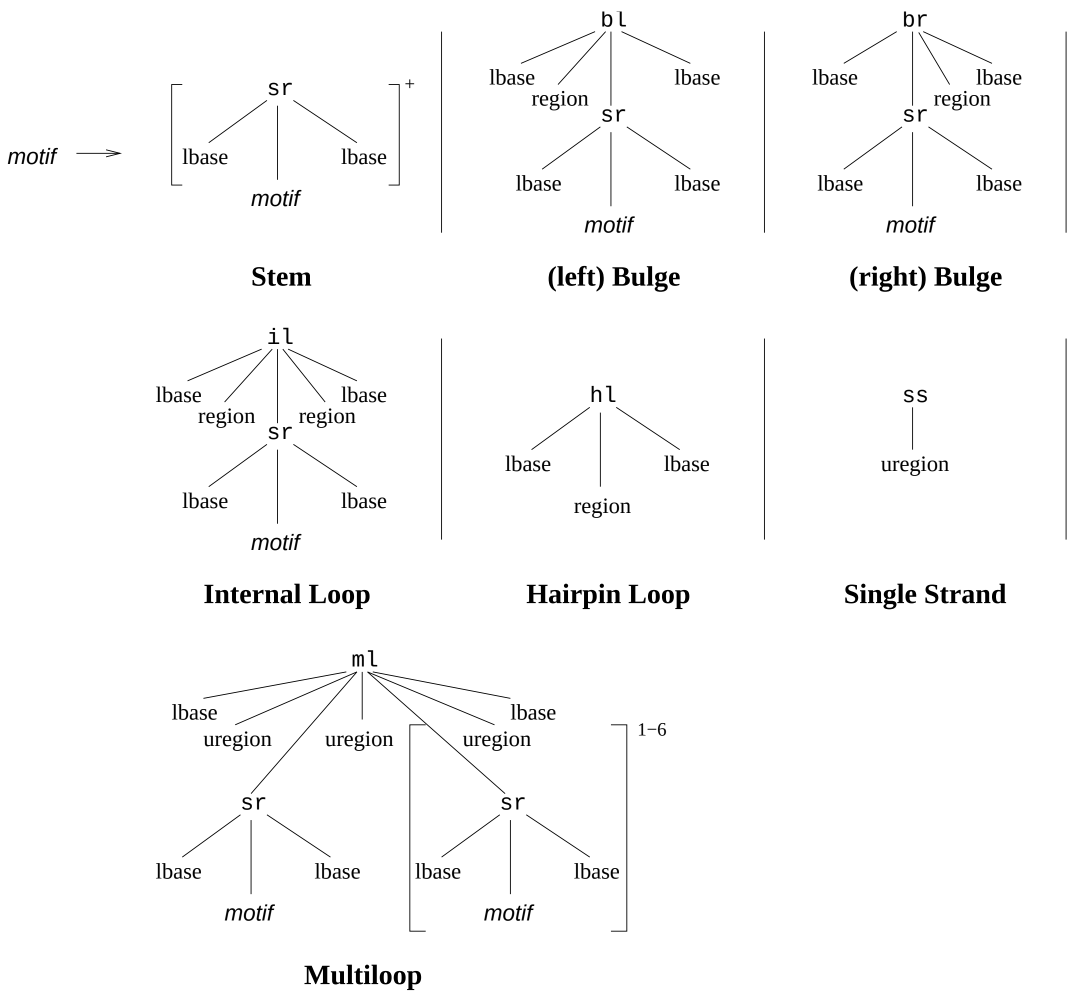


Figure 3.1: Each structural element is a subtree of the overall structure tree: Nonterminal grammar symbols in *italics*, functional operators in `courier` and parsers in standard font.

Therefore, a `motif` always generates the root of a subtree while a `tail` combines different sibling subtrees. `tail` and `motif` are derived from `rnastruct` which is the root element of the complete language of RNA secondary structures. An RNA motif is a particular secondary structure and thus an instance of this grammar tree. An example is shown in Figure 3.2.

The ADP syntax for the RNA motif grammars is quite simple. A parent node is a functional operator which precedes the line followed by the grammar operator `<<<` denoting the application of the functional operator on the following grammar clause. Additionally, an evaluation algebra is defined in Haskell syntax that interprets these operators as

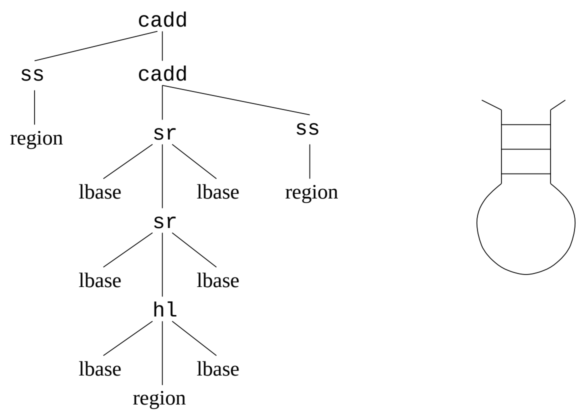


Figure 3.2: A simple hairpin motif and its corresponding ADP code tree.

functions. When using ADP for modeling RNA motifs, there is no need to redefine the algebra functions for every motif. A general scoring algebra can be applied to all motifs and its functions are integrated into the relevant grammar clauses. A child node of the overall motif tree is a terminal symbol which acts as parsers for the input sequence (e.g. `lbase` recognizes a single nucleotide whereas `region` accepts 1 or more nucleotides). The grammar operator `~~~` is a parser combinator that can be read as "next to". It defines an order upon the parsers contained in the grammar clause. It is used to combine the different child elements on the same level of a grammar tree. Similarly, the `|||` operator combines different clauses in an OR - fashion.

```
tail = cadd <<< motif ~~~ tail |||
                motif
```

Despite the simplicity of the syntax, a non-programmer and even some computer scientists not familiar with functional programming may find it difficult to write ADP grammars for RNA motifs. This gave rise to an even more abstract domain specific language placed on top of ADP: Graphical ADP.

3.2.4 Graphical Algebraic Dynamic Programming for RNA motif search

Graphical Algebraic Dynamic Programming is a DSL for the subdomain of modeling RNA motifs. Relying on the ADP grammars for RNA structures, we identified several repetitive structure elements. These grammatical repetitions can also be found in the visual descriptions of RNA motifs commonly used by biologists. Hence, we chose to define a visual

DSL exclusively for RNA motifs that can be handled by a graphical editor. I identified the necessary building blocks of RNA motifs and designed graphical representations for them. An important aspect to consider was the fact that all building blocks had to be closed by basepairs on all sides. That way, all building blocks can be combined with each other without the potential of creating energetically unfavorable single basepairs or running into ambiguity issues (see Chapter 6). Figure 3.3 gives an overview of the standard and compound building blocks used within the Locomotif system and their corresponding code template trees.

Every building block resembles a template tree for ADP grammars and thus, a user with no programming experience can design RNA motifs via this visual DSL. Actually, since the user is probably an expert in the field of RNA biology, s/he will have used the DSL for a long time already and just obtains a means to creatively employ it to generate useful programs.

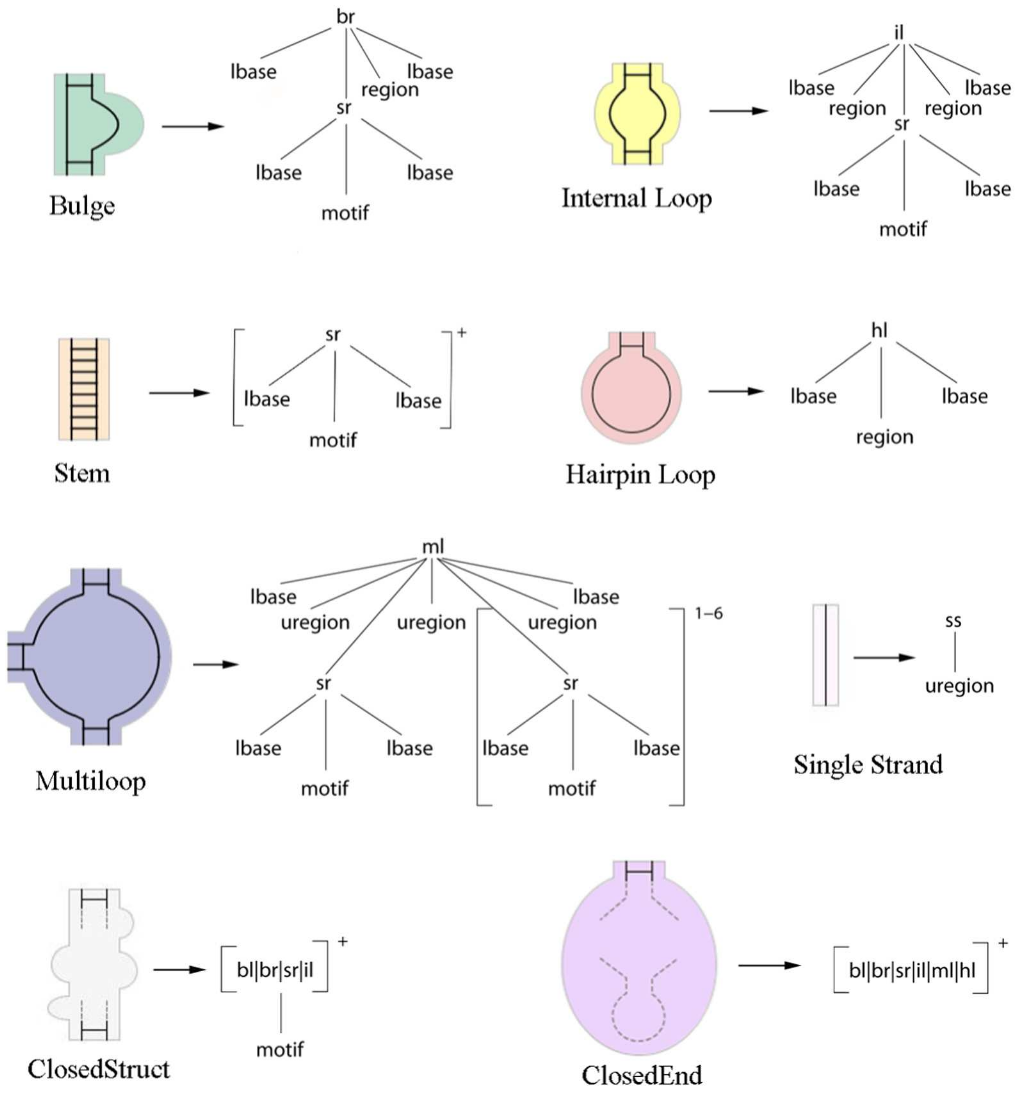


Figure 3.3: Every visual building blocks corresponds to a building block template tree containing the relevant ADP code. The left bulge was omitted in this figure as the visual Bulge building block represents both kinds. ClosedStruct and ClosedEnd represent subtrees composed from a limited selection of building blocks as specified within the brackets.

Chapter 4

Searching RNA motifs

4.1 Overview

The currently available programs for searching RNA motifs can roughly be divided in two classes. In the first class, the motif specification is provided by an expert user. These programs have means for defining a motif descriptor via some simple patterns or a more complex declarative language. Here, an evolution from simple sequence patterns as known from standard sequence analysis problems to intricate modular languages based on regular expressions can be observed. The search algorithms are more or less standard pattern-matching or tree traversal approaches. The second class are those programs where information on the motif is not provided directly by the user, but rather extracted from the available data. Here, the search algorithm depends strongly on the method of information extraction, being either alignment-based or relying on genetic algorithms. Furthermore, there are several programs used for special types of noncoding RNA that belong to either or both of the two classes.

Here, I will first give an introduction to descriptor-based programs and then continue with those using inherent information. At the end, I will mention some specific programs.

4.2 Descriptor-based programs

In descriptor-based programs, the information on the motif is specified by an expert user via a descriptive language. The language is in all cases of a simple declarative kind incorporating repetitive patterns and/or regular expressions. Different search algorithms are employed ranging from standard string/pattern matching algorithms to depth-first tree searches.

The earliest programs used for searching RNA motifs focused only on complex sequence motifs [DHS84] or sequence motifs separated by spacers [MM93]. Other programs offered more complex pattern languages where helices were represented by palindromic repeats [SSA92]. Thus, basic structural information could be integrated in the motif query.

RNAMOT The program RNAMOT [GMC90, LGC94] was an early program with a descriptor file composed of structural elements. Here, a motif is defined via a list of helical and single stranded elements and, for each of the elements, positions, size boundaries and sequence motifs can be specified. The structural elements are searched in an order specified by the user that can be different from the order of structural elements in the motif. Thus, elements with strict sequence requirements can be favored to avoid unnecessary deep searches. If the algorithm cannot continue the search with a structural element, it traces back and tries to find another location for the previous one.

Palingol Palingol [BKV96] is a search environment based on a declarative programming language for describing secondary structures and search algorithms to scan sequence databases. Here, the user describes the motif as a list of helices and can define both local and global contraints on the helices. The list must then be translated in the Palingol syntax and the Palingol interpreter builds an evaluation tree for the constraints. A HelixSearch program calculates for every sequence in the database an ordered list of all helices. Finally, in a branch-and-bound procedure, the Palingol engine finds subsets in the list of helices that match the required constraints of the descriptor.

PatScan PatScan [DLO97] is a web-based program that can be used to search motifs in EMBL or SWISS-PROT. The descriptive language allows for the labeling of pattern units which can then be used to denote helices by reverse complements (`p1 ... ~p1`). It can incorporate not only mismatches, insertions and deletions, but also alternative constructs or nonstandard pairing rules. The pattern matching algorithm `scan_for_matches` is freely available.

PatSearch PatSearch [PLD00] uses the `scan_for_matches` algorithm from PatScan and essentially the same descriptive language. As an advancement, it offers a statistical assessment of the significance of the results via comparisons with a Markov chain simulation.

HyPa The HyPa program [GSKS01] is still under development. As of now, the descriptive language is very complex and powerful. It is based on regular expressions and is essentially modular in nature. Thus, patterns can be reused. Approximate motif descriptions are supported and the user can define scoring functions and constraints. Currently,

```
parms
  wc += gu;
descr
  h5 (tag='5p_helix', len=4)
  ss (len=4, seq=''GNRA'')
  h3 (tag='3p_helix')
```

Figure 4.1: A simple RNAMotif descriptor file for a standard helix forming a hairpin. GU basepairs are allowed via the parameter section.

the search strategy is based on the PatScan and RNAMotif algorithms, but an original HypaSearch algorithm is under development.

RNAMotif

RNAMotif [MEG+01] is the most widely used program for searching RNA motifs based on user-defined input. The descriptor contains four sections: First, a parameter section for default variables, then the main descriptor containing the criteria to generate a match, a sites section for more complex relations among elements of the descriptor and the score section. RNAMotif first produces all hits that fulfill the criteria of the descriptor and sites sections. Then, the score section is used to filter and rank the matches. Thus, in case of a loose descriptor, the program can generate a very large intermediate result file.

The descriptor contains a sequence of helical and single stranded regions ordered from 5' to the 3' end. For each helix, there is both a `h5` helical region on the 5' strand and the corresponding `h3` helical region on the 3' strand. Each structural element can be constrained both in size and with sequence motifs in brackets following the element. Figure 4.1 shows a descriptor file for a simple standard hairpin.

A binary search tree is built from the descriptor file where the root of the tree is the helix itself, the left subtree is the motif contained in the interior of the helix and the right subtree is the motif that follows the helix. Then, all implicit and explicit length constraints are checked whether they are compatible. Finally, bounds are computed for all structural elements according to these length contraints.

The search phase is a straightforward depth-first search using the binary search tree. Starting with the left most position of the target sequence, it is checked whether it contains any instances of the left-most submotif, i.e. the root of the tree. Then, it recursively finds all solutions for the interior of that motif, i.e. the left subtree. If all interior regions have been searched, the algorithm is applied to the region following the left-most submotif.

When all candidates at a particular recursion level have been examined, the search algorithm backs up and continues the search on any unexamined candidates from the previous level. Once all candidates at the original level have been tried, the overall search is started again one position to the right on the target sequence. Thus, the algorithm has a runtime of $O(n^{2*k})$ where k is the number of helices.

The quality of the results depends strongly on the quality of the descriptor and an adequate scoring function.

4.3 Homology-based programs

In contrast to descriptor-based approaches, here, the motif is not defined by an expert user, but rather extracted from the available data. The advantage of these approaches is that even the expert might not be able to determine which features are truly relevant for a functional RNA motif. Plus, no time has to be spend time on learning a descriptive language to define the desired motif. On the other hand, these programs are only reliable if the available data is adequate, and only those motifs can be searched that are already rather well-known.

There are programs based on an input of many example sequences which are aligned to obtain information on the consensus structure of the sequences. Other programs only take one sequence with its structure as input and use general statistical data to evaluate program results. Here, I will first describe some approaches based on multiple sequences and then continue with examples for single sequence input.

ERPIN ERPIN [GL01] takes a sequence alignment with secondary structure annotation as input. For every helix in the structure a log-odds-score profile matrix is calculated which has 16 rows (all possible types of basepairs) and n columns (n being the length of the helix). For every single stranded region a similar profile matrix with just 5 rows (4 nucleotides plus a gap) is calculated. These profile matrices are filled according to the log score of the observed versus the expected frequencies of the bases/basepairs. The search algorithm calculates for every position in the target sequence the helix score by comparing its profile to the target sequence. Then, the single strand score is obtained via the alignment of its profile with the target sequence between the helical regions.

Infernal

Infernal [Edd02] takes a multiple sequence alignment of an RNA family with its consensus structure as input. Using this data, it calculates a “Covariance Model” for searching a

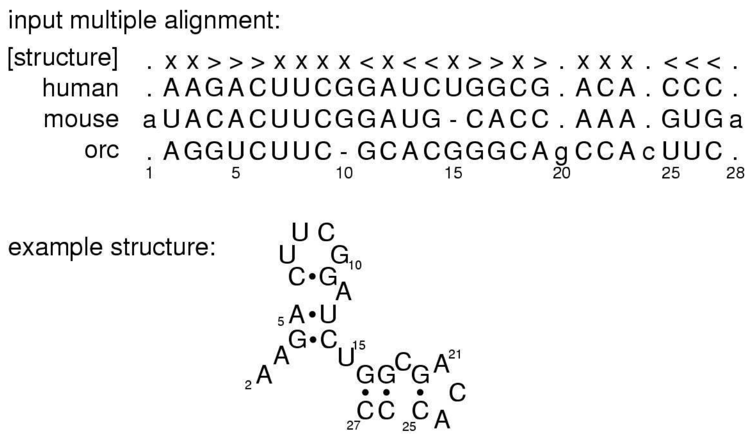

input multiple alignment:
[structure] . x x > > > x x x x < x < < x > > x > . x x x . < < < .
human . A A G A C U U C G G A U C U G G C G . A C A . C C C .
mouse a U A C A C U U C G G A U G - C A C C . A A A . G U G a
orc . A G G U C U U C - G C A C G G G C A g C C A c U U C .
1 5 10 15 20 25 28
example structure:

Figure 4.2: A small input alignment including three sequences and the consensus structure. Figure taken from [Edd02].

sequence database to find more family members. Covariance models take the variation of both sequence and structure in all input members into account and provide a scoring scheme to align the covariance model to a new potential family member. They are in principal based on modeling RNA secondary structures with SCFGs. Yet, while SCFGs model RNA secondary structures in general, a covariance model represents a particular family of secondary structures, i.e. an RNA motif. In SCFGs there are production rules for modeling basepairs and single stranded regions whereas a covariance model is an automaton that produces a particular basepair or a particular base at a certain position in the target sequence. Bases are emitted and the states of the model represent the consensus structure of the RNA motif as formed by the input sequences and its possible deviations in the target sequence. Of course, in the most general form, a covariance model can simply represent the class of all secondary structures using uniform emission probabilities and transition probabilities based on the production probabilities of a SCFG.

Computing a covariance model The first step in computing the covariance model is the assignment of basepairs and unpaired columns in the input alignment (see Figure 4.2) to nodes of a guide tree. The guide tree (see Figure 4.3) is simply the parse tree for the consensus structure of the alignment and it serves as the skeleton for the covariance model. In total, there are 8 different types of nodes representing basepairs, single stranded regions, bifurcations and start and end of the input alignment.

Since a guide tree only represents the consensus structure, in order to compare this structure to a target sequence, structural deviations must be taken into account. There-

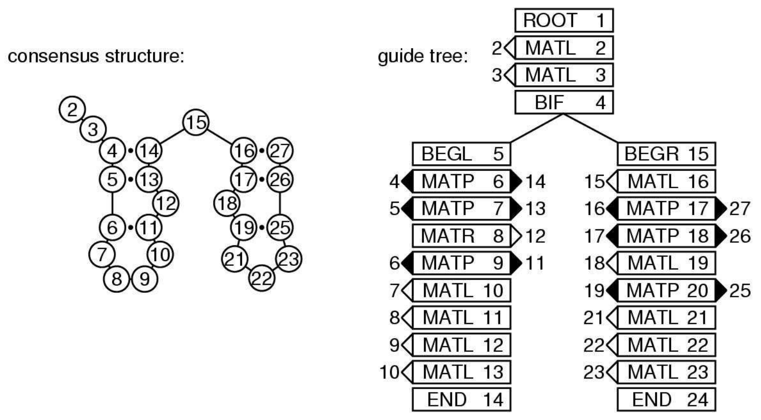


Figure 4.3: The guide tree is based on the consensus structure of the input alignment. Figure taken from [Edd02].

fore, the guide tree is expanded to include insertions and deletions as well as matches and mismatches which leads to the covariance model. Here, for every node of the guide tree, the covariance model contains several states depending on the kind of node (see Figure 4.4). For a MATP node (base pair), while aligning the model to a target sequence, one of four basic states (so-called split states) is found: a basepair in the target sequence (MP), a single left base (ML), a single right base (MR) or no bases at all (D). Additionally, any number of insert states on both the left and the right side (IL, IR) can be used.

In the end, the covariance model is a large directed graph with many parallel states for every column of the input alignment and the appropriate insert states in addition to the column. Then, each sequence of the input alignment is converted to a CM parse tree. Counts for observed state transitions and singlet/pair emissions are collected and converted to transition and emission probabilities. Finally, the target sequence is processed as a sequence of states of the covariance model. For every state, the transition probability is used to score the structure of the target sequence and the emmision probability evaluates the specific bases that occur at this position in the target sequence.

Search algorithm The CYK algorithm is used to align a SCFG with a target RNA sequence. The same algorithm is used to align the CM to a target sequence. Yet, this algorithm calculates a large Dynamic Programming matrix making it unfeasible to use for searching RNA motifs. Eddy devised a Divide and Conquer Strategy to reduce the space complexity with the cost of a small increase in time complexity (for details refer to [Edd02]. Then, the memory requirement of the program is reduced to $O(N^2 logM)$ for a model of M states and a query sequence of length N. The original time complexity of $O(MN^2+BN^3)$

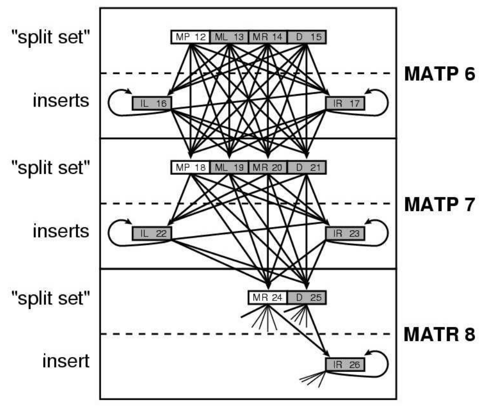


Figure 4.4: Every node of the guide tree is translated into several states of the covariance model to incorporate matches, deletions and insertions. Figure taken from [Edd02].

with B representing the number of bifurcations is raised to an upper bound of $O(M^2N^3)$. Yet, in a recently submitted manuscript, Nawrocki and Eddy introduce a query-dependent banding method that provides a four-fold speed-up for typical RNA queries [NE07].

In addition to these programs based on several input sequences, others were developed to deal with those cases where only one input sequence is available:

RSEARCH RSEARCH [KE03] uses the covariance model data structures and the search algorithm from Infernal. Yet the covariance model can not be calculated from an input alignment, but only from a given structure. Also, parametrization does not depend on the input sequences, but is done with the RIBOSUM substitution matrix calculated from empirical frequencies of a structure-annotated small subunit ribosomal RNA alignment.

FastR FastR [BZ04] also searches structural homologs based on the input of one RNA sequence with a known secondary structure. Here, structural filters are used to eliminate large portions of the database. Thus, the computation time is superior to RSEARCH. The alignment to the target sequence is based on a Nussinov-like counting model and employs a banded alignment for efficient computation. For scoring, the RIBOSUM matrix developed for RSEARCH is used.

RAGA

RAGA [NOH97] is another approach that uses two homologous RNA sequence (master and slave) with one given secondary structure (master structure) as input. The program does RNA alignment by a genetic algorithm. The chosen optimization function takes the quality of the primary sequence alignment, the stability of the folding of the slave sequence imposed by the master structure and any necessary gaps in the alignment into account.

The population consists of pairwise alignments between master and slave sequence. The initialization is performed based on Dynamic Programming with Added Noise, thus producing a diverse, yet good scoring generation zero. Mimicking natural evolution, individuals are selected according to their fitness (optimization function) for breeding. Offspring is generated both by mutation (altering of one parent) and crossover (combination of two parents). Parallelization is achieved by using islands for breeding and exchanging solutions between the different parallel groups. The fittest member of the population is a candidate solution. The procedure continues until no further improvements for the solution candidate are observed for some generations.

4.4 Specific motif search programs

In addition to these general purpose search programs, there are several tools tailored to the use for specific types of noncoding RNAs. These include programs for searching group I intron cores, snoRNAs, tmRNAs, tRNAs and micro RNAs.

Citron [LDM94] is a rule-based system searching for group I intron cores. Individual signals of these large motifs are identified and combined to more complex patterns until the entire core structure is assembled. Given the complexity of these large secondary structure elements, a definition of these RNA motifs in a general motif search program would be highly complex as well. Also, searching the entire structure at once would be far less efficient than focusing on individual parts of the motif and combining them step by step. Thus, in case of such large motifs, a general search strategy should also be divided into different steps in order to achieve higher search efficiency.

SnoRNAs [LE99] have different elements of functional importance within one large loop region. Several sequence motifs occur within a specific distance from each other and a section of the loop pairs with a ribosomal RNA along a variable stretch of basepairs. It seems very hard to find snoRNAs with a general motif search program, since the secondary structure or these RNA motifs is not distinctive.

A program for searching tmRNAs, BRUCE [LBA02], is tailored to the specific secondary structure of tmRNAs which contain a tRNA-like domain and an unstructured mRNA-functional part. The algorithm focuses on the structured part starting with a specific sequence motif and aims to find the remaining secondary structure around this starting point. The mRNA-like domain is determined by another sequence motif that must be found in a range of up to around 500 bases from the first sequence motif. As this motif relies strongly on sequence motifs and in part does not exhibit characteristic structural features, a general motif search program should provide means to integrate these characteristics into the motif definition. A filtering step relying on the sequence motifs should be very beneficial as well.

The program tRNAscan-SE [LE97] for searching tRNAs is also based on covariance models. Here, a filtering step is used to restrict the search space making the program both reliable and efficient. Due to the characteristic secondary structure of a tRNA in combination with specific sequence motifs, it should be possible to define a good description of the motif within a general motif search program. Yet, in order to efficiently search for these motifs, filtering steps must be incorporated to restrict the search space in a similar way as tRNAscan-SE.

The miRseeker [LTWR03] program for detecting microRNAs in *Drosophila* is a multi-step filtering strategy relying on deep expert knowledge. First, conserved sequence regions between different *Drosophila* species are determined using a global alignment tool. Then, the conserved regions are folded with mfold and the results are compared to the expected structures for microRNAs. These are further filtered using divergence patterns for the two *Drosophila* species. Thus, the good results of this approach stem from the integration of expert knowledge on this particular class of RNA motifs restricted to *Drosophila*. It is unlikely that a general motif search program will produce comparable results without the same amount of expert knowledge integration into the search strategy.

Conclusively, in order to provide a general search program for these and other classes of RNA motifs, a great amount of flexibility must be provided. Optimally, the expert user should be able to specify which parts of the motif are of highest importance and thus determine a search order or a filtering phase before general folding procedures are undertaken. Especially sequence motifs and their spatial relation are promising candidates for such a filtering phase. Also, large motifs could be decomposed into different search programs whose results are then combined to determine the presence of the overall motif.

4.5 Discussion

It is difficult to make a general statement on the quality of one approach versus the other. The choice of which search program to use should be based on the available data.

The programs tailored for specific kinds of RNA generally produce good results as they are adapted to the best available knowledge on that kind of RNA to the date of their development. tRNAscan-SE e.g. works very well because the underlying covariance model is based on an input of more than 1000 different tRNA sequences producing a very reliable model.

For all those RNA motifs that lack specific search programs another approach must be chosen. Here, again, the choice depends on the available data. If the RNA belongs to a large RNA family that exhibits a characteristic structure, then a good covariance model can be generated producing reliable search results. On the other hand, if little data is available and not much is known about which parts of the motif are relevant for its function, then a single-sequence based approach such as RAGA or RSEARCH should be chosen. Finally, if not much data, but good knowledge on the motif is existent, then an expert should define the motif in a descriptor language (e.g. RNAMotif) and search based on the description. Yet, this requires learning the descriptor language which can be a hindrance for the biological expert. Here, we provide an improvement in allowing to draw the motif in a graphical language that can be understood intuitively by a biologist. Furthermore, the search strategy is based on current knowledge on the thermodynamics of RNA secondary structures with comparable efficiency to that of RNAMotif or Infernal. Our approach is presented in detail in Chapter 5.

Chapter 5

Locomotif System

In recent years many new noncoding RNAs have been found and many more are believed to exist. These are active molecules with a function of their own that depends both on their structure and some specific sequence motifs. Also, there are classical RNA motifs responsible for regulatory control mechanisms. Using the fact that these motifs are composed of typical structure parts such as stems, bulge loops or single stranded regions, we chose to develop a graphical editor for defining RNA motifs. The motifs should then be translated into a description used to search them. Having the ADP framework ready for RNA folding with the thermodynamic algebras already implemented, it is not difficult to imagine using this approach for searching RNA motifs. Instead of a general RNA folding grammar, we need particular grammars for every motif, but can rely on the same thermodynamic algebras.

My approach of visual definition of RNA motifs and subsequent generation of search programs is based on two attributes of RNA motifs:
First, they are tree-like structures with a root (the start of the underlying sequence, termed the “5’ end”) and corresponding substructures. This aspect facilitates both the storage of structures in the graphical editor as well as the translation into tree-like XML documents. And second, RNA motifs are composed of a limited set of “building blocks”, namely stems, bulges, internal loops, hairpin loops, multiloops (multifurcations) and single stranded regions. This allows us to construct any motif by placing the required building blocks next to each other.

Organization of this chapter I first give an overview of the architecture of the graphical programming system focusing on the client-server approach and the different programming techniques employed. Then, I give an idea of typical usage of the system, and present the major part of my work, the graphics frontend. I will talk about the techniques used for visualization and describe several aspects of implementing the system in detail. Afterwards, I will introduce the principles of translating a motif to XML and continue with the declarative grammars generated by the graphical editor. Finally, I will give an overview of the techniques for compiling them into executable C code. As the compiler is not part of my work, I will only briefly mention its function and refer to other publications [GMS04, SG06, GS06].

5.1 Overall system architecture

5.1.1 Logical decomposition

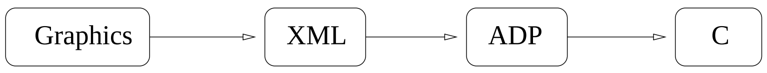


Figure 5.1: The graphical programming system is composed of 4 layers from graphics via XML and ADP code to an executable C-program.

The graphical programming system consists of four stages illustrated in Figure 5.1. The main part of the system and focus of my work is the graphical editor for molecular RNA motifs. It is implemented in Java, utilising Java Graphics2D for the visual components. A detailed introduction to the use and implementation of the graphical editor is given in Sections 5.2 and 5.3. The editor serves as the interface through which the user visually defines an RNA motif. The resulting view of an IRE motif construction with the editor is shown in Figure 5.2. The motif is then translated into an XML representation containing all biologically relevant information (see Figure 5.3). Subsequently, the XML code is translated into a declarative ADP program for the motif described in detail in Section 5.5. This translation is also implemented in Java. Finally, the ADP grammar is compiled to an executable C-program, the motif matcher, by the ADP compiler as summarized in Section 5.6. Using the motif matcher, new ocurrences of the RNA motif can be located in input RNA/DNA sequences.

5.1.2 Client - Server architecture

The four-tier composition of the programming system is devised to allow for an integration within a client-server architecture as shown in Figure 5.4. Offering the Locomotif system in a client-server setting has the advantage that we do not need to provide an installation

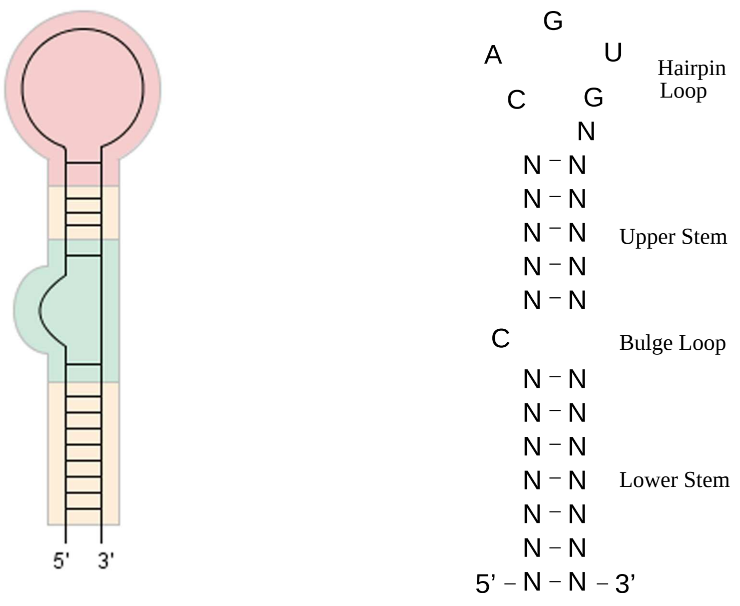


Figure 5.2: The visual representation of the IRE in the editor closely resembles the traditional plots that biologists are accustomed to.

of the compiler for different machines. Furthermore, an extension of the system to genome-wide scans as described in Chapter 7 requires a lot of storage and computational power which our server can provide.

The graphical editor (Locomotif) is accessible on our webserver, BiBiServ[1], as an application through Java Web Start. Running under any common browser and operating system, Java Web Start automatically checks for updates of the application and downloads all needed resources without requiring any installation procedures from the user.

The XML representation of the designed RNA motifs serves as a secure transport layer between Client (Graphical Editor) and Server. Using an XML schema, we can verify on the server side that only valid information is sent from the client to our server. This eliminates both the risk of subsequent compilation errors and any issues regarding the security of information content sent to the server. Eliminating this step and sending ADP code directly would pose a threat, since correctness cannot be checked automatically.

A unique identifier (ID) is returned to the client through which it can access the generated search tool for a certain time period (depending on the most recent access to the tool). The ID is stored during the current use of the programming system and can

[1]Bielefeld Bioinformatics Server, http://bibiserv.techfak.uni-bielefeld.de/{locomotif}

```xml
<?xml version="1.0" encoding="UTF-8"?>
<rnamotif xmlns="http://bibiserv.techfak.uni-bielefeld.de/xsd/20060306/rnamotif"
         searchtype="global">
  <neighbor>
    <stem allowinterrupt="false">
      <neighbor>
        <bulge length="1" bulge-pos="5prime">
          <neighbor>
            <stem length="3" allowinterrupt="false">
              <neighbor>
                <hairpinloop length="8">
                  <seqmotif>cagugn</seqmotif>
                </hairpinloop>
              </neighbor>
            </stem>
          </neighbor>
          <seqmotif>c</seqmotif>
        </bulge>
      </neighbor>
    </stem>
  </neighbor>
</rnamotif>
```

Figure 5.3: The XML document contains all relevant information on the motif structure.

also be saved in a file. Optionally, the user can choose to have this information sent to an email address together with a link to a submission web page.

The next step on the server side is the translation of the information stored in the XML document into declarative ADP code and the subsequent compilation to executable C-code.

Once compilation of the matcher is completed, the search program can be run through the editor where the user can input RNA sequences directly or upload FASTA-files together with the obtained ID. Alternatively, the matcher can be accessed via the submission web page. The search tool is found according to the unique ID and applied onto the given sequence. The result is then presented in an extra window of the editor or on a web page. The user can repeat this process on as many sequences as desired.

Web Services The server implementation is done using the Web Services technology provided by the BiBiServ framework. The Web Service client is written in Java using the AXIS library[2]. WSDL (Web Services Description Language) is used to define the methods available for the client to call. The Locomotif wsdl file[3] offers four different operations:

[2]http://ws.apache.org/axis

[3]http://bibiserv.techfak.uni-bielefeld.de/wsdl/Locomotif.wsdl

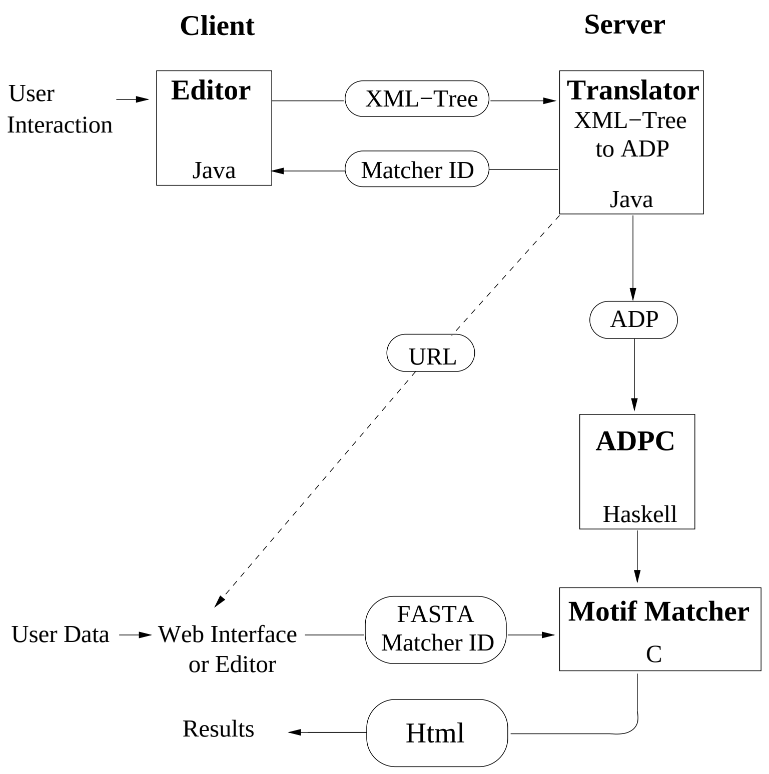


Figure 5.4: Overall structure of the graphical programming system. Rectangles are programs, ovals are data formats.

- `request` is used to obtain a BiBiServ ID and send the XML document to the server.
- `response` is given the ID to check whether the result is already available. In this case, if the compilation is finished, the answer is a success message.
- `run` is called to use a compiled motif matcher that must be specified by the BiBiServ matcher ID obtained via `request`. A new ID is returned for the matcher results.
- `matchresult` is given the result ID. It checks whether the run operation has terminated and upon success, receives the results.

Within Locomotif, the request operation can be used as soon as the motif definition is finished, i.e. only the open 5' and 3' ends remain. The XML information is stored in a DOM document (for details see Section 5.4) which is then sent to the server using the `request` method in a separate thread. Then, the `response` method is automatically called in another thread until it receives note that the compilation on the BiBiServ terminated successfully. This usually takes only a few seconds during which a progress frame is visible. When sending a sequence to the server, the `run` method is used in the same way to send data to the server (both sequence and matcher ID). The `matchresult` method is automatically called in a separate thread until the results of the matcher run are available. During this time another progress frame informs the user of the ongoing computation which may take some time depending on the motif and input size.

If the webpage is used for submitting the sequence and matcher ID, a similar perl script is employed to retrieve the specified information and send it to the server (using the same operations specified in the wsdl file).

5.2 Using the Locomotif System

The graphical editor provides the interface through which we specify RNA motifs by placing motif building blocks next to each other. After selecting the desired building block through buttons, it is attached to the mouse cursor. Figure 5.5 shows a snapshot of the editor during the movement of a building block of the IRE motif to its final location. Once dragged to the desired location, we can drop the building block onto the underlying canvas. If a motif structure is already available nearby, the subsequent building blocks snap into any correct position as if attracted in a magnetic fashion. Regarding the IRE, it takes only 4 drag-and-drop operations to construct the overall structure of the motif.

At any time in the process of building an RNA structure, we can open an editing interface by double-clicking on a building block. Here, we can add details such as the size of the building block or a sequence motif contained in it. In the case of the IRE, for 3 building blocks (hairpin loop, bulge and upper stem), the editing interfaces must be opened and the necessary information relevant to the motif added (see Figure 5.6). The internally stored data is shown via a tooltip when hovering over a building block with the mouse cursor. If desired we can also access the complete information on the RNA motif in a separate window (see Figure 5.7).

Several user IO and editing operations are available for making changes to the current view of the RNA motif. These include standard file IO modes such as creating a new projects, saving, loading and restoring it. Export functionality is provided for several image formats. The current RNA structure in the editor can be rotated, moved or zoomed in and out. Individual building blocks can be detached or deleted from the structure and the orientation (5'-3') of the RNA motif can be changed.

As soon as the motif construction is finished, i.e. the only remaining open ends are the 5' and the 3' end, translation into XML is possible. First though, we have to specify

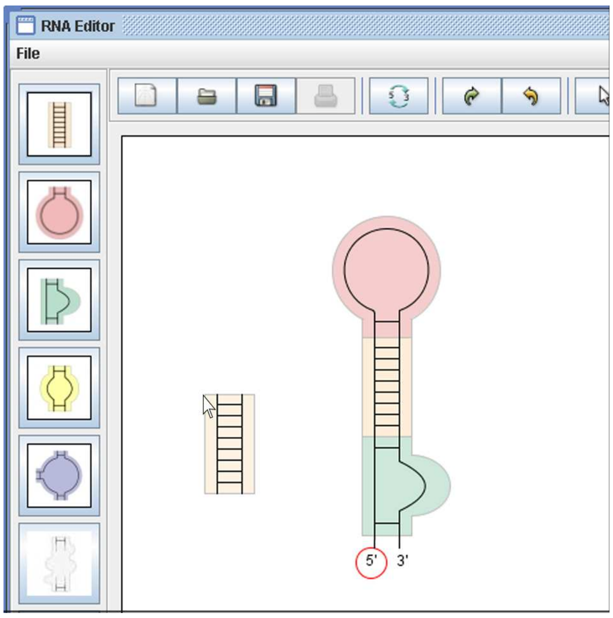


Figure 5.5: The lower stem element is attached to the mouse cursor and will snap into the correct position once in proximity of the open end.

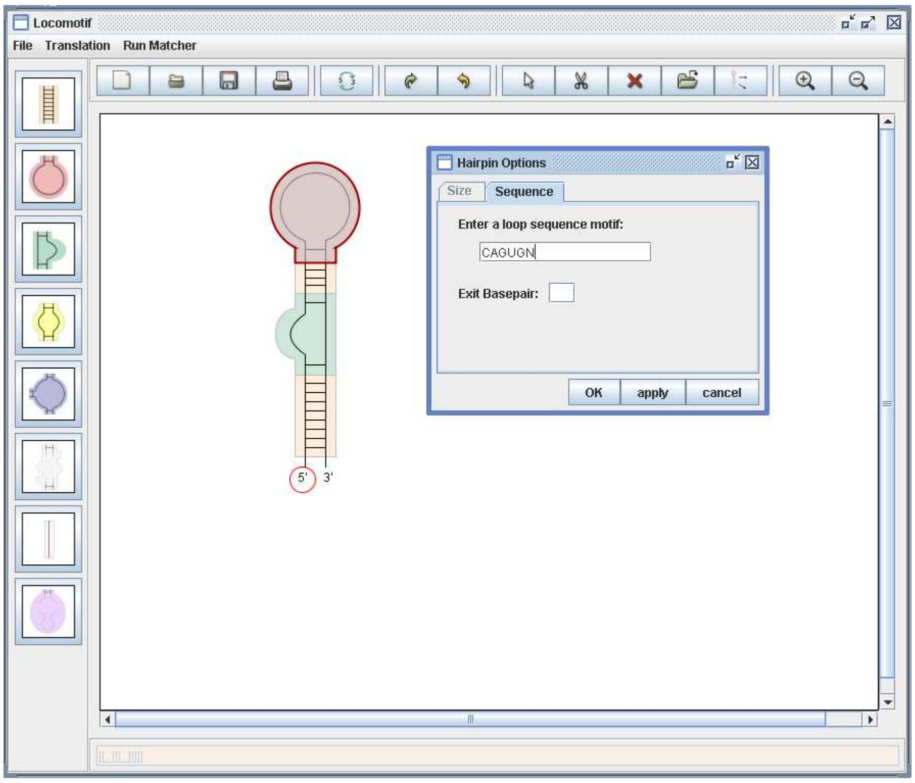
some global search parameters by clicking on the red circle around the 5' end indicating the "motif head". A small user interface opens where we must specify a project name and whether we are interested in a global or a local search. Here, a global search refers to finding the motif within a larger sequence whereas a local search aims to fold the total given sequence into the specified motif structure.

Then, the XML document can be sent to the server where it is translated into ADP and compiled. Afterwards, we choose one or more RNA sequences for localizing the motif with the generated program. The results are presented in form of the annotation strings introduced in Chapter 2 (Figure 5.8).

Typically, we will compile the motif, perform some test runs, and return to the editor to refine the motif.

5.3 Implementation of the Motif Editor

The motif editor is implemented in Java 1.5 relying mostly on the java.awt.geom package for the visualization methods. First, I introduce some terminology used in this Section and give an overview of the general framework of the editor software. Then, I briefly address the background of the visualization techniques employed before going into more detail on



Enter a loop sequence motif: CAGUGN

Exit Basepair:

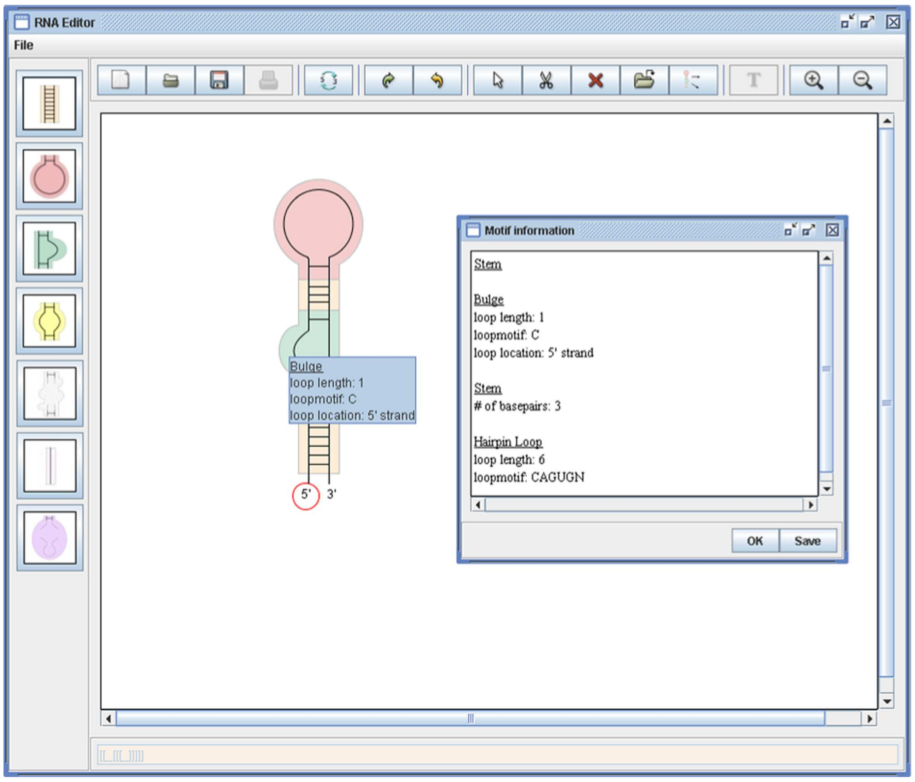
Figure 5.6: Sequence and size information can be added via graphical user interfaces.

the attributes and technology of the building blocks. Afterwards, I describe the handling of user interactions and account for the semantic integrity of the motif descriptions. Further, I explain the methods used for traversals through the motif structure and introduce the online shape strings included in the editor. I finish this Section with a few words on project maintenance within the system.

5.3.1 Building Blocks and their Shapes

In the description of the graphical editor, I rely on some terminology to refer to the individual components of the graphics and implementation details. Italics are used to indicate a Java class of the same name.

The "building blocks" introduced in Chapter 3 are the distinctive elements an RNA motif is composed of. The words "building block" are used as general concepts, i.e. not in relation to implementation details. The graphical view of these building blocks is termed *RnaShape* in relation to the name of the corresponding Java class. Figure 5.9 shows



Bulge
loop length: 1
loopmotif: C
loop location: 5' strand

Stem

Bulge
loop length: 1
loopmotif: C
loop location: 5' strand

Stem
# of basepairs: 3

Hairpin Loop
loop length: 6
loopmotif: CAGUGN

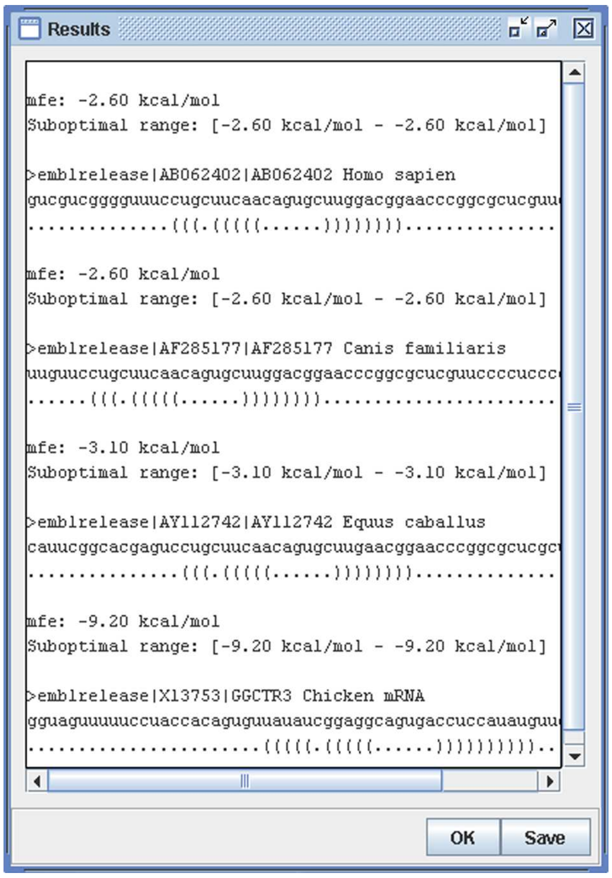
Figure 5.7: Information on the building blocks is shown either via a tooltip when hovering over the building block or in an editable window upon request.

the graphical views of the building blocks and Figure 5.10 contains the structure subtree templates that each of them represents. I use the term *BuildingBlock* when referring specifically to the data structures responsible for storing information on the properties of the building blocks.

Graphically, every *RnaShape* can be connected to other *RnaShape*s at its open ends. When describing the implementation of the *RnaShape*, I refer to these ends as the "exits" of the *RnaShape*.

5.3.2 Framework of the motif editor

The main class of the editor, the *EditorGui*, is based on javax.swing. It provides the buttons and menu, the framework for the graphics, and event handling for the translation to different languages (implemented in *Translator*) and the client-server communication (implemented in *RNAEditorClient*). The *DrawingSurface* embedded in the *EditorGui* is the principal component of the visualization. Extending a *JPanel*, it keeps track of all



```
mfe: -2.60 kcal/mol
Suboptimal range: [-2.60 kcal/mol - -2.60 kcal/mol]

>emblrelease|AB062402|AB062402 Homo sapien
gucgucgggguuuccugcuucaacagugcuuggacggaacccggcgcucguu
..............(((.(((((......))))))))...............

mfe: -2.60 kcal/mol
Suboptimal range: [-2.60 kcal/mol - -2.60 kcal/mol]

>emblrelease|AF285177|AF285177 Canis familiaris
uuguuccugcuucaacagugcuuggacggaacccggcgcucguuccccuccc
......(((.(((((......))))))))......................

mfe: -3.10 kcal/mol
Suboptimal range: [-3.10 kcal/mol - -3.10 kcal/mol]

>emblrelease|AY112742|AY112742 Equus caballus
cauucggcacgaguccugcuucaacagugcuugaacggaacccggcgcucgc
...............(((.(((((......))))))))..............

mfe: -9.20 kcal/mol
Suboptimal range: [-9.20 kcal/mol - -9.20 kcal/mol]

>emblrelease|X13753|GGCTR3 Chicken mRNA
gguaguuuuuccuaccacaguguuauaucggaggcagugaccuccauauguu
.......................(((((.(((((......))))))))))..
```

Figure 5.8: For each sequence, the location of the motif (if found) is given by the annotation string: `.` at the beginning and end are bases framing the motif, the first `(` and the last `)` are the first basepair of the motif structure and thus the start and end position of the motif within the sequence.

relevant information of the graphics using *MouseListeners* to capture user interaction. The class has a number of essential variables for handling the visualization of the current element attached to the mouse cursor, a selected element, and the overall motif structure. Furthermore, it stores references to the *RnaShape* of the start and the end point of the motif needed to start a traversal over the entire RNA motif structure. Additionally, it holds a Vector-based data structure, *FreeMagnets*, that stores all open ends of the RNA motif, the *Magnets*, for possible sites of building block addition. The RNA motif itself is stored within another Vector-based data structure, the *RnaStructure*. It provides traversal methods over the *RnaShapes* as well as information on the overall area of the RNA motif. For Vector-based data structures, generic data types are employed to ensure that only

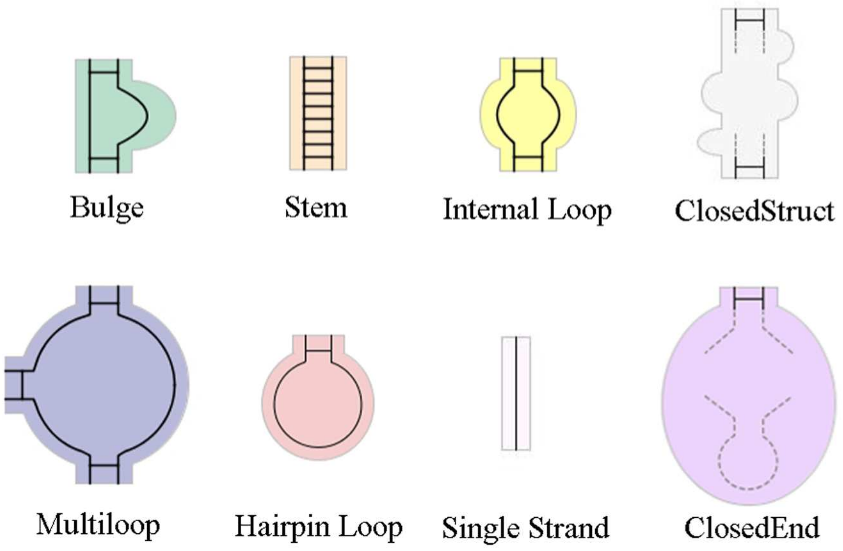


Figure 5.9: An RNA motif can be composed of 6 different kind of building blocks. Additionally, a ClosedStruct represents any number of bulges, stems and/or internal loops and a ClosedEnd stands for any closed structure part without single strands.

correct data is stored in these Vectors and to minimize the need for casting objects.

For every building block, a subclass of *BuildingBlock* is used to store the information relevant for that building block. Also, an implementing class of the abstract class *RnaShape* is available for showing each building block in the *DrawingSurface*. Finally, an editing interface, i.e. an extended *JFrame* is provided for every type of building block. For the Multiloop, the editing interface contains three *JPanel*s for selecting the exits (*SelectorPanel*), the sizes (*SizeSelectPanel*) and the sequence motifs (*SeqSelectPanel*).

The code generation parts of the program rely on a *Translator* class responsible for translating the graphics into XML. Any further translations to ADP or Html are based on the XML trees. All Client - Server communication is implemented within the *RNAEditorClient*. All file IO is contained in the *EditorIO* class. Saving and loading information from files is based on *ObjectOutput/InputStreams* realized via implementations of the *Serializable* interface in all classes that contain data which must be stored. File filters are used to restrict the interfaces of the *JFileChooser*, e.g. an *ADPFilter*, a *MatcherFilter* or an *RNAFilter* for storing the overall project.

At the bottom of the user interface a small panel is integrated that shows an online

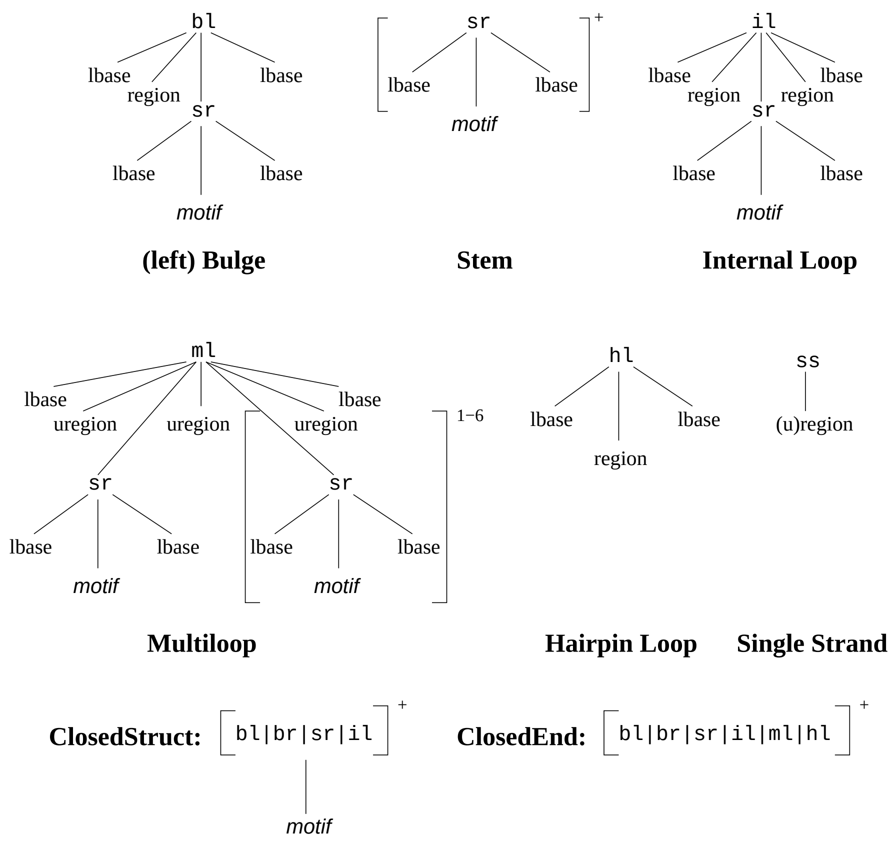


Figure 5.10: The tree representations of the building blocks are the templates used for translation of the motif parts into the ADP grammar. Each tree is part of the overall motif tree that can be obtained by replacing the *motif* nonterminals with the root of the next subtree in the order of the motif. Two versions of a bulge exist depending on the location of the loop region. (Here, only the left bulge is shown).

translation into an adapted abstract shapes string ([GVR04, SVR$^+$06]). The translation is based on methods implemented in the *Translator* and is updated whenever changes are made to the motif structure.

A *RunPanel* is provided for calling a generated matcher. Via this user interface, sequences can be specified directly or loaded from a file and the matcher ID can be chosen. During the compilation of a matcher or the search phase, a *ProgressFrame* informs the user about the ongoing tasks. The results are presented in an additional *ResultPresenter* class extending a *JFrame*.

Finally, some helper classes are needed to take care of storing *Basepair*s and ensuring the usage of *Iupac* symbols.

5.3.3 Visualization technology

The visualization methods are based upon Java's *Graphics2D* class of the java.awt package. Within the *paintComponent* method of the *DrawingSurface*, a *Graphics2D* object is used to *draw* or *fill* the geometric components according to its settings (e.g. color, stroke).

The *RnaStructure* has a *drawStructure* method which obtains the *Graphics2D* object of the *DrawingSurface.* It is included within the *paintComponent* method which is automatically called whenever the view changes. It iterates over all building blocks of the structure, calling their own *show* methods with the original *Graphics2D* object as a parameter.

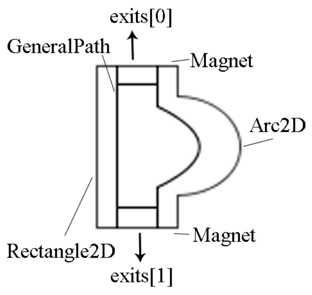


Figure 5.11: Each *RnaShape* has a number of variables responsible for the view of the *RnaShape* and its location in the motif structure tree.

For each building block, a rough visual outline represents its type and a fine plot resembles the known graphical representations for the motifs that biologists are accustomed to (see Figure 5.11). For the outline a combination of *Rectangle2D* and *Arc2D* or *Ellipse2D* depending on the type of building block is stored in an *Area.* The fine plot showing the sequence strand(s) of the building block is done with a *GeneralPath* object. Within the show method of each *RnaShape*, the *Graphics2d* object *g2* is used by simply calling methods to draw and fill the components:

```
//prints the background color of the RnaShape
g2.setPaint(color);
g2.fill(area);

//draws the outline around the RnaShape on top of the background
g2.setPaint(Color.lightGray);
g2.draw(area);

//draws the fine plot on top of the background and the outline
g2.setPaint(Color.black);
g2.draw(rnapath);
```

The order of the function calls is important, since any further drawing in the same place on the screen will overwrite previous ones unless transparent. If the *RnaShape* is the start or end element of the structure, the 5' and/or 3' end tags and the "motif head" circle are added to the building block.

5.3.4 Motif Building Blocks

There are three different levels for each type of building block. The first level is comprised of an editing interface for every type of building block providing means to control its properties. The second is an internal data level for storing information on the building blocks. The third level is responsible for the visual aspects of the building blocks. This division is similar to a Model-View-Controller (MVC) architecture [Ree79] which is based on the separation of concerns regarding the visualization from those restricted to storing the relevant information. The model contains data structures to store the information that is to be presented. In our case, the *BuildingBlock* and its implementing classes keep track of everything relevant to the biologist. The information stored in the *BuildingBlock* classes is later used to replace the nonterminals in the template trees (Figure 5.10) with terminal values. The view of this data is realized with the *RnaShape* that presents the stored information to the user (see Figure 5.9). Here though, the overall motif structure is stored via the exit references of the *RnaShape* (see Figure 5.11), i.e. the classes responsible for the view also fulfill some modeling properties. It might be possible to separate these properties from the view and store them in the model, leading to a true MVC architecture, but I do not plan on establishing this separation since the exit references are widely used within the *RnaShape* classes. Finally, the controller is found in the editing interface through which the user interacts with the view and inputs information stored in the model. Yet, direct interaction with the view is enabled as well, as this allows us to construct the secondary structures of the motif. The chosen architecture makes it very easy to add changes to the individual building blocks and keeps the overall system quite accessible for a programmer.

Attributes of the Building Blocks

Each building block has several attributes that can be specified by the user. This includes both local and global size restrictions or information on the sequence of the strand(s) contained in the building block. Local size information is visualized for stems (the number of pairing bases are shown) and for bulges, internal loops and hairpin loops (the size of the loop segment). Global size restrictions are imposed on the entire substructure rooted at the building block with the restriction.

BuildingBlock Every *BuildingBlock* stores information on its orientation indicating whether the default 5'-3' direction is maintained or changed by the user. Additionally, it keeps track of general length requirements that can either be unrestricted, an exact length or a minimum and/or maximum length. Other variables store a minimum and/or maximum global length restriction. In addition to these, the subclasses have more refined variables which are described in the following pages.

RnaShape The four "standard" building blocks (stem, bulge loop, internal loop and ClosedStruct) possess two open ends. While the data level of these building blocks is different, the methods for visualization and user interaction are to a large extent the same. Therefore, the methods for those four building blocks are implemented in the abstract class *RnaShape* and overwritten in the implementing classes of the multiloop having 3-8 open ends, the hairpin loop and ClosedEnd loop with only one open end and the single strand that connects or extends different motif parts.

Each *RnaShape* has a number of variables illustrated graphically in Figure 5.11. The *Magnets* describe the location of the open end(s) of the *RnaShape*. They are used when attaching an *RnaShape* to the existing motif structure. Also, each building block has a reference to its neighbors (1,2, or 3-8) stored as "exits". Another important variable of each *RnaShape* is the *AffineTransform* which is used to tell the *Graphics2D* object responsible for showing the *RnaShape* how to draw it. I will go into further detail on the functions of the *AffineTransform* object in Section 5.3.6.

The angle *theta* stores the degree of rotation of this building block. By default, the value of *theta* is 0. If a rotation is performed upon the building block (or the overall *RnaStructure*), *theta* is adjusted according to the degree of rotation. For all methods that require interactions with the exits of the building block, *theta* is needed. It is used to tell on which side of the building block an addition or removal is made. Also, during a traversal, *theta* is used to determine the angle through which a building block is entered (i.e. the appropriate entrance point and thus the nearest exit in case of the multiloop).

For the visualization of the *RnaShape* four basic coordinates are stored: *xloc*, *yloc*, *width* and *height*. These are used as the parameters for the *Rectangle2D* and *Arc2D/Ellipse2D* that make up the outer structure of the building block. Most implementations of the *RnaShape* have additional variables for visualizing the particular building block. The overall outline of the building block is stored as an *Area* while the fine plot is stored as a *GeneralPath*. The *xloc* and *yloc* parameters are coordinates within the *DrawingSurface* and all other parameters are used in relation to those two (see Figure 5.12).

The location of the 5'/3' end is calculated within the show method based upon the

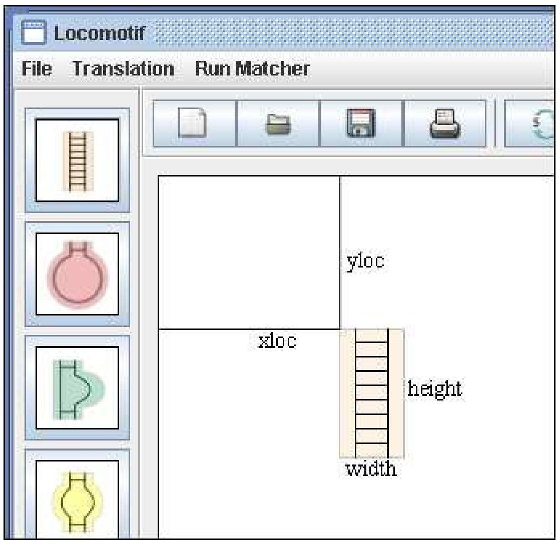


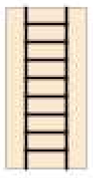
Figure 5.12: A building block is stored by its coordinates within the *DrawingSurface*.

main coordinates and the internally stored information on the angle of the start point, i.e. the available open exit. Overall, it depends on three things: the start/endangle, the presence and exact location of a single strand at that exits and the main coordinates:

```
//it is the start element: draw the 5’ end
if(isstartelement){

  //on which side of the building block is the 5’ end located
  if(startangle == ((theta+offsetangle)%360)){

    //if there is a building block attached to this one,
    //it must be a single strand shape
    if(exits[0] != null){

      //find out which strand the single shape is attached to and
      //write the 5’ to the appropriate location
      ...
        g2.drawString("5’",(float)xloc+28,(float)yloc-15);
      ...
    }

    //nothing is attached to the exit, i.e. it contains 5’ and 3’ end
    else{
```

```
//depending on the orientation, the 5’ end can be located
//on either side of the exit
if(standardorientation){
  g2.drawString("5’",(float)xloc+28,(float)yloc-15);
  ...
```

Stem A stem or stacking region is basically a sequence of basepairs whereas all other building blocks contain loop regions (the single strand being a loop itself). This significant structural difference of stems is visualized in the graphics by showing the individual basepairs of the stem. If its size is large, dots indicate the presence of even more basepairs. If no size restriction is given, a standard number of 8 basepairs is shown, but this value is not included in the XML output. The user can select an exact number or a minimum and/or maximum number of basepairs. Optional basepairs (i.e. if a maximum is given) are shown as dashed lines in the graphics.



A stem can either store a sequence of specific basepairs or a sequence motif on one of its strands. In both cases the start of the sequence must coincide with the start of the building block. This need arises from the ADP grammars where every base is enumerated explicitly. Several alternative rules for every possible beginning of the sequence motif could be provided, but for now, I chose to restrict the use of sequence motifs. A different location of the motif can still be achieved by placing three stems next to each other: the first representing all bases up to the motif, the second for the motif itself and the third containing all bases following the motif.

Furthermore, the user can specify whether the stem is continuous or if it might be interrupted. The default value is a continuous stem with an uninterrupted sequence of basepairs. Yet, in nature, stacking regions in RNA motifs are sometimes interrupted by loops of 1 or 2 bases. As long as no restrictions are imposed on the stem, i.e. size restrictions or sequence motifs, the user can choose to regard the stem as discontinuous. Then, small loops of a maximum of 2 bases on one strand or both strands are allowed within the stem building block. The fact that discontinuous stems cannot contain sequence motifs or size restrictions is again due to the ADP grammars as described in Section 5.5.

The outline of the stem is a simple *Rectangle2D* which is based on the four basic coordinates. The strands of the fine plot are done with the *rnapath* and an additional *GeneralPath* is employed to draw the basepairs.

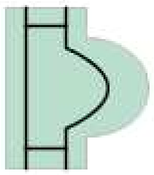
The stem is the only building block whose *height* changes according to size changes.

So if the user enlarges or reduces the stem, not only does its size change, but its neighbors must be moved as well. This shifting move is described in Section 5.3.6.

Bulge Loop A bulge consists of an opening and a closing basepair and a loop region. In addition to the standard size variables, the location of the loop is stored in the *Bulge* building block. A motif can be specified for the loop. Here, the location is flexible as the ADP grammar does not require a fixed `lbase` position, but rather a general `region`. Also, the bases of the two basepairs can be restricted.

Visually, the *BulgeShape* consists of the basic *Rectangle2D* with an added *Arc2D* on the side of the loop region. Therefore, it includes additional variables for the location and size of the *Arc2D* that are adjusted according to size specifications.

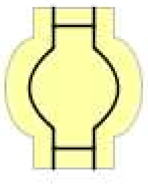
For the bulge loop, a size change is reflected only in the extent of the bulge, i.e. the *Arc2D*, since the amount of basepairs included in the building block is constant. The *GeneralPath rnapath* represents the two strands, one is a straight line and the other a curve. For the loop segments instead of line commands, quad curves are used:



```
//draw the straight line
rnapath.moveTo(x + (w/4), y);
rnapath.lineTo(x + (w/4), y + h);
//draw the loop
rnapath.moveTo(x + w - (w/4), y + h);
rnapath.lineTo(x + w - (w/4), y + h - (us+5));
rnapath.quadTo(x - (lw - 80)/2 - 1 + lw, y + us + cp/2,
               x + w - (w/4), y + (us+5));
rnapath.lineTo(x + w - (w/4), y);
//draw both basepairs
rnapath.moveTo(x + (w/4), y + us/2 + 2);
rnapath.lineTo(x + w - (w/4), y + us/2 + 2);
rnapath.moveTo(x + (w/4), y + h - us/2 - 2);
rnapath.lineTo(x + w - (w/4), y + h - us/2 - 2);
```

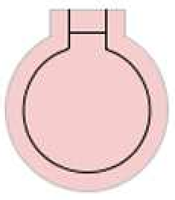
Since the location of the bulge can be changed by the user, the graphical representation must take this into account. Additional parameters are needed to determine the location of the bulge in accordance with the current orientation of the structure. If the user chooses to change the orientation, it will also have an effect on the bulge as the loop is now located on the other strand, while the visualization remains unaltered.

Internal Loop An internal loop is basically a bulge loop with an additional loop region, yet it is more complicated to store and maintain.



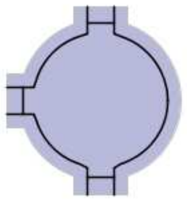
Additional size variables are needed to incorporate information for both flexible regions. The same holds for sequence motifs present in either of the two strands. Any size change is reflected in the appropriate loop by changing the extent of the *Arc2D* and the corresponding *GeneralPath*. In order to determine which strand is the 5' and which is the 3' strand, both the location of the start element with respect to the internal loop and the overall orientation of the structure must be taken into account. Visualization is done with the basic *Rectangle2D* and two additional *Arc2D*s.

Hairpin Loop A hairpin loop is different from other building blocks as it is the end of a structure part. It has only one exit and thus only one basepair. The loop varies in size with a minimum of 3 bases due to requirements of the energy rules for RNA secondary structures.



The visualization is slightly different as the large basic *Rectangle2D* is missing. Instead, a much smaller *Rectangle2D* is shown for the exit area requiring an additional *rheight* variable. The loop region is formed by an *Ellipse2D*. Instead of the exit array, an *RnaShape exit* is employed. The fine plot is also somewhat different: while the strands within the exit region are done in the traditional way, the loop region is realized via an *Arc2D* that is appended to the *GeneralPath* at the appropriate position. Since the hairpin contains only one exit, many methods of the *RnaShape* class are overwritten in *HairpinShape*.

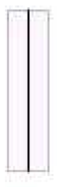
Multiloop Storage of the multiloop requires a completely different set of variables than all other building blocks. The key feature of a multiloop is its diverse number of possible exits: anywhere from 3 to 8. In order to be flexible about the number of exits at all times, I chose to represent them via arrays of size 8 for all types of variables: exact lengths, minimum and maximum lengths, sequence motifs and basepairs. No biological examples with more than 8 stems protruding from a multiloop are known, so an upper limit of 8 should be sufficient for all cases. The values are set to -1 or null, if no exit is available at the particular location. An additional array is used to indicate the presence of an exit. The indices of the arrays are arranged to store data for a standard degree of rotation and no readjustment is made when a rotation of the structure occurs. The angle theta is used to determine the correct index of any of those arrays when needed.



On the visual side, the multiloop is similar to the hairpin loop: an *Ellipse2D* forms the basic element of the building block. Then, a small *Rectangle2D* is created for the top open exit (using the same variables as the hairpin loop). For every available exit, this *Rectangle2D* is rotated around the center of the *Ellipse2D* using an *AffineTransform* (see Section 5.3.6) to create the other exits. Therefore, for the multiloop, we only need to know the exact coordinates of one exit in relation to the main circle and can obtain all other exits from it. For the *rnapath* several instances of the exit strand lines (including the basepair) and the *Arc2D* loop are rotated and appended to the *GeneralPath*. Nearly all methods of the *RnaShape* are overwritten in the *MultiShape* in order to incorporate the different composition of this building block.

Single Strand The single strand *BuildingBlock* has no additional variables to the super class except for a sequence motif that can be stored. The *SingleShape* though is quite different from its superclass, the *RnaShape*.

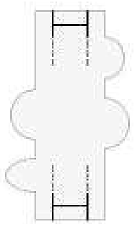
A single strand does not have a fixed shape, but can be drawn by the user with a great degree of freedom. The basic variables are needed for its initial view, a slim *Rectangle2D*. Once attached to the structure, only the *rnapath* is still shown which is realized by redrawing the *GeneralPath* according to the current location of the mouse cursor. Upon a mouse click, the final shape is stored with the *rnapath* ending at the mouse cursor position. The outer shape is then drawn around the *rnapath* using another *GeneralPath* object as the parameter for the *Area*.



Only straight lines are used since it would require extensive calculations to determine the optimal path for the single stranded region. This can lead to problems if start and end point are on the same horizontal line. Then, the outline of the single strand will be nearly invisible and it is very hard to click into this area to open the editing interface.

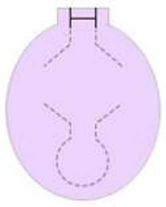
If a single strand connects two structure part, this connection is an entirely single stranded section of the underlying sequence. Yet, it is possible to allow foldings within the connection that can contain additional motif structure parts. Then, the connecting line is not drawn as a straight line, but dashed.

Compound Blocks The *ClosedStruct* does not have any additional variables, since the user cannot restrict its size or input a sequence motif.



The overall view of the *ClosedStructShape* is also constant as it just indicates a stacking region interrupted by looped regions. In order to draw the shape, several additional variables are necessary to determine the location of the *Arc2D*s and their sizes. The basic structure is again a *Rectangle2D*. The *GeneralPath* of the ClosedStruct is interrupted to indicate the fact that the building block is a placeholder for a stacking region that can contain any number of loops of various sizes.

Another compound block is the *ClosedEnd* which corresponds to an entire closed structure part.



It does not only contain a stacking region with interruptions, but can also include multiloops and hairpin loops. Actually, it is a complete motif part on its own ending in hairpin loops at every branch. The only building block excluded from the *ClosedEnd* is the single strand. Again, this building block does not have additional variables, since only global size restrictions can be imposed upon the building block. Its visual outline is similar to a hairpin loop consisting of a large *Ellipse2D* and a small exit *Rectangle2D*. The *GeneralPath* was designed to indicate that the structure part is closed, i.e. it ends in a hairpin loop, but otherwise both strands can fold into any type of motif, i.e. they diverge and reconnect.

5.3.5 From Building Blocks to the RNA structure

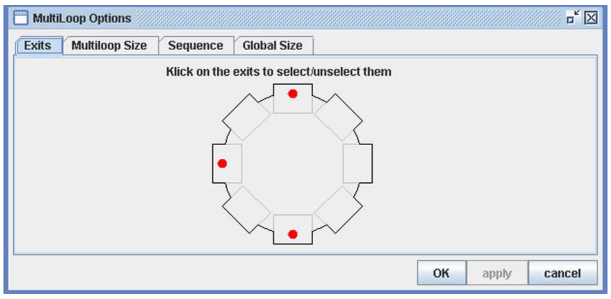
All *RnaShape*s are stored in the Vector-based data structure *RnaStructure*. It does not impose an order on the individual building block, but simply holds a reference to each of them and keeps track of the overall *Area* of the motif. Using a *Vector*, adding or removing elements from the *RnaStructure* is straightforward and does not require any manual reordering procedures.

The true structure of the motif is obtained through references of the building blocks to all their neighbors. Thus, the motif is internally as a double-linked list with a particular start element (the 5' end). Using the start element as the root of a then tree-like structure, traversals can be performed.

When two building blocks are attached to each other, it represents a connection of their sequence strands. Hence, the addition of the hairpin loop closes a motif part by connecting the open strand ends of its neighbor. In the end, only the start and the end point remain open and are connected to each other through all building blocks of the structure.

A little more sophisticated than the standard building blocks with two open ends and the hairpin loop or ClosedEnd building block are both the single strand and the multiloop.

Multiloop A multiloop must incorporate a variable number of exits at locations chosen through a visual interface by the user (see Figure 5.13). Open exits are indicated by red dots, those with building blocks attached with gray dots. The user can remove any exit by clicking on a red dot or add a new exit by clicking in an empty rectangle.



Klick on the exits to select/unselect them

Figure 5.13: The location of the exits of a multiloop can be adjusted via a visual interface.

Single Strands Single stranded regions and their representation as *SingleShapes* do not possess the same double-stranded open ends as the other building blocks. If a single strand is added to either end, one strand of the neighbor is extended. The other, if not already taken by another single strand, remains open and can only be connected to another single strand. The neighboring building blocks do not store the *SingleShape* directly, but rather have a reference to a *DoubleSingleShape*. It keeps track of both strands that can either hold a single strand or otherwise contain a null reference. Refer to Figure 5.14 for an example. That way, I can use the standard two-neighbor (1 for hairpin loop, 3-8 for multiloop) storage for single strand connections. The *SingleShape* itself also has two exits that keep track of which one is closer to the 3' or to the 5' end of the motif.

Addition or removal of single strands is problematic as it could disrupt the entire motif structure by placing the 5' and 3' end on distinct structure parts. Therefore, I chose to impose limits on the usage of the single strands. It can only be added to the 5' or 3' end and extend these or connect them to other structure parts. Removals of other building blocks can only be made once all single strands have been deleted. While these restrictions can be somewhat bothersome, it is difficult to ensure correctness of the structures otherwise. I discuss the problems of using single strands and potential solutions in Chapter 7.

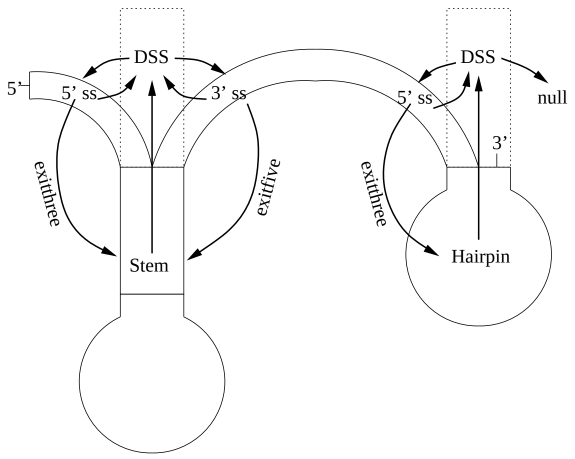


Figure 5.14: The stem/hairpin stores a reference (indicated as arrows) on a *DoubleSingleShape* which keeps track of both single strands connected to the stem/hairpin. The single strands store both the neighboring building blocks and the *DoubleSingleShape*.

5.3.6 Handling user interactions

Connecting building blocks

The connection of two building blocks is based on the afore mentioned *Magnet*s. A *Magnet* stores the *Line2D* of the open end, i.e. the exact location where another building block can be attached. Additionally, it stores an angle indicating the side of the building block where the *Line2D* is located, one of 8 different faces from 0,45,... to 315 degrees. A building block can be attached if one of its exits has the same angle as the *Magnet* or the main exit (*theta*) is rotated until it fits to the *Magnet* (see Figure 5.15). Finally, each *Magnet* has a reference on its parent, i.e. the building block it belongs to. Using this information, if a new building block element is in proximity of a *Magnet*, the exact location that the building block must be moved to for attachment to the existing structure can be calculated. The *Line2D* has the x and y coordinates while the angle tells us wether the building block must be rotated for proper fitting. Whenever the building block is in proximity of a *Magnet*, the `snap` function is called. It moves and rotates the building block to the appropriate position based on its coordinates and *theta* angle (cases A-C refer to Figure 5.15:

```
public void snapTo(Magnet m){
   Line2D.Double line = m.getLine();
   //start point of the line
```

```
    double x1 = line.getX1();
    double y1 = line.getY1();
    //end point of the line
    double x2 = line.getX2();
    double y2 = line.getY2();

    //case A
    if(m.getAngle()==theta){
      changeLocation(x2,y2,false);
    }

    //case B
    else if(m.getAngle()==((theta+offsetangle)%360)){
      changeLocation(x1,y1-height,true);
    }

    //case C
    else if(currentrotation == 0){
      //the top of the building block is attached to the end of the
      //Magnet's line
      changeLocation(x2,y2);
      //then it is rotated to fit to the Magnet using the
      //AffineTransform of the building block
      at.rotate(StrictMath.toRadians(m.getAngle()),xloc,yloc);
      rotations.add(new Rotation(m.getAngle(),1));
      theta = StrictMath.abs(m.getAngle());
      adjustMagnets();
    }
  }
```

Using the parent of the *Magnet*, the new building block is added to the structure by creating new neighbor references. Then, the *Magnet* is removed from the *FreeMagnets*. If a building block is detached or deleted from the structure, the neighbor references are set to null and the *FreeMagnets* are updated accordingly.

In order to incorporate single strands, *Magnets* have a boolean flag describing whether any kind of building block or only single strands can be added to the *Magnet*. The addition of a single strand is somewhat more complicated as it can be attached to two points at a standard *Magnet*. Using the *Area* of the single strand, it can be determined on which side of the *Magnet* it is located. Then, the *Magnet* is removed from the *FreeMagnets* and only the remaining half is added again. Only single strands can be attached to a half *Magnet*,

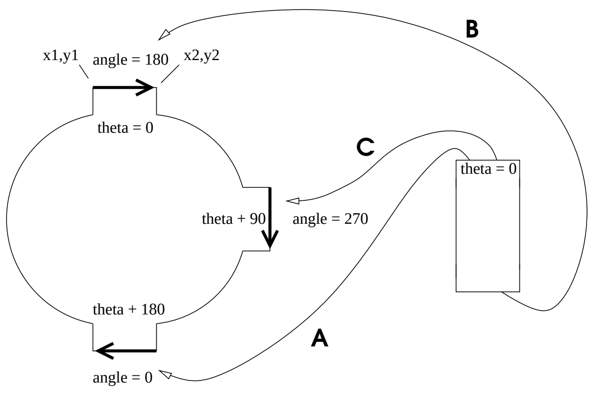


Figure 5.15: Each `RnaShape` has an orientation based on `theta`. All other exits are stored as `theta+offsetangle`: for standard building blocks, the offsetangle is 180 degrees. If the face of an exit is the same as the angle of a `Magnet`, the building block can be attached to it (cases A and B). Otherwise, it must be rotated: in case C, the building block is rotated by 270 degrees.

so for every building block addition this parameter must be checked. Furthermore, to prohibit circular structures (i.e. those with no open ends left), every *Magnet* also has a boolean flag indicating whether a hairpin loop or ClosedEnd can access it.

Updating the start element If a building block is added to the 5' end (or 3' end) of the structure, a new start or end element must be determined. For standard building blocks with two open ends, this is straightforward: it is simply its remaining open exit.

In case of a multiloop addition, the nearest exit is searched within the multiloop according to the current orientation of the structure. In order to minimize any disruption to the existing motif structure, it is always chosen by following the strand connected to the previous 5' end to its end, i.e. the 5' end is transported to the multiloop. This way, the closing basepair of the multiloop is the start (and end) element and the previous motif structure is contained in the first arm of the multiloop (notation is based on shape strings as explained in Section 5.3.9):

```
[[[[_]]_]] -> [_[[[[[_]]_]]]_[ R ]_]
```

A hairpin loop or ClosedEnd closes a structure part. In this case a traversal upon the structure is necessary to find a new open end. Again, this is done in a fashion to minimize reordering of the motif structure. The traversal follows the 5' strand to its end, first entering and exiting the hairpin, and then, it continues until an open end is found.

In a multiloop, again, the nearest exit is chosen according to the entry point and the orientation of the structure. Hereby, only some parts of the motif, i.e. those between the new start and the multiloop (shown in red), are added around the overall structure. Those elements belonging to the previous start element up to the multiloop are now placed in the appropriate branch (in blue) and all branches in between the old and the new start exit of the multiloop are shifted towards the 3' end (shown in gray):

```
[[_[[[_[[ R ]]]]]_[[_[ R ]_]]_]] -> [_[[_[[[_]]]_[[[_[[ R ]]]]]_]]_]
```

During the traversal to the new start element, the building blocks that are passed switch sides (the previous 5' strand is now the 3' strand and vice versa). Sequence motifs or basepairs must be switched and for bulges or internal loops, the loop regions are exchanged.

If a single strand is added to the 5' end and connected to another structure part, this entire part is prepended to the overall structure. If the single strand connects the 3' with another structure part, this part is appended to the overall structure. As the previously distinct structure part already had an orientation, this might be disrupted by the single strand connection. In this case, sequence motifs and loops of this structure part must be switched as well.

Transforming the structure: *AffineTransform*

A great advantage to the concept of interactive visualization is the Java built-in class *AffineTransform* of the java.awt.geom package. It is responsible for calculating linear coordinate transformations. Several transformation can be appended such that an *Affine-Transform* object can represent a sequence of translations, scales and rotations. They are used to incorporate user interactions such as moving the structure around, zooming in and out or rotating either the structure or the element attached to the mouse cursor. In fact, internally, the *RnaShape*s are always based on the same coordinates, yet when shown, the painting *Graphics2D* object is transformed by the associated *AffineTransform* of the *RnaShape*. The different user interactions performed via affine transformations are the following:

Zooming Zooming is managed in the *DrawingSurface* which stores the current zoom-factor. Whenever the mouse is moved within the *JPanel*, the *paintComponent* method of the class is invoked. Within this method, the *Graphics2D* object is transformed according to the zoomfactor. As the same *Graphics2D* object is responsible for drawing all parts of the structure, the zoomfactor is inherited for all building blocks or elements shown in the visualization. This is done simply by passing the *Graphics2D* object *g2* to the *RnaS-*

tructure rnastruct which calls the individual *show* methods of the *RnaShape*s with the *Graphics2D* object as a parameter:

```
AffineTransform zoom = new AffineTransform();
zoom.setToScale(zoomfactor,zoomfactor);
g2.transform(zoom);
rnastruct.drawStructure(g2,fixed);
```

Moving When the building block attached to mouse cursor is moved, the coordinates of the building block are changed according to the location of the mouse cursor. In this case, no affine transformations are calculated, but rather the x and y coordinates are changed directly. All other coordinates depend on these two and must not be altered.

When the element is first dropped onto the canvas, the coordinates are stored. Subsequent movements of the entire motif are performed with an *AffineTransform movetransform* within the *RnaStructure*. When the structure moves around, the *AffineTransform* calculates the translation between the old location and the mouse cursor position:

```
//mx and my are the mouse cursor coordinates
//xmove and ymove store them to calculate the shift
movetransform.setToTranslation(mx-xmove,my-ymove);
xmove = mx;
ymove = my;
for(RnaShape shape : structure){
   shape.changeLocation(movetransform);
}
```

The translation is passed to all of the *RnaShape*s stored within the *RnaStructure* which recalculate their location. Thus, again the x and y coordinates are adjusted. Yet, they cannot be calculated directly as they do not coincide with the mouse cursor location. Instead, the change of location of this point is used to create an *AffineTransform* object that transforms the coordinates of every *RnaShape*.

Rotation The most complex transformation is the rotation of the entire structure and of the element connected to the mouse cursor. The later can occur both actively with the user clicking the appropriate button or passively when snapping to a *Magnet*. Since a rotation requires not only the degree of rotation, but also the point around which to rotate, the sequence of rotations of each building block must be preserved. Therefore, every *RnaShape* has a *Vector* that keeps track of every *Rotation* that occurred, storing both the degree as well as the coordinates of the rotation center. Whenever the *RnaShape* is rotated around its center, the rotation is simply appended to the *AffineTransform* object *at* of the class and stored in the *Vector rotations*:

```
at.rotate(StrictMath.toRadians(degree),xloc+width/2,yloc+height/2);
rotations.add(new Rotation(degree,3));
```

Once the building block is attached to the structure, all previous rotations are replaced by one rotation based on the angle describing its orientation at that time.

When the entire structure is rotated, the rotation is prepended. These rotations move the center point of each *RnaShape* to another location. Since the rotations of the individual *RnaShape*s depend on their center point, its location must be calculated before performing rotations around it.

```
AffineTransform buf = new AffineTransform(at);
at.setToIdentity();
at.rotate(StrictMath.toRadians(degree),xmiddle,ymiddle);
rotations.add(0,new Rotation(degree,xmiddle,ymiddle));
at.concatenate(buf);
```

If a movement is then exerted on the building block, the rotations have to be recalculated, as the center points have changed. Thus, the *Rotation Vector* is used to repeat the transformations of the *AffineTransform* object in correct order.

Such accumulation of translations and rotations bears the danger of slowing down the visualization of a motif that has undergone many iterations of refinement. So far, this effect has not been observed, and I have not yet thought about possible remedies.

Shifts If the size of a stem is reduced or enlarged, its exits are moved to another location. Consequently, any neighboring building blocks must be adjusted. An *AffineTransform* object is created that performs a translation according to the displacement of the appropriate *Magnet*. For convenience, the bottom exit of the stem is chosen, as a size change is then simply a change in the *height* of the stem:

```
//change the height of the stem and recalculate its area
height = ((double)seqlength) * 10;
area = new Area(new Rectangle2D.Double(xloc,yloc,width,height));
adjustPath();
//store the old magnet locations
oldmoffset = moffset.clone();
oldmtheta = mtheta.clone();
adjustMagnets();
...
```

```
Line2D.Double oldline = oldmtheta.getLine();
Line2D.Double newline = mtheta.getLine();
//calculate the translation of the bottom magnet
movetransform.setToTranslation(newline.getX1()-oldline.getX1(),
                               newline.getY1()-oldline.getY1());
//move the neighboring structure at the bottom
traverseShift(movetransform, ((theta+offsetangle)%360));
```

Yet, a *SingleShape* is not a regular geometric form whose coordinates can be recalculated when part of a motif structure. Recall, that the final *SingleShape* is a combination of two *GeneralPath* objects which depend on the location of the mouse cursor or a *Magnet* at that time. Therefore, it is first checked whether a single strand is part of the substructure starting at the bottom exit of the stem. Since the use of single strands is restricted to combining different structure parts, they cannot occur on both sides of a stem. Therefore, if a single strand is located at the bottom exit of the stem, the *height* of the stem is changed and afterwards an affine translation is used to move it so that its bottom side is located at its original place. Then, the building blocks at the top of the stem are shifted.

```
if(findSS((theta+offsetangle)%360)){
  Line2D.Double oldline = oldmtheta.getLine();
  Line2D.Double newline = mtheta.getLine();
  //calculate the shift of the stem
  movetransform.setToTranslation(oldline.getX1()-newline.getX1(),
                                 oldline.getY1()-newline.getY1());
  //move the stem to its old bottom end
  changeLocation(movetransform);
  oldline = oldmoffset.getLine();
  newline = moffset.getLine();

  //calculate the translation of the top magnet
  movetransform.setToTranslation(newline.getX1()-oldline.getX1(),
                                 newline.getY1()-oldline.getY1());
  //move the neighboring structure on top
  traverseShift(movetransform, theta);
}
```

The extent of the shift is determined by the translation of the appropriate *Magnet*. A shift traversal is performed with an *AffineTransform* representing the translation.

Magnet Update In all these cases not only the *RnaShape*s are transformed, but also their *Magnet*s are moved. Therefore, whenever a building block is transformed, its *Magnet*s must be adjusted afterwards. This is done with the *RnaShape*'s *AffineTransform* that stores any current transformations effective for the building block. First, the *Line2D*s are reinitialized according to the current coordinates. Then, their end points are transformed according to the current state of the *AffineTransform*. New *Magnet*s are created based on the transformed end points. Finally, the *FreeMagnets* array is updated with the new *Magnet*s.

```
Line2D.Double line = new Line2D.Double(xloc,yloc,xloc+width,yloc);
Point2D p1 = line.getP1();
Point2D p2 = line.getP2();
at.transform(p1,p1);
at.transform(p2,p2);
//FreeMagnets fm
if(fm.contains(moffset)){
   fm.remove(moffset);
   moffset = new Magnet(new Line2D.Double(p1,p2),this,
                        (theta+offsetangle)%360,
                        moffset.getIsHairpinAccessible());
   fm.add(moffset);
}
...
```

Deleting or Detaching elements

Deletion and detachment of an element from the structure is achieved using the same steps for both operations. The only difference is the fact, that after a deletion, the building block is completely removed from the screen, whereas after a detachment, it is drawn next to the mouse cursor and can be placed elsewhere. In this case, all properties stored for the building block remain intact. Here, I will use the term "remove" to refer to both kinds of operations.

A building block can easily be removed from the structure by removing it from the *RnaStructure Vector* and removing its *Magnet*s from the *FreeMagnets Vector*. Yet in order to prevent disruption of the motif structure, it is only possible to remove elements from the ends of the structure. Here, a hairpin loop is also considered an "end" of the structure. A multiloop can only be removed from the structure, if all other exits of the multiloop are unoccupied. Thus, it is not possible to delete e.g. a bulge within the structure and replace it with an internal loop. Instead, all building blocks up to the bulge must be detached and placed elsewhere on the screen. Then, the bulge can be deleted and replaced with an internal loop. Afterwards, the original building blocks can be picked up from the screen (by detaching them) and added to the modified structure. The reason for this restriction is the fact, that one cannot pick up entire structure parts and add them to another part. If a building block was removed within the structure, it would currently be impossible to combine the resulting structure parts directly. There are two solutions to this problem that could be implemented in future versions of the software: Either by allowing deletions within the structure and then combining the different parts by detaching individual building blocks from one part and adding them to another. Or, favorably, by implementing the different structure parts as distinctive mobile elements that can be moved around individually. I will go into more detail on the later option in Chapter 7.

Furthermore, if single strands are contained in the structure, all of them must be deleted before any other building block can be removed from the structure. These deletions must occur from the ends, i.e. if several structure parts are connected via single strands, only those single strands next to either the 5' or the 3' end can be deleted. This way, I can ensure that the 5' and 3' ends cannot be placed on two different structure parts.

Effectively, the usage is restricted to a mode where different parts are first defined and then, single strands are used to connect or extend them. It would be highly desirable to find a solution that allows for greater freedom when removing elements from the structure, but this would take extensive effort in ensuring consistency in the motif structure. This problem is also addressed in Chapter 7.

All these restrictions and several cases must be taken into account when removing a building block:

First, all single strands must be removed from the structure beginning with those extending either the 5' (3') end. Then, those single strands connecting the 5' (3') end to another structure part can be removed. Basically, the 5' (3') end is carried along, jumping from one structure part to the next. That way, it is not possible to separate both ends onto disconnected motif parts.

For deletion of single strands and all other building blocks, the following rules hold:

> If the last building block on the screen is being removed, any future building block must initialize the *RnaStructure*.
>
> Else, if the building block is a hairpin loop or ClosedEnd AND the start ele-

ment, it cannot have any neighbors (Recall that any single strands must have been removed before.). Then, a new start and end element must be determined within an additional structure part. All additional structure parts are stored by their start element in a generic *Vector<RnaShape> furtherstarts*. Whenever elements are added to an additional structure part, the start is updated accordingly. Therefore, all that needs to be done here, is to remove the first element from the *furtherstarts* and store it as the 5' and 3' end.

Else, in all other cases

If the building block is the start element, then its neighbor must be the new start element. If there is no neighbor, then the first element from the *furtherstarts* is removed and stored as the 5' element.

And if the building block is the end element, then its neighbor must be the new end element. If there is no neighbor, the 3' end is the same as the 5' end.

And if the building block is contained in the *furtherstarts*, i.e. it is a start element of structure part, it is removed from the *Vector*. If present, its neighbor is chosen as the new start element for this structure part.

Finally, the online shape string is updated. In case of a detachment, the neighbor references of the detached element are removed and it is drawn next to the mouse cursor.

Building block selection

Whenever a building block is selected, a dark red *Rectangle2D* is drawn around it and a transparent gray color is placed upon it. This is implemented in the *DrawingSurface* and simply drawn/filled upon the standard view of the *RnaShape*. The selection indicates which building block is currently edited and can be used to delete or detach a selected building block by clicking on the appropriate button.

Orientation changes

The orientation of the structure can be changed at all times. The system is based on a standard orientation stored in a boolean flag. The data level is not aware of the orientation.

It simply stores the relevant information on the 5' and 3' strand. Yet, the view must reflect the current orientation by drawing the loops of bulge and internal loop at the appropriate side and by writing the 5' and 3' tags to the correct strands. As mentioned before, any traversals depend on the orientation when passing through a multiloop. This is described in detail in Section 5.3.8.

In standard orientation, the 5' end is located at the upper right strand on top of all building blocks. If not located at the face on top of a building block, it is found at the same relative location after rotating to the appropriate angle. When the orientation is changed, 5' and 3' end are exchanged and the orientation is changed within every building block. In case of the bulge, the loop must be switched to the other strand, such that the view remains the same, but the bulge is then stored on the other strand. In case of the internal loop, both strands must be exchanged on the data level as they can have different restrictions. In all other building blocks, only the flag is updated. For all *RnaShape*s, it is checked whether a *DoubleSingleShape* is located at one of its exits. Since these are not stored in the *RnaStructure*, but act only as placeholders for the neighbor references to *SingleShape*s, their *changeOrientation* function must be called separatly. For them, the *5'ss* and *3'ss* must be exchanged (see Figure 5.14). Also, if the user wishes, any stored sequence motifs are reversed when the orientation is changed.

5.3.7 Semantic integrity of motif descriptions

I carefully designed the mechanics of the editor such that it is impossible to construct motifs that lead to semantically incorrect ADP programs. This is why one cannot extend a single strand by another single strand, or construct a hairpin loop from a stem and a single strand that connects its bottom ends. Although such descriptions would be equivalent graphically, the energy functions associated with them would be inappropriate.

One aspect that has not been taken care of yet is a possible source of ambiguity in the generated matcher grammars. This problem arises whenever stems with no size restrictions are placed next to other stems, ClosedStructs or ClosedEnds. I will go into more detail on this problem and present possible means to solve it in Chapter 6.

Summarizing the design decisions, it is clear that I have taken great care that the user can draw all biologically plausible motifs, but no others. No semantic checking is required when graphics are translated into programs.

5.3.8 Traversing through the structure

Using the reference to the start element stored in the *DrawingSurface* class, we can invoke a traversal through the structure for different purposes. On one hand, a traversal is necessary for completed RNA motifs when translating them into XML code. On the other hand, they are also required when a hairpin is attached to the 5' end leading to an adjustment of the start element. Finally, any adjustment is immediately reflected in the shape string shown at the bottom of the editor (see Section 5.3.9).

Recall that the structure is stored as a tree with the start point being the root of the tree. A traversal is thus a depth-first walk through the entire tree. Yet, we do not have to walk back to a branching point (a multiloop), but can end the walk at every leaf of the tree (hairpin loop). If single strands are used to connect two or more structure parts, we actually have a number of trees that are traversed in order from start to end point.

Internally, a traversal is based on the neighbor references and an angle describing the direction of the traversal. The angle describes the side through which a building block is left. In standard building blocks this direction does not change, yet in a multiloop, it is necessary to locate the entrance and exit angle within the multiloop. Then, the order in which the multiloop must be processed depends on the orientation of the structure as previously described.

```
protected String traverse(int angle, int type){
  int index = //calculated using the entrance angle

  if(standardorientation){
    for(int j=0;j<7;j++){
      i++;
      i = i%8;
      //there is an exit at i
      if(this.offsetangle[i] == 1){
         //translate and call traverse for neighbor
         ...
      }
  }
  else{
    for(int j=0;j<7;j++){
      i--;
      if(i<0){
        i=7;
      }
      if(this.offsetangle[i] == 1){
         //translate and call traverse for neighbor
         ...
      }
    }
  }
}
```

When a single strand must be parsed, the traversal first enters the *DoubleSingleShape* that is stored as the neighbor of that building block and then enters either its 5' or the 3' strand depending on the direction of the traversal. If the traversal is continued in another structure part, the startangle of the first building block of that structure part determines the direction of the traversal.

5.3.9 Online shape strings

Below the main *DrawingSurface*, there is a *JTextField* that shows an adapted abstract shape string ([GVR04, SVR$^+$06]) for the secondary structure. The following symbols were introduced in addition to standard shape strings:

- `R` represents a site where a building block addition is possible
- `{-` and `-}` represent the opening and closing basepairs of a ClosedStruct.
- `{---}` represents a ClosedEnd.
- different structure parts are separated by `----`. They are shown in order of their addition to the *DrawingSurface.*

The shape strings fulfill several purposes. First of all, it is a precise, short notation of RNA secondary structures that is related to the well-known dot-bracket strings. The user can see by these strings that the graphics s/he is designing represents the correct secondary structure. Any changes made to the motif structure, e.g. bulge location change, addition or removal of building blocks, (dis)connection of different parts, are immediately reflected in the shape string. This was very helpful when programming the editor and serves great benefit in debugging it. Problems in the connections within the structure and errors occurring through orientation changes or other user interactions can be analyzed using this concise notation.

5.3.10 Project maintenance

The Locomotif system allows to work on different projects with the options to save the current motif definition and load it from file. While making changes to the structure, the user can always restore to the previously stored version of the project.

Loading and Storing projects

All relevant information for both the data and visual level are written to a `.rna` file using *ObjectOutputStreams*. Any object that is to be stored via these streams must implement the *Serializable* interface.

BuildingBlock For the data level it suffices to implement the *Serializable* interface in the *BuildingBlock* class which is inherited in all subclasses. No methods must be overwritten here, as all variables are written to file using the default *writeObject* method.

RnaShape For the visual level, I chose to implement the *Serializable* interface in all individual *Shape* classes and overwrite the *writeObject* and *readObject* methods. The visualization depends on some variables that are used to create the *Area* and *GeneralPath* for the building block. Therefore, it suffices to write these variables to file and recreate the geometric compounds upon loading from file.

Using the *ObjectOutputStream* and *ObjectInputStream* is a convenient way to allow project maintenance, since nothing needs to be done, but to implement an interface and optionally specify what is to be stored via the *writeObject* and *readObject* methods. Yet, this strategy does impose risks regarding persistency. By default, every serializable class automatically obtains a unique identifier which is changed whenever the class is adapted, e.g. by adding a new field. If a user stores a secondary structure and returns to the Locomotif system after changes were introduced, an exception would occur when s/he tries to load the old data file. In order to prevent these issues, I chose to control the versioning of the system by adding a *serialVersionUID* to all classes that have to be serialized. That way, it is no problem to use older files in a new Locomotif version, but any future versions must ensure compatibility with the older ones. If this cannot be ensured, because major changes must be made to a class, then the user must be informed and refused when trying to load an older version file to avoid system crashes.

Printing

At any time during motif definition, the user can chose to print the current view of the secondary structure to an image file. The type of available image formats depends on the machine in use. The implementation is simple: Instead of using a *Graphics2D* object in the *paintComponent* method of the *DrawingSurface* to draw the structure on the screen, a *Graphics2D* object is created for a *BufferedImage*. Then, the drawing and filling steps are exactly the same as the ones within the *paintComponent* method and the resulting *BufferedImage* is written to a file in the desired format using an *ImageIO* object.

5.4 Compilation to XML

XML with its treelike document structure is well-fitted for storing information on RNA motifs. In our case, it is simply a sequence of building blocks in order from 5' to 3' end. An example file can be seen in Figure 5.3. The XML code is sent to the server as a DOM (Document Object Model) document which can be handled comfortably using the org.jdom package. Jdom offers object-oriented handling of DOM documents as well as creating and manipulating the individual elements of the document. Since the Jdom tree has the same order as the RNA motif, it can be constructed iteratively during the traversal. The root element of the Jdom tree is constructed upon the beginning of the traversal. Each building block in order of the traversal is translated into a Jdom *Element* which is appended as a child to the previous one, thereby creating a tree structure: Multiloops lead to several neighboring children, i.e. a fork in the tree, whereas connecting single strands lead to new structures parts, i.e. different trees connected to each other on the same level. Actually, a multiloop does the same thing within the motif tree: connecting different subtrees on the same level of the motif. Each building block *Element* stores size information as attribute nodes and has child element nodes for sequence motifs. Once the traversal is completed, we construct a DOM *Document* from the Jdom tree using a *DOMOutputter* object provided with the org.jdom package. A pretty print version of the Jdom tree is obtained similarily via an *XMLOutputter*.

An XML schema checks whether the information sent to the server is of the correct format. A visual overview of the schema is given in Figure 5.16. Only XML documents describing RNA motifs in the exact terminology defined by the schema are accepted. Thus, we can be sure that no incorrect DOM document is sent to the BiBiServ which would lead to errors when trying to translate it into ADP code. Complete documentation of the XML schema can be found on the BiBiServ[4]

[4]Locomotif Schema, http://bibiserv.techfak.uni-bielefeld.de/locomotif/documentation.html.

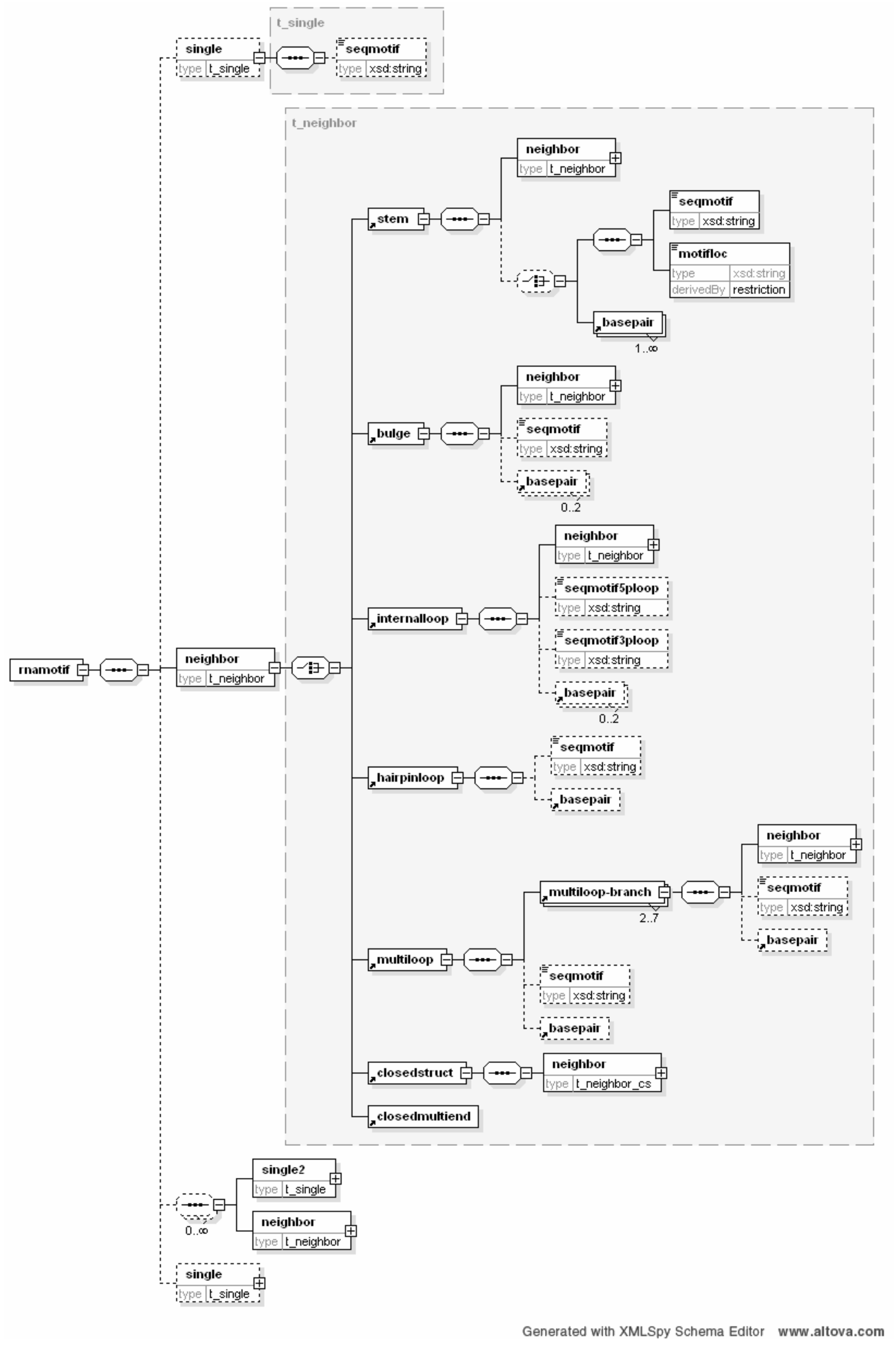


Figure 5.16: The complete schema for the Locomotif system: an `rnamotif` is a `sequence` of a `neighbor`, optionally framed by `single` strands and connected to other `neighbors`. `t_neighbor` includes all 7 blocks whereas `t_neighbor_cs` is restricted to `stem`, `bulge` and `internalloop`.

5.5 Declarative level

For the translation to ADP, the DOM document is processed using the Jdom libraries and translated into declarative code. For each building block element of the XML document, there is a code section in the ADP grammar that describes the structure of the motif. These sections are based on the template trees shown in Figure 5.10. The grammar block for each building block can be appended to the overall ADP file as it occurs in the Jdom tree. In the specific methods for processing the different types of building blocks, the attributes (e.g. length specifications) and child elements (e.g. sequence motifs) of each building block are taken care of. Additionally, a generic header is included in the final code with the algebra functions used to evaluate the grammar.

A motif description in ADP notation has a declarative and an operational semantics. I refer to [GMS04] for the definitions and only give an informal explanation here.

A few lines of ADP code generated from our IRE motif are shown in Figure 5.17. Their declarative meaning can be cast in a narrative form as follows:

> The overall motif is `rnastruct`, which is a `motif0` embedded somewhere in an RNA sequence.
> `motif0` is a helix named `stem0`.
> `stem0` consists of any number of base pairs, enclosing a `motif1`.
> `motif1` is a bulge named `bulge1`.
> `bulge1` is a base pair enclosing a left bulge of exactly one base “C”, and a `motif_b2`.
> `motif_b2` is a single base pair enclosing `motif2`.
> `motif2` is a helix named `stem2`.
> `stem2` is a series of three base pairs enclosing `motif3`.
> `motif3` is a hairpin named `hairpin3`.
> `hairpin3` is a base pair enclosing a loop region of exactly 6 bases, “CAGUGN”.

Thus, this describes the syntax of the motif in the fashion of a context free grammar.

The functions `sr, bl, hl, ...` attached via `<<<` describe the computation of free energies from the energy of embedded substructures and from local contributions by base pairs, bulges and loops. The function `h`, attached via the `...` operator, indicates that a choice between alternative motif matches is to be made, based on lowest energy.

The operational semantics of ADP is defined via the technique of parser combinators [Hut92]. Operationally, the symbols `motif0, stem0` etc. denote parsers, which recognise a particular submotif and evaluate its free energy. The three-letter operators `<<<, |||, ~~~, -~~, ~~-` are implemented parser combinators [GMS02]. The keywords `tabulated`

```
rnastruct = listed ( sadd <<< lbase -~~ rnastruct |||
                     addss <<< motif0 ~~~ uregion     ... h);



motif0 = stem0     ... h;
stem0 = tabulated ( (sr <<< lbase -~~ stem0 ~~- lbase |||
                     sr <<< lbase -~~ motif1 ~~- lbase) `with` basepairing ... h);

motif1 = bulge1    ... h;
bulge1 = tabulated ( (bl <<< lbase -~~
                           ((region `with` size (1,1))
                                    `with` contains_region "C" ) ~~~
                             motif_b2  ~~-
                             lbase) `with` basepairing     ... h);
motif_b2 = (sr <<< lbase -~~ motif2 ~~- lbase) `with` basepairing ... h;

motif2 = stem2     ... h;
stem2 = tabulated ( ((sr <<< lbase -~~
                           ((sr <<< lbase -~~
                                  ((sr <<< lbase -~~
                                           motif3 ~~-
                                               lbase) `with` basepairing) ~~-
                                    lbase) `with` basepairing)  ~~-
                             lbase) `with` basepairing)    ... h);

motif3 = hairpin3  ... h;
hairpin3 = (hl <<< lbase -~~
                 ((region `with` size (6,6)) `with` contains_region "CAGUGN") ~~-
                   lbase) `with` basepairing ... h;
```

Figure 5.17: Concrete ADP description for the IRE motif, as generated from the XML encoding of the graphics.

and `listed` mark certain parts of a motif for tabulation, to avoid excessive recalculation.

Each building block corresponds to an ADP code template, further refined by information taken from its attributes. I demonstrate one example. The bulge building block has a code template of the form

```
bulge$_i$ = tabulated ( ( bl <<< lbase -~~ region # ~~~ motif_b$_{i+1}$ ~~- lbase)
                                        `with` basepairing   ...h);
```

The symbol # could be the empty string, if no attributes were defined. In our example, # is refined to the clause

```
‘with‘ size(1,1) ‘with‘ contains_region "C",
```

reflecting the specification that the bulge consists of a single C nucleotide. The $ symbols must be instantiated to create unique names, connecting program clauses in the same way as building blocks are connected in the overall motif.

Specific bases from sequence motifs in stems or basepairs are replaced by using the `iupac_base` parser instead of the `lbase`. It requires a iupac code as an argument that restricts the allowed bases in the target sequence. For loop motifs, the `region` parser is refined with a `contains_region` filter function that receives the stored motif as a parameter.

Global size restrictions are simply added to the building block of their origin using a `‘with‘ (min/max)size` command:

```
motif1 = bulge1 ‘with‘ minsize 20;
```

In this case, the entire substructure rooted at `bulge1` is made up of at least 20 bases. Similarily, local constraints on a loop region can be added to the `region` parser.

Handling these special cases requires the use of many if/then/else blocks in the methods of the *ADPTranslator*. Spaceholder strings or arrays are initialized with the standard values

```
String region = "region";
```

and then replaced by the required enhancements according to the attributes of the element.

```
region += "‘with‘ minsize 6";
```

Then, during translation, the placeholder strings/arrays can be used unaware of their specific content allowing for general translation methods.

ADP building blocks

There are several obstacles for the different building blocks involving their translation into ADP code blocks. Tabulation keywords and choice functions were omitted for clarity.

Stem A stem is the most complex building block in ADP code as every base has to be enumerated explicitly when length or sequence constraints are given. Basically, a stem is a sequence of basepairs in the form

```
stem$_i = (sr <<< lbase -~~ nextnonterm ~~- lbase) `with` basepairing;
```

Depending on the size restriction, different ways of using this rule are employed:

- No restriction: A recursive rule must be provided (`nextnonterm` = `stem$`$_i$) and a rule leading to the next motif building block (`nextnonterm` = `motif$`$_{i+1}$) must be added using the `|||` - OR combinator.
- Exact length: All basepairs of the stem are explicitly listed and the `nextnonterm` in the innermost position refers to the next motif building block. Here is an example template for a stem with 3 basepairs:

  ```
  stem$_i = (sr <<< lbase -~~ ((sr <<< lbase -~~
               ((sr <<< lbase -~~ motif$_i+1 ~~- lbase) `with` basepairing )
                                  ~~- lbase) `with` basepairing)
               ~~- lbase) `with` basepairing;
  ```

- Minimum length: The minimum number of basepairs are explicitly listed and then the rule for no restriction is used.
- Maximum length: The maximum number of basepairs are explicitly listed, but after every basepair follows either the next basepair or the `nextnonterm`, i.e. the next motif building block.
- Minimum and Maximum length: A combination of the previous two rules is used. First, the (minimum - 1) number of basepairs is explicitly listed with the `nextnonterm` at the innermost position being a `maxstem$`. The `maxstem$` explicitly lists the remaining number of basepairs to the maximum value, but can be interrupted after every basepair with the `nextnonterm`, i.e. next motif building block. Here is an example for a stem having 2 - 4 basepairs:

  ```
  stem0 = tabulated ( ((sr <<< lbase -~~ maxstem0 ~~- lbase)
                              `with` basepairing)  ... h);
  maxstem0 = (sr <<< lbase -~~ ( motif1 |||
                             (sr <<< lbase -~~ ( motif1 |||
                                (sr <<< lbase -~~ motif1 ~~- lbase)
                                        `with` basepairing)
                                 ~~- lbase) `with` basepairing)
                              ~~- lbase) `with` basepairing  ... h;
  ```

Alternatively, a stem can be discontinuous in which case generic rules are used that cannot be refined. It contains at least one basepair and can include left or right bulges or internal loops, all having maximum loop sizes of 2. Using the additional `stembp$`$_i$ rule requires the presence of at least two basepairs in between loop regions, thus disallowing isolated basepairs.

```
stem$_i = (sr <<< lbase -~~ stem$_i ~~- lbase |||
          bl <<< lbase -~~ (region ‘with‘ maxsize 2) ~~~ stembp$_i~~- lbase  |||
          br <<< lbase -~~ stembp$_i ~~~ (region ‘with‘ maxsize 2) ~~- lbase |||
          il <<< lbase -~~ (region ‘with‘ maxsize 2) ~~~ stembp$_i
                                 ~~~ (region ‘with‘ maxsize 2) ~~- lbase |||
          sr <<< lbase -~~ motif$_{i+1} ~~- lbase) ‘with‘ basepairing  ...h;

stembp$_i = (sr <<< lbase -~~ stem$_i ~~- lbase |||
            sr <<< lbase -~~ motif$_{i+1} ~~- lbase) ‘with‘ basepairing  ...h;
```

Bulge and Internal Loop For the bulge loop, the location of the loop region must be encoded in the ADP grammar. Basically, this determines whether the `region` parser is situated to the left or the right of the `motif_b$` nonterminal. For efficiency reasons, all `region` parsers within bulge and internal loops are automatically restricted to a `maxsize` of 30 unless stronger restrictions were made by the user. These restrictions are omitted here for clarity.

```
bulge$_i = ( bl <<< lbase -~~ region ~~~ motif_b$_{i+1} ~~- lbase)
                    ‘with‘ basepairing;

bulge$_i = ( br <<< lbase -~~ motif_b$_{i+1} ~~~ region ~~- lbase)
                    ‘with‘ basepairing;
```

For the internal loop, a loop region is located on both sides of the next `motif_b$`, but there may be different constraints acting on the 5’ and the 3’ strand. The `region` parsers must be instantiated accordingly.

```
internal$_i = ( il <<< lbase -~~ region ~~~ motif_b$_{i+1} ~~~ region
                       ~~- lbase) ‘with‘ basepairing;
```

Hairpin Loop In the hairpin loop, the `region` parser is always used in combination with a `minsize 3` requirement unless a higher minimum or exact value was given.

```
hairpin$_i = ( hl <<< lbase -~~ (region `with` minsize 3) ~~- lbase)
                       `with` basepairing;
```

Multiloop The multiloop's function is to hold several structure parts together and this function is reflected in its ADP code. It starts with a basepair tying together the different structure parts whose inner nonterminal is a `ml_tail$`:

```
multiloop$_i = (ml <<< lbase -~~ ml_tail$_i ~~- lbase)
                        `with` basepairing;
```

This tail begins with a `uregion`, i.e. a possibly empty region followed by a basepair and the next internal tail:

```
ml_tail$_i = ssadd <<< uregion ~~~ ml_nexttail$_{i+1};
```

Then, the `ml_nexttail$` nonterminal refers to the next motif part followed by another tail of the multiloop:

```
ml_nexttail$_{i+1} = mlcons <<< ml_motif_bp$_{i+1} ~~~ ml_tail$_{i+1};
```

The `ml_motif_bp$`$_{i+1}$ nonterminal leads to a basepair enclosing another motif building block:

```
ml_motif_bp$_{i+1} = (sr <<< lbase -~~ ml_motif$_{i+1} ~~- lbase)
                          `with` basepairing  ... h;
```

Here, the functions `ml` and `mlcons` incur an energy penalty as the more favorable rod-like structure is interrupted by the multiloop. The final `ml_nexttail` is followed by a `ml_motif_bp$`$_j$ and a `uregion` ending the multiloop.

Single Strand The single strand can either extend 5' or 3' end or connect different structure parts. In the later case, it can either be a straight connection or one allowing internal basepairings. If no internal folding is allowed, the single strand is the most straightforward building block, as it is simply a `region` that can be restricted the same way as all other regions in the ADP code. Nevertheless, it is encoded as a separate motif building block and not part of any other code section.

However, if folding of the single strand is allowed, the resulting ADP code is quite different. The single strand can still be a straight `region` or folded to another structure part.

```
motif$_i$ = ss <<< region ||| struct   ...h;
```

A `struct` is an entire `closed` motif part contained somewhere within the single strand and can be followed by another `struct` or the remaining unpaired single strand `region`.

```
struct = tabulated ( sadd <<< lbase -~~ struct |||
                     cadd <<< closed ~~~ struct |||
                    addss <<< closed ~~~ region   ...h);
```

The `closed` motif part can contain all types of building blocks except for single stranded regions.

```
closed = stack ||| hairpin ||| leftB ||| rightB ||| iloop ||| multiloop  ...h;
```

It is not necessary to include nonterminals for the compound building blocks as they represent the same or a smaller class of building blocks. A `stack` is replaced by one basepair followed by `closed`. The bulge and internal loops represent one basepair followed by a `region` on one or both strands followed by a `stack`. Hairpin and multiloop are defined as described in the paragraphs above, except that every nexttail of the multiloop leads to a `stack` instead of the next motif building block.

```
ml_nexttaila = mlcons <<< stack ~~~ ml_tailb  ...h;
```

No indices are needed here, since this code section describes the general, unrestricted folding of a sequence segment, framed by restricted motif parts. The entire code section is included only once in the ADP grammar to avoid redefinition and recalculation of the nonterminals.

ClosedStruct The ClosedStruct is a compound building block for any number of stems, internal or bulge loops. Thus, in its ADP code, rules for all these types of building blocks are included similar to the unrestricted folding of a single strand described above. However, it must be ensured that it contains at least one basepair, i.e. it is nonempty. This is done by including the basepair in the only recursion leading to the next motif building block.

```
closed$_i$ = stack$_i$ ||| iloop$_i$ ||| bulgeR$_i$ ||| bulgeL$_i$ |||
          (sr <<< lbase -~~ motif$_{i+1}$ ~~- lbase) `with` basepairing   ...h;
```

The rules for `stack$`i and the other nonterminals refer back to the original `closed$`$_i$. Internally, the ClosedStruct rules can be reused, but within the entire motif grammar it must obtain a unique index. The building block has a specific location within a motif structure part and is thus followed by a particular motif building block. Therefore, every ClosedStruct within the motif is unique and must be treated as such.

ClosedEnd The ClosedEnd building block on the other hand closes a motif part. It does not enclose any specific motif building blocks. Therefore, the ClosedEnd is defined only once for the entire ADP motif grammar and its definition is based on the same `closed` rules responsible for general folding of a motif part within a connecting single strand. Thus, the `closed` rules need to be included only once for ClosedEnds and foldable single strands combined.

Actually, if the user chooses to use only the ClosedEnd building block to define a motif, the resulting ADP program does not search a motif, but rather computes general RNA folding. In case of a global search, the best folding of a subsequence is computed and in case of a local search, the entire sequence is folded to the thermodynamically optimal secondary structure.

5.6 Direct compilation to target code

In the generated form, a motif can be compiled either as a Haskell program or directly from ADP notation into C. In the later case, the compiler makes the ADP approach independent of Haskell as a host language. (The compiler, however, is written in Haskell.) The main motivation for this quite substantial effort has an intellectual, a pragmatic and an efficiency aspect. It results from our previous experience with the ADP method.

- While the key concepts of the ADP technique – grammars and algebras – can be understood without a Haskell background, the concrete syntax is strongly influenced by the origin of ADP as a Haskell-embedded domain specific language. The intended user community in bioinformatics typically has no Haskell experience, and is provably reluctant to acquire it.

- When an ADP program is developed, one makes simple errors like omitting the application of an algebra function or a grammar symbol in the right-hand side of a rule. In this case, Haskell confronts us with an error message from the type system, which naturally is unaware that it is dealing with ADP code. Hence, it cannot give meaningful help. Even Haskell experts find it easier to ignore the details of the type error and just re-inspect the ADP code.

- Efficiency of the resulting dynamic programming algorithm depends critically on the choice of nonterminal symbols to be tabulated, and their annotation via the keyword `tabulated` (actually a Haskell function application) is essential. Choosing a good or optimal number of tables is not easy for a human, and has been shown to be an NP-complete problem in general [SG06]. Our compiler makes substantial optimizations regarding the number and size of tables. In principle, these improvements could be embedded in the ADP source program, and the compiler would act as a pre-processor for ADP in Haskell. However, we chose to generate C code directly.

The concerns of syntax and type errors are eliminated by the use of the graphical programming system, but efficiency becomes even more of a concern. The graphics do not offer annotations on tabulation, because we cannot expect a user to provide them. Some annotation is generated by the translation rules, but in general, this is now the task of the compiler. Thereby, it optimizes not only constant factors, but takes responsibility for the asymptotic efficiency of the algorithm. A number of challenges it has to face and the status of the compiler project have been described recently in [GS06, Ste06]. While the development of the graphical programming system relied on the Haskell embedding, it is now based on the new compiler developed by Peter Steffen.

From our contact with the users of the graphical programming system, we expect a flow of requests for extending the motif description language. This will require a joint extension of the graphical level and the ADP notation. Again, liberating ADP from its host language will allow us to choose a more convenient syntax in such extensions.

Chapter 6

Effective Ambiguity Checking in RNA Motif Search

6.1 Preface

The paper included here was published in BMC Bioinformatics in 2005 [RSG05]. It is joined work with Peter Steffen and Robert Giegerich. Robert came up with the topic and the undecidability result included in Section 6.5, Peter focused on the partial proof technique presented in Section 6.3.3 and my efforts concentrated on the testing procedures described in Section 6.3.2. The paper deals with the problem of ambiguity in grammars for RNA secondary structures which also arises in the generated Locomotif matcher grammars. It identifies typical sources of ambiguity and presents means to detect ambiguity in context free grammars. Since the Locomotif matcher grammars for RNA motif searches are generated automatically, ambiguity issues are mostly eliminated and the user does not need to worry about them. Yet, there are some cases when ambiguity can still be introduced into the ADP grammars and the user must be aware that the search results might be corrupted in these cases. A further description of these issues is included in Section 6.6 of this chapter.

6.2 Background

6.2.1 The ambiguity problem in biosequence analysis

Biosequence analysis problems are typically optimization problems – we seek the best alignment of two protein sequences under a similarity score, or the most stable secondary structure of an RNA molecule under a thermodynamic model. In such a problem, there is a “good” and a “bad” type of ambiguity. The good one is that there are many solutions to choose from. The bad one is that our algorithm may find the same solution several times, or even worse, it may study seemingly different solutions, which in fact represent the same object of interest. The cause of all these phenomenona has been called ambiguity, because it is closely related to the ambiguity problem of formal languages. It is not quite the same problem, however. In striving for avoidance of ambiguity, we want to get rid of the bad type and retain the good.

Ambiguity is not a problem with a dynamic programming (DP) algorithm that returns a single, optimal score, together with a solution that achieves this score, and does not make assertions about other solutions in the search space. Then, it does not matter whether this solution is analyzed several times, or that there are other solutions achieving the optimal score. In other cases, ambiguity can cause a DP algorithm to return an “optimal” answer which is plainly wrong. In the presence of ambiguity, the Viterbi algorithm cannot report the most likely structure [DE04], a folding program cannot produce a complete and non-redundant set of suboptimal structures [WFHS99], and statistics like counts, sum over all scores (by an Inside-type algorithm), or expected number of feasible or canonical structures [Gie00] cannot be computed.

6.2.2 Previous work

The phenomenon of ambiguity has been formalized and studied in [Gie00] in a quite general framework of dynamic programming over sequence data. There, it is shown that for a proof of non-ambiguity, a canonical model of the studied domain is required. The canonical model plays an essential role. It is the mathematical formalization of the real-world domain we want to study, and “canonical” means one-to-one correspondence. Any formal proof can only deal with the formalization of the real-world domain, and when the one-to-one correspondence does not hold, all proofs of (non-)ambiguity would be meaningless for the real world. In general, it may be quite difficult to find a canonical model for some real-world domains. Our case, however, is easy. When RNA secondary structure is our domain of study, base pair sets or the familiar dot-bracket strings can serve as a canonical model, as they uniquely represent secondary structures. To ensure non-ambiguity, there must exist an injective (i.e. one-to-one) mapping from derivation trees (according to the grammar underlying the DP algorithm) to the canonical model. While such a mapping may be easy to specify, the proof of its injectivity remains a problem.

Recently, Dowell and Eddy have re-addressed this problem [DE04] in the framework of stochastic context free grammars (SCFGs). In a probabilistic framework, ambiguity matters when a best, i.e. most likely solution is computed. This solution is wrong if several "different" solutions represent the same real-world object. Dowell and Eddy experimented with two ambiguous SCFGs, and showed that the quality of results may range from just slightly wrong to totally useless. After having shown that one cannot get by with ignoring ambiguity, they provide four non-ambiguous SCFGs for RNA structure analysis; however, a proof of their non-ambiguity is not given. Instead, they suggest a testing approach to check for the presence of ambiguity, which, of course, cannot prove its absence.

In this contribution, we first review the ambiguity problem in the framework of SCFG modeling, explain some of its sources, prove its algorithmic undecidability, and suggest three ways to deal with it: ambiguity avoidance, testing for ambiguity, and, best of all when successful, a mechanical proof of absence.

6.2.3 Formalization of ambiguity

We formalize the problem at hand in two steps, going from context free grammars (CFGs) to stochastic context free grammars, and then differentiating between syntactic and semantic ambiguity.

Formal grammars

A formal language is a subset of the set of all strings over a finite alphabet. Formal languages are typically described by formal grammars. In general, a formal grammar consists of an alphabet, a set of nonterminal symbols, and a set of production rules. There exist various grammar types, differing in the laws for construction of these production rules. The expressive power of a grammar type depends on these laws. In 1956, Noam Chomsky introduced a hierarchy of formal grammars that ranks grammar types by their expressive power, the Chomsky hierarchy [Cho56]. It consists of four levels: regular grammars, context-free grammars, context-sensitive grammars, and unrestricted grammars. Here, we only address context-free grammars. These are suitable to describe the pseudoknot-free secondary structure of RNA. When considering pseudoknots, context-sensitive grammars are needed.

Context free grammars

A context free language is described by a context free grammar G, given by a set of terminal symbols (the alphabet), a set of nonterminal symbols, including a designated axiom symbol, and a set of production rules of the form $X \to \alpha$, where X is a nonterminal symbol, and α is a string of terminal and nonterminal symbols. α may be the empty string, denoted ε. Starting with the axiom symbol, by successive replacement of nonterminal symbols by right-hand sides of corresponding productions, we can derive a set of terminal strings. They constitute the language of the grammar, denoted $L(G)$. Without loss of generality, derivations are canonized by replacing, in each step, the leftmost nonterminal symbol in the string obtained so far. Each such derivation can uniquely be represented as a derivation tree, and if the same terminal string has two different derivation trees, the grammar is called ambiguous.

Our first example is Dowell and Eddy's grammar $G1$ [DE04] to describe RNA secondary structures:

$$\begin{aligned}
G1: \qquad & S \rightarrow aSu \,|\, uSa \,|\, cSg \,|\, gSc \,|\, gSu \,|\, uSg \\
& S \rightarrow aS \,|\, cS \,|\, gS \,|\, uS \\
& S \rightarrow Sa \,|\, Sc \,|\, Sg \,|\, Su \\
& S \rightarrow SS \\
& S \rightarrow \varepsilon
\end{aligned}$$

In the following, we shall use a shorthand notation, where a stands for any base A,C,G,U, while a and $\hat{a}$ occurring in the same rule stand for either one of the base pairs (A,U), (U,A), (C,G), (G,C), (G,U), or (U,G).

$$G1: \qquad S \rightarrow aS\hat{a} \,|\, aS \,|\, Sa \,|\, SS \,|\, \varepsilon$$

Four different derivation trees of the grammar *G1* are shown in Figure 6.1. As they all emerge from the same terminal string `acaggaaacuguacggugcaaccg`, this grammar is ambiguous.

Stochastic context free grammars

Stochastic context free grammars associate a (non-zero) probability with each production, such that the probabilities for all alternative productions emerging from the same non-terminal symbol add up to 1. As a string is derived, probabilities of the involved rules multiply.

We extend the CFG $G1$ to a SCFG by the following example probabilities:

$$\begin{aligned}
P_{S\rightarrow aS\hat{a}} &= 0.2 \\
P_{S\rightarrow aS} &= 0.2 \\
P_{S\rightarrow Sa} &= 0.2 \\
P_{S\rightarrow SS} &= 0.2 \\
P_{S\rightarrow \varepsilon} &= 0.2
\end{aligned}$$

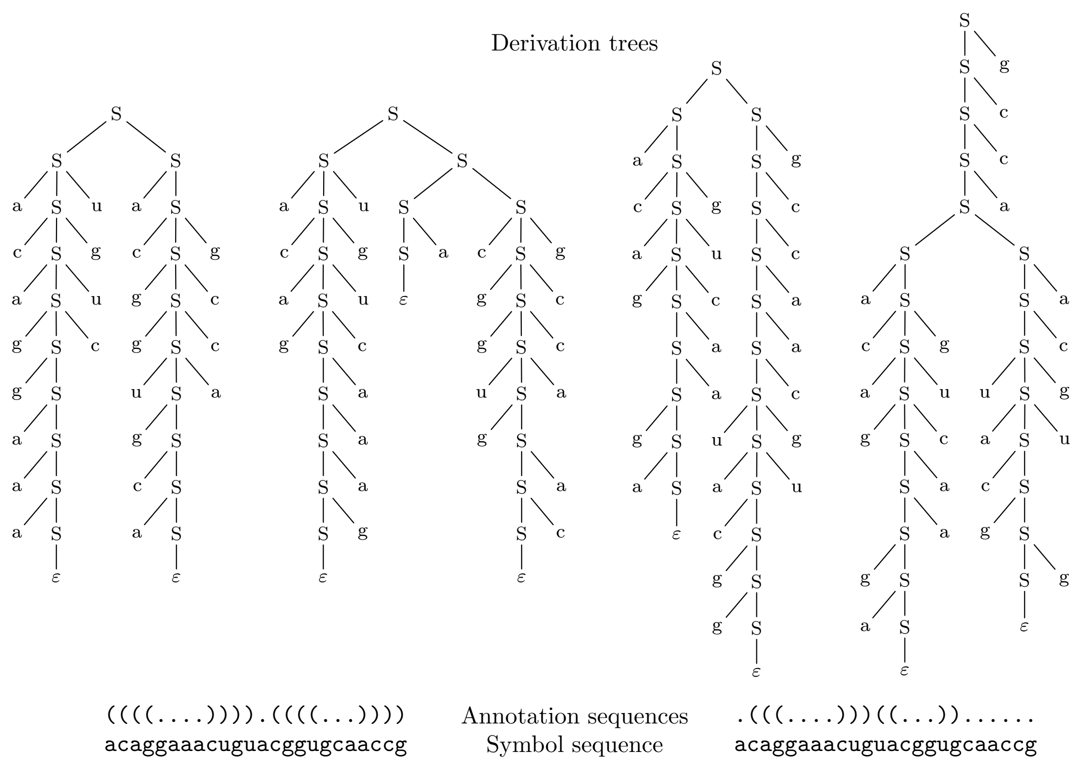


Figure 6.1: Four derivation trees for RNA sequence “acaggaaacuguacggug-caaccg”, representing the annotation sequences `((((....)))).((((...))))` and `.(((....)))((...)).......`

For simplicity, we chose probabilities independent of certain bases. In SCFG design, often also non-canonical base pairings are allowed with a low probability.

For grammar $G1$, the derivations shown in Figure 6.1 have probabilities of $5.24 \cdot 10^{-14}$, $2.1 \cdot 10^{-15}$, $4.19 \cdot 10^{-16}$ and $4.19 \cdot 10^{-16}$ (from left to right).

All derivations for a string can be constructed by a CYK-type parser [AU73]. The parser may compute the overall probability of a given string, summing up probabilities over all its derivations, in which case it is called the Inside algorithm. Or, the parser can return the most likely derivation of the input string, in which case it is known as the Viterbi algorithm. For grammar $G1$, the corresponding CYK-based Viterbi algorithm is shown here:

```
Input: Sequence $x = x_1 \dots x_n$
Initialization: for $1 \leq i \leq n$
$S(i,i) = P_{S \to \varepsilon}$

Iteration: for $1 \leq i < j \leq n$
```

$$S(i,j) = \max \begin{cases} S(i+1, j-1) * P_{S \to x_i S x_j} \\ S(i+1, j) * P_{S \to x_i S} \\ S(i, j-1) * P_{S \to S x_j} \\ \max_{i \leq k < j} \{S(i,k) * (S(k+1,j) * P_{S \to SS})\} \end{cases}$$

Syntactic versus semantic ambiguity

Above, we introduced the formal language-theoretic notion of ambiguity: if the same symbol sequence has two or more different derivation trees, the grammar is called ambiguous. For clarity, we will refer to it as *fl-ambiguity.* In this sense, grammar $G1$ (and every other grammar in this manuscript) is in any case fl-ambiguous. This is demonstrated by the fact that the four derivation trees of Figure 6.1 all belong to the same symbol sequence. We now need to refine this notion of ambiguity.

In modeling with SCFGs, derivations do not merely produce strings, but they represent objects of interest themselves. With RNA, a derivation of an RNA sequence represents a possible secondary structure of this sequence. A more compact representation of a secondary structure is the widely used dot-bracket notation, as shown at the bottom of Figure 6.1. In the following, we will use the term *annotation sequence* for the dot-bracket string representing one secondary structure of the underlying RNA sequence. The one-to-one correspondence between (molecular) structures and (in silico) annotation sequences qualifies the latter as a canonical model of the grammar.

By the term *syntactic ambiguity* we denote the fact that typically an RNA sequence has many secondary structures, i.e. annotation sequences, hence many derivations. Figure 6.1 shows two example annotation sequences of the same RNA sequence.

Semantic ambiguity exists when there are, for some sequence, several derivations that represent the same annotation sequence, and hence, the same secondary structure. This is our point of study. In this case, the probability of a certain annotation sequence is split up into the probabilities of its multiple derivations. In Figure 6.1, this is exemplified by the two derivations on the left that both represent the annotation sequence `((((....)))).((((...))))`, and the two derivations on the right, that both represent the annotation sequence `.(((....)))((...)).......` Thus, grammar $G1$ is syntactically as well as semantically ambiguous.

Semantic ambiguity is the “bad”, syntactic ambiguity the “good” type of ambiguity in SCFG modeling and dynamic programming that was mentioned above. On the pure formal language level, they cannot be distinguished – both are manifest as fl-ambiguity. The bad ambiguity hides with the good, which is why its presence is sometimes overlooked.

Semantic ambiguity is not a problem with the Inside algorithm, as a probability sum over all derivations is computed anyway. With the Viterbi algorithm, we can certainly obtain the most likely derivation, but we do not know whether it represents the most likely annotation sequence. Some other annotation sequence may be more likely, but as its probability is the sum of many different derivations, none of these derivations may come out optimal. And even if the most likely annotation sequence is returned by the Viterbi algorithm, its computed probability is too small when there are further derivations of this annotation sequence.

As Dowell and Eddy have shown, this happens in practice and the effects are severe. For correct modeling with SCFGs, we need grammars that are syntactically, but not semantically ambiguous.

Semantic ambiguity in dynamic programming

Our treatment here extends to all dynamic programming algorithms that fall into the class known as algebraic dynamic programming (ADP) [GMS04]. However, some definitions must be refined, as the ADP approach uses so-called yield grammars rather than (S)CFGs. We will not introduce the ADP formalism here, but remain within the SCFG terminology. Still, we shall refer to some DP algorithms that are not based on SCFGs, where our treatment also applies.

6.2.4 SCFGs for RNA secondary structure analysis

We will further exemplify the above using the grammars $G1$ to $G6$ studied by Dowell and Eddy:

$$G1: \quad S \to aS\hat{a} \mid aS \mid Sa \mid SS \mid \varepsilon$$

$$\begin{aligned} G2: \quad & S \to aP^{a\hat{a}}\hat{a} \mid aS \mid Sa \mid SS \mid \varepsilon \\ & P^{b\hat{b}} \to aP^{a\hat{a}}\hat{a} \mid S \end{aligned}$$

$$\begin{aligned} G3: \quad & S \to aS\hat{a} \mid aL \mid Ra \mid LS \\ & L \to aS\hat{a} \mid aL \\ & R \to Ra \mid \varepsilon \end{aligned}$$

$$
\begin{aligned}
G4{:}\quad & S \to aS \,|\, T \,|\, \varepsilon \\
& T \to Ta \,|\, aS\hat{a} \,|\, TaS\hat{a} \\[1em]
G5{:}\quad & S \to aS \,|\, aS\hat{a}S \,|\, \varepsilon \\[1em]
G6{:}\quad & S \to LS \,|\, L \\
& L \to aF\hat{a} \,|\, a \\
& F \to aF\hat{a} \,|\, LS
\end{aligned}
$$

Dowell and Eddy showed that grammars $G1$ and $G2$ are semantically ambiguous, while $G3$ to $G6$ passed a partial test for non-ambiguity.

6.3 Results and Discussion

In this section, we first review some sources of ambiguity and suggest three ways to deal with it: ambiguity avoidance, testing for ambiguity, and, best of all when successful, a mechanical proof of absence.

6.3.1 Sources of ambiguity, and how to avoid them

We first study some standard patterns that give rise to ambiguity in our grammars. Thereafter, we make some observations with respect to the potential of testing procedures.

Three simple cases

Ambiguity does not sneak into our grammars by chance and non-awareness. There are two competing goals in grammar design, and both may foster ambiguity.

Small grammars have the advantage that they require fewer parameters and can be trained more quickly. Larger grammars allow a more sophisticated distinction of cases, hence providing a more fine-tuned model. However, if the underlying "distinct" cases lead to the same annotation sequence, then the grammar is ambiguous. This case is witnessed by grammar $G2$, where along with the introduction of base pair specific rules, another degree of ambiguity is introduced.

Often, non-ambiguous grammars require more space in their implementation via a CYK parser. For example, the non-ambiguous Wuchty algorithm (RNAsubopt, [WFHS99]) requires four tables for storing intermediate results, while the ambiguous Zuker-Stiegler recurrences (Mfold, [ZS81]) require only two. Two other cases in point are (a) and (b) below, while (c) shows that the non-ambiguous grammar can also be smaller.

Ambiguity can have many sources. Here, we present three common situations that lead us to write ambiguous rules, but can be easily avoided.

(a) Lists of adjacent elements of the same type, $\{S^n\}$:
Consider $S \rightarrow SS|U$ versus $L \rightarrow LS|S, S \rightarrow U$. The left-hand rule generates the language $\{S^n\}$ in an ambiguous way. For example, S^3 has the two derivations $\mathbf{S} \rightarrow S\mathbf{S} \rightarrow SSS$ and $\mathbf{S} \rightarrow \mathbf{S}S \rightarrow SSS$, where the generating nonterminal symbol is written in bold face. By contrast, with the right-hand rules there is only the derivation $\mathbf{L} \rightarrow \mathbf{L}S \rightarrow \mathbf{L}SS \rightarrow SSS$. The price for non-ambiguity is the new nonterminal symbol L, more parameters in the training set, and possibly another DP table in the implementation.

(b) Embedded elements, $\{a^m T a^n\}$:
Consider $R \rightarrow aR|Ra|T$ versus $R \rightarrow aR|V$, $V \rightarrow Va|T$.

For a given string $a^m T a^n$, the first two alternatives of the left-hand rule produce the initial string a^m and the terminal a^n in arbitrary order, while the right-hand rules produce a^m completely before a^n, allowing for only one derivation. An analog case is the embedding $\{a^m T b^n\}$. As above, an extra nonterminal symbol is required to achieve non-ambiguity.

(c) ε-rules, $L \rightarrow \varepsilon$:
Sometimes it is tempting to add a special case by using ε. Consider $L \rightarrow LS|S|\varepsilon$, which generates $\{S^n \mid n \geq 0\}$ by adding an ε-rule to the non-ambiguous rules in (a). Now, each string of length > 0 has two derivations, e.g. $\mathbf{L} \rightarrow \mathbf{L}S \rightarrow S$ and $\mathbf{L} \rightarrow S$. The solution here is to drop the middle alternative, $L \rightarrow S$.

The general case of ε-rules may be more tricky to handle. In general, all context free languages can be described without ε-rules, except possibly one for the axiom symbol. However, if ε-rules were used relentlessly, eliminating them without affecting the language may require a major redesign of the grammar.

Degree of ambiguity and consequences for testing

Dowell and Eddy showed that semantic ambiguity produces sometimes mildly, sometimes drastically false results. For example, they showed that the CYK algorithm for the semantically ambiguous grammar $G1$ does not give the optimal secondary structure for about 20% of a sample set of 2455 sequences. The same experiment for grammar $G2$ even gave a rate of 98% false results. The explanation of the difference in effect lies with the degree of ambiguity. The degree of ambiguity of a given annotation sequence is the number of its derivations, i.e. a degree of 1 means that this annotation sequence is not ambiguous. Depending on the involved productions, a particular string can have a constant, polynomial, or exponential number of derivations. The latter is the rule rather than the exception. It is easy to calculate for the left production rule of case (b) above that the sequence $\{a^mTa^n\}$ has $\binom{m+n}{n}$ derivations starting from S. Moreover, if derivations emerging from T are also ambiguous, the degrees of ambiguity multiply.

Studying sources of ambiguity helps to better understand the nature of the error. Depending on the grammar, certain types of RNA structures may have their probability split up over a large number of derivations, while others are unaffected. This makes it difficult to judge the amount of testing required, and the confidence achieved with the approaches presented in the next section.

6.3.2 Testing for ambiguity

Performing a test for semantic ambiguity allows us to obtain more confidence in the grammar, although testing cannot prove non-ambiguity, but only ambiguity.

Algorithmic arsenal for ambiguity testing

First, we create several variants of the Inside and Viterbi algorithms, which are our algorithmic arsenal for testing. $G1$ serves as the expository example here; for any other grammar, recurrences can be given in an analogous way:

Input: Sequence $x = x_1 \dots x_n$
Initialization: for $1 \leq i \leq n$
$S(i,i) = P_{S \to \varepsilon}$

Iteration: for $1 \leq i < j \leq n$

$$S(i,j) = \quad H \begin{cases} S(i+1,j-1) \circ P_{S \to x_i S x_j} \\ S(i+1,j) \circ P_{S \to x_i S} \\ S(i,j-1) \circ P_{S \to S x_j} \\ H_{i \leq k < j}\{S(i,k) \circ (S(k+1,j) \circ P_{S \to SS})\} \end{cases}$$

Scoring schemes:

Viterbi:	$H = \max$
	$\circ = *$ (multiplication)
	$P_{V \to \alpha} =$ rule probability
Inside:	$H = \sum$
	$\circ = *$ (multiplication)
	$P_{V \to \alpha} =$ rule probability
Counting:	$H = \sum$
	$\circ = *$ (multiplication)
	$P_{V \to \alpha} = 1$
Base pair	$H = \max$
maximization:	$\circ = +$
	$P_{S \to x_i S x_j} = 1$
	$P_{V \to \alpha} = 0$ for all other rules

By different interpretations of the operations H, $\circ$ and P, different scoring schemes can be plugged in. The recurrences may also be "conditioned" by annotating the symbol sequence x with a given annotation sequence s [DE04]. In that case, the rule $S \to aS\hat{a}$ is only allowed when the bases involved are annotated to form a base pair in s. This version of the recurrences will be denoted by G_s.

Using the first two scoring schemes, we obtain the Viterbi and the Inside algorithm. Using the other two, we obtain an algorithm for counting the number of derivations for the input string, and an algorithm for base pair maximization. Base pair maximization will not be used in the sequel, it is included only to indicate the swiftness of transition from SCFG modeling to other DP-based analyses. These algorithms are available at the accompanying website [AMB], where readers are welcome to practice their insight on ambiguity matters.

In the following, we write $G(\sigma, x)$ for running the CYK parser based on grammar G with scoring scheme σ on input x.

We recalled above that the formal treatment of semantic ambiguity requires a canonical representation of the objects under study. For RNA secondary structures, there is an obvious choice, our annotation sequences in the widely used dot-bracket notation (cf. Figure 6.1). Each secondary structure (excluding pseudoknots) is uniquely represented by such a string. The scoring scheme *Dotbracket* makes the CYK algorithm report all the structures it has analyzed for a given input sequence by producing their annotation sequences.

$$\begin{aligned} \textit{Dotbracket}:\quad & H = \text{collect} \\ & v \circ p = p(v) \\ & x +\!\!+\, y = xy \\[4pt] & P_{S\to x_i S x_j}(x) = \text{‘}\texttt{(}\text{’} +\!\!+\, x +\!\!+\, \text{‘}\texttt{)}\text{’} \\ & P_{S\to x_i S}(x) = \text{‘}\texttt{.}\text{’} +\!\!+\, x \\ & P_{S\to S x_j}(x) = x +\!\!+\, \text{‘}\texttt{.}\text{’} \\[4pt] & P_{S\to\varepsilon} = \text{‘’} \\ & P_{S\to SS}(y)(x) = x +\!\!+\, y \end{aligned}$$

Here, the objective function H merely collects lists of dot-bracket strings, each $P_{V\to\alpha}$ is a function adding dots or brackets to strings. $+\!\!+$ is string concatenation. $P_{S\to SS}$ is also string concatenation, but has the unusual type *String → (String → String)*, in order to fit into our recurrences smoothly. Here, $\circ$ expects a dot-bracket string as its left argument, a function as its right argument, and applies the latter to the former. For example, the function calls $P_{S\to SS}(P_{S\to gSu}(P_{S\to\varepsilon}))(P_{S\to aS}(P_{S\to\varepsilon}))$ generate the annotation sequence "`.()`" for the symbol sequence "`agu`". The reader may verify (using the aforementioned website) that $G1(\textit{Dotbracket}, \text{“}\texttt{agu}\text{”}) =$ ["`(.)`","`(.)`","`.()`","`...`","`...`", etc.], where the duplicate entries result from the ambiguity of $G1$. For example, the annotation sequence "`...`" is found 48 times.

Using these algorithms in concert for some RNA sequence x, we obtain from $G(\textit{Viterbi}, x)$ the probability of the most likely derivation for x, from $G(\textit{Counting}, x)$ the number of possible derivations, and from $G(\textit{Dotbracket}, x)$ the complete list of the annotation sequences associated with these derivations – possibly containing duplicates in the case of semantic ambiguity.

Testing procedures

Brute force testing: Checking for duplicates in $G(\textit{Dotbracket}, x)$. We can simply enumerate the dot-bracket representation of all structures exhaustively for a given input string and check for any repeats. There are some duplicates in $G(\textit{Dotbracket}, x)$ if and only if x can fold into an ambiguous annotation sequence (which may be precluded by its nucleotide content). Performing this test on a large number of inputs x should give a good hint whether ambiguity is present. Of course, enumerating the annotation sequences for all possible derivation trees creates voluminous output, and the automated check for duplicates requires some careful programming. Hence, this test is practical only for short sequences.

Sampling structures from sample sequences: Test $G(\mathit{Viterbi}, x) = G_s(\mathit{Inside}, x)$? Dowell and Eddy suggested a testing procedure that relies on a comparison of the results from the Viterbi and the Inside algorithms, where the latter is conditioned on the most likely annotation sequence s returned by the Viterbi run. $G_s(\mathit{Inside}, x)$ sums up probabilities over *all* derivations representing annotation sequence s. The tested equation therefore holds if and only if the annotation sequence s has exactly one derivation tree. If there are more than one, the Inside algorithm will return a higher probability than the Viterbi run, which indicates ambiguity of s (and hence G). Similarly, $G_s(\mathit{Counting}, x)$ directly computes the number of derivations for s, where a result larger than 1 signals ambiguity.

Dowell and Eddy suggest to run the test also for a sample of suboptimal annotation sequences for x. As a variant, we can do the same test based on a *minimizing* Viterbi run (setting $H = \min$). Since the minimizing Viterbi run gives us the least probable derivation tree, we may have a higher chance to find an ambiguous one (if present) than in the maximizing run.

In any case, this test works with samples of suboptimal annotation sequences for a test set of sequences, and it is difficult to give general guidelines how much testing is required. The four grammars $G3$ – $G6$ passed the Dowell-Eddy test in [DE04], and in the next section we shall prove their non-ambiguity. In this sense, we can state that this test has already worked quite well in practice. However, the eternal dilemma of testing persists – only if we confirmed the above equation for *all* x, semantic non-ambiguity would be assured.

Structure counting for sample sequences: Test $G(\mathit{Counting}, x) = R(\mathit{Counting}, x)$? An even stronger test is possible when we have a reference grammar R available that generates the same language and is known to be semantically non-ambiguous. Grammar G will produce counts that are larger than those of R if and only if G allows ambiguous derivations for x. Still, if this test succeeds, this does not imply non-ambiguity of G. But this test is much more thorough than our previous one, as the entire structure space of each tested x is analyzed. For example, a sequence of length 30 has an expected number of 175550 feasible structures [Gie00]. Thus, one run of this test has the testing power of 175550 runs of the previous one. Several non-ambiguous reference grammars for RNA are known – the critical part here is to assure that our grammar G to be tested describes the same language as R. Both grammars must impose the same restrictions on loop sizes, lonely base pairs, etc. This may be obvious in many cases, but in general, language equivalence is an undecidable problem in formal language theory.

Just-In-Time testing: Test $G(\mathit{Counting}, x) = R(\mathit{Counting}, x)$? While testing cannot guarantee the non-ambiguity of the grammar, we can convert the previous idea to a test that ensures for each application run that the results are not affected by ambiguity. Prior to running $G(\mathit{Viterbi}, x)$ for a given x, we test whether the property $G(\mathit{Counting}, x) = R(\mathit{Counting}, x)$ holds. This costs a constant factor in runtime, but solves the problem in the sense that when we get a positive test, we know the Viterbi result is correct for this input. If the grammar is ambiguous, this will be detected with the first application where it occurs.

6.3.3 Proving non-ambiguity

Proving the absence of ambiguity in a grammar is of course better than any test procedure.

Semantic ambiguity in dynamic programming is undecidable

Ambiguity of context free grammars is well-known to be algorithmically undecidable [CS63]. There exists no program that can determine for an arbitrary grammar G whether or not G is fl-ambiguous. Here, the problem is to decide whether a given SCFG is *semantically* ambiguous. It is not surprising that this problem is not easier:

Theorem 1 *Semantic ambiguity in dynamic programming is formally undecidable.*

Proof. We show that for a given CFG G there exists a DP problem and an associated canonical model such that the DP algorithm is semantically ambiguous if and only if the grammar is fl-ambiguous. Given an algorithm to decide ambiguity for DP problems, we could hence decide ambiguity for context free grammars, which is impossible. Details are given in Section 6.5. □

While this result rules out an automated proof procedure for arbitrary grammars used in SCFG modeling, there might still be the possibility to design such a procedure for a restricted class of grammars, say all grammars which describe RNA secondary structures. However, no such method is currently known.

Hand-made proof of non-ambiguity

A hand-made de-novo proof of the non-ambiguity of a new grammar G requires an inductive argument on the number of parses corresponding to the same annotation sequence. We constructed one such proof for the grammar published in [Gie00]. It is not mathematically deep, but rather a tedious exercise, and the likelihood to produce errors or oversights is high. An easier approach is the use of a known, non-ambiguous “reference” grammar R, such that $L(G) = L(R)$. By showing that a one-to-one mapping between parse trees of G and R exists, it is possible to prove the non-ambiguity of G. Such a proof remains manageable if the grammars are rather similar and the correspondence between derivations is easy to maintain. For grammars that are rather distinct, the proof is as messy as the de-novo proof.

Mechanical proof of non-ambiguity

We now present a mechanical technique that is a *partial* proof procedure for the case of modeling RNA structure with SCFGs: If it succeeds, it proofs non-ambiguity, if it fails, we do not know. We shall show that the method succeeds on several relevant grammars.

The technique described in the following comprises two steps. First, we remove the syntactic ambiguity of the grammar and reduce a possibly existent semantic ambiguity to fl-ambiguity. Then we use a parser generator to check the transformed grammar for fl-ambiguity. This test can prove non-ambiguity of a large number of grammars.

Ambiguity reduction. Paired bases can always also be unpaired – this creates the syntactic (good) ambiguity. For example, grammar $G1$ has four rules of the form $S \to aS$, one for each base A, C, G, U, and six rules of the form $S \to aS\hat{a}$ for the six valid base pairs. Used in concert, they create the "good" ambiguity that allows us to parse "CAAAG" either as "`(...)`" or as "`.....`".

Remember that the dot-bracket notation is a canonical representation for RNA secondary structure. For any G, we denote by G^* the transformed grammar that arises when we replace base pairs a, $\hat{a}$ by "`(`" and "`)`", and other base symbols by "`.`". Take for example

$$G5 : S \to aS \,|\, aS\hat{a}S \,|\, \varepsilon \quad \text{(11 productions)},$$

which is transformed to

$$G5^* : S \to \text{`.'}S \,|\, \text{`('}S\text{`)'}S \,|\, \varepsilon \quad \text{(3 productions)}.$$

This transformation removes the syntactic ambiguity of $G5$ by differentiating between paired and unpaired bases and reduces the semantic ambiguity – if present – to fl-ambiguity of $G5^*$. Note that the transformation from G to G^* works for any grammar for RNA structure, as long as we can identify the corresponding bases of a base pair.

Theorem 2 *Let G^* be derived from G according to the above rules. Then, G^* is fl-ambiguous if and only if G is semantically ambiguous.*

Proof. Every dot-bracket string describes exactly one possible secondary structure. If G^* is fl-ambiguous, there exist different derivations in G^* for the same dot-bracket string z.

Then, for an RNA sequence x compatible with z, using the corresponding productions there are different derivations in G which represent the same secondary structure z. This is equivalent to semantic ambiguity of G. If G^* is non-ambiguous, only a single derivation exists for every z in $L(G^*)$. A single derivation exists in G for a compatible RNA sequence x, and hence, G is semantically non-ambiguous. $\square$

Non-ambiguity proof. By the transformation described above, the task of proving semantic non-ambiguity of G is transformed to the task of proving fl-non-ambiguity of G^*. As stated above, this question is undecidable in general. However, compiler technology provides a partial proof procedure: If a deterministic parser can be generated for a grammar, then it is non-ambiguous [AU73]. We shall apply a parser generator to G^*.

Simply speaking, a parser generator takes a file with a context free grammar as input, and generates a program which implements the parser for this grammar. This parser must be deterministic, and, in contrast to our CYK parsers, it only exists for non-ambiguous grammars. There are many such generators available; we will focus on the class of $LR(k)$ grammars [Knu65] and their parser generators. A context free grammar is called $LR(k)$ if a deterministic shift reduce parser exists that uses k symbols of lookahead. By definition, an $LR(k)$ grammar is non-ambiguous, and for a given k it is decidable whether a grammar is $LR(k)$. This decision can be assigned to a parser generator. Given the grammar and the lookahead k, a parser generator tries to construct a parser that uses k symbols of lookahead. When successful, the non-ambiguity of the grammar is proved. When the grammar is not $LR(k)$, the generator will not be able to create a deterministic parser and reports this situations in form of "shift-reduce" and "reduce-reduce"-conflicts to the user. In this case, we do not know whether the parser generator might be successful for a larger k, and the question of ambiguity remains undecided.

Applications. For our experiments, we used the `MSTA` parser generator of the COCOM compiler construction toolkit [COC]. `MSTA` is capable of generating $LR(k)$ parsers for arbitrary k. Note that compiler writers prefer other parser generators like `yacc` [Joh75] and `bison` [BIS], which for efficiency reasons only implement $LR(1)$ parsers. We, however, are not planning to run the parser at all. Its successful construction is the proof of non-ambiguity; for applying our SCFG, we need the original grammar and its CYK parser.

`MSTA` accepts input files in the widely used `yacc` format. The following shows the input file for grammar $G5$:

```
%%
S : '.' S
  | '(' S ')' S
  |
```

Feeding this file into `MSTA` with $k = 1$ yields a deterministic shift-reduce parser for grammar $G5$. This proves that $G5$ is $LR(1)$, has a deterministic $LR(1)$ parser, and is therefore non-ambiguous.

Grammar	k	SR/RR conflicts
G1	1	24/12
G1	2	70/36
G1	3	195/99
G2	1	25/13
G2	2	59/37
G2	3	165/98
G3	1-3	4/0
G3	4	16/0
G3	5	0/0
G4	1	0/0
G5	1	0/0
G6	1	0/0
G7	1-6	5/0
G7	7	0/0
G8	1	0/0

Table 6.1: Number of shift-reduce (SR) and reduce-reduce (RR) conflicts when feeding example grammars G1 to G8 into parser generator `MSTA`. A 0/0 entry indicates a successful proof of non-ambiguity. Note that for increasing k, the number of conflicts may remain constant or even grow before it goes down to 0/0.

Table 6.1 summarizes the results for grammars $G1$ to $G6$. For $G1$ and $G2$, the results only show that both grammars are not $LR(1)$, $LR(2)$ or $LR(3)$. Although no real proof, the number of conflicts growing with k gives a strong hint at the ambiguity of these grammars.

Grammar $G3$ is $LR(5)$ and $G4$ to $G6$ are $LR(1)$. Therefore, we have proved mechanically that the four "good" grammars studied by Dowell and Eddy are definitely non-ambiguous. The two additional grammars $G7$ and $G8$ from [DE04], not reproduced here, were also included in the study and proved to be non-ambiguous.

In Table 6.1 we also report on the number of conflicts found by the parser generator for increasing values of k. While the nature of these conflicts is not relevant for us, the table shows that various behaviors are possible. Their numbers may grow ($G3$) or may remain constant ($G7$) before they go to zero for some k.

Experience from a larger example. The parser generator test works quite well for the small grammars we presented so far. However, there exist cases where, due to the finite lookahead of the generated parser, the parser generator reports conflicts while the grammar is in fact non-ambiguous. In the following, we report on one such case, and show how to deal with this situation.

In his thesis [Vos04], Björn Voss introduced a new grammar that promises to handle dangling bases of multiloop components in a non-ambiguous way. With 28 nonterminal symbols and 79 rules, the grammar is quite large. In such a case, mechanical assistance is strongly required. Our first approach with the parser generator succeeded, except for one small part of the grammar for which it reports a conflict. Figure 6.2 shows two example derivations where this conflict occurs.

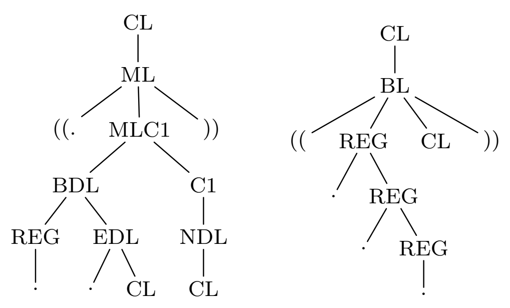


Figure 6.2: Two example derivations of a grammar taken from [Vos04]. The left side is part of a multiloop derivation, the right side part of a left bulge.

The central nonterminal of the grammar is CL, which splits up into closed structures like hairpin loops, bulges, and multiloops. Due to the necessity to handle dangling bases in a non-ambiguous way, the rules for multiloops are the most complicated of this grammar. Altogether, 11 nonterminals and 35 rules are used exclusively for this purpose. The construction of these rules guarantees, that every derivation of a multiloop *must* lead to at least two closed substructures. One of these derivations is shown on the left side of Figure 6.2. Therefore, a derivation of a multiloop can by no means conflict with a derivation of a left bulge, which must include a *single* closed substructure. However, the parser generator runs into a conflict here. Consider the following annotation sequence:

```
((...((...((...))..((...))..))))
ggaaaggaaaggaaaccaaggaaaccaacccc
```

Here, the string "`((...((`" appears two times in the annotation sequence. The first appearance denotes a left bulge, the second the beginning of a multiloop. The decision which of these two is given can only be made after the first closed substructure is completely processed. Since the generated parser can only read a limited number of input characters ahead (k), the parser generator is not able to construct a deterministic parser for this situation and reports a conflict.

However, we can circumvent this problem by extending the alphabet of the annotation sequence by an additional character (say, '`:`') for unpaired bases in left bulges[1]:

```
((:::((...((...))..((...))..))))
ggaaaggaaaggaaaccaaggaaaccaacccc
```

Since a multiloop's derivation can not conflict with that of a bulge, this modification does not alter the ambiguity or non-ambiguity of the grammar. The important difference is that positional information is turned into symbolic information.

After this modification, the parser generator runs smoothly through the grammar, which proves its non-ambiguity.

6.4 Conclusions

In this work, we have presented testing methods and a partial proof procedure to analyze the semantic ambiguity of SCFGs. We have shown that the problem is not decidable for dynamic programming over sequence data in general, and that hence there is no standard solution that works for all cases. It remains open whether specifically for the class of grammars that describe RNA secondary structure, this problem is decidable. We have proposed several tests, and a partial, mechanical proof procedure. We mechanically proved that the six grammars that passed Dowell and Eddy's test for non-ambiguity are actually non-ambiguous. We also reported on a proof of the non-ambiguity of a new and large grammar for RNA secondary structures, whose sophistication makes it inadvisable to rely solely on human reasoning.

We want to point out that the non-ambiguity proofs for the grammars studied here do not solve the problem of ambiguity for modeling of RNA secondary structures once and for all. New scientific interests and research questions will always demand new grammars. An example is a grammar that is restricted to a special class of structures of an RNA family. This allows us to define a thermodynamic matcher, which uses the minimum free energy as a scoring scheme and focuses only on a specific realm of secondary structures. Here, for every new RNA family, a new grammar must be devised. This demonstrates a

[1]For the same reason, this modification is also necessary in the rules for internal loops.

continuous need for new, specialized grammars. Every time we develop a new grammar, the dragon of ambiguity raises its head, but with the weapons presented here, we can be confident to defeat it.

6.5 Ambiguity in DP is undecidable

Dynamic programming is a very general programming technique, and its scope is not precisely circumscribed. We prove our undecidability result for the well defined class of algebraic dynamic programming [GMS04] problems, which of course implies undecidability in general. Simply speaking, a DP problem is given by a grammar G and a scoring scheme σ (not necessarily stochastic), as was exemplified in Section 6.3.2.

Theorem 3 *Semantic ambiguity in dynamic programming is formally undecidable.*

Proof. For an arbitrary context free grammar G, we can construct a DP problem where $L(G)$ serves as the canonical model, and show that the context free grammar G is ambiguous if and only if the DP problem is semantically ambiguous.

Let G be a context free grammar. Without loss of generality, we can assume that each production is either of the form $A \to t$, generating a terminal symbol, or of $A_0 \to A_1 \dots A_n, n \geq 0$, generating a series of nonterminal symbols. We construct a scoring scheme σ for grammar G such that $G(\sigma, x)$ computes all derivation trees for x. Similar to the scoring scheme *Dotbracket*, we set $H =$ collect and $x \circ f = f(x)$. For each production π we use a unique tree label T_π. We define $P_{A_0 \to A_1 \dots A_n}(a_n) \dots (a_1) = T_{A_0 \to A_1 \dots A_n}(a_1, \dots, a_n)$, and $P_{A \to t} = t$.

$$\begin{array}{ll} \textit{Derivation trees} & H = \text{collect} \\ & x \circ f = f(x) \\ & T_\pi = \text{unique tree label} \\ & P_{A_0 \to A_1 \dots A_n}(a_n)...(a_1) = \quad T_{A_0 \to A_1 \dots A_n}(a_1, ..., a_n) \\ & P_{A \to t} = t \end{array}$$

By construction, $G(\sigma, x)$ constructs the list of all derivation trees for x. The canonical mapping ν (from derivation trees to their derived strings) is simply given by $\nu(T_{A_0 \to A_1 \dots A_n} (a_1, \dots, a_n)) = \nu(a_1) \dots \nu(a_n)$ and $\nu(t) = t$. By construction, the domain of ν are the derivation trees of G, its range is $L(G)$. Hence, ν is injective if and only if G is non-ambiguous. Could we formally decide the semantic ambiguity of an arbitrary DP problem, we could do so for the problem given by G and σ, and hence, ambiguity of context free languages would be decidable. $\square$

6.6 Ambiguity in Locomotif

In the Locomotif approach, grammar building blocks are fused together to obtain the overall motif grammar. Therefore, in order to ensure ambiguity, every grammar building block in itself must be nonambiguous and their connection must not lead to ambiguity. In our case, semantic ambiguity arises, if the RNA motif is found several times at the same location in the target sequence by the use of different grammar rules from the same or different building blocks. For most building blocks, this is excluded by restrictive grammar rules that do not allow for options. An internal loop has a precise structure with variations only in the lengths of its loop regions.

```
iloop$_i = (il <<< lbase -~~ region ~~~ motif_il$_{i+1} ~~~ region
                  ~~- lbase) ‘with‘ basepairing
motif_il$_{i+1} = (sr <<< lbase -~~ motif$_{i+1} ~~- lbase)
                       ‘with‘ basepairing
```

It starts with a basepair followed by loop region on both strands followed by another basepair. It does allow for syntactic ambiguity, because the `region` and `lbase` parsers can be applied to different sections of the target sequence. (Or the other way round, the same part of the target sequence can be folded into different parts of the RNA motif). Yet, it does not allow for semantic ambiguity, because the parsers can not be mixed up. We have exactly one base of a pair, followed by an unpaired region, followed by exactly one basepair enclosing the rest of the motif, followed by another unpaired region, followed by the pairing partner of the first base. A specific part of the target sequence can only be folded into an internal loop in exactly one way. The same argument holds for the bulge loop, hairpin loop, multiloop and single strand. Even though these building blocks have a different composition, they also exhibit a clearly defined structure that does not allow for semantic ambiguity.

The Stem, ClosedStruct and ClosedEnd building blocks by themselves also do not create semantic ambiguity. They do contain recursive rules and are thus not as restrictive in structure. Yet, a particular instance of the stem building block (having e.g. 4 basepairs) can only be constructed by using the first rule 3 times and the second rule 1 time:

```
stem$_i = ((sr <<< lbase -~~ stem$_i ~~- lbase |||
            sr <<< lbase -~~ motif$_{i+1} ~~- lbase) ‘with‘ basepairing)
```

A ClosedStruct allows for any combination of internal loops, stems and bulge loops. Yet, a particular combination can only be constructed by a particular set of rules. The same holds for the ClosedEnd which additionally contains multiloops and hairpin loops.

A problem arises in the combination of these variable building blocks. If an internal loop is placed next to another building block, no semantic ambiguity can occur because of the precise structure of the internal loop. Yet, if a stem is placed next to a ClosedStruct, a basepair can potentially be generated by either of these building blocks. Both contain rules to generate stacking regions. If the size of the stacking region of the stem is not restricted, a basepair can either be generated via the stacking region of the stem or of the ClosedStruct. Thus, there are several ways to produce the exact same basepair and we are dealing with semantic ambiguity. If an exact size is given for the stem, all its basepairs are enumerated and we are safe. In all other cases, i.e. minimum and/or maximum number of basepairs or no size restrictions, semantic ambiguity occurs whenever the neighboring building block also carries the potential for ambiguity. This potential is present in stems without exact size restrictions and in the compound building blocks, i.e. the ClosedStruct and the ClosedEnd. Any direct combination of these types of building blocks will lead to semantic ambiguity. While one cannot combine two compound building blocks, it is possible to use a ClosedStruct or ClosedEnd next to a stem or two stem next to each other.

It would be possible to prohibit the connection of these building blocks, yet I chose not to do so. The reason is that sometimes the user might just want to define a motif of this kind. An example would be a hairpin-like structure that contains at least 3 basepairs in the stem and then no further information is given until it ends in a particular hairpin loop. Here, it is necessary to position a stem next to a ClosedStruct. The same thing occurs whenever the user wants to input sequence motifs in stems which are always placed at the beginning of the stems. Thus, longer stems might have to be divided into different parts. In all these cases, semantic ambiguity can be avoided by restricting the stem to an exact size. Since the user might change the size restrictions at the very end, I cannot simply prohibit the use of a stem building block next to a another stem or compound building block. A useful measure would be to employ an ambiguity filter before the motif is translated to XML and then furthermore to the ADP grammar. It could either require the user to change the size restrictions, possibly making the motif definition more complicated as a division of the stem into several building blocks might be necessary. Also, this might restrict the user in a way s/he does not wish. Alternatively, a warning message might be given to the user informing him/her about potential ambiguity issues. This would raise awareness of such problems in biosequence analysis in our users and is in my opinion the method of choice.

Chapter 7

Future Work

The current version of the Locomotif system allows the user to define RNA motifs and search them in RNA or DNA sequences. Any non-pseudoknotted motif structure can be defined, enhanced by size or sequence restrictions, and an executable matcher will be generated. Nevertheless, the system can still be improved in many ways.

7.1 Visualization

Single Strands A single strand is drawn as a straight line with a surrounding rectangular box. Whenever a single strand is used to connect two structure parts, the size of the surrounding box depends on the location of the structure parts. In the worst case, both are in line with each other as shown in Figure 7.1. Then, the surrounding box collapses onto the connecting line making it nearly impossible to select the building block. (Actually, the only remedy is to zoom into the view several times and try to select the line). Especially in these cases, but also in general, it would be much nicer, if the single strand was visualized as a curved segment appropriate to the surrounding structure. Of course, to draw such a curve, we need to know about the exact location of all structure parts to avoid conflicts. If the single strand is located on top of two motif parts, it needs to be drawn above them. In other cases, an S - shaped curve or even more complex connections would be needed. Therefore, to implement this improvement, the whole visual structure must be taken into account and an optimal path must be calculated.

7.2 Program enhancement

Whenever essential changes to the editor are made, we also need to adapt the translation to XML and the corresponding XML schema. Also, the translation to ADP on the server

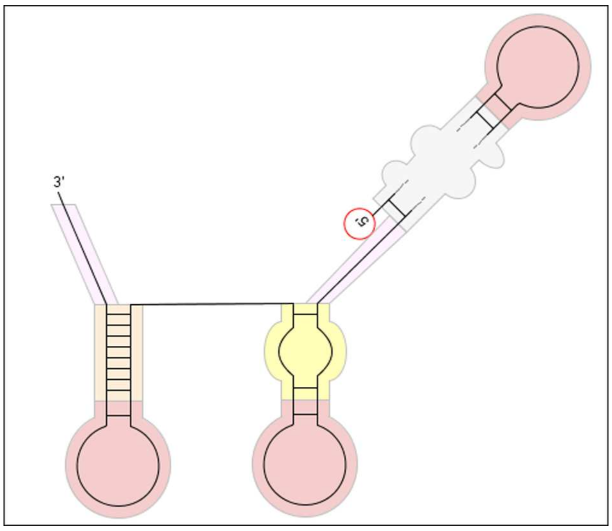


Figure 7.1: The box of the single strand connecting the last two motif parts (internal loop and stem) is barely visible.

must be changed to include new functionality. Compatibility with previous versions is ensured as long as only optional features are added to the system. For the ADP translation, we need to be sure that no elements are required to be present in the XML document which were not included in previous versions. This would lead to *NullPointerExceptions* when trying to translate them. In that case, we must require the use of the newest version of the program. As long as it is only offered as a Web Start application, this should be no problem except for the unlikely case of someone using the editor while changes are made on the server.

Simple pseudoknots As mentioned above, it is only possible to define pseudoknot-free motifs. Pseudoknots result from non-nested basepairings within an RNA structure. These types of motif components occur frequently in nature and it would be desirable to include them as far as possible. There are several classes of pseudoknots differing in complexity regarding both their visual representation and the computational effort necessary to fold a sequence. For some classes, it is not possible to visualize them in two dimensions without highly intertwined tertiary connections. In other cases, even though the connections could be projected to 2D without intersections, no building block could represent them. Yet, there are simple types of pseudoknots for which a building block could be designed. This is possible as long as the entire range of the pseudoknot is confined within one building block (see Figure 7.2).

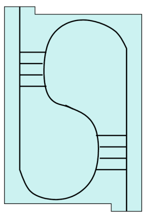


Figure 7.2: This building block represents a class of simple pseudoknots where the bases of the hairpin loops pair with a previous or following single strand to form a stacking region.

The two single exits of the building block would have to be connected to other structure parts using (possibly empty) single strands. Thus, no internal definition of other motif building blocks contained within the pseudoknot is possible. This would require an even more complex building block with additional double-stranded exits within the loops.

The only way to realize far reaching tertiary connections would be to use a line mechanism similar to drawing the single strands to connect individual pairing bases. Some thought would have to be given as to whether and how that could be realized visually and to what extent an ADP grammar could include it. A good starting point would be to use the classes of pseudoknots for which efficient matcher programs can be generated based on [RG04] as a guideline and then try to find a visualization for them.

Groups Another thing that should be taken into account is the fact that RNA motifs often belong to families whose members are similar in structure. The IRE e.g. exists in two forms: one having a single C-bulge on the 5' strand, the other having an internal loop starting with a C on the 5' strand. It is no problem to include such groups of building blocks within the ADP code for the matcher. Therefore, we should provide a way for the user to define a choice of building blocks at a certain location within the motif. To do this, we would have to adapt both the internal data structures as well as the visualization. For the storage level, this is rather straightforward. An *RnaShape* currently stores a specific neighbor reference to the next building block. A data class must be added similar to the *DoubleSingleShape* that acts as a placeholder neighbor which is stored in the *RnaShape*. This placeholder must hold a reference to all building blocks that can occur at the location.

To include groups in the visualization, we have to develop a means to represent optionality. The easiest way would be to use a placeholder visual building block with a general look that does not reflect the types of building blocks it contains. Then, the editing interface must provide means to specify constraints for all the building blocks represented by the placeholder. A better way would be to show all types of building blocks on the screen. An option could be to use a placeholder shape for the general view. Then, upon selecting the placeholder, the rest of the motif structure fades into the background. At the same time, all the building blocks contained within the placeholder could be visualized next to each other in a box standing out from the background. One of them would be drawn in the center in normal size and the user can open its editing interface by selecting it. All other members of the group would be drawn in a miniature size next to the current selection and can be chosen by clicking on them. Then, the view would change with a new selection in the middle and the previous one as a miniature to the side. That way, the user can interact directly with each building block of the group, yet the general view of the motif is not overloaded with information.

7.3 Improve interactivity

Direct interaction with structure parts In the current version of the software, it is only possible to interact with individual building blocks. These can be moved around, rotated, added to structure parts and removed from them. Also, the entire motif can be moved or rotated. Yet, it is not possible to interact with a selection of building blocks at the same time. The reason for this restriction is the fact that the entire structure is stored within one *RnaStructure* object. All interaction with the view is handled within the *DrawingSurface* and given to the *RnaStructure* if necessary. A building block attached to the mouse cursor is treated directly within the *DrawingSurface*. Specific methods are available to rotate it, add it, remove it and so on. Whenever the user interacts with the entire structure, similar methods are called within the *RnaStructure*. In order to separate different motif parts from each other, as many *RnaStructure*s as necessary could be used, each responsible for the operation acting on its members. Traversal methods would have to be adapted: instead of using the *startelement* and the members of the *furtherstarts Vector* as the origins for the traversal, a start element for each *RnaStructure* would have to be stored eliminating the need for the *furtherstarts* alltogether. Then, a problem arises whenever a building block is to be attached to the structure. Currently, only the neighbor references are updated and the block is stored in the *RnaStructure*. Using several storage capacities, we would need to know which *RnaStructure* the building block is added to. This could be achieved by giving each *RnaShape* a reference to its *RnaStructure*.

The implementation of direct interaction with different structure parts has several advantages other than just moving them around individually. We could treat these objects similar to building blocks and allow for the addition of an entire mobile structure part to another one. Just as well, an entire structure part could be detached or deleted at once.

Selection could be done using well-known concepts of drawing a selection rectangle with the mouse cursor or based on the Shift and/or Ctrl keys. Several problems with removals of building blocks could be solved. A building block could be removed from within the structure, because one of the remaining parts could be picked up and added to the other one. Also, we would not need to require that all single strands must be removed, but only that the structure part we are deleting from is not connected to any more single strands. Overall, adding this feature gives the user much more flexibility, but it will require some effort implementing it. Much of the *DrawingSurface* must be adapted, the *RnaShape*s have to be adjusted and a means to treat *RnaStructure*s as building blocks has to be introduced.

7.4 Search quality and time

Ambiguity checking In Chapter 6, I introduced the ambiguity problems prevalent in the generated motif grammars of the Locomotif system. While I cannot prevent the construction of potentially ambiguous motifs, a check for ambiguity could be included. Since the occurrence of problems is restricted to combinations of specific building blocks, it suffices to compare each pair of neighbors. If a combination of stems or a stem and a compound building block occurs, we must check the length restrictions within the stems. As long as an exact length is given, everything is fine. Otherwise, the user should be informed of the ambiguity issue and be recommended to add a size restriction to the stem in question.

Result significance In order to evaluate the results of a search program, we need to take the significance of the motif into account [MG02]. If the user defines a very simple motif based e.g. just on a stem and a hairpin loop, it is not surprising to find hits even in a small arbitrary sequence. Currently, we are only searching the minimum free energy, i.e. only the best hit(s) are produced. If we do allow for suboptimal structures, then in such cases, we will get even more hits of similar energy values. Therefore, a measure for the significance of the results should be included. This measure must take both the motif definition and the target sequence into account. Using just the motif definition, we could already inform the user about a potential significance issues of the motif and recommend him/her to further restrict the definition. Then, with the results, the target sequence can be added into consideration to give a complete significance analysis as introduced for thermodynamic matchers in [HHG06].

Search efficiency Both time and space complexity depend on the size and complexity of the RNA motif and the length n of the target sequence. An upper bound for the space needed is $O(|G| * n^2)$ and the search time is bounded by $O(|G| * n^3)$ with $|G|$ indicating the number of tabulated nonterminals.

At the moment, the search is done entirely by the generated matcher program. By using the ADP compiler instead of the ghc, we were able to improve their efficiency based on the built-in optimization of the number of nonterminals to be tabulated. The resulting programs are sufficient to search small sequences, but not good enough for sequences on chromosome or genome level. One way to improve the runtime of the generated matchers, is to use a window functionality that searches a sequence based on window segments. The compiler already includes an option to use this function within the generated matcher, so I only need to include it as an option in the interface for running a matcher.

In addition, a filtering step based on sequence motifs specified for some building blocks can be included in the system. A script could be written that parses the XML document and extracts all relevant sequence motifs. A screen would have to be added that determines whether the combined sequence motif is a significant filter. Then, using an established sequence search algorithm, all occurrences of the sequence motif or, if possible, the combination of different ones, will be determined. Then, the generated motif search program only needs to be run around those hits. The whole procedure should be automated so that a user must only upload the sequence and wait for the results.

An additional opportunity to speed up the search time lies within the motif structure itself. For example, running an IRE matcher on an arbitrary sequence of 10000 bases (using the window functionality described above) takes about one second. Running a simple matcher including a multiloop on the other hand, requires twelve seconds for the same sequence (in window mode). Thus, in case of large motifs, we could allow the user to define a search order for different motif parts. Then, different matcher programs for the individual parts could be generated, and could be run in the specified search order. Thus, we would establish a structural filter for the search using the most important structure part to identify the best hits within the sequence as a starting point around which to search for the other parts. In the end, a standard search would have to be computed with a matcher for the entire motif, but we could restrict the sequence length for that search.

This might also work as an automated filter for large motifs that could be decomposed into different parts by the system. Yet, we have to be sure that these parts are significant enough to act as filters and be aware of the fact, that a hit will always be produced unless impossible because of hard restrictions such as sequence motifs. It is not straightforward to define a barrier for a good MFE value as it depends on the structure of the motif being searched and the target sequence. Here, we need to evaluate both the significance of a motif matcher and the significance of a hit to establish a useful filter.

Additional run parameters As mentioned above, the generated matchers can be run with additional parameters not yet included in the Locomotif system. The compiler automatically provides a number of parameters such as the window functionality or the option to search for suboptimal structures. Other possibilities include restrictions on the energy range, a maximal loop length or output formatting options. As the compiler generates these options for every matcher anyway, I intend to enhance the interface for running a matcher to include them. This will require some effort in adapting the webservice functions, but should not pose a big problem.

7.5 Application

Finally, another important aspect of future work for the Locomotif system is a survey of how well it works. This should take both aspects of the software into account: how well does the interactive editor work and how do the generated programs perform. The first aspect is best tested by letting the experts work with the program and obtaining their feedback as to what is missing and needs to be done. Some answers to efficiency issues can be obtained by comparison with other applications of ADP for RNA analysis. Yet, we need to apply the generated matchers on several good motif candidates that allow us to evaluate their performance and compare it to other motif search programs based both on the quality of the results and the efficiency of the programs.

7.6 Known Issues

Debugging the user interface. While I already performed extensive debugging of the graphical editor, some bugs still remain within the user interface. These do not occur frequently, but can be triggered by complex sequences of interacting with a motif structure. Combinations of rotations, building blocks snapping to a *Magnet* and movements of the structure can lead to uncaught exceptions. The program does not crash, but the graphical view does not function properly anymore. Using the "New" or "Open" buttons will recreate a correct interface, but the current project cannot be saved.

In an interactive graphical user interface, it is not possible to automate the debugging process. Any problems depend on a certain order of user invoked commands that must be taken into account when determining the source of an error. As a developer interested in debugging the program, I can take note of all interactions to find the trigger for an exception. A biologist using the system on the other hand, usually does not remember the exact handling steps performed. Therefore, I intend to implement a logging level that captures all user interactions. Then, any user that experiences such problems can be asked, if s/he is willing to share the logging information to alleviate future debugging work.

Compilation In some rare cases, the compiler seems to enter endless loops while compiling a motif. On the BiBiServ, the compiler was installed in May 2006 and has not been updated since. The reason for this is the fact that first of all, installation of the compiler on the server is not trivial and must be done by the BiBiServ administrators. Also, with every compiler update, some minor modifications of the ADP syntax might be necessary. A newer version of the compiler exists that can usually compile the same motif, but I cannot be sure that other rare instances do not cause similar problems. Furthermore, we recently discovered a bug in the `iupac_base` parser that leads to incorrect folding energies. Since Peter Steffen already fixed this bug, I intend to ask for an update of the compiler as soon as I adapted the ADP syntax to the requirements of the new version. Still, some issues might remain and extensive testing of all known issues and common motifs must be performed, possibly demanding further compiler updates.

Chapter 8

Conclusion

I developed a graphical programming system as a means for biologists to write search programs for RNA motifs without the need to learn any advanced computer science skills. While descriptor-based motif search programs are available [MEG$^+$01, GSKS01], they require the motif definition via a cryptic (for biologists) specification language, and do not implement the established thermodynamic model within the search algorithm.

In my work, I have used and combined several programming paradigms in a productive way. Taking the graphical information exchange used by biologists as a guideline, I analyzed common RNA motifs to determine how to decompose them into visual building blocks. I designed the different types of building blocks and came up with the idea to use a drag and drop mechanism with magnets attracting the visual blocks to the motif structure. I implemented the graphical editor including its internal translations to XML and shape strings. The resulting user interface is easy to use and available as a Java Web Start application. I achieved the convenience of use through complex graphics implementations based on the java.awt.geom package. The object-oriented approach allowed me to handle complexity on distinct levels for information storage, visual representation and user interaction.

While the graphical editor is the frontend to the user, the main idea for the programming system stems from our work on declarative programming in biosequence analysis. The composition of RNA motifs from building blocks is visible in similar repetitive grammar blocks in ADP programs for general RNA folding. The semantics of the graphical building blocks closely resemble the semantics of the declarative level.

Three components are essential to build a correct and efficient molecular matcher, and in my work, they are brought together from three different sources:

- The actual *motif definition* is provided by the domain experts graphically, and is translated to the control structure of a dynamic programming algorithm. By itself, this constitutes a purely combinatorial search method without optimization criteria, and with potentially exponential runtime complexity.

- The underlying *physical model* is imported from a library of evaluation algebras used already in other folding programs; I defined its (nontrivial) integration into the control structure of the combinatorial search algorithm within the semantics of the graphical building blocks.
- Good and often optimal *asymptotic runtime efficiency* of the generated motif matcher is contributed by the compiler, whose table design algorithm is often superior to human efforts on the larger examples.

I integrated the programming system in a client-server architecture, eliminating any need for the user to install the compiler or to run the matchers via the command line.

The software environment has reached a stage where it is time to step back from the implementation details and let our users play with it[1]. Even though some improvements are still in plan for the graphical editor, a biologist can use it to define and search RNA motifs. Hopefully, this will be the beginning of a fruitful dialogue giving me feedback on how to continue in this line of work.

[1]Locomotif is available at http://bibiserv.techfak.uni-bielefeld.de/locomotif

Bibliography

[AMB] Effective ambiguity checking in biosequence analysis. http://bibiserv.techfak.uni-bielefeld.de/adp/ambiguity/.

[AU73] A.V. Aho and J.D. Ullman. *The Theory of Parsing, Translation and Compiling.* Prentice-Hall, Englewood Cliffs, NJ, 1973. I and II.

[BFF$^+$05] A. F. Bompfünewerer, C. Flamm, C. Fried, G. Fritzsch, I. L. Hofacker, J. Lehmann, K. Missal, A. Mosig, B. Müller, S. J. Prohaska, B. M. R. Stadler, P. F. Stadler, A. Tanzer, S. Washietl, and C. Witwer. Evolutionary patterns of non-coding RNAs. *Th. Biosci.*, 123:301–369, 2005.

[BIS] Bison parser generator. http://www.gnu.org/software/bison/.

[BKV96] B. Billoud, M. Kontic, and A. Viari. Palingol: a declarative programming language to describe nucleic acids' secondary structures and to scan sequence databases. *Nucleic Acids Res.*, 24(8):1395–1403, 1996.

[BZ04] V. Bafna and S. Zhang. FastR: Fast database search tool for non-coding RNA. *CSB 2004 Proceedings*, 2004.

[Cho56] N. Chomsky. Three models for the description of language. *IRE Transactions on Information Theory*, 2:113–124, 1956.

[COC] Cocom tool set. http://cocom.sourceforge.net/.

[CS63] N. Chomsky and M.P. Schützenberger. The algebraic theory of context-free languages. In P. Braffort and D. Hirschberg, editors, *Computer Programming and Formal Systems*, pages 118–161. Amsterdam: North-Holland, 1963.

[Dan02] T. Dandekar, editor. *RNA, the Underestimated Molecule.* Springer, 2nd edition, 2002.

[DE04] R.D. Dowell and S.R. Eddy. Evaluation of several lightweight stochastic context-free grammars for RNA secondary structure prediction. *BMC Bioinformatics*, 5(71), 2004.

[DHS84] J. Devereux, P. Haeberli, and O. Smithies. A comprehensive set of sequence analysis programs for the VAX. *Nucleic Acids Res.*, 12:387–395, 1984.

[DLO97] M. Dsouza, N. Larsen, and R. Overbeek. Searching for patterns in genomic data. *TIG*, 13(12):497–498, 1997.

[EBH$^+$01] V.A. Erdmann, M.Z. Barciszewska, A. Hochberg, N. de Groot, and J. Barciszweska. Regulatory RNAs. *Cell. Mol. Life Sci.*, 58:960–977, 2001.

[Edd01] S. R. Eddy. Non-coding RNA genes and the modern RNA world. *Nature Reviews*, 2:919–929, December 2001.

[Edd02] S. R. Eddy. A memory-efficient dynamic programming algorithm for optimal alignment of a sequence to an RNA secondary structure. *BMC Bioinformatics*, 3(18), July 2002.

[Gie00] R. Giegerich. Explaining and controlling ambiguity in dynamic programming. In *Proc. Combinatorial Pattern Matching*, volume 1848 of *Springer Lecture Notes in Computer Science*, pages 46–59. Springer, 2000.

[Gil86] W. Gilbert. Origin of life: The RNA world. *Nature*, 319:618, 1986.

[GL01] D. Gautheret and A. Lambert. Direct RNA motif definition and identification from multiple sequence alignments using secondary structure profiles. *J. Mol. Biol.*, 313:1003–1011, 2001.

[GMC90] D. Gautheret, F. Major, and R. Cedergren. Pattern searching/alignment with RNA primary and secondary structures: an effective descriptor for tRNA. *Comput. Appl. Biosci.*, 6(4):325–331, 1990.

[GMS02] R. Giegerich, C. Meyer, and P. Steffen. Towards a discipline of dynamic programming. In S. Schubert, B. Reusch, and N. Jesse, editors, *Informatik bewegt*, GI-Edition - Lecture Notes in Informatics, pages 3–44. Bonner Köllen Verlag, 2002.

[GMS04] R. Giegerich, C. Meyer, and P. Steffen. A discipline of dynamic programming over sequence data. *Science of Computer Programming*, 51(3):215–263, 2004.

[GS06] R. Giegerich and P. Steffen. Challenges in the compilation of a domain specific language for dynamic programming. In *Proceedings of the 2006 ACM Symposium on Applied Computing*, 2006.

[GSKS01] S. Gräf, D. Strothmann, S. Kurtz, and G. Steger. HyPaLib: a Database of RNAs and RNA Structural Elements defined by Hybrid Patterns. *Nucleic Acids Res.*, 29(1):196–198, 2001.

[GVR04] R. Giegerich, B. Voss, and M. Rehmsmeier. Abstract Shapes of RNA. *Nucleic Acids Res.*, 32(16):4843–4851, 2004.

[HFS+94] I. L. Hofacker, W. Fontana, P. F. Stadler, S. Bonhoeffer, M. Tacker, and P. Schuster. Fast Folding and Comparison of RNA Secondary Structures. *Monatshefte f. Chemie*, 125:167–188, 1994.

[HHG06] T. Höchsmann, M. Höchsmann, and R. Giegerich. Thermodynamic matchers: Strengthening the significance of RNA folding energies. In *Computational Systems Bioinformatics, Proceedings of the Conference CSB 2006*, August 2006.

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

[Joh75] S.C. Johnson. YACC: Yet another compiler compiler. *Computing Science TR*, 32, 1975.

[KE03] R. Klein and S. Eddy. RSEARCH: Finding homologs of single structured RNA sequences. *BMC Bioinformatics*, 4(44), 2003.

[KEBM95] Z. Kikinis, R. S. Eisenstein, A. J. E. Bettany, and H. N. Munro. Role of RNA secondary structure of the iron-responsive element in translational regulation of ferritin synthesis. *Nucleic Acids Res.*, 23(20):4190–4195, 1995.

[Knu65] D.E. Knuth. On the translation of languages from left to right. *Information and Control*, 8(6):607–639, 1965.

[LBA02] D. Laslett, Canback B., and S. Andersson. BRUCE: a program for the detection of transfer-messenger RNA genes in nucleotide sequences. *Nucleic Acids Res.*, 30(15):3449–3453, 2002.

[LDM94] F. Lisacek, Y. Diaz, and F. Michel. Automatic identification of group I intron cores in genomic DNA sequences. *J. Mol. Biol.*, 235:1206–1217, 1994.

[LE97] T.M. Lowe and S.R. Eddy. tRNAscan-SE: a program for improved detection of transfer RNA genes in genomic sequence. *Nucleic Acids Res.*, 25(5):955–964, 1997.

[LE99] T.M. Lowe and S.R. Eddy. A computational screen for methylation guide snoRNAs in yeast. *Science*, 283:1168–1171, 1999.

[LG90] A.I. Lamond and T.J. Gibson. Catalytic RNA and the origin of genetic systems. *Trends Genet.*, 6(5):145–149, May 1990.

[LGC94] A. Laferrière, D. Gautheret, and R. Cedergren. An RNA pattern matching program with enhanced performance and portability. *Comput. Appl. Biosci.*, 10(2):211–212, 1994.

[LTWR03] E.C. Lai, P. Tomancak, R.W. Williams, and G.M. Rubin. Computational identification of Drosophila microRNA genes. *Genome Biology*, 4(R42), 2003.

[Mat03] J. S. Mattick. Challenging the dogma: the hidden layer of non-protein-coding RNAs in complex organisms. *BioEssays*, 25(10):930–939, 2003.

[MEG$^+$01] T. J. Macke, D. J. Ecker, R. R. Gutell, D. Gautheret, D. A. Case, and R. Sampath. RNAMotif, an RNA secondary structure definition and search algorithm. *Nucleic Acids Res.*, 29(22):4724–4735, 2001.

[MG02] C. Meyer and R. Giegerich. Matching and significance evaluation of sequence-structure patterns in RNA. *J. Physical Chemistry*, 216:1–24, 2002.

[MM93] G. Mehldau and G. Myers. A system for pattern matching applications on biosequences. *Comput. Appl. Biosci.*, 9:299–314, 1993.

[MSZT99] D. Mathews, J. Sabina, M. Zuker, and D. Turner. Expanded sequence dependence of thermodynamic parameters improves prediction of RNA secondary structure. *J. Mol. Bio.*, 288:911–940, 1999.

[NE07] E.P. Nawrocki and S.R. Eddy. Query-dependent banding (QDB) for faster RNA similarity searches, 2007. Manuscript submitted.

[NOH97] C. Notredame, E. A. O'Brian, and D. G. Higgins. RAGA: RNA sequence alignment by genetic algorithm. *Nucleic Acids Res.*, 25(22):4570–4580, 1997.

[PLD00] G. Pesole, S. Liuni, and M. D'Souza. Patsearch: a pattern matcher software that finds functional elements in nucleotide and protein sequences and assesses their statistical significance. *Bioinformatics*, 16(5):439–450, 2000.

[PS05] L. Pachter and B. Sturmfels, editors. *Algebraic Statistics for Computational Biology.* Cambridge University Press, August 2005.

[Ree79] T. Reenskaug. Models - Views - Controllers, December 1979. Technical note, Xerox PARC, A scanned version on http://heim.ifi.uio.no/~trygver/mvc/index.html.

[RG04] J. Reeder and R. Giegerich. Design, implementation and evaluation of a practical pseudoknot folding algorithm based on thermodynamics. *BMC Bioinformatics*, 5(104), 2004.

[RG06] J. Reeder and R. Giegerich. A graphical programming system for molecular motif search. In *Generative Programming and Component Engineering, Proceedings of GPCE 2006*, October 2006.

[Rot90] G. Rote. Path problems in graphs. *Computing Supplements*, 7:155–189, 1990.

[RSG05] J. Reeder, P. Steffen, and R. Giegerich. Effective ambiguity checking in biosequence analysis. *BMC Bioinformatics*, 6(153), 2005.

[RSHG04] M. Rehmsmeier, P. Steffen, M. Höchsmann, and R. Giegerich. Fast and effective prediction of microRNA/target duplexes. *RNA*, 10:1507–1517, 2004.

[SG06] P. Steffen and R. Giegerich. Table design in dynamic programming. *Information and Computation*, 204(9):1325–1345, 2006.

[SSA92] P.R. Sibbald, H. Sommerfeldt, and P. Argos. Overseer: a nucleotide sequence searching tool. *Comput. Appl. Biosci.*, 8:45–48, 1992.

[Ste06] P. Steffen. *Compiling a Domain Specific Language for Dynamic Programming.* PhD thesis, University of Bielefeld, October 2006.

[Sto02] G. Storz. An expanding universe of noncoding RNAs. *Science*, 296:1260–1263, May 2002.

[SVR$^+$06] P. Steffen, B. Voß, M. Rehmsmeier, J. Reeder, and R. Giegerich. RNAshapes: an integrated RNA analysis package based on abstract shapes. *Bioinformatics*, 22(4):500–503, 2006.

[Vos04] B. Voss. *Advanced Tools for RNA Secondary Structure Analysis.* PhD thesis, University of Bielefeld, 2004.

[WFHS99] S. Wuchty, W. Fontana, I. Hofacker, and P. Schuster. Complete Suboptimal Folding of RNA and the Stability of Secondary Structures. *Biopolymers*, 49:145–165, 1999.

[ZS81] M. Zuker and P. Stiegler. Optimal Computer Folding of Large RNA Sequences using Thermodynamics and Auxiliary Information. *Nucleic Acids Res.*, 9:133–148, 1981.