Declarative Grammars and Constraints for Intelligent Biosequence Analysis

ZHUAN CHEN

May, 2000

Declarative Grammars and Constraints for Intelligent Sequence Analysis

A thesis submitted to the College of

Graduate Studies and Research

in Partial Fulfillment of the Requirements

for the Degree of Master of Science

in the Department of Computer Science

University of Saskatchewan

Saskatoon, Saskatchewan

By

Zhuan Chen

Spring, 2000

©Copyright Zhuan Chen, 2000. All rights reserved.

PERMISSION TO USE

In presenting this thesis in partial fulfillment of the requirements for a Postgraduate degree from the University of Saskatchewan, I agree that the Libraries of this University may make it freely available for inspection. I further agree that permission for copying of this thesis in any manner, in whole or in part, for scholarly purposes may be granted by the professor or professors who supervised my thesis work or, in their absence, by the Head of the Department or the Dean of the College in which my thesis work was done. It is understood that any copying or publication or use of this thesis or parts thereof for financial gain shall not be allowed without my written permission. It is also understood that due recognition shall be given to me and to the University of Saskatchewan in any scholarly use which may be made of any material in my thesis.

Requests for permission to copy or to make other use of material in this thesis in whole or part should be addressed to:

Head, Department of Computer Science

University of Saskatchewan

57 Campus Drive

Saskatoon, Saskatchewan

Canada S7N 5A9

ABSTRACT

The field of molecular biology has generated a wealth of information. Biological sequences embody a complicated linguistic system that challenges computational techniques to model and analyze it. Logic programming (LP) and Definite Clause Grammars (DCGs), combined with constraint logic programming (CLP), are efficient, powerful, and flexible tools for natural language processing. This research explores the application of DCGs and CLP to modeling, analyzing and designing biological sequences, especially triplex DNA. Triplex DNA is able to target specific nucleotide sequences through oligonucleotide-based triple helix formation. A possible application of triplex DNA technology is to cure genetic diseases by triplex-mediated transcriptional inhibition.

This research develops a software system for automated triplex DNA formation region (TFR) analysis and triplex DNA oligonucleotide (TFO) design. Molecular biology rules governing triplex DNA are distilled from the literature and (subsequently) implemented with DCGs and CLP. Two types of TFRs (continuous and discontinuous purines/pyrimidines) are identified in human *Calmodulin* genes. The TFRs are subsequently evaluated based on position in the gene, binding proficiency, and uniqueness. The TFRs are ranked, most promising ones selected, and TFOs corresponding to these regions are designed.

ACKNOWLEDGMENTS

I would like to thank my research supervisor Dr. Anthony Kusalik. Without his supervision, guidance, encouragement, this project would not have occurred. I gratefully acknowledge his suggestions on this dissertation.

I would like to thank the members of my advisory committee, Dr. Mark Keil and Dr. Robert Hickie for their invaluable time, resources, advice and suggestions.

Special thanks to Dr. Joseph F. Angel, Dr. Keith Bonham, Dr. James Maher III, and Mr. Shawn Ritchie for their help and encouragement for this project.

A lot of thanks to my wife, my parents and my sister, for their understanding, patience, and support throughout the completion of this research.

To my friends and colleagues in the Department of Computer Science and elsewhere, who are simply too numerous to mention, your time and support made my life much brighter.

TABLE OF CONTENTS

PERMISSION TO USE..i

ABSTRACT ii

ACKNOWLEDGMENTS..iii

TABLE OF CONTENTS iv

LIST OF TABLES...viii

LIST OF FIGURES...x

LIST OF PROGRAMS..xii

CHAPTER 1. INTRODUCTION 1

CHAPTER 2. BACKGROUND...3

2.1 Logic Grammars..3

2.1.1 Context free grammars...4

2.1.2 Definite clause grammars...6

2.2 Constraint logic programming..10

2.3 Biological sequences...12

2.3.1 Nucleic acid structures..12

2.3.2 Gene structure...14

2.3.3 Triplex DNA..15

2.4 Biosequence language applications..26

2.4.1 Non-logic programming approaches to biosequence languages..26

2.4.2 Applications of DCGs to biosequence languages 26

2.4.3 Applications of constraints to biosequence languages.................29

2.5 Calmodulin genes 30

2.6 Reasoning with incomplete knowledge 31

2.7 Conclusions 33

CHAPTER 3. THE EXPERIMENT..35

3.1 Objectives of the experiment 35

3.2 Knowledge rules and reasoning...36

3.2.1 TFR type...37

3.2.2 TFR length...37

3.2.3 TFR position...38

3.2.4 Guanine content 38

3.2.5 TFR uniqueness..38

3.2.6 Multiple overlapping SD TFRs 39

3.2.7 TFO designing 39

3.3 Methodology...40

3.4 System design 40

3.4.1 Modular decomposition...41

3.4.2 Data...44

3.4.2.1 Input and output...44

3.4.2.2 Internal data flows and data structures 45

3.4.3 Knowledge implementation...47

3.4.3.1 CP TFR identification..48

3.4.3.2 SD TFR identification..51

3.4.3.3 CP and SD TFR evaluation..52

3.4.3.4 TFO design...55

3.4.3.5 Incomplete knowledge handling....................................56

3.5 Instrumentation...56

3.6 Validation of results...57

CHAPTER 4. RESULTS 58

4.1 Data verification...58

4.2 Validation result with testing sequence...60

4.3 CP TFRs from the *CaM* genes...64

4.4 SD TFRs from the *CaM* genes...64

4.5 Designed TFOs..65

4.6 Runtime analysis 71

4.7 Summary...71

CHAPTER 5. DISCUSSION 72

5.1 Extendable system...72

5.2 Contributions 73

5.3 Future work 74

REFERENCES 77

APPENDIX I: TERMINOLOGY...80

APPENDIX II: *CaM I* GENE...83

APPENDIX III: *CaM II* GENE...86

APPENDIX IV: *CaM III* GENE..89

APPENDIX V: IDENTIFIED CP9 TFRs FROM *CaM* GENES 92

APPENDIX VI: UNIQUE CP9 TFRs FROM *CaM* GENES........................96

APPENDIX VII: IDENTIFIED GENE SUBLIST CP TFR GROUPS FROM THE *CaM* GENES...98

APPENDIX VIII: IDENTIFIED SD4 TFRs FROM THE *CaM* GENES.100

APPENDIX IX: IDENTIFIED SD GENE SUBLIST TFR GROUPS FROM *CaM* GENES 107

APPENDIX X: IDENTIFIED MULTIPLE OVERLAPPING SD4 TFRs FROM *CaMI* GENE...108

APPENDIX XI: IDENTIFIED MULTIPLE OVERLAPPING SD4 TFRs FROM *CaMII* GENE...109

APPENDIX XII: IDENTIFIED MULTIPLE OVERLAPPING SD4 TFRs FROM *CaMIII* GENE..110

APPENDIX XIII: TFRs IN UPSTEAM OF THE EXON 1 OF THE *CaM III* GENE...111

APPENDIX XIV: SOURCE CODE OF THE PROGRAM 112

LIST OF TABLES

Table 4.1. Identified CP9 TFRs, corresponding TFOs, and other TFR characteristics from the testing sequences. 61

Table 4.2. Identified SD4 TFRs, corresponding TFOs and other TFR characteristics from the testing sequences 62

Table 4.3. Identified multiple overlapping SD4 TFRs from the testing sequences. 62

Table 4.4. Uniqueness analysis of identified CP9 TFRs from the testing sequences. 63

Table 4.5. Uniqueness analysis of identified SD4 TFRs from the testing sequences. 63

Table 4.6. Designed TFOs for the three front CP9 TFRs (sorted by position) of each *CaM* gene...67

Table 4.7. Designed TFOs for the three front CP9 TFRs (sorted by length) of each *CaM* gene...68

Table 4.8. Designed TFOs for the three front SD4 TFRs (sorted by position) in the three *CaM* genes..69

Table 4.9. Designed TFOs for the three front SD4 TFRs (sorted by length) in the three *CaM* genes..70

Table V.(A. to C.) Identified CP9 TFRs from *CaM* genes. 92

Table VI. Unique CP9 TFRs from *CaM* genes. 96

Table VII. Identified gene sublist CP TFR groups from the *CaM* genes.........98

Table VIII (A. to C.) Identified SD4 TFRs from the *CaM* genes................100

Table IX: Identified SD gene sublist TFR groups from C*aM* genes.............107

Table X: Identified multiple overlapping SD4 TFRs from *CaMI* gene..........108

Table XI: Identified multiple overlapping SD4 TFRs from *CaMII* gene.......109

Table XII: Identified multiple overlapping SD4 TFRs from *CaMIII* gene....110

LIST OF FIGURES

Figure 2.1. DCG derivation tree for [The man that Mike saw laughed] using Program 2.4. 8

Figure 2.3. Simplified diagram of eukaryotic gene structure and the process of gene expression. 15

Figure 2.4. The ribbon model of TFO-oriented triplex DNA formation. 17

Figure 2.5. Basic features of a pyrimidine motif-based triple helix interaction 18

Figure 2.6. Basic features of a purine motif-based triple helix interaction. 19

Figure 2.7. Triplex DNA formation between a single basepair inversion interrupted target site and abasic site TFOs. 22

Figure 2.8. Triplex DNA formation between a single basepair inversion interrupted target site and natural DNA base TFOs 23

Figure 2.9. Examples of triplex DNA formation by alternate-strand recognition. 25

Figure 3.1. Relationship among the ten modules. 42

Figure 3.2. Process control flow of the interface 43

Figure 3.3. A diagram shows the basic data flows and data structures 44

Figure 3.4. DNA sequence in GenBank format 45

Figure 3.5. An example of the system output. 45

Figure 3.6. An example of the input/2 data structure 46

Figure 3.7. Grouped input data structure. 46

Figure 3.8. The `region/7` data structure..47

Figure 3.9. An example of the `region/7` data structure for the input from Figure 3.6. 47

Figure 3.10. An example of multiple overlapping SD TFRs...........................51

Figure 4.1. Three *CaM* open reading frame DNA sequences and the deduced amino acid sequence 59

Figure 4.2. Three synthesized testing sequences for validating results...........60

LIST OF PROGRAMS

Program 2.1. A CFG in BNF. 4

Program 2.2. A Prolog program corresponding to Program 2.1....... 5

Program 2.3. An DCG corresponding to Program 2.1....... 6

Program 2.4. A DCG recognizes the sentence [The man that Mike saw laughed]. 7

Program 2.5. A DCG which recognizes arithmetic expressions and computes the value of the expression....... 9

Program 2.7. A DCG describing complementarity in nucleic acids....... 27

Program 2.8. A DCG hierarchically specifying the basic characteristics of gene structure (see Figure 2.3 for gene structure)....... 28

Program 3.1. The program segment for CP TFR parsing....... 49

Program 3.2. The program segment for CP TFR identification. 50

Program 3.3. The program segment for sublist grouping. 53

Program 3.4. The program segment for unique TFR identification. 54

Program 3.12. The program segment for purine TFO designing....... 55

Program 3.13. The program segment for pyrimidine TFO designing. 55

CHAPTER 1. INTRODUCTION

Data generated by molecular biology research, especially the Human Genome Project, is growing at an explosive rate. The sequence databases will have grown to more than 4 billion base pairs by the end of this year. It has led to a great deal of interest in using computational techniques to analyze these data. Intelligent or advanced computational methods, such as logic grammars and logic programming, have been applied to molecular biology data analysis with good success (Schulze-Kremer, 1996).

Logic programming languages have features that make them useful tools in molecular biology data analysis. Such features include built-in pattern expression, recognition, and manipulation: capabilities unmatched in conventional languages (like FORTRAN, C/C++) and built-in capabilities for non-deterministic search, important in advanced matching searches. Logic grammars, developed simultaneously with logic programming from research on natural language processing, make Prolog a powerful and expressively programming language. Constraint solving, on the other hand, provides an ability to work in domains specific to a problem (e.g. sets, Boolean, integers, real numbers, etc.) and to provide efficient solution over those domains.

This research explores the power of logic grammar and constraint logic programming for biological sequence analysis and design, with focus on triplex DNA formation regions (TFRs) and triplex formation oligonucleotides (TFOs). Molecular

biology research keeps extending our understanding of the triplex DNA, and our current knowledge about it is incomplete. Artificial intelligent techniques are used to handle the incomplete knowledge issue. Two types of TFRs, continuous purines/pyrimidines and discontinuous with one mismatch, are identified in the three human *Calmodulin* (*CaM*) genes. Identified TFRs are evaluated according to the position in the gene, TFR length, and guanine contents. TFOs corresponding to these TFRs are then designed. All these processes are automated, i.e. done by software.

The rest of the thesis is organized as follows. The Chapter 2 covers both computer science and molecular biology backgrounds. On the computer science side, declarative grammars, with focus on definite clause grammars (DCGs), constraint solving, and their usage for analyzing biosequence languages, are discussed. On the molecular biology side, biological sequence structures, especially triplex DNA, are described. The *Calmodulin* genes are also introduced in the Chapter 2. Objectives of the experiment and design and implementation issues are discussed in the Chapter 3. In Chapter 4, results of the experiment and evaluation are presented. Extensibility, contributions and future work are discussed in Chapter 5. A glossary of terms, the *Calmodulin* gene sequences, tables of TFR results, and source code can be found in the appendices. New terminology introduced in this thesis is italicized when it is used for the first time. It is assumed that the reader has basic computer science (especially logic programming and grammars) and a molecular biology background knowledge.

CHAPTER 2. BACKGROUND

In this chapter, logic grammars and constraint logic programming are described, followed by introduction of relevant material regarding biological sequences. Biosequence language applications are then explored. Finally, motivations of this research are discussed.

2.1 Logic Grammars

A language contains a set of sentences/strings of finite length. Such sentences/strings are composed of symbols of some alphabet. However, not all combinations of symbols are well-formed strings in the language. In order to define a language, a number of formalisms have been developed. Grammars are such formalisms.

Historically, Prolog (PROgrmmation en LOGique) and logic grammars were developed for natural language processing (Clocksin and Mellish, 1994). Logic grammars, being simply axiomatic systems, are declarative in nature, and hence are easy to understand and create. Logic grammars can be transformed easily to logic programs that can be executed to analyze and synthesize the defined languages. Furthermore, analysis and synthesis of language statements can be very efficient since the grammar is executable as Prolog.

As described by Abramson and Dahl (1989), logic grammars can be thought of as rules “rewrite α into β, denoted:

$$\alpha \rightarrow \beta$$

where α and β are strings of terminals and nonterminals.” Four important characteristics distinguish logic grammars from traditional grammars: (1) the form of grammar symbols, (2) the use of variables, (3) the possibility of including tests in a rule, and (4) metalevel extensions. These characteristics indicate that logic grammars are more expressive than traditional grammars.

The first logic-based grammatical formalism was Metamorphosis Grammars or MGs introduced in 1975 (reviewed by Saint-Dizier, 1994). However, the complexity of MGs made them somewhat difficult to use for simple natural language applications. A special case of MGs is definite clause grammars or DCGs (Pereira and Warren, 1980), designed for easier implementation without loss in power.

2.1.1 Context free grammars

One popular class of simple, though usefully expressive grammars, context free grammars (CFGs), are introduced in this section. Though not typically categorized as such, CFGs are a simple type of logic grammar. The following is an example of a CFG in Backus-Naur form (BNF):

```
<sentence>        ::=   <noun-phrase>, <verb-phase>
<noun-phrase>     ::=   [the], <noun>
<verb-phrase>     ::=   [runs]
<noun>            ::=   [car]
<noun>            ::=   [dog]
```

Program 2.1. A CFG in BNF.

A simple sentence derived by the grammar is [the car runs] since:

```
<sentence>      ⇒   <noun-phrase>, <verb-phrase>.
                ⇒   [the], <noun>, <verb-phrase>.
                ⇒   [the car], <verb-phrase>.
                ⇒   [the car runs].
```

By representing strings as lists of ground terms the whole grammar above can be formulated as the following Prolog program:

```
sentence(S) :- noun_phrase(NP), verb_phrase(VP),
                  append(NP, VP, S).
noun_phrase([the | NP]) :- noun(NP).
verb_phrase([runs]).
noun([car]).
noun([dog]).
append([ ], H, H).
append([H | Res1], L, [H | Res2]) :- append(Res1, L, Res2).
```

Program 2.2. A Prolog program corresponding to Program 2.1.

This program correctly describes the given language. With a query :- sentence(S), the program generates strings in the language, such as [the, car, runs] and [the, dog, runs] bound to S. The program can also recognize statements like [the car runs] via success of the query :- sentence([the, car, runs]).

In the context of Prolog, a logic program can be used directly to specify exactly the same language as any CFG (Nilsson and Matuszynski, 1990; Deransart and Maluszynski, 1993).

2.1.2 Definite clause grammars

The name "Definite Clause Grammars" (DCG) was first used by Pereira and Warren in their paper (Pereira and Warren, 1980). The name refers to the fact that the DCG rules can be easily translated into definite clauses. That is, DCGs can be transformed into effective Prolog programs. A comparison between DCGs and the augmented transition network (ATN) formalism (Pereira and Warren, 1980) suggested that DCGs could be at least as efficient as ATNs, but that the DCG formalism was clearer, more concise and more powerful.

The form of DCG rules can be specified by grammar rules. Some authors (Abramson and Dahl, 1989) specify that DCGs into a single, nonterminal symbol on the left-hand side, i.e.:

$$S \rightarrow \beta$$

where S is a nonterminal logic grammar symbol, and β is a string of terminals, nonterminals and procedure calls. Most Prolog systems also implement an extension of DCGs allowing rules of the form:

$$S \ \tau \rightarrow \beta$$

where τ is a string of terminals (Clocksin and Mellish, 1994). The example CFG (Program 2.1) can be rewritten as an DCG as follows:

```
sentence        →    noun-phrase, verb-phase.
noun-phrase     →    [the], noun.
verb-phrase     →    [runs].
noun            →    [car].
noun            →    [dog].
```

Program 2.3. An DCG corresponding to Program 2.1.

This DCG parallels the CFG formulation; for example, `sentence` and `<sentence>` are nonterminals, and `[run]`, `[car]` and `[dog]` are terminals. The form of grammar symbols is different than that in traditional grammars. For example, angle brackets (<>) are not used in DCGs, and '→' is used instead of '::='. The DCG in Program 2.3can be automatically translated into a Prolog program similar to Program 2.2.

The following example (Program 2.4) shows how DCGs deal with more complex languages, for example a language containing relative clauses. This grammar will appropriately recognize the sentence `[The man that Mike saw laughed].` Again, Program 2.4 can be automatically translated into a Prolog program before execution. The DCG derivation tree for the sentence [`The man that Mike saw laughed`] using Program 2.4 is shown in Figure 2.1 (modified from Abramson and Dahl, 1989). The numbers in the tree of Figure 2.1 are the rule numbers from the grammar.

```
(1)  sentence           →    sent([ ]).
(2)  sent(E)            →    noun-phrase, verb-phase(E).
(3)  noun-phrase        →    determiner, noun, relative.
(4)  noun-phrase        →    proper_name.
(5)  verb-phrase([ ])   →    verb.
(6)  verb-phrase(E)     →    trans_verb, direct_object(E).
(7)  relative           →    [ ].
(8)  relative           →    relative_pronoun, sent(E).
(9) direct_object(elided)
                        →    [ ].
(10) direct_object([ ])→    noun-phrase.
(11) determiner         →    [the].
(12) noun               →    [man].
(13) proper_name        →    [mike].
(14) verb               →    [laughed].
(15) trans_verb         →    [saw].
(16) relative_pronoun   →    [that].
```

Program 2.4. A DCG recognizes the sentence [`The man that Mike saw laughed`].

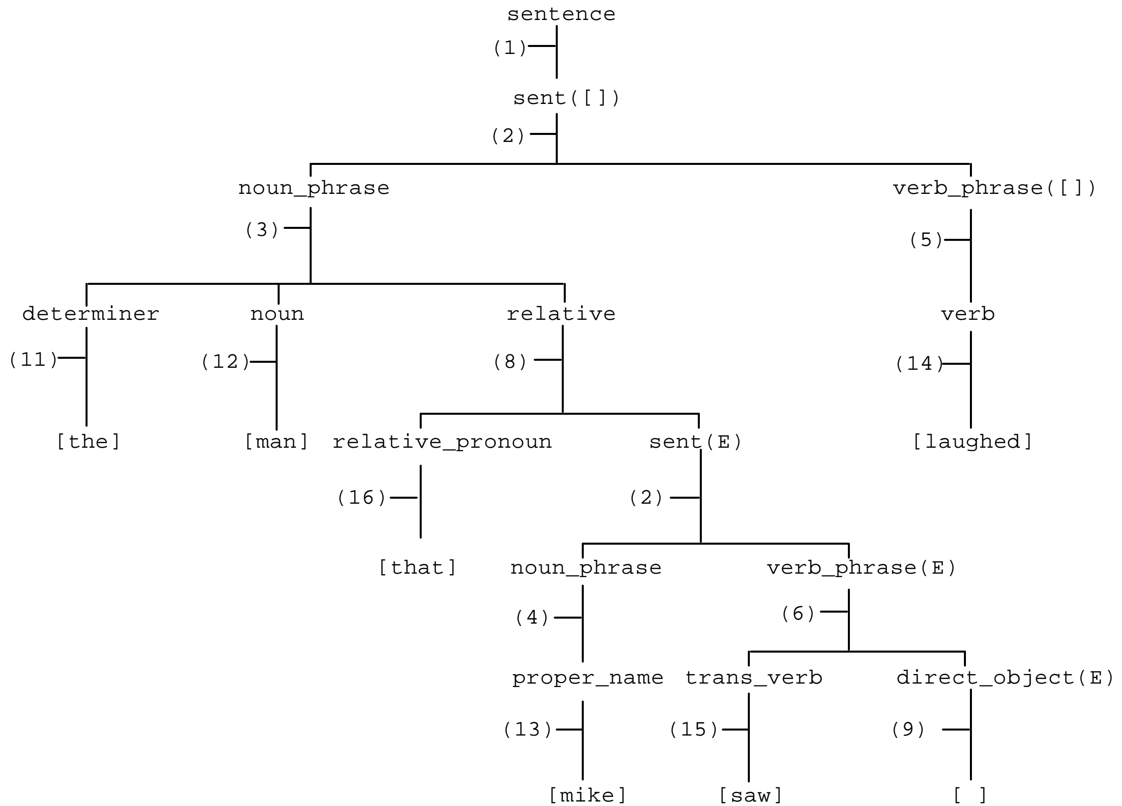

Figure 2.1. DCG derivation tree for [The man that Mike saw laughed] using Program 2.4.

Recall the four characteristics indicating that logic grammars are more expressive than traditional grammars, which is supported by Program 2.4 (DCGs) and Program 2.2 (CFGs). Program 2.4 illustrates characteristic (1) since, for example, atoms (ground literals) represent nonterminals and some nonterminals have arguments. Characteristic (2), the use of variables, is shown in rules (2), (6) and (8). Characteristic (3), the inclusion of tests in a rule, is not shown in Program 2.4. In order to show this characteristic, rule (12):

```
(12) noun            →    [man].
```

can be rewritten as:

```
(12a) noun            →    [W], {is_a_noun(W)}.
(12b) is_a_noun(man).
```

The rule (12a) includes a test is_a_noun(W) and a variable W. The test in a rule is typically generalized to any Prolog predicate. The calls are enclosed by curly brackets, {}. Given the ability to use variables and arbitrary Prolog predicates in rules, characteristic (4), metalevel extensions, is easily achieved. However, Program 2.4 does not make use of such extensions.

The following example (Program 2.5) is a grammar which recognizes arithmetic expressions but, more than that, also computes the value of the expression. That value is an argument of each nonterminal symbol.

```
(1) expr(X)      →    term(Y), [+], expr(Z), {X is Y + Z}.
(2) expr(X)      →    term(Y), [−], expr(Z), {X is Y − Z}.
(3) expr(X)      →    term(X).
(4) term(X)      →    factor(Y), [*], term(Z), {X is Y * Z}.
(5) term(X)      →    factor(Y), [/], term(Z), {X is Y / Z}.
(6) term(X)      →    factor(X).
(7) factor(X)    →    [X], {integer(X)}.
```

Program 2.5. A DCG which recognizes arithmetic expressions and computes the value of the expression.

In rule (1), the value of X is specified as Y + Z. The last rule (7) states that any string which consists of a single terminal which is an integer, is a "factor" with the same value as the terminal.

Much work in the area of logic programming for language processing is derived from DCGs, such as Definite Clause Translation Grammars (DCTGs) and Discontinuous Grammars (DGs). DCTGs are an extension of DCGs that permit automated construction of the parse tree (Saint-Dizier, 1994). DGs can deal with multiple phenomena involving discontinuity such as left and right extraposition and free word order (Abramson and Dahl, 1989).

2.2 Constraint logic programming

Logic programming languages and logic grammars have many characteristics valuable for programming. One of the most important is the declarative nature of the programs. This characteristic makes it easy, compared to imperative languages, to program the high-level logic of the problem or solution, rather than deal with the details of the implementation. Another important characteristic is the built-in capability for backtracking search which makes them well-suited for non-deterministic search problems (Clocksin and Mellish, 1994).

Although LP languages and logic grammars have the advantages discussed above, there are some limitations in standard Prolog which then carries over to grammar formalisms implemented using the language. One limitation is that Prolog's backtracking execution model is inefficient for certain kinds of problems. The backtracking method makes it easy to program, but if the search is done naively, it can be expensive. Another limitation is performing numeric calculations. For example, a Prolog query "`:- 56 = Y + Z`" simply returns `no`. The reason for such an error is that Prolog only deals with un-interpreted symbols: it recognizes "`Y + Z`" as a term built from the symbols "`Y`", "`+`", and "`Z`" rather than sum of `Y` and `Z`. Since DCGs are implemented in Prolog, they are usually afflicted with the same shortcomings. These important limitations of Prolog and DCGs are addressed by incorporating constraints and constraint solving.

Constraint programming (CP) and Constraint Satisfaction Problems (CSP) emerged from the studies of mathematics, operations research and artificial intelligence

in the early 1960s. As a combinations of CP/CSP and LP, constraint logic programming (CLP) originated in the mid-1980s (Jaffar and Maher, 1994; Van Hentenryck, 1989). CLP allows working in domains specific to a problem (e.g., real numbers) and thus improves efficiency, expressivity, generality, and reusability of logic programs (Saint-Dizier, 1994). To language processing, CLP introduces a greater expressive power which permits the description of linguistic knowledge at a higher level of abstraction, linguistic adequacy and modularity (Saint-Dizier, 1994).

Constraint logic programs are made up of rules, which allow definition of predicates that are simply user-defined constraints or relations. These constraints are checked for consistency during the evaluation of a goal, and constraint propagation allows the pruning of search tree branches before exploration of a branch. Using constraints to eliminate branches of a search tree can solve the inefficient backtracking problem discussed above.

The aforementioned numeric calculation problem can be solved by using a CLP supporting constraints over the domain of real numbers (e.g., CLP($\Re$), (Jaffar and Maher, 1994)). The real number constraints make interpretation of numbers and standard mathematical operators part of the language. Using CLP($\Re$), the previous query ":- `56 = Y + Z`" gives result `Z = 54`, if `Y` has the value `2`. A CLP over integers could solve this query as well.

DCGs can be extended with constraints – the procedure calls will be arbitrary goals of an underlying CLP. The two rules from the Program 2.6 (Section 2.1.3) can be used as an example:

```
(1) expr(X)     →     term(Y), [+], expr(Z), {X is Y + Z}.
(2) expr(X)     →     term(Y), [−], expr(Z), {X is Y − Z}.
```

This grammar can not be used to generate strings because of the numeric calculation problem. For example, when X is 56 and Y is known, this grammar can not generate the value of Z. With the power of CLP and assuming a complete solver, the following modified grammar could solve the problem:

```
(1a) expr(X)     →     term(Y), [+], expr(Z), {X = Y + Z}.
(2a) expr(X)     →     term(Y), [−], expr(Z), {X = Y − Z}.
```

When X is known and either Y or Z is known, the above grammar can find the value of the other and generate strings from the language.

Other important constraint domains, such as Boolean, rational number and finite domains, can also be used. In general, constraints enhance DCGs capability (e.g., expressivity and efficiency) to describe languages. DCGs extended by CLP can be called constraint logic grammars (CLGs).

2.3 Biological sequences

With the rapid development of molecular biology, biological sequences are becoming a major research domain of computational linguistics. The linguistic character of biological sequences has been analyzed in a number of formal and practical perspectives (Searls, 1993). This section is an introduction to the background of biological sequences and the application of DCGs to biosequences.

2.3.1 Nucleic acid structures

Two important biological polymers are nucleic acids, the substances that preserve and transmit genetic information, and proteins, the products generated from the

transmitted information. Nucleic acids are built of monomers called nucleotides, while proteins are constructed from amino acids. The arrangement of monomers in these biopolymers is linear. Only nucleic acids will be discussed in the following sections. The same principles can be applied to proteins, though the latter are more complicated as there are 20 amino acids (20 members of alphabet) with more types of bonds.

There are two closely related genetic information-carrying molecules: deoxyribonucleic acid (DNA) and ribonucleic acid (RNA). DNA and RNA each consist of only four different monomers, called nucleotides. A nucleotide has three parts: a phosphate group, a pentose (a five-carbon sugar molecule), and an organic base. One difference between RNA and DNA is that RNA contains ribose (sugar), while DNA contains deoxyribose. The other difference between the DNA and RNA monomers is found in one of their bases. The base components of nucleic acids are either purines (Pu: adenine, A; guanine, G) or pyrimidines (Py: uracil, U; thymine, T; cytosine, C). The bases A, G, and C are found in both DNA and RNA; T is found only in DNA; and U is found only in RNA. Nucleic acids are linked together in chains that may be millions of units long. The number of possible nucleic acid sequences with n bases is 4^n.

Molecules of DNA normally exist in anti-parallel chains that are held together by hydrogen bonds and hydrophobic interactions. The bases on opposite strands are held by complementarity between bases A–T (two hydrogen bonds) and G–C (three hydrogen bonds). Such bonds are called Watson-Crick hydrogen bonds. Base pairing can also occur on certain conditions through Hoogsteen hydrogen bonds, which will be discussed in Section 2.3.3. Since RNA is chemically similar to DNA, it takes on similar

configurations as DNA with base pairing A–U and G–C. An RNA strand can also bind to a DNA strand by base pairing A_{RNA}–T_{DNA}, G–C and U_{RNA}–A_{DNA}.

2.3.2 Gene structure

Greatly simplified, a eukaryotic gene (a functional unit of DNA) can be viewed as containing three regions: upstream or promoter region, region to be transcribed, and downstream region. The upstream region contains CAAT and TATA boxes. The TATA box is a region of DNA usually formed with nucleotides TATA*N*A, where *N* indicates that any of four bases may appear at this position. The CAAT box usually contains nucleotides PyCAAT, where Py is one of pyrimidine bases (C or T). The transcript region contains a CAP site, exons, introns, and a polyA site. The CAP site normally contains nucleotides A and C. Introns are spliced out in the process of gene expression, while only exons are left and further translated. The polyA site is typically composed of nucleotides AATAAA. The region after the transcript region is the downstream region. The processing of genetic information from gene to protein consists of two steps: the first is to transcribe genetic information from DNA to messenger RNA (mRNA), and the second is to translate the message from mRNA into protein. Basic gene structure and its information processing is shown in Figure 2.3 and Program 2.8. More information about nucleic acids, proteins, genes, and gene expression, can be found in any good molecular biology textbook, such as that by Wolfe (1993).

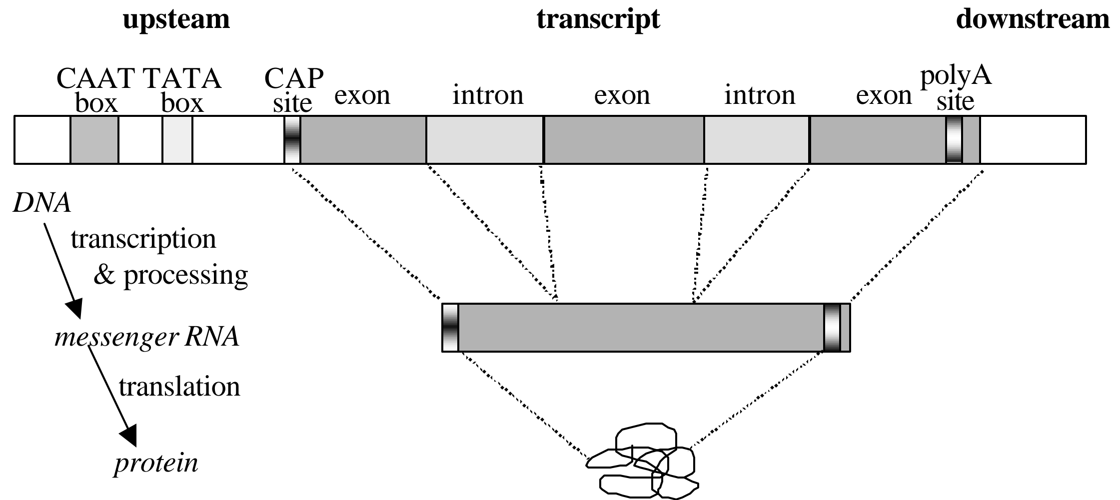

Figure 2.3. Simplified diagram of eukaryotic gene structure and the process of gene expression.

2.3.3 Triplex DNA

This section introduces the basic structures of triplex DNA and issues involved in TFO design. Biochemistry details not critical for computational modeling are omitted. More information about triplex study methods, triplex structure features, and possible biological roles of triplexes can be found in Soyfer and Potaman's book (1996).

The formation of a three-stranded, or triple-helical, DNA occurs when pyrimidine or purine bases occupy the major groove of the DNA double helix forming Hoogsteen pairs with purines of the Watson-Crick basepairs (Figure 2.4). Triplex DNA can be categorized into two types: (1) intermolecular triplexes – formed between triplex-forming oligonucleotides (TFOs) and target sequences on duplex DNA, and (2) intramolecular triplexes – formed in homopurine/homopyrimidine regions of supercoiled DNAs, which are the major elements of H-DNAs, unusual DNA structures. Molecular biology experiments have implicated that triplex DNA can inhibit DNA

transcription and replication, generate site-specific mutations, cleave DNA, and induce homologous recombination *in vitro* (Chan and Glazer, 1997; Frank-Kamenetskii, 1995; Soyfer and Potaman, 1996). The ability to target specific sequences of DNA by TFOs provides a promising tool for genetic manipulation and anti-gene drugs.

A three-letter notation is used to describe a triplex: the first letter signifies the base in a pyrimidine or pyrimidine-rich (Py) strand of the duplex, the second letter signifies the corresponding base in the purine or purine-rich (Pu) strand of the duplex, and the third letter (often after *) denotes the corresponding base in a triplex forming strand (usually TFO) (Figures 2.5 and 2.6). The number of nucleotides is usually described by a subscripted number such as 10 in $T_{10}A_{10}*T_{10}$.

A TFO is categorized in either the pyrimidine or purine motif depending on its base composition and binding orientation relative to its DNA target site (Figures 2.5 and 2.6). A TFO of pyrimidine motif binds parallel to the purine strand of the DNA via Hoogsteen bonds as shown in Figure 2.5. There are two canonical base triplets in the pyrimidine motif (PyPu*Py): TA*T and $C^{+}G*C$. $C^{+}G*C$ requires protonated cytosine C^{+}, which is usually formed in low pH conditions. In the purine motif, a TFO binds antiparallel to the purine-rich strand in the DNA via reverse Hoogsteen bonds (Figure 2.5C). Three canonical base triplets can be formed in this motif (PyPu*Pu): TA*A, CG*G, and TA*T (T appears here as an exception because T is a Py, not a Pu. More exceptions exist in triplex DNA formation; however, only canonical forms are discussed in this work.).

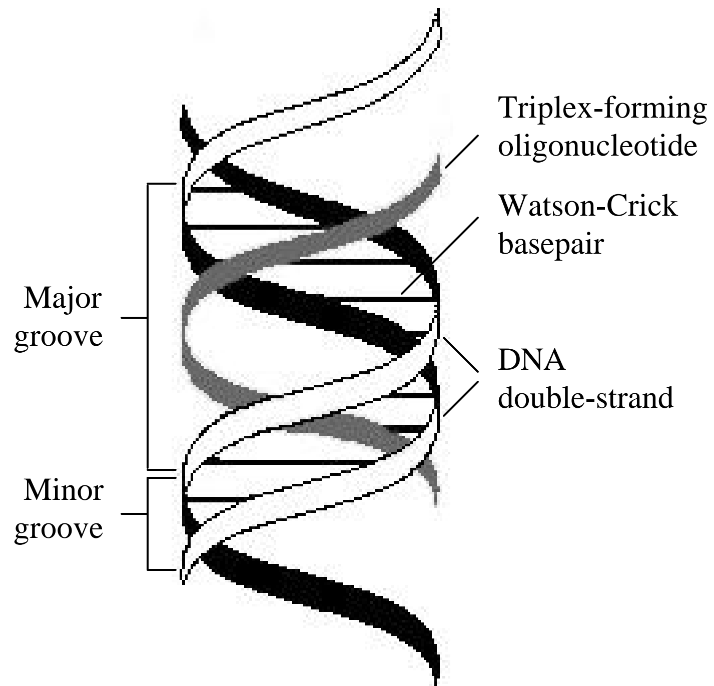

Figure 2.4. The ribbon model of TFO-oriented triplex DNA formation. A TFO (gray strand) is in the major groove of DNA. The two black-white ribbons represent the sugar-phosphate backbones of the DNA double-strand. The horizontal lines are the traditional Watson-Crick basepairs. Hoogsteen hydrogen bonds are omitted.

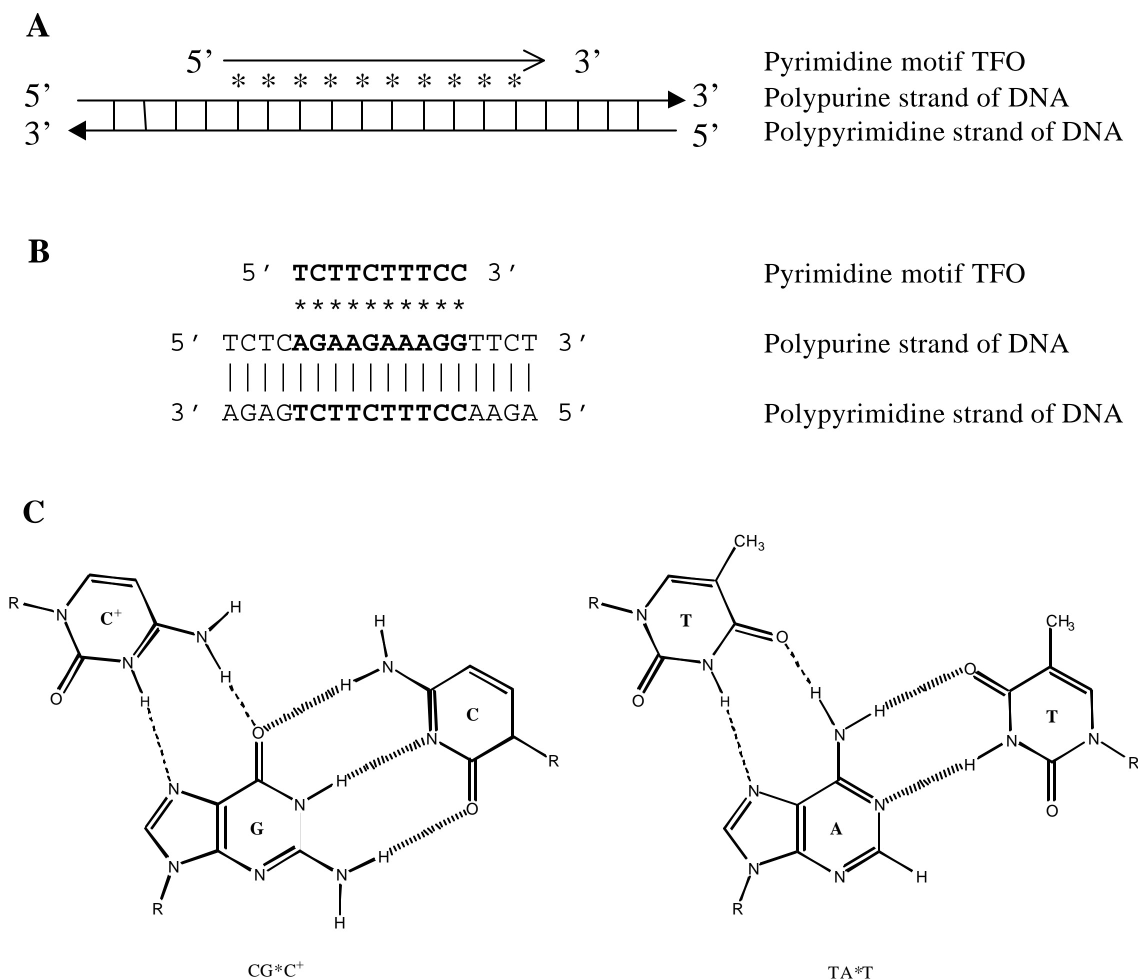

Figure 2.5. Basic features of a pyrimidine motif-based triple helix interaction. (A) Binding of a TFO in a parallel orientation to the polypurine strand of DNA. Arrows indicate individual strands of the triple helix and point in the 5’to 3’ direction. (B) An example of pyrimidine motif triplex DNA structure. Bold characters: triplex DNA forming region; *: Hoogsteen hydrogen bonds; vertical |: Watson-Crick hydrogen bonds. (C) Two canonical base triplets are TA*T and CG*C. …: Hoogsteen hydrogen bonds; \\\: Watson-Crick hydrogen bonds.

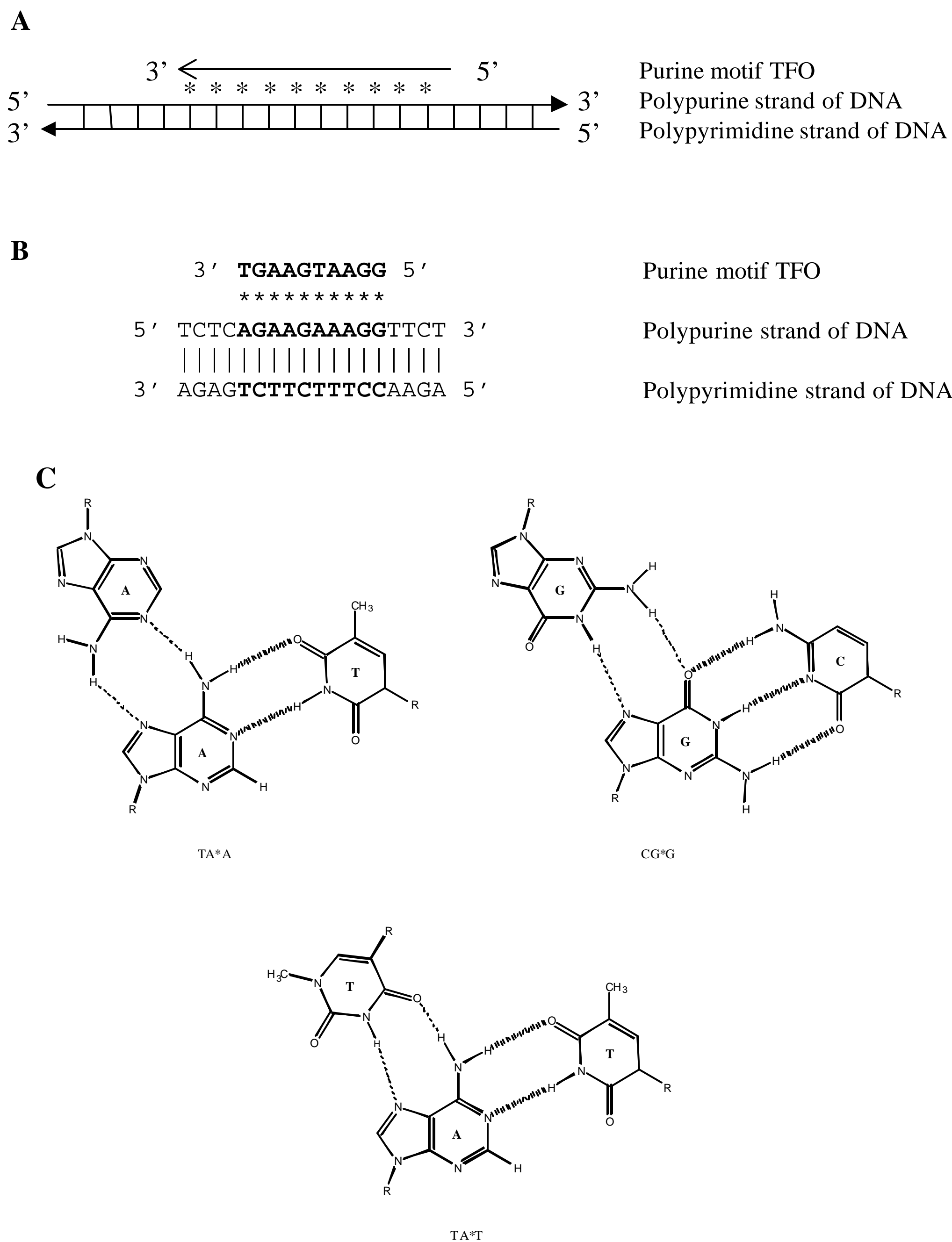

Figure 2.6. Basic features of a purine motif-based triple helix interaction. (A) Binding of a TFO in an antiparallel orientation to the polypurine strand of DNA. Arrows indicate individual strands of the triple helix and point in a 5' to 3' direction. (B) An example of purine motif triplex DNA structure. Bold characters: triplex DNA forming region; *: Hoogsteen hydrogen bonds; vertical |: Watson-Crick hydrogen bonds. (C) Three canonical base triplets are TA*A, CG*G, and TA*T. …: Hoogsteen hydrogen bonds; \\\: Watson-Crick hydrogen bonds.

Triplex formation is highly specific; for example, a 16-mer (or 16 continuous) pyrimidine motif can bind to a single 16 base pair (bp) triplex formation site *in vitro* among more than $3x10^9$ bp of genomic DNA (Strobel et al., 1991). Both purine and pyrimidine motifs are stabilized by certain chemical conditions, such as pH value (C^+G*C) and concentration of divalent cations (Mg^{2+}, Ca^{2+}, and Zn^{2+}).

Knowledge is limited on sequence and base composition requirements for TFO triplex stability. It is suggested that purine oligonucleotides must generally be G rich (greater than 65%) for a purine motif to form stable triplexes. Long contiguous of cytosines (Cs) in a pyrimidine motif may destabilize the triplex (Chan and Glazer, 1997). There is no clear oligonucleotide length requirement for TFOs. One study showed the formation of triplex with 10-mer oligonucleotides, such as $T_{10}A_{10}*T_{10}$ and $C_{10}G_{10}*G_{10}$ (reviewed by Soyfer and Potaman, 1996). Other oligonucleotide sequence and length studies using the purine motif suggest that at least 12-14 uninterrupted purines are needed to obtain adequate triplex binding (reviewed by Chan and Glazer, 1997).

It is often difficult to find sufficiently long tracts with purines in one strand providing stable triplex formation within biologically important DNA sequences, especially in promoter regions. Thus some strategies have been uncovered to make more polypurine sites available for triplex binding. One such strategy is to use an abasic site in TFOs. In triplex formation regions (TFRs), a single basepair inversion (T or C inverted to A or G, respectively) can interrupt a continuous purine or pyrimidine sequence. Such a single discontinuity (SD) can cause a mismatch with a TFO designed

for a homopurine or homopyrimidine TFR (Figure 2.7). To avoid this problem, an abasic linker, shown as "L" in Figure 2.7, is used in a TFO to skip over basepair inversions (Soyfer and Potaman, 1996). An abasic linker, for example 1,2-dideoxy-D-ribose, can bind to an A/G or T/C base pair to form imperfectly matched base triads. However, the binding strength of this imperfectly matched triplex DNA is weaker than that of regular canonical triplex DNA.

Molecular biology studies have also uncovered that SD sites can form triplex complexes without using an abasic linker (Gowers and Fox, 1999). To date, the following triplet combinations have been identified (Figure 2.8). For a CG inversion, GC*C, GC*T can be formed in a pyrimidine motif, and GC*T in a purine motif (Hone and Dervan, 1991). For a TA inversion, AT*G is stable in a pyrimidine motif (Gowers and Fox, 1999). There are currently no natural bases for specifically binding a TA inversion in a purine motif (Gowers and Fox, 1999).

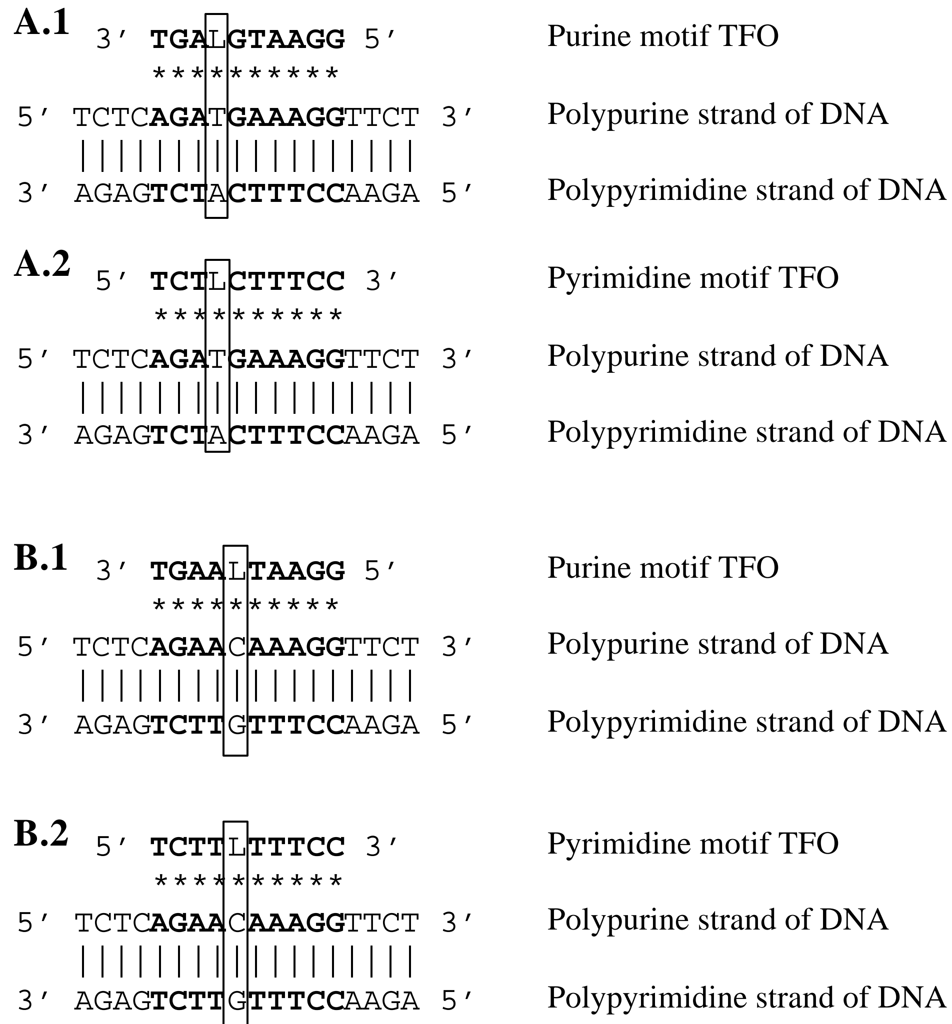

Figure 2.7. Triplex DNA formation between a single basepair inversion interrupted target site and abasic site TFOs. (A) TA to AT inversion in the target site is shown as un-bolded characters. (A.1) shows a possible purine motif TFO. (A.2) shows a possible pyrimidine motif TFO. (B) CG to GC inversion in target site is shown as un-bolded characters. (B.1) shows a possible purine motif TFO. (B.2) shows a possible pyrimidine motif TFO. L: linker.

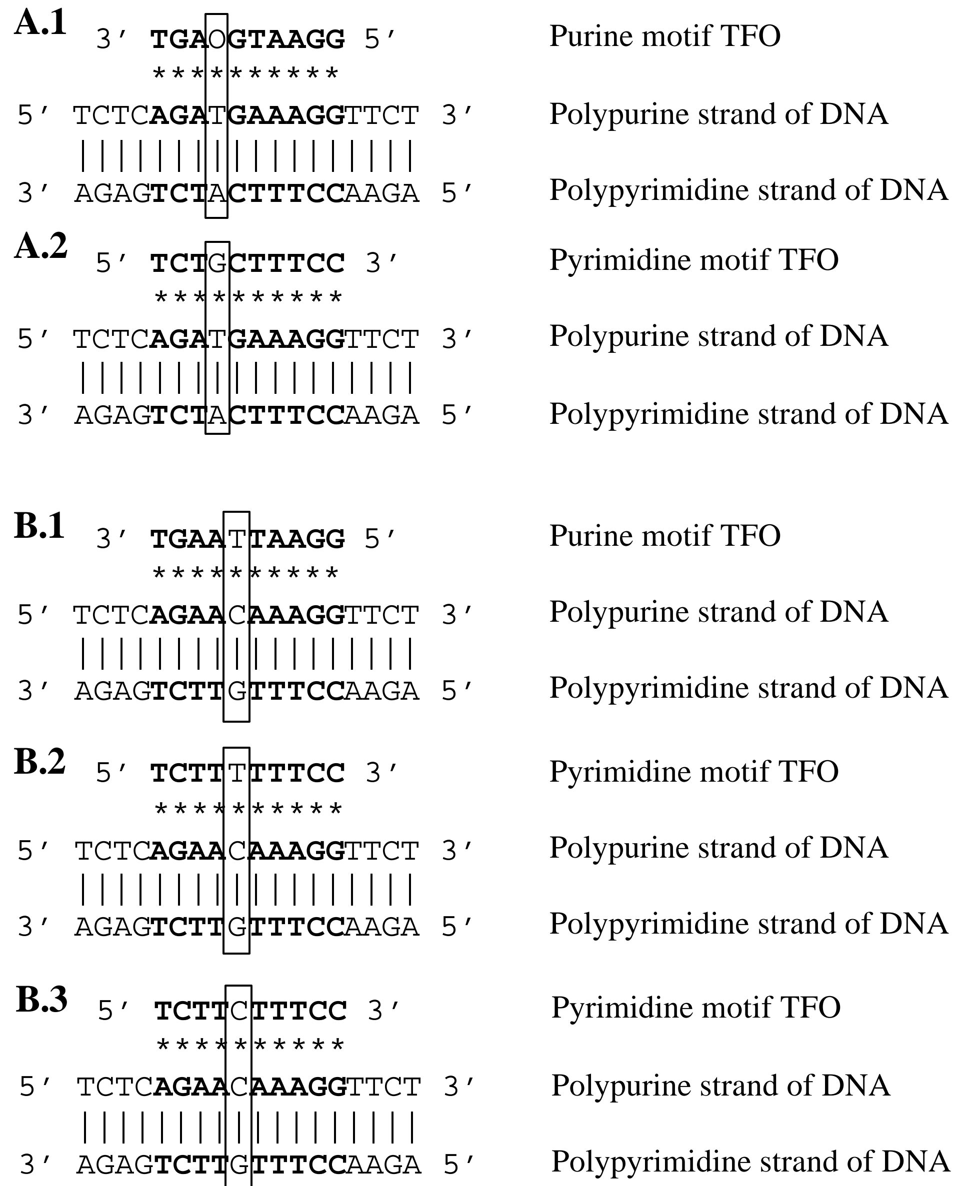

Figure 2.8. Triplex DNA formation between a single basepair inversion interrupted target site and natural DNA base TFOs. (A) TA to AT inversion in the target site is shown as un-bolded characters. (A.1) shows a no purine motif TFO, O: none. (A.2) shows a pyrimidine motif TFO. (B) CG to GC inversion in target site is shown as un-bolded characters. (B.1) shows a purine motif TFO. (B.2 & B.3) show a possible pyrimidine motif TFO.

Another approach is alternate-strand recognition. Two shorter adjacent (homo) Pu and Py tracts, with purines in different strands, are possible in a segment of DNA. Alternate-strand recognition uses an oligonucleotide which can consecutively recognize homopurine sites on opposite strands of DNA. Such alternate-strand recognition requires a minimum of two separate oligonucleotides to be linked 3'-3' or 5'-5' when the TFOs are of the same motif, or to be continuously 5' to 3' when utilizing two different motifs (Figure 2.9). A linker, such as a tetramethylene linker, which minimally alters the phosphate backbone but supplies enough flexibility to accommodate a strand cross-over, separates the different oligonucleotides. DNA sequences which have been most successfully targeted by this approach conform to the 5'-$(Pu)_m(Py)_n$-3' design (Figure 2.9). The four possible TFO designs for the TFR target are shown in Figure 2.9. In order to form stable triplex, the length of both the Py and Pu portions in a 5'-Pu-Py-3' should be at least 4 bases (Frank-Kamenetskii, 1995).

The proficiency rules to evaluate TFRs are under-determined in the literature. There is only some incomplete knowledge about the TFR proficiency for TFO design. For example, it is generally known that the length of a triplex forming region is important, i.e., the longer the region, the more binding strength for its corresponding TFO. It is a focus of this thesis research to identify these proficiency rules and implement reasoning based on them.

Transcriptional inhibition by TFOs binding at or downstream of gene promoters provides a potential antigene technique for genetic therapy. Several mechanisms exist. Blocking of the binding of transcription factors, including SP1, by triplex DNA formation is believed the most promising mechanism. Preventing transcription

elongation is another mechanism (Chan and Glazer, 1997). Thus, when designing TFOs, molecular biologists usually pay more attention to promoter regions, though without ruling out the rest of the gene.

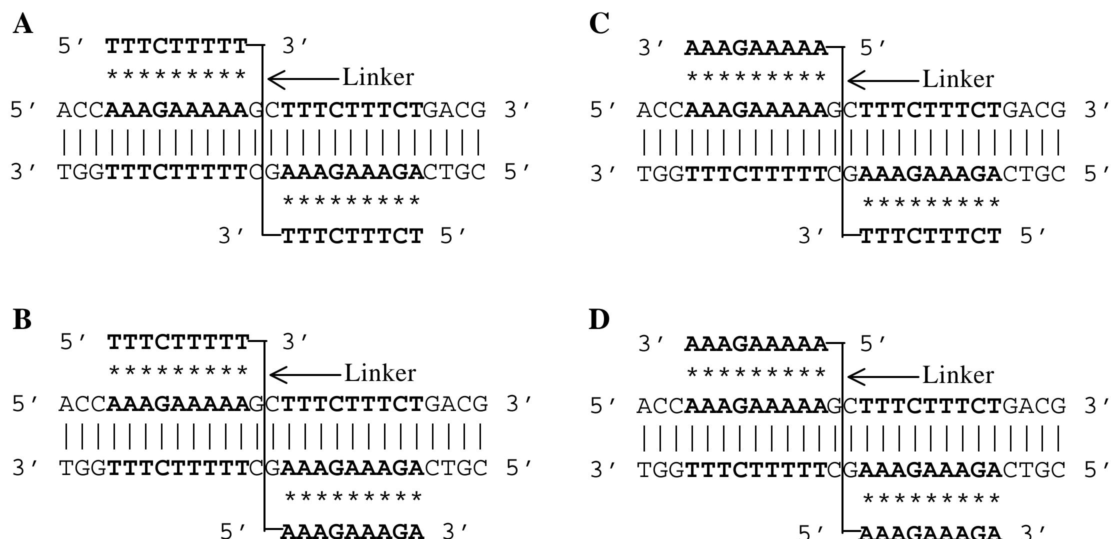

Figure 2.9. Examples of triplex DNA formation by alternate-strand recognition. In this example, a DNA sequence with a 5'-$(Pu)_{10}(Py)_{10}$-3' site can form a triplex with four different alternate-strand TFOs. The TFOs contain two 9-mer polynucleotides linked by linkers. (A) 3' linked, or 5'-$(Py)_9$-3'-linker-3'-$(Py)_9$-5' alternate-strand TFO. (B) and (C) Continuous 5' to 3', or 5'-$(Py)_9$-3'-linker-5'-$(Pu)_9$-3' alternate-strand TFO. (D) 5' linked, or 3'-$(Pu)_9$-5'-linker-5'-$(Pu)_9$-3' alternate-strand TFO.

Guanine (G) content is another important consideration during TFO design. The reason for this rule is that a pyrimidine motif is pH-dependent, because of the requirement for cytosine protonation (C^+). For this reason, A-rich continuous purine regions appear to form more stable complexes with continuous pyrimidine oligonucleotides near neutral pH (7.0 – 7.5) than do G-rich targets. On the other hand, purine motif recognition seems to be dominated by the complexes of G*GC base triplets. Formation of purine motif complexes does not require acidic conditions, but

does require G-rich TFRs (Maher, 1992, 1996). Other research on sequence and base composition requirements for purine motif stability also suggests that purine oligonucleotides must generally be guanine rich (>65%) to form stable triplexes (Chan and Glazer, 1997).

2.4 Biosequence language applications

2.4.1 Non-logic programming approaches to biosequence languages

Other computational approaches, including Hidden Makov Models (HMMs) and Neural Networks (NNs), have generated interesting results when applied to problems involving biosequences. Using NN algorithms, Steeg predicated RNA secondary structure based on the primary structure (Steeg, 1993). HMMs have been applied to protein sequence alignments (Baldi et al, 1994). However, HMMs can not capture long-range dependencies (DCGs have no such limitation). This weakness of HMMs could be overcome by using HMM/NN hybridization. However, both HMM and NN need to be trained with a large number of data. When sufficient training data is not available, both HMM and NN have to deal with a large number of unstructured parameters. In this case, computational efficiency can be a problem.

2.4.2 Applications of DCGs to biosequence languages

DCGs have been successfully applied in language processing. DCGs can also be used for analysis of the language of biosequences. For example, Searls successfully

applied DCGs to describe genes and the gene transcription process (Searls, 1993). DCGs thus can provide a very useful computational tool to describe the gene language.

As an example, the base complementarity of nucleic acids, as discussed in Section 2.3.1, can be described in DCGs as shown in Program 2.7 (comments are enclosed by `/*` and `*/`). The left-side nonterminals are nucleotide bases in DNA, RNA, or Hoogsteen bonds. The right-side terminals are the corresponding complementary bases. This grammar can be used to describe base-pairing characteristics of Watson-Crick DNA, RNA / RNA and RNA / DNA hybridization, and Hoogsteen bonds in triplex DNA.

```
/* DNA and DNA */
dna(t)      →     [a].
dna(a)      →     [t].
dna(g)      →     [c].
dna(c)      →     [g].

/* RNA/RNA or RNA/DNA complementarity */
rna(u)      →     [a].
rna(a)      →     [t].
rna(g)      →     [c].
rna(c)      →     [g].

/* Hoogsteen bonds */
hoogsteen(a)     →     [a].
hoogsteen(a)     →     [g].
hoogsteen(t)     →     [a].
hoogsteen(g)     →     [g].
hoogsteen(c)     →     [g].
```

Program 2.7. A DCG describing complementarity in nucleic acids.

The simplified gene grammar (Program 2.8, modified from Searls, 1993) can describe the basic characteristics of gene structure and gene expression hierarchically. For example, in promoter region (down to the cap site), there are two “boxes”: the CAAT box (`caat_box`) and the TATA box (`tata_box`). There are about 40 to 50 base pair nucleotides between the CAAT box and the TATA box, and another 19 to 27

between the TATA box and the cap site (cap_site). The rule for the tata_box specifies the string TATA*N*A, where *N* indicates that any of four bases may appear at this position. The rule for caat_box contains a mixture of nonterminals and terminals specifying the string PyCAAT, where Py is one of the pyrimidine bases (C or T, as shown in rule pyrimidine). This grammar is incomplete; the rules for nonterminals exon(Seq), intron, and downstream are not specified. However, they follow the same style and structure as the rules given.

```
 (1) gene          →    upstream, xscript, downstream.
 (2) upstream      →    caat_box, basepairs(N), tata_box,
                        basepairs(M),
                        {N >= 40, N <= 50, M >= 17, M <= 27}.
 (3) xscript       →    cap_site, xlate, polyA_site.
 (4) caat_box      →    pyrimidine, [c, a, a, t].
 (5) tata_box      →    [t, a, t, a], base, [a].
 (6) cap_site      →    [a, c].
 (7) xlate(Seq)    →    exon(Seq).
 (8) xlate(Seq)    →    exon(X1), intron, xlate(Xn),
                        {append(X1, Xn, Seq)}.
 (9) basepairs(1)→      base.
(10) basepairs(X)→      base, {X is X1 + 1}, basepairs(X1).
(11) base          →    pyrimidine.
(12) base          →    purine.
(13) polyA_site    →    [a, a, t, a, a, a].
(14) pyrimidine    →    [c].
(15) pyrimidine    →    [t].
(16) purine        →    [a].
(17) purine        →    [g].
```

Program 2.8. A DCG hierarchically specifying the basic characteristics of gene structure (see Figure 2.3 for gene structure).

The first rule for gene in this grammar is an uncluttered context free statement at a highly abstract level. The immediately succeeding rules present how the grammar can be “broken out” into its components in a clear, hierarchical fashion. The rules for tata_box and caat_box specify lists of terminals (nucleotide bases), sometimes combined with nonterminal atoms like pyrimidine or base. The features of

DCGs used in this example can also be applied to describe regions forming secondary or tertiary structures in DNA, RNA, and proteins.

Searls' pioneering work uses formal grammars (DCGs) to describe gene structures (Searls, 1993). Their pattern recognition parser, called GENLANG, has been successfully used for the recognition of tRNA genes, group I introns, and protein-encoding genes (Dong and Searls, 1994).

2.4.3 Applications of constraints to biosequence languages

As discussed in Section 2.2, DCGs and standard Prolog have limitations. For example, rules `(2)`, `(9)`, `(10)`, `(11)`, and `(12)` of Program 2.8 can operate in only one direction. For instance, `basepairs/1` can recognize a sequence given X. However, it can not process the other direction: to count the number of bases given a sequence. This limitation can be removed with constraints. The above rules of the `gene` grammar can be modified using finite domain constraints as follows:

```
 (2a) upstream      →  caat_box,
                       {[N] :: [40..50], [M] :: [19..27]},
                       basepairs(N), tata_box, basepairs(M).
 (9a) basepairs(1)  →  base.
(10a) basepairs(X)  →  base, {X #= X1 + 1}, basepairs(X1).
(11a) base          →  [Py], {[Py] :: [c, t]}.
(12a) base          →  [Pu], {[Pu] :: [a, g]}. /* Pu: purine */
```

Such modifications improve the expressivity of the `gene` DCG since with constraints it can be used in either direction.

Several constraint-based programs have been developed for sequence pattern searching in biosequences. For example, Palingol (Billoud et al., 1996) is a declarative programming language (with LISP-like syntax) whose data types and search engines have been adapted for nucleic acid secondary structures. CBSDL, a constraint-based

structure description language for biosequences, is designed to search for secondary structures in biosequences (Eidhammer et al., 1997; Gilbert et al., 1997). Both Palingol and CBSDL have been successfully used for searching general and well-understood secondary structures (for example, tandem repeats and stem loops). For protein sequences, Clark et al. (1993) employed parallel constraint logic programming to predicate protein topology structure with good results.

2.5 Calmodulin genes

Calmodulin (CaM) is known to act as the main regulator of Ca^{2+}-dependent signaling in eukaryotic cells. This involves a wide variety of fundamental cellular processes including intermediary metabolism, muscle contraction and cell division (Vandonselaar et al., 1994). Humans possess three *CaM* genes that are differentially regulated yet encode an identical protein (see Figure 4.1 for the amino acid sequence).

The *CaMI* gene is composed of about 8357 base pairs (bp) (Rhyner et al., 1994; see Appendix II: *CaMI* GENE for detail). Two fragments can be retrieved from GenBank (NCBI, 2000). One fragment, with GenBank accession number U11886, contains exon 1 of the *CaMI* (Appendix II: *CaMI* GENE). The other fragment contains exons 2, 3, 4, 5 and 6 and has GenBank accession number U12022 (Appendix II: *CaMI* GENE). There are 5245 bp in the *CaMII* gene. Four fragments can be retrieved from GenBank, with accession numbers as follows: U94725 (containing a 5' flanking region and exon 1), U94726 (containing exon 2), U94727 (containing partial intron 2), and U94728 (containing exons 3-6) (Toutenhoofd et al., 1998; Appendix III: *CaMII* GENE). The *CaMIII* gene contains three fragments – GenBank accession numbers X52606

(containing exon 1), X52607 (containing exon 2), and X52608 (containing exons 3-6) − with a total 6000 bp (Koller et al., 1990; Appendix IIII: *CaMIII* GENE). The fragments of each gene are combined together for the full gene sequence analysis.

The regulation of expression and function of each *CaM* gene is an area of active research. In order to study the function of each *CaM* gene, one has to "shut off" the other two *CaM* genes. Transcriptional inhibition by triplex DNA, as discussed in Section 2.3.3, is one of many ways to study a gene's function. Generally, it is difficult to design an appropriate triplex DNA oligonucleotide. For *CaM* genes, it is especially hard because *CaM* genes are highly conserved throughout evolution, i.e., their DNA sequences share a great number of similarities as each *CaM* gene encodes the same amino acid sequence. A well-designed triplex DNA oligonucleotide should inhibit transcription of only one *CaM* gene, without interfering in the transcription of the other two *CaM* genes. The thesis of this research is that a triplex DNA-based transcriptional inhibition technique can be used with appropriate designed TFOs.

2.6 Reasoning with incomplete knowledge

Knowledge in molecular biology is sometimes incomplete. Triplex DNA analysis also faces this problem. Reasoning techniques dealing with incomplete knowledge include induction, abduction, deduction, default reasoning, probabilistic modeling, and fuzzy logic (Poole et al., 1998). The remainder of this section will discuss these techniques and their potential usage in triplex DNA analysis. The reader is directed to Poole et al. (1998) and Sombe (1990) for more detail about reasoning with incomplete knowledge.

Induction learns from given examples and extends the knowledge to general rules. In contrast, deduction is a process which concludes logical consequences of a knowledge base. Abduction explains observed facts with one of several possible reasons; for example, “The e-mail did not go through? The network communication must be down!” The explanation in this example is only one of several possible explanations (e.g., the operating system might be frozen). Default reasoning deals with generalized knowledge that may have exceptions. It involves making assumptions that an individual or situation is not exceptional unless it can be proven to be exceptional. An example of default reasoning is "if I click the 'SEND' button, my e-mail will be sent". A number of implicit assumptions are included in this reasoning: the mouse is not broken, the operating system is not frozen, network communication is functioning, etc. Usually, everything is assumed to be normal.

The above reasoning techniques are not guaranteed to result in true conclusions because the information used is not complete. In induction, the conclusion is usually based on only part of the (ultimately) available observations. When more observations become available, the conclusion may no longer be true. In deduction, the reasoning result is true with a given knowledge base. When new knowledge rules are added into the knowledge base, the result might be false. In abduction, several alternative explanations might exist, and the chosen explanation might not be right. In default reasoning, the conclusion could be wrong if the state of affairs is not normal as it is assumed to be.

Probabilistic modeling can be employed to deal with uncertainty. This technique requires a set of evidence in order to calculate the likelihood of a hypothesis. Fuzzy

logic can also be used. It utilizes concepts such as ‘slow’ and ‘nice’, and incorporates quantifiers such as “very” and “slightly”.

In triplex DNA analysis, molecular biology experiments are continuously generating new knowledge. Thus, the knowledge base regarding triplex DNA is hardly complete. However, probabilistic modeling and fuzzy logic was not employed in triplex DNA analysis, because background knowledge of triplex DNA is not sufficient.

Reasoning techniques like induction, abduction, deduction, and default reasoning can be used in triplex DNA analysis. CLG, as the combination of LP, DCGs and CLP, is a nature system for implementing of the above reasoning techniques. CLG also allows quick development, updating and maintenance of a knowledge base and its reasoning rules. Thus, CLG is a good candidate for triplex DNA analysis. Further, CLG will be able to handle approximate rules (Poole et al., 1998) by using probabilistic modeling and fuzzy logic techniques when more observations are obtained in the triplex DNA area.

2.7 Conclusions

In this chapter, background about logic grammars, CLP, and biosequences has been discussed. It is noteworthy that no existing software or package for triplex DNA analysis and TFO design was found. CLG is a suitable computational tool for the identification of potential triplex DNA formation regions and the design of TFOs. CLG is chosen over procedural programming languages, such as Perl or C, because of its higher expressive and abstraction level, which allows more rapid program development. The latter is important because rules for designing TFOs – rules which will need to be

implemented in software – have not yet been explicitly specified by the biological community. Distillation of such rules is part of this thesis work, and will be characterized by imprecision and the need for iterative refinement.

CHAPTER 3. THE EXPERIMENT

3.1 Objectives of the experiment

A goal in finding a thesis topic was to develop a useful/practical program to solve some specific and relatively new molecular biology problem. Triplex DNA analysis and TFO design is one such problem. This problem was originally raised by Dr. Robert A. Hickie in Department of Pharmacology, College of Medicine, University of Saskatchewan. One goal of Dr. Hickie's research group is to study gene functions of *Calmodulin* (*CaM*, Vandonselaar et al., 1994) by triplex-mediated transcriptional inhibition. However, no existing computational software can perform the required triplex DNA analysis and TFO design. Molecular biologists typically identify potential TFRs by eye or with limited help from conventional DNA sequence analysis software such as MacVector 6.5 (Oxford Molecular Group, 1998). Using MacVector, the user can search for a TFR by querying for a certain number of purines or pyrimidines. For example, if a user wants to search for 9 or more continuous pyrimidines, the user would query with 9 pyrimidines: "`YYYYYYYYY`". MacVector will then highlight regions with 9 continuous pyrimidines. However, the user has to identify TFRs with more than 9 continuous pyrimidines by eye or by issuing another query. This example is a relatively simple task in triplex DNA analysis. For sophisticated TFR identification and TFO design, one needs to issue many queries, most of them much more complex (e.g.

involving approximate matches). For this objective, then, it is practically impossible to use MacVector-like systems. To identify an appropriate TFR among several genes and design a corresponding TFO by eye or with MacVector-like software is time-consuming, error-prone, and impracticable.

The goal of this research is to develop a software system. This system can identify potential TFRs in three *CaM* genes, evaluate these TFRs according to position, binding proficiency and uniqueness, and design TFOs specific to these regions. Ideally, the correctness of the TFO will be confirmed by experiment in a molecular biology laboratory, when resources (funding and personnel) are available.

3.2 Knowledge rules and reasoning

It is significant to this research that knowledge in triplex DNA field is incomplete at this time. As a result, the software follows reasoning techniques such as induction, deduction, and default reasoning. Deduction is employed to generate logical conclusions from the triplex DNA knowledge base. This software does not employ abduction technique, but may use it in future when an explanation is necessary for observations having several possible explanations. Default reasoning technique is used for knowledge that may have exceptions. An example is the rule “a TFO binding to a TFR with longer length will inhibit a gene’s transcription more efficiently than one with shorter length”. An assumption for this rule is: the TFR length has the most important impact on transcription inhibition of any other factor. Such an assumption is not true when a shorter TFR is located in the promoter region and a longer TFR is downstream of the gene. Hence, the software of this research uses induction, deduction, and default

reasoning. The rest of this section discusses knowledge rules abstracted from the molecular biology literature, and how these knowledge rules are used for TFR identification and TFO designing in this research. Because this knowledge base is incomplete, each rule should be considered in the context of the other rules in order to obtain a systematic view of triplex DNA analysis and TFO designing.

3.2.1 TFR type

In order to implement triplex DNA analysis, triplex DNA knowledge rules are abstracted to cover general cases. Two types of triplex DNA forming regions are studied in this thesis project. One is a continuous region with nine or more Pu or Py bases, called *continuous Pu/Py* or *CP* in this thesis. The other is the *single discontinuity* (*SD*), where the region contains one gap of base inversion (Section 2.3.3). The alternate-strand recognition is not part of this research since its search algorithm is similar to that of the SD recognition, and potential sequences from the above two methods, found during preliminary investigation, are believed to be numerous enough for this study.

3.2.2 TFR length

Generally, the length of a TFR is important for triplex DNA formation. As discussed in Section 2.3.3, the minimum length of a TFR can vary from 10 to 14. The longer the TFR, the better TFR/TFO binding. In the CP TFR category, 9 bases (CP9) is sufficient as the lower boundary. In the SD TFR category, the lower boundary of the number of continuous purine/pyrimidine on each side of the single discontinuity site is 4 (SD4). Thus, the total number of nucleotides is 9 (4 + 1 + 4 = 9) or more in SD TFRs.

Under the condition of CP9 and SD4, CP and SD TFRs can be compared based on the same length. This is the reason 9, instead of 10, is chosen.

3.2.3 TFR position

As discussed in Section 2.3.3, the position of a TFR in a gene is an important issue. It is generally believed that a TFR in the promoter region is preferred for transcription inhibition over those in downstream regions (Chan and Glazer, 1997).

3.2.4 Guanine content

Guanine (G) content and its roles in triplex DNA formation are discussed in Section 2.3.3. It is accepted that low G content (or high A content) in the TFR is good for the pyrimidine motif, and high G content (or low A content) is good for the purine motif. However the boundary between "high" and "low" is not very clear. Here 75% of G is considered as high, and 25% is considered low.

3.2.5 TFR uniqueness

One purpose of this research is to design TFOs which effectively inhibit one particular *CaM* gene without affecting the other two *CaM* genes. If one TFR sequence region can be found in other longer TFRs (either in the same gene or not), they are not unique. A TFO corresponding to that TFR may bind to all of the other TFRs. In order to design appropriate TFOs, uniqueness analysis of identified TFRs is necessary. A

unique TFR is not necessarily better than a non-unique one because of the effects of other rules (such as TFR position and G content).

It is believed that if a shorter TFR is a subsequence of other, longer TFRs in the same gene, a TFO responding to the shorter TFR may bind to the shorter TFR plus all other TFRs (called *parent TFRs,* Ritchie and Bonham, 1998). The efficiency of transcription inhibition will probably be higher than a single TFO/TFR binding, if other conditions are the same.

3.2.6 Multiple overlapping SD TFRs

Two or more SD regions can overlap each other. This is called *multiple overlapping SD TFRs* (*MOSTs*). With increased length, multiple overlapping SD TFRs provide good binding strength for corresponding TFOs. Identifying multiple overlapping SD TFRs gives the user one more TFRs (and corresponding TFOs).

3.2.7 TFO designing

Purine motif TFO design follows the triplex formation rules in Figure 2.6 of Section 2.3.3. There is only one solution for a CG duplex: CG*G. Two TFO candidates are available for a TA duplex: TA*A and TA*T. It is not clear which of these two is better for binding strength. TA*T is chosen in this research since it has obtained good results in other work (Ritchie and Bonham, 1998). TFOs designed for both CP and SD TFRs share the same triplex formation rules except the single discontinuity or single inversion positions (Figure 2.8). In such positions, triplex AC*T is used for TFO designing. In the case of an AT duplex, no known match is available.

The pyrimidine motif-based triple helix interactions shown in Figure 2.5 are pyrimidine TFO designing principles. Two canonical base triplets are TA*T and CG*C, which are used for both CP and SD TFRs. For SD TFRs, rules AC*C and AT*G are used for TFO designing in single discontinuity or single inversion positions.

3.3 Methodology

With the combination of the powers of LP, DCGs and CLP, CLG is a suitable tool for this research. SICStus Prolog (SICS, 2000), with its various constraint libraries, is chosen as the implementation platform.

A goal for the software was ease of system extensibility and program modification upon discovery of new knowledge in molecular biology fields. For easy upgrading, separate modules are used for CP and SD identification, CP and SD evaluation, and purine and pyrimidine motif TFO design. Each module can be executed independently. In order to provide friendly usage to the molecular biologist, a dialogue interface was implemented. Such an interface can be converted into a graphic user interface in future. The interface provides the user flexibility to identify CP or SD TFRs, sort the identified TFRs according to different rules, and design purine or pyrimidine motif TFOs. The details of interface and modules are given in Section 3.4.

3.4 System design

The knowledge rules are discussed in Section 3.2. This section discusses the system design strategy and describes modular decomposition, data manipulation, interfacing between modules, and knowledge implementation.

3.4.1 Modular decomposition

The system contains ten modules. The relationship among these ten modules is outlined in Figure 3.1. Module one is the dialogue interface. Module two is CP TFR identification, and module three is SD TFR identification. Modules four to seven are used to evaluate TFRs – based on TFR position, TFR length, guanine content, and uniqueness analysis, respectively. Modules four to seven can be used for either CP TFRs or SD TFRs. The next module, module eight – evaluation on multiple SD TFRs – is relevant to SD TFRs only. Module nine is for purine motif TFO designing and module ten is pyrimidine motif TFO designing. The dialogue interface (Module one) controls how other modules are executed. For example, the user can choose to identify CP or SD TFRs, or both. Once TFRs are identified, the user can sort them according to one of the evaluation rules, or a combination of rules. Then, the user can choose to design a purine motif or pyrimidine TFOs, or both. The details can be found in source code listed in Appendix XIV.

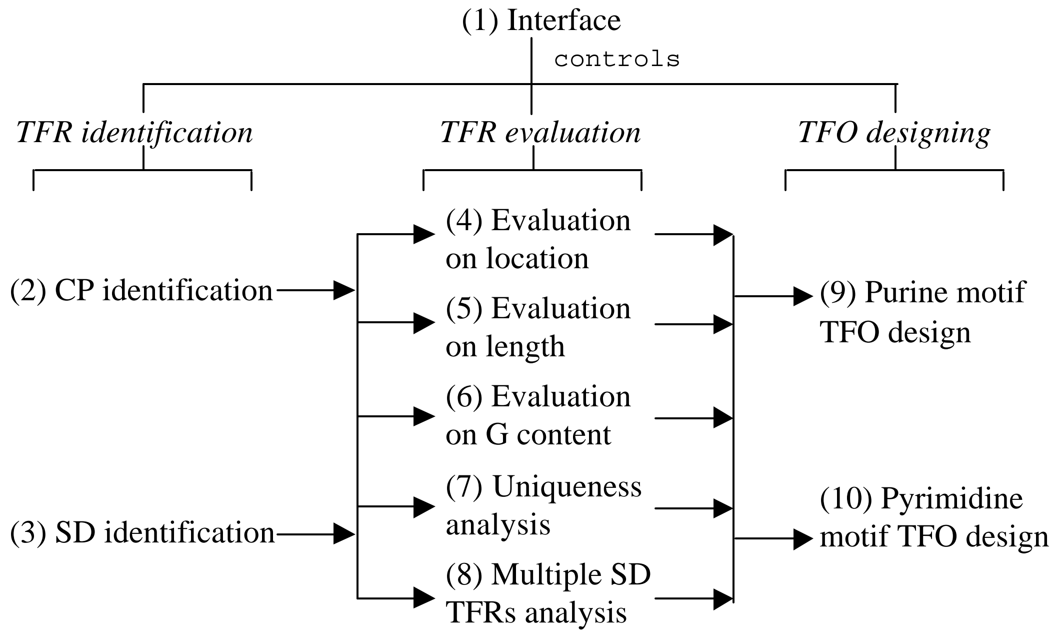

Figure 3.1. Relationship among the ten modules. The interface controls how three components process. The arrows show the information flow direction.

A flexible input/output interface is used to provide a user-friendly system. The interface controls three components: TFR identification, TFR evaluation, and TFO designing. In the TFR identification component, six choices are listed and the user can simply enter the number of an appropriate choice. Next, the user is asked to choose evaluation methods. The process flow of the input prompts of the interface is shown in Figure 3.2. To evaluate TFRs, the user can choose one of modules (4), (5) or (6), combined with module (7) and/or (8). Module (8) is only available for SD TFRs identified by module (3). The user can "bypass" an evaluation method by typing "0". In each evaluation method, the output can be sorted according to the TFR position in a gene, length of TFR, or G content. Once appropriate TFRs have been identified and

evaluated, the user can choose to design TFOs with a purine motif or pyrimidine motif, or both. Output is saved in a text file.

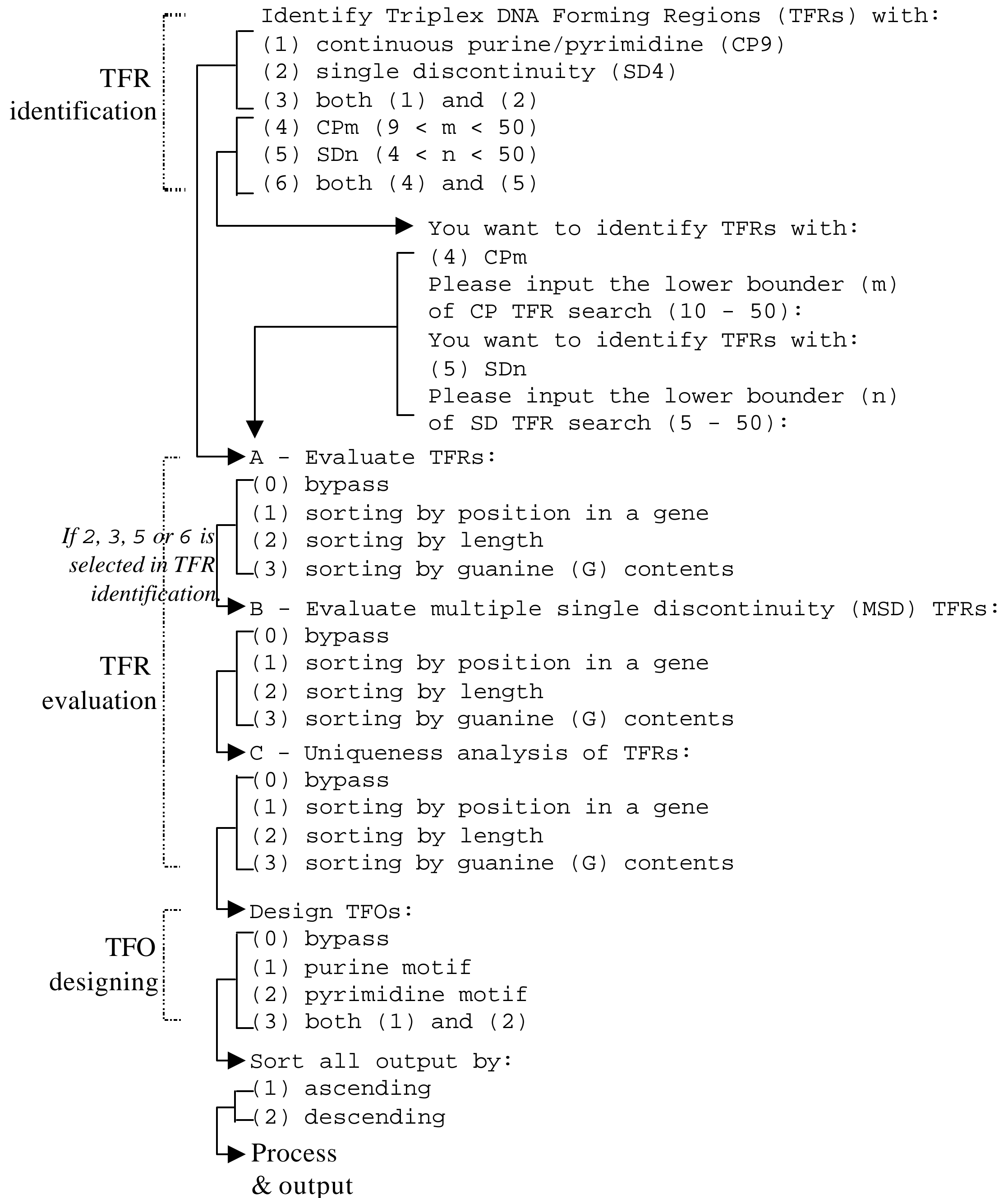

Figure 3.2. Process control flow of the interface. The arrow lines indicate the process flow direction. The dash lines show the three components: TFR identification, TFR evaluation, and TFO designing.

3.4.2 Data

This section describes data flows and data structures. Using a simple example, the diagram in Figure 3.3 shows the basic data flows and data structures.

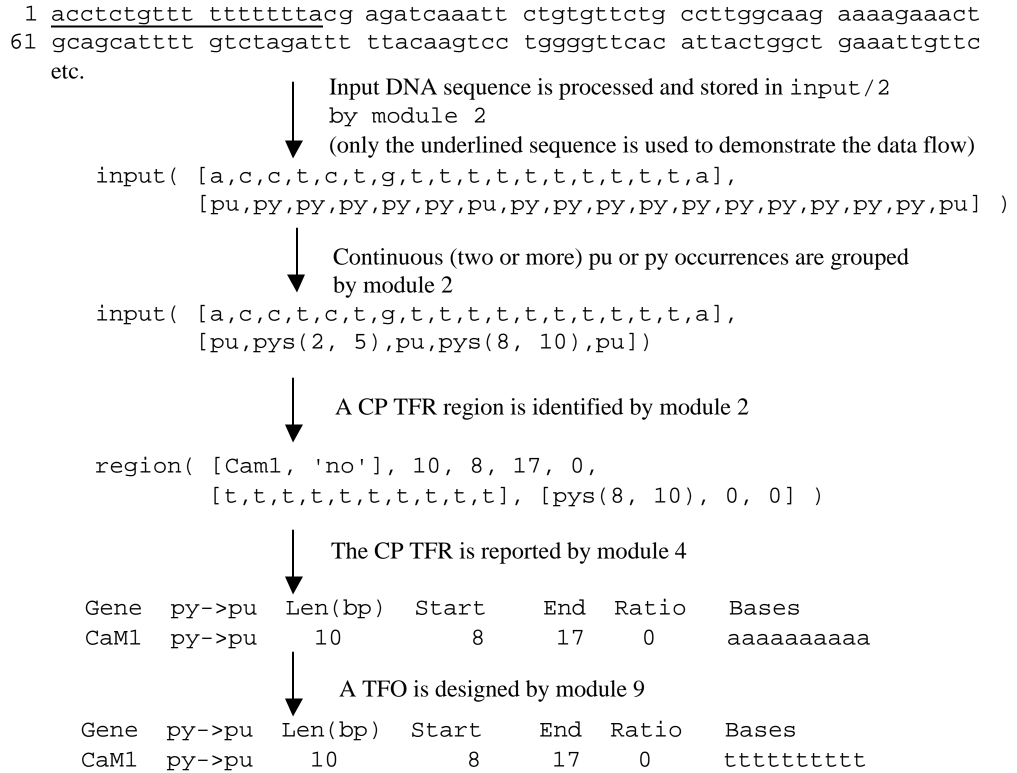

Figure 3.3. A diagram shows the basic data flows and data structures. Modules in Figure 3.1 are indicated.

3.4.2.1 Input and output

This system takes a text file containing the DNA sequence in the GenBank format (Figure 3.4 and Appendices II, III, and IV). In GenBank data format, sixty bases are usually in a line of six blocks, with ten bases in each block. The number in the first column is the position of the first base in the line.

```
  1 cccgggccct gtaaaacaga agatcaaatt ctgtgttctg ccttggcaag aaaagaaact
 61 gcagcatttt gtctagattt ttacaagtcc tggggttcac attactggct gaaattgttc
121 tttctctact ttacagaaaa atggaaaaca ctagtaaact taaagattta aatattattt
    etc.
```

Figure 3.4. DNA sequence in GenBank format.

After program execution, the result is saved in a text file. The file contents vary according to the user input. A typical output file contains a title, total number of TFRs, and sorted TFRs. The fields in the output are arranged as gene name, py to pu conversion, sequence length, TFR start and end positions, G content ratio, and DNA bases (Figure 3.5).

```
CaM1 all CP9 TFRs sorted by position are:
  Total number of regions:48

Gene  py->pu  Len(bp)  Start    End  Ratio   Bases
CaM1      no    11        48      58   18     aagaaaagaaa
CaM1  py->pu    10       118     127   30     agagaaagaa
CaM1  py->pu    11       346     356   27     aagggaaaaaa
etc.
```

Figure 3.5. An example of the system output.

3.4.2.2 Internal data flows and data structures

All white spaces and numbers in a DNA sequence are ignored. Ignoring numbers ensures that other data formats (e.g., DNA sequence without line numbering) can be used by this system. Only characters `A` to `Z` and `a` to `z` (`Char >= 65, Char < 123`) are collected. Characters `a` and `g` are then converted into ‘`pu`’(purine), while `t` and `c` are converted into ‘`py`’ (pyrimidine). The results are stored in a data structure `input/2` (Figure 3.5):

```
input( [c,c,t,t,g,g,c,a,a,g,a,a,a,a,g,a,a,a,c,t],
[py,py,py,py,pu,pu,py,pu,pu,pu,pu,pu,pu,pu,pu,pu,pu,pu,py,py] )
```

Figure 3.6. An example of the `input/2` data structure (data from Figure 3.4, bases 41 – 60).

Further processing gathers continuous (two or more) pu or py occurrences into groups `pus(M, N)` or `pys(J, K)`, respectively. M or J indicates the start position of the group, while N or K is the total number of nucleotides in the group. A single '`pu`' or '`py`' is left without any change. For example, the Figure 3.6 `input/2` data would be converted to that shown in Figure 3.7.

```
input( [c,c,t,t,g,g,c,a,a,g,a,a,a,a,g,a,a,a,c,t],
       [pys(1, 4),pu(5,2),py,pys(8,11),py(19,2)])
```

Figure 3.7. Grouped `input` data structure.

All pyrimidine TFRs are converted into antiparallel strings in order to compare G contents. This conversion is implemented as follows. When a pyrimidine TFR is found, the sequence is reversed and its characters are replaced by their complement characters. Once a pyrimidine TFR is converted into its antiparallel sequence, the atom '`py->pu`'replaces the default '`no`' in the result (Figures 3.5 and 3.8).

The system searches data in data structure `input/2` (Figure 3.7) for CP and/or SD TFRs. When a CP TFR is identified, the start and end position, the length, and the nucleotide characters are "sliced" out and recorded in data structure `region/7`.

`region/7` contains 7 fields (Figure 3.8). The first field contains two parts: an input file name and a flag for pyrimidine to purine complementarity. This flag is either 'no', meaning "not converted", or 'py-pu' meaning "converted" as discussed further in

Section 3.2.5. The fields `Length`, `PosiStart`, or `PosiEnd` indicate the length, start position, or end position of the CP TFR, respectively. `Gratio` indicates the G content of this region. `HomoGATC` contains the nucleotide characters of this CP region, while `HomoPuPy` contains the original grouped data. The example in Figure 3.6 can be identified and recorded as shown in Figure 3.9.

```
region( [InputFile, 'no'], Length, PosiStart, PosiEnd,
          GRatio, HomoGATC, HomoPuPy )
```

Figure 3.8. The `region/7` data structure.

```
region( [Cam1, 'no'], 11, 8, 18, 18,
         [a,a,g,a,a,a,a,g,a,a,a],
         [pus(8, 11), 0, 0] )
```

Figure 3.9. An example of the `region/7` data structure for the input from Figure 3.6.

The data structure `region/7` is used for interfacing between modules. For example, the field `PosiStart` is used by module 4 (Figure 3.1) for sorting TFRs by their location. Evaluation of TFR based on length is implemented by module 5 by comparing `Length`. The field `Gratio` is used by module 6 for ordering TFRs according to their G contents. The TFO design component designs TFOs using `HomoGATC` field.

3.4.3 Knowledge implementation

Knowledge implementation of CP and SD TFR identification, TFR evaluation, TFO design, and incomplete knowledge handing is discussed in this section. Program segments are presented for discussion. The comments in CLP and CLG program

segments are either bracketed by `/* */`, or are initiated by a `%` (extending to the end-of-line).

3.4.3.1 CP TFR identification

This search function focuses on the basic features of triplex DNA formation. As discussed in Section 2.3.3 and 3.2, the basic feature is that nine or more continuous Pu or Py are necessary. Grammar rules are used to identify CP TFRs (Programs 3.1 and 3.2). Both Programs 3.1 and 3.2 belong to module (2) in Figure 3.1. As shown in `search_puspys/4`, an input contains some bases (`bases`), one CP TFR (`one_pus_or_pys/5`), followed by arbitrary number of bases which may contain more bases and CP TFRs. An input can also be arbitrary number of bases without a CP TFR present or an empty list (`[]`). A `bases` nonterminal contains empty list, a `pu`, a `py`, a pu group (`pus(P1, L1)`), a py group (`pys(P2, L2)`), or more bases (`bases`) (in the program, an underscore begins the name of any variable which occurs only once in a rule). `search_puspys/4` is "eager" in that it is formulated to find a CP TFR region as soon as possible. The predicate `bases` is lazy because it wants to find an empty list (`[]`) sooner.

`one_pus_or_pys/5 is used to` identify CP TFRs (Program 3.2). The lower boundary (default 9) is entered from the input interface. When a CP TFR is identified, the start and end position, the length, and the nucleotide characters are "sliced" out and recorded in data structure `region/6`.

```
/* grammar rules for 9 or more pus/pys
 * eager for more regions
 * input:
 *      InputFile, input DNA file name
 *      ShortL, containing structure shortL(N, GATC)
 *              N starts at 1
 *      Mcp, the threshold of CP TFR
 * output:
 *      Region1 and MoreRegions, containing one CP TFR
 */
search_puspys( InputFile, ShortL, Mcp,
               [ Region1 | MoreRegions ]  ) -->
     bases,
     one_pus_or_pys( InputFile, ShortL, Mcp,
                     ShorterL, Region1 ),
     search_puspys( InputFile, ShorterL, Mcp, MoreRegions ).
%no triplex regions, red cuts
search_puspys( _InputFile, _ShortL, _Mcp, [ ] ) --> bases, !.
search_puspys( _InputFile, _ShortL, _Mcp, [ ] ) --> [ ], !.


%lazy bases
bases --> [ ].
bases --> base, bases.

%green cuts
base --> [ pu ], !.
base --> [ py ], !.
base --> [ pus( _P, _L ) ], !.
base --> [ pys( _P, _L ) ], !.
```

Program 3.1. The program segment for CP TFR parsing.

```
/* a pus(N,L) found, L >= 9
 * then get the region and store in region
 * input:  InputFile, shortL( Position, GATC_Posi), Mcp
 * output: shortL( PosiStart, GATC_Rest)
 *         region( [InputFile, 'no'], Length,
 *                  PosiStart, PosiEnd,
 *                  HomoGATC, HomoPuPy )
 * InputFile: the text file name, e.g., CaM1
 * Position: the point of input sequence
 * GATC_Posi: list of g,a,t,c
 * Mcp: the threshold of continuous pu/py
 * PosiStart: start position pus(PosiStrat, Length1)
 * GATC_Rest: list of g,a,t,c of the above
 *            pus(PosiStrat,  Length1)
 * PosiEnd: end position of the pus(PosiStrat, Length1)
 * HomoGATC: list of g,a,t,c of pus(PosiStrat, Length1)
 * HomoPuPy: list of the identified pus(PosiStrat, Length1)
 */
one_pus_or_pys( InputFile, shortL( Position, GATC_Posi ),
                Mcp, shortL( PosiStart, GATC_Rest ),
                region( [InputFile, 'no'], Length,
                        PosiStart, PosiEnd,
                        HomoGATC, HomoPuPy ) ) -->
     [ pus( PosiStart, Length1 ) ],
     { Length1 >= Mcp,
       PosiEnd is PosiStart - 1 + Length1,
       Length = Length1,
       HomoPuPy = [pus( PosiStart, Length ), 0, 0],
       PosiStart2 #= PosiStart - Position + 1,
       PosiEnd2 #= PosiEnd - Position + 1,
       /* two-step slice:
        * slice1( GATC_Posi, PosiStart2,
        *         PosiEnd3, GATC_Rest )
        *   GATC_Posi: input list
        *   PosiStart2: the slicing start point in
        *      the list GATC_Posi
        *   PosiEnd3: the slicing end point in the
        *      list GATC_Posi
        *   GATC_Rest: the resulting list containing
        *      elements from PosiStart2 to the end of GATC_Posi
        *      GATC_Rest is used for further search
        * slice2( GATC_Rest, PosiEnd4, HomoGATC )
        *   GATC_Rest: input list
        *   PosiEnd4: end slicing position
        *   HomoGATC: return list with elements from
        *      start to PosiEnd4 of GATC_Rest
        */
       slice1( GATC_Posi, PosiStart2,
               PosiEnd3, GATC_Rest ),
       PosiEnd4 #= PosiEnd2 - PosiEnd3,
       slice2( GATC_Rest, PosiEnd4, HomoGATC ) }.
```

Program 3.2. The program segment for CP TFR identification.

3.4.3.2 SD TFR identification

SD TFR identification is similar to CP TR identification, except that SD TFRs can overlap each other (called multiple overlapping SD TFRs). Figure 3.10 shows an example of such multiple overlapping SD TFRs. The two SD TFRs are: [pys(2,5),pu,pys(8,10)] and [pys(8,10),pu,pys(19,5)].

```
input( [a,c,c,t,c,t,g,t,t,t,t,t,t,t,t,t,t,a,c,t,t,c,c],
       [pu, pys(2,5),pu,pys(8,10),pu,pys(19,5)] )
```

Figure 3.10. An example of multiple overlapping SD TFRs.

Usually, a DCG performs only a single pass over its target input. In the case above, this would mean that only one SD TFR could be identified: when the first SD TFR ([pys(2,5),pu,pys(8,10)]) is found, the rest of input sequence is [a,c,t,t,c,c]. In order to identify the second SD TFR, ([pys(8,10),pu,pys(19,5)]), the program could prepend the sequence pys(8,10) to the input and parse it again, but that is not efficient.

In the SD TFR search situation, a simple method can solve the problem. Once a SD TFR is determined , the system searches for the rest of multiple overlapping SD TFR fields. For example, when a SD TFR is identified (e.g., [pys(P0,L0),pu,pys(P1,L1)]), the program continues to search for a single pu and pys(P2,L2). If a field containing pu and pys(P2,L2) is identified, the region [pys(P0,L0),pu, pys(P1,L1),pu,pys(P2,L2)] is recognized as a multiple overlapping SD TFR. The program then recursively looks for further pu and pys(P,L).

3.4.3.3 CP and SD TFR evaluation

Identified CP and SD TFRs are evaluated based on rules of binding proficiency and the uniqueness of each TFR. Three sorting methods – position in the gene, TFR length, and guanine contents – are used to evaluate TFRs, as discussed in Sections 2.3.3 and 3.2.

All sort operations are carried out by merge sort since merge sort is a stable sort algorithm and one of the fastest sorts when implemented as a logic program (Nilsson and Matuszynski, 1990).

The uniqueness analysis is to determine whether a TFR query is a substring of any other TFRs (called parent TFRs) in the three genes. When such a case occurs, the TFR query and identified parent TFRs are grouped together and reported as a *sublist TFR group*. The TFRs which are not substrings of any parent TFRs are *unique TFRs*. When a TFR query is identified as a substring of one or more TFRs in the same gene, such a sublist TFR group is called a *gene sublist TFR group* (note that a gene sublist TFR group may also contain parent TFRs from the other genes). All sublist TFR groups are reported to the user (see examples in Section 4.2).

The uniqueness analysis is implemented as follows. All TFRs identified from the *CaM* genes are merged together and sorted by their length in ascending order. Each item, starting from the first, is used as a query. Suppose the list contains m items and the current item is nth. Starting from the first (indexing at 1) item in the list, each item is compared with other items from $n+1$ to $m-1$ to check whether it is a sublist of the other items (Program 3.3, module (7) in Figure 3.1). Once sublists are identified, they are grouped together into a single list-of-lists for further analysis or report.

```
/************************************************
 * check sublist: search the sorted (by length) list with
 * the first element, if the first element is a sublist
 * of a TFR, they are grouped together;
 * then search with the second element …
 * through the rest of the list
 * check_sublist( SortList, SubList ):
 * Input: SortList - sorted (by length, ascending) TFR list
 * Output: SubList - grouped TFR list
 */
% nothing to do if list is empty
check_sublist( [], [] ).
% don't search the last one
check_sublist( [H|R], [SubList2|SubList] ) :-
     base_sublist( [H|R], SubList2 ),
     check_sublist( R, SubList ).

/* base_sublist( SortList, OneGroupList):
 *   Input: SortList - sorted (by length, ascending) TFR list
 *   Output: OneGroupList - grouped list containing the first
 *                         element and its parent TFRs
 * An accumulator technique is used for building up the
 * output argument
 */
% add the query to the end list
base_sublist( [H], [H] ).
% if the query is a sublist of the second item
% output the second item in the Output list
base_sublist( [region( Gene1, N1, S1, E1, R1, Base1, PuPy1 ),
               region( Gene2, N2, S2, E2, R2, Base2, PuPy2)
               |R],
             [region( Gene2, N2, S2, E2, R2, Base2, PuPy2 )
             |SubList] ):-
     %my_sublist/2 determines if Base1 is a sublist of Base2
     my_sublist( Base1, Base2 ),
     base_sublist( [region( Gene1, N1, S1, E1, R1, Base1,
                   PuPy1 )|R], SubList ).
% skip one if not a sublist
base_sublist( [H1, _H2|R], SubList ) :-
      base_sublist( [H1|R], SubList ).
```

Program 3.3. The program segment for sublist grouping.

The unique TFRs, which are not sublists of other TFRs, are determined by executing `unique/2`. `unique/2` checks the length of a sublist group (Program 3.4, module (7) in Figure 3.1) from the start of the list-of-lists (i.e., from short to longer queries). If the length is equal to 1, the TFR in this sublist group is identified as unique.

When the length is greater than 1, all TFRs in this sublist group is removed from the list-of-lists. Some removed (longer) TFRs may be single sublists, but are also removed by this process.

```
/* to identify unique TFRs
 * unique( Input, Output ):
 *   input: Input, a list containing TFR sublists
 *   output: Output, a list containing lists with only
 *         one element (single TFR)
 */
unique( [], [] ).
unique( [H|R1], [H1|R2] ) :-
     length( H, Length ),
     Length == 1,
     H = [H1],
     unique( R1, R2 ).
unique( [H|R1], R2 ) :-
     length( H, Length ),
     Length > 1,
     my_select( H, R1, R3 ),
     unique( R3, R2 ).

/* to remove (longer) single sublists from the rest of list
 * because these (longer) single sublists are a part of shorter
 * sublist.  E.g.,
 *   [ [[a,g,t,c], [a,a,g,g,a,g,t,c]], [a,a,g,g,a,g,t,c] ]
 * the single sublist [a,a,g,g,a,g,t,c] is removed because
 * it is not really unique (it's substring matches [a,g,t,c]
 * my_select( H, R1, R3 )
 *   Input:  H  - query sublist, each element will be
 *              used to search the input list R1
 *           R1 - list containing TFR sublists
 *   Output: R3 - rest of R1 list with un-unique single
 *              sublist removed
 */
my_select( [_], R, R ).
my_select( [H|R1], R2, R3 ) :-
     select2( [H], R2, R2a ),
     my_select( R1, R2a, R3 ).


select2( _H, [], [] ).
select2( H, [H|R1], R ) :-
     select2( H, R1, R).
select2( H, [H1|R1], [H1|R2] ) :-
     H \== H1,
     select2( H, R1, R2 ).
```

Program 3.4. The program segment for unique TFR identification.

3.4.3.4 TFO design

Purine and pyrimidine motif TFO designing follows the triplex formation rules discussed in Section 3.2.7. Programs 3.12 (belonging to module (9) in Figure 3.1) and 3.13 (belonging to module (10) in Figure 3.1) show the TFO design implementation. An input list of bases containing the TFR is supplied as an argument. The grammar rules operate on each base, generating the corresponding base (in the TFO) in each case. The rules are run in the forward direction, i.e. generating strings.

```
/* CP TFO design */
r_tfo_match( g ) --> [g].
r_tfo_match( a ) --> [t].
%r_tfo_match( a ) --> [a].  //only a-t match is used

/* SD TFO design */
r_tfosd_match( g ) --> [g].
r_tfosd_match( a ) --> [t].
r_tfosd_match( c ) --> [t].
/* There is no match for a T discontinuity,
 * "nnnNOT_AVAIABLEnnn" is used to indicate
 * this fact and set as a marker in the result.
 */
r_tfosd_match( t ) --> [nnnNOT_AVAIABLEnnn],!.
```

Program 3.12. The program segment for purine TFO designing.

```
/* CP TFO design */
y_tfo_match( a ) --> [t].
y_tfo_match( g ) --> [c].

/* SD TFO design */
y_tfosd_match( a ) --> [t].
y_tfosd_match( g ) --> [c].
y_tfosd_match( c ) --> [c].
y_tfosd_match( t ) --> [g].
```

Program 3.13. The program segment for pyrimidine TFO designing.

3.4.3.5 Incomplete knowledge handling

As discussed in Section 3.2, triplex DNA knowledge is incomplete. Deduction is used to identify CP, SD and MSD regions, and to design TFOs using established knowledge rules. Default reasoning technique is mainly used for TFR evaluations. Recall that TFR evaluation is based on TFR position, TFR length, guanine content, and uniqueness analysis, respectively. In each TFR evaluation method, the system considers the rule to be evaluated in the method only, and assumes that all other rules have no effect on TFR preference. For example, when the system evaluates TFRs based on their positions, the system focuses on the positions only and ignores all other rules.

3.5 Instrumentation

The experiment was mainly carried out on SUN Microsystems "Ultra5" workstations. Runtime analysis of the program is not a focus of this thesis, so only total CPU time was measured. CPU time, the measurements of tasks, such as input/output, CP/SD TFR searching, uniqueness analysis and TFO design (Section 4.6), were taken over a single continuous period on a stand-alone or idle workstation. This was to minimize artifacts and measurement overhead. The workstation was running the SunOS 5.6 operating system. The CPU time measurements, recorded by `statistics(runtime, Time)` in SICStus Prolog, for each task are listed in the next chapter.

3.6 Validation of results

In order to justify the correctness and completeness of TFR search and TFO design program, the system of programs was tested and evaluated. For this purpose, three short DNA sequences containing predetermined triplex DNA formation regions were created. These short DNA sequences were used as the testing sequences. The TFRs identified by the program were compared with the known regions for correctness and completeness. The test DNA sequences with known TFRs and the results obtained with the system are shown in Section 4.2.

For *CaM* genes, identified TFRs were evaluated by eye. For example, an identified TFR can be verified by checking the position, length and DNA bases in the *CaM* gene sequences. Once the triplex DNA formation regions were extracted, it was easy to verify the correctness by eye since these regions were usually short (up to twenty-nine bases). The completeness was hard to test for in this case. It is assumed that if the search is complete for the synthesized sequences, it is also complete for the real genes. The binding strength of designed TFOs to the gene targets will be tested in a molecular biology laboratory when resources are available.

CHAPTER 4. RESULTS

The system discussed in Chapter 3 was validated using three test sequences. This system then analyzed the three *CaM* genes. The results obtained with the system are discussed in the following sections. The raw output from the program was reformatted and arranged to generate the tables.

4.1 Data verification

Before conducting a search on the three *CaM* gene sequences, these sequences were analyzed to identify possible noise in the sequence data. The exons (regions coding protein sequence) of the three *CaM* genes were extracted from the complete gene and transformed into their amino acid sequences. The results (Figure 4.1) confirm that the three genes encode identical amino acid sequences, which suggests that the sequence data contain no noise or a very low noise level.

```
CaMI   atg gct gat cag ctg acc gaa gaa cag att gct gaa ttc aag gaa gcc ttc tcc cta ttt
CaMII  *** *** **c **a *** **t **a **g **g *** **a **a *** **a **a **t **t **a **a ***
CaMIII *** *** **c **g *** **t **g **g **g *** **a **g *** **g **g **c **c **c **c ***
        M   A   D   Q   L   T   E   E   Q   I   A   E   F   K   E   A   F   S   L   F

CaMI   gat aaa gat ggc gat ggc acc atc aca aca aag gaa ctt gga act gtc atg agg tca ctg
CaMII  **c **a *** **t *** **a **t **a **a **a *** **a t*g *** **t **a *** **a **t **t
CaMIII **c **g *** **a *** **c **t **c **c **c *** **g t*g *** **a **g *** **a **c **g
        D   K   D   G   D   G   T   I   T   T   K   E   L   G   T   V   M   R   S   L

CaMI   ggt cag aac cca aca gaa gct gaa ttg cag gat atg atc aat gaa gtg gat gct gat ggt
CaMII  **g *** **t **c **a *** **a **g t*a *** **c *** **t *** **a **a *** **t *** **t
CaMIII **a *** **c **c **t *** **a **g c*g *** **t *** **c *** **g **g *** **a *** **g
        G   Q   N   P   T   E   A   E   L   Q   D   M   I   N   E   V   D   A   D   G

CaMI   aat ggc acc att gac ttc ccc gaa ttt ttg act atg atg gct aga aaa atg aaa gat aca
CaMII  **t **c **a *** *** *** **t **a **t c** **a *** *** **a *** **a *** **a **c ***
CaMIII **c **g **c *** *** *** **g **g **c c** **c *** *** **c *** **g *** **g **c ***
        N   G   T   I   D   F   P   E   F   L   T   M   M   A   R   K   M   K   D   T

CaMI   gat agt gaa gaa gaa atc cgt gag gca ttc cga gtc ttt gac aag gat ggc aat ggt tat
CaMII  **c *** **a **a **a **t a*a **a **a *** **t **g *** **t *** *** **c *** **c **t
CaMIII **c *** **g **g **g **c c*a **g **g *** **t **c *** **c *** *** **g *** **c **c
        D   S   E   E   E   I   R   E   A   F   R   V   F   D   K   D   G   N   G   Y

CaMI   atc agt gca gca gaa cta cgt cac gtc atg aca aac tta gga gaa aaa cta aca gat gaa
CaMII  **t **t **t *** **a **t **c **t **g *** **a *** c*t **a **g **g t*a **a *** **a
CaMIII **c **c **c *** **g **g **t **c **a *** **g *** c*g **g **g **g c*g **c *** **g
        I   S   A   A   E   L   R   H   V   M   T   N   L   G   E   K   L   T   D   E

CaMI   gaa gta gat gaa atg atc aga gaa gca gat att gat gga gac gga caa gtc aac tat gaa
CaMII  **a **t *** **a *** *** **g **a **a **t **t *** **t **t **t **a **a **c *** ***
CaMIII **g **g *** **g *** *** **g **g **t **c **c *** **a **t **c **g **c **t *** ***
        E   V   D   E   M   I   R   E   A   D   I   D   G   D   G   Q   V   N   Y   E

CaMI   gaa ttc gta cag atg atg act gca aaa tga
CaMII  *** **t *** **a *** *** **a *** **g ***
CaMIII *** **t *** **g *** *** **t *** **g ***
        E   F   V   Q   M   M   T   A   K  ///
```

Figure 4.1. Three *CaM* open reading frame DNA sequences and the deduced amino acid sequence. The amino acid is shown in capital letter, e.g., M, A, etc. The identical nucleotides among three genes are indicated by `*`. The end of the deduced amino acid sequence is indicated by `///`.

4.2 Validation result with testing sequence

In order to verify the triplex DNA analysis system, three test sequences (Figure 4.2) were synthesized with data taken from the *CaM* genes. In sequence # 1, there are 3 CP TFRs (6-17, 48-58, and 67-86), 3 SD TFRs (26-35, 31-41, and 91-104) and 1 multiple overlapping SD TFR (26-41). Sequence # 2 contains 2 CP TFRs (8-17 and 31-45). Sequence # 3 contains 2 CP TFRs (1-11 and 119-139) and 2 SD TFRs (1-18 and 78-88), where one CP TFR (1-11) is a part of a SD TFR (1-18). Both sequence # 1 (6-17 vs. 67-86) and # 2 (8-17 vs. 31-45) contain 1 gene sublist CP TFR group. One sublist CP TFR group forms across two sequences: # 2 (8-17) vs. # 2 (31-45) and # 3 (119-139). There is 1 gene sublist SD TFR group in sequence # 1 (26-35 vs. 91-104). Recall that a gene sublist TFR group forms where a TFR is a sublist of another TFR in the same sequence or gene. Two CP TFRs and three SD TFRs are unique among the three sequences.

```
# 1
1  cccggcctct ctccttcaga agatcaaagt gagaacaag agcttggcaag aaaagaaact
61 gcagcatttt cctctctcct tcctctagat agaaagtga gaagattttac

# 2
1  aagtcctgga gaggaagttc acgcatccat aaggagagg aagagattact ggctgaaatt
61 gttc

# 3
  1 tttctctctt tacttcttat ggaaaacact agtaaactt aaagatttaaa tattatttcc
 61 taaaaggaat ataataaaaa atggaaaaca ctagtatatt taaagattta aatagtctgg
121 agaggaagag aaggagaag
```

Figure 4.2. Three synthesized testing sequences for validating results. CP TFRs are underscored. SD TFRs are double-underscored. Multiple overlapping SD TFRs are bolded. CP TFRs as part of a SD TFR are italicized.

Using the data of Figure 4.2 as input, the triplex DNA analysis system successfully and correctly identified all TFRs regions (Tables 4.1 and 4.2, re-formatted from raw output). Other TFR characteristics in the three testing sequences are also successfully identified by the system. The results of the validation are listed in Tables 4.3 – 4.5. Table 4.3 shows the determined multiple overlapping SD TFRs. Table 4.4 displays the result of uniqueness analysis of identified CP9 TFRs. The result of uniqueness analysis of identified SD4 TFRs is shown in Table 4.5. The results indicate that the system is correct and complete for the given testing sequences.

Table 4.1. Identified CP9 TFRs, corresponding TFOs, and other TFR characteristics from the testing sequences.

Sequence No.	Start	End	Length	Conversion	G ratio	Sequence
#1	6	17	12	py->pu	58	gaaggagagagg
TFO	*Purine motif*					*gttggtgtgtgg*
TFO	*Pyrimidine motif*					*cttcctctctcc*
#1	48	58	11	no	18	aagaaaagaaa
TFO	*Purine motif*					*ttgttttgttt*
TFO	*Pyrimidine motif*					*ttcttttcttt*
#1	67	86	20	py->pu	45	agaggaaggagagaggaaaa
TFO	*Purine motif*					*tgtggttggtgtgtggtttt*
TFO	*Pyrimidine motif*					*tctccttcctctctcctttt*
#2	8	17	10	no	60	ggagaggaag
TFO	*Purine motif*					*ggtgtggttg*
TFO	*Pyrimidine motif*					*cctctccttc*
#2	31	45	15	no	46	aaggagaggaagaga
TFO	*Purine motif*					*ttggtgtggttgtgt*
TFO	*Pyrimidine motif*					*ttcctctccttctct*
#3	1	11	11	py->pu	27	aaagagagaaa
TFO	*Purine motif*					*tttgtgtgttt*
TFO	*Pyrimidine motif*					*tttctctcttt*
#3	21	119	139	no	52	ggagaggaagagaaggagaa g
TFO	*Purine motif*					*ggtgtggttgtgttggtgtt g*
TFO	*Pyrimidine motif*					*cctctccttctcttcctctt c*

Table 4.2. Identified SD4 TFRs, corresponding TFOs and other TFR characteristics from the testing sequences. '–': Not identified / not available

Sequence No.	Start	End	Length	Conversion	G ratio	Sequence
#1	26	35	10	no	30	aaagtgagaa
TFO	*Purine motif*					-
TFO	*Pyrimidine motif*					*tttcgctctt*
#1	31	41	11	no	36	gagaacaagag
TFO	*Purine motif*					*gtgttttgtg*
TFO	*Pyrimidine motif*					*ctcttcttctc*
#1	91	104	14	no	35	agaaagtgagaaga
TFO	*Purine motif*					-
TFO	*Pyrimidine motif*					*tctttcgctcttct*
#2	-	-	-	-	-	-
TFO	*Purine motif*					-
TFO	*Pyrimidine motif*					-
#3	1	18	18	py->pu	27	aagaagtaaagagagaaa
TFO	*Purine motif*					-
TFO	*Pyrimidine motif*					*ttcttcgtttctctcttt*
#3	76	88	13	no	15	aaaaaatggaaaa
TFO	*Purine motif*					-
TFO	*Pyrimidine motif*					*ttttttgcctttt*

Table 4.3. Identified multiple overlapping SD4 TFRs from the testing sequences.

Sequence No.	Start	End	Length	Conversion	G ratio	Sequence
#1	26	35	10	no	30	aaagtgagaa
#1	31	41	11	no	36	gagaacaagag

Table 4.4. Uniqueness analysis of identified CP9 TFRs from the testing sequences. TFRs are sorted by their positions or query positions in the sublist TFR group.

Features	Sequence No.	Start	End	Length	Conversion	G ratio	Sequence
All CP	#3	1	11	11	py->pu	27	aaagagagaaa
	#1	6	17	12	py->pu	58	gaaggagagagg
	#2	8	17	10	no	60	ggagaggaag
	#2	31	45	15	no	46	aaggagaggaagaga
Total: 6	#1	48	58	11	no	18	aagaaaagaaa
	#1	67	86	20	py->pu	45	agaggaaggagagaggaaaa
Sublist TFR group	#1	67	86	20	py->pu	45	agaggaaggagagaggaaaa
	#1	6	17	12	py->pu	58	gaaggagagagg
	#2	31	45	15	no	46	aaggagaggaagaga
Total: 2	#3	21	119	139	no	52	ggagaggaagagaaggagaag
	#2	8	17	10	no	60	ggagaggaag
Unique CP TFRs	#3	1	11	11	py->pu	27	aaagagagaaa
Total: 2	#1	48	58	11	no	18	aagaaaagaaa
Gene sublist TFR group	#1	67	86	20	py->pu	45	agaggaaggagagaggaaaa
	#1	6	17	12	py->pu	58	gaaggagagagg
Total: 2	#2	31	45	15	no	46	aaggagaggaagaga
	#2	8	17	10	no	60	ggagaggaag

Table 4.5. Uniqueness analysis of identified SD4 TFRs from the testing sequences. TFRs are sorted by their positions or query positions in the sublist TFR group.

Features	Sequence No.	Start	End	Length	Conversion	G ratio	Sequence
	#3	1	18	18	py->pu	27	aagaagtaaagagagaaa
All SD	#1	26	35	10	no	30	aaagtgagaa
	#1	31	41	11	no	36	gagaacaagag
Total: 5	#3	76	88	13	no	15	aaaaaatggaaaa
	#1	91	104	14	no	35	agaaagtgagaaga
Sublist TFR group	#1	26	35	10	no	30	aaagtgagaa
Total: 1	#1	91	104	14	no	35	agaaagtgagaaga
Unique SD TFRs	#3	1	18	18	py->pu	27	aagaagtaaagagagaaa
Total: 3	#1	31	41	11	no	36	gagaacaagag
	#3	76	88	13	no	15	aaaaaatggaaaa
Gene sublist TFR group	#1	26	35	10	no	30	aaagtgagaa
Total: 1	#1	91	104	14	no	35	agaaagtgagaaga

4.3 CP TFRs from the *CaM* genes

Following the validation testing, the system was used to determine regions with nine or more continuous Pu/Py bases (CP9) in the three *Calmodulin* genes. TFR length, position, guanine content, and uniqueness analysis were also analyzed. As shown in Table V.A-C (Appendix V), the system identifies 48 potential triplex DNA regions in the *CaMI* gene, 34 in the *CaMII* gene, and 49 in the *CaMIII* gene. The longest potential triplex DNA region is 23 bases in the *CaMI* gene, 29 bases in the *CaMII* gene, and 25 bases in the *CaMIII* gene.

Uniqueness analysis identifies 58 unique TFRs, as shown in Table VI (Appendix VI). There are 20, 13, and 25 unique TFRs in *CaMI, CaMII*, and *CaMIII*, respectively. The longest TFR contains 19 bases. The G content analysis was carried out, indicating that G content ratio ranges from 0% to 90%.

The uniqueness analysis determined 20 CP gene sublist TFR groups (Table VII, Appendix VII). Some groups contain query and parent TFRs from the same gene only. The gene names of these groups are bolded in Table VII. One such TFR group example is *CaM2* 1536-1555 and *CaM2* 231-240. A TFO corresponding to TFR *CaM2* 231-240 will bind to TFR *CaM2* 1536-1555, which may inhibit transcription of the *CaM2* gene more efficiently than single one TFR/TFO interaction.

4.4 SD TFRs from the *CaM* genes

This system identified SD4 regions in the three *Calmodulin* genes. As shown in Table VIII (Appendix VIII), the system identified 70 SD4 TFRs in the *CaMI* gene, 39 in the *CaMII* gene, and 80 in the *CaMIII* gene. The longest length of potential triplex

DNA region is 27 bases in the *CaMI* gene, 37 bases in the *CaMII* gene, and 23 bases in the *CaMIII* gene.

Uniqueness analysis identified 154 unique TFRs among three *CaM* genes (data not shown). There are 60, 26, and 68 TFRs in *CaMI, CaMII*, and *CaMIII*, respectively. The longest TFR contains 27 bases. G content ratio ranges from 0% to 90%. The uniqueness analysis also uncovered 13 gene sublist SD TFRs groups (Table IX, Appendix IX).

The system identified 8 groups of multiple overlapping SD TFRs in the *CaMI* gene, 6 groups in the *CaMII* gene, and 9 groups in the *CaMIII* gene (Table X – XII, Appendix X – XII). The longest group is in the *CaMIII* gene, which covers 4 overlapping SD TFRs including 35 nucleotide bases.

The CP9 and SD4 TFRs upstream of the exon 1 of *CaMIII* gene are marked in Appendix XIII. There are 10 CP9 and 27 SD4 TFRs in this region. Based on the number of TFRs in such short region, it is necessary to raise the lower boundary from 9 to 15 (or higher) in CP category or 4 to 7 (or higher) in SD category when more specific TFRs are needed.

4.5 Designed TFOs

Once appropriate TFRs were identified, the TFO design was implemented according the knowledge rules discussed in Section 3.2. This section presents designed purine and pyrimidine motif TFOs. Only three TFRs from each category, sorted by position or length, are discussed.

According to the positions in the gene, the three front CP9 TFRs and their TFOs in the *CaM* genes are as listed in Table 4.6. These TFOs should be considered for molecular biology experiments to obtain more knowledge about TFR/TFO interactions. However, they are not the best choices overall. For example, the first TFR in the *CaMI* gene, starting at 48, contains only 18% G. As discussed in Section 3.2, a TFR/TFO is considered as a good candidate if G content is from 25% to 75%. In this case, this TFR/TFO can be tested in a molecular biology experiment to confirm the 25-75% G "rule". The best TFR/TFO candidate in this category is the first TFR in the *CaMIII*, starting at 158. This TFR has a length of 17 bases, good G content (29%), and obviously well-positioned in the promoter region (before exon 1: 1607-1609). This TFR/TFO candidate is highly recommended for molecular biology testing.

Table 4.7 lists TFR/TFO candidates according to the length. Among them, the TFR (*CaM1* 7204 – 7221) has a length of 18 bases and contains 55% G, which makes it a good candidate. Positioning downstream of promoter may help this TFR/TFO interaction prevent transcription elongation. Other TFRs in this table are also worth further investigation in a molecular biology laboratory, except *CaM1* 7258 – 7280, *CaM1* 8041 – 8060, and *CaM2* 3253 – 3281 because of their low G contents.

SD TFRs in the *CaM* genes were identified and their TFOs designed according to knowledge rules in Section 3.2. The three front (sorted by position) SD4 TFRs in each *CaM* gene and their purine and pyrimidine TFOs generated by the system are listed in Table 4.8. All these TFO candidates are worthy of further evaluation. Notice that *CaM3* 153 – 174, a SD4 by itself, contains a CP9 (*CaM3* 158 – 174) which is a

candidate as discussed above. It would be interesting to compare these two TFR/TFO binding behaviors.

The SD4 TFRs are also sorted according to their lengths. Table 4.9 lists the three longest TFR/TFO candidates of each *CaM* gene. Notice that the *CaM3* 153 – 174 stands out again. This TFR/TFO candidate has been listed three times in different categories, which makes it a very suitable candidate for laboratory experimentation.

Table 4.6. Designed TFOs for the three front CP9 TFRs (sorted by position) of each *CaM* gene.

Sequence No.	Start	End	Length	Conversion	G ratio	Sequence
CaM1	48	58	11	no	18	aagaaaagaaa
TFO	*Purine motif*					*ttgttttgttt*
TFO	*Pyrimidine motif*					*ttcttttcttt*
CaM1	118	127	10	py->pu	30	agagaaagaa
TFO	*Purine motif*					*tgtgtttgtt*
TFO	*Pyrimidine motif*					*tctctttctt*
CaM1	346	356	11	py->pu	27	aagggaaaaaa
TFO	*Purine motif*					*ttgggtttttt*
TFO	*Pyrimidine motif*					*ttccctttttt*
CaM2	45	55	11	no	63	gaaggggaagg
TFO	*Purine motif*					*gttggggttgg*
TFO	*Pyrimidine motif*					*cttccccttcc*
CaM2	231	240	10	no	60	gagagaggga
TFO	*Purine motif*					*gtgtgtgggt*
TFO	*Pyrimidine motif*					*ctctctccct*
CaM2	357	368	12	py->pu	25	agaaaagaaaag
TFO	*Purine motif*					*tgttttgttttg*
TFO	*Pyrimidine motif*					*tcttttcttttc*
CaM3	158	174	17	py->pu	29	agagaagaagagaaaaa
TFO	*Purine motif*					*tgtgttgttgtgttttt*
TFO	*Pyrimidine motif*					*tctcttcttctcttttt*
CaM3	250	261	12	no	33	aaaaggaaaagg
TFO	*Purine motif*					*ttttggttttgg*
TFO	*Pyrimidine motif*					*ttttccttttcc*
CaM3	294	302	9	no	55	agggaagga
TFO	*Purine motif*					*tgggttggt*
TFO	*Pyrimidine motif*					*tcccttcct*

Table 4.7. Designed TFOs for the three front CP9 TFRs (sorted by length) of each *CaM* gene.

Sequence No.	Start	End	Length	Conversion	G ratio	Sequence
CaM1	7258	7280	23	no	13	aaaaaaaagaaaaaagaaaaaag
TFO	*Purine motif*					*ttttttttgttttttgttttttg*
TFO	*Pyrimidine motif*					*ttttttttcttttttcttttttc*
CaM1	8041	8060	20	py->pu	15	ggaaaaaaaaaaaaaaaaga
TFO	*Purine motif*					*ggttttttttttttttttgt*
TFO	*Pyrimidine motif*					*ccttttttttttttttttct*
CaM1	7204	7221	18	py->pu	55	agaggggggaaaaaggag
TFO	*Purine motif*					*tgtggggggtttttggtg*
TFO	*Pyrimidine motif*					*tctcccccctttttcctc*
CaM2	3253	3281	29	py->pu	0	aaaaaaaaaaaaaaaaaaaaaaaaaaaaa
TFO	*Purine motif*					*ttttttttttttttttttttttttttttt*
TFO	*Pyrimidine motif*					*ttttttttttttttttttttttttttttt*
CaM2	1536	1555	20	no	60	gaaggaagggagagagggag
TFO	*Purine motif*					*gttggttgggtgtgtgggtg*
TFO	*Pyrimidine motif*					*cttccttccctctctccctc*
CaM2	1520	1534	15	no	53	gggaaggagagaaag
TFO	*Purine motif*					*gggttggtgtgtttg*
TFO	*Pyrimidine motif*					*cccttcctctctttc*
CaM3	5103	5127	25	py->pu	32	agaggggaaagaaaaaaaaagagaa
TFO	*Purine motif*					*tgtggggtttgtttttttttgtgtt*
TFO	*Pyrimidine motif*					*tctcccctttcttttttttt ctctt*
CaM3	3249	3268	20	py->pu	70	gggggaagaagggggaggag
TFO	*Purine motif*					*gggggttgttgggggtggtg*
TFO	*Pyrimidine motif*					*cccccttcttccccctcctc*
CaM3	4391	4409	19	py->pu	47	agggaagaaggagaaagag
TFO	*Purine motif*					*tgggttgttggtgtttgtg*
TFO	*Pyrimidine motif*					*tcccttcttcctctttctc*

Table 4.8. Designed TFOs for the three front SD4 TFRs (sorted by position) in the three *CaM* genes.

Sequence No.	Start	End	Length	Conversion	G ratio	Sequence
CaM1	13	23	11	no	18	aaaacagaaga
TFO	*Purine motif*					*tttttgttgt*
TFO	*Pyrimidine motif*					*ttttctcttct*
CaM1	36	44	9	py->pu	33	aaggcagaa
TFO	*Purine motif*					*ttggttgtt*
TFO	*Pyrimidine motif*					*ttccctctt*
CaM1	118	132	15	py->pu	26	aaagtagagaaagaa
TFO	*Purine motif*					-
TFO	*Pyrimidine motif*					*tttcgtctctttctt*
CaM2	82	90	9	py->pu	55	agggtagga
TFO	*Purine motif*					-
TFO	*Pyrimidine motif*					*tcccgtcct*
CaM2	493	503	11	no	27	gaaatgaagaa
TFO	*Purine motif*					-
TFO	*Pyrimidine motif*					*ctttgcttctt*
CaM2	579	589	11	py->pu	27	aaagaatgaag
TFO	*Purine motif*					-
TFO	*Pyrimidine motif*					*tttcttgcttc*
CaM3	32	41	10	py->pu	80	ggaggtgggg
TFO	*Purine motif*					-
TFO	*Pyrimidine motif*					*cctccgcccc*
CaM3	127	142	16	py->pu	43	ggagaggtaaagaaga
TFO	*Purine motif*					-
TFO	*Pyrimidine motif*					*cctctccgtttcttct*
CaM3	153	174	22	py->pu	22	agagaagaagagaaaaataa aa
TFO	*Purine motif*					-
TFO	*Pyrimidine motif*					*tctcttcttctctttttgtt tt*

Table 4.9. Designed TFOs for the three front SD4 TFRs (sorted by length) in the three *CaM* genes.

Sequence No.	Start	End	Length	Conversion	G ratio	Sequence
CaM1	4943	4969	27	no	33	gggaggagaaaaggataaaa gaaaaaa
TFO	*Purine motif*					-
TFO	*Pyrimidine motif*					*ccctcctcttttcctgtttt ctttttt*
CaM1	1402	1424	23	py->pu	56	gaagggaggaagagcagagg gag
TFO	*Purine motif*					*gttgggtggttgtgttgtgg gtg*
TFO	*Pyrimidine motif*					cttccctccttctcctctcc ctc
CaM1	4959	4978	20	no	15	aaaagaaaaaataaaagaag
TFO	*Purine motif*					-
TFO	*Pyrimidine motif*					*ttttctttttgttttcttc*
CaM2	1520	1555	36	no	55	gggaaggagagaaagtgaag gaagggagagagggag
TFO	*Purine motif*					-
TFO	*Pyrimidine motif*					*cccttcctctctttcgcttc cttccctctctccctc*
CaM2	4531	4554	24	no	45	gaggaaagagaggaatggaa agag
TFO	*Purine motif*					-
TFO	*Pyrimidine motif*					*ctcctttctctccttgcctt tctc*
CaM2	2164	2182	19	no	36	aaggagtggaaaaaaagga
TFO	*Purine motif*					-
TFO	*Pyrimidine motif*					*ttcctcgcctttttttcct*
CaM3	5358	5380	23	py->pu	60	gagggaagaggcagggaaga ggg
TFO	*Purine motif*					*gtgggttgtggttgggttgt ggg*
TFO	*Pyrimidine motif*					*ctcccttctccctcccttct ccc*
CaM3	153	174	22	py->pu	22	agagaagaagagaaaaataa aa
TFO	*Purine motif*					-
TFO	*Pyrimidine motif*					*tctcttcttctcttttttgtt tt*
CaM3	5175	5195	21	no	57	aaagggtggggagagggaag a
TFO	*Purine motif*					-
TFO	*Pyrimidine motif*					*tttcccgcccctctcccttc t*

4.6 Runtime analysis

The performance of the system was measured using the built-in predicate `statistics(runtime, Runtime)`. It took 570 milliseconds CPU time to read in the three input files, search for CP TFRs, and save the output files. The TFO design for all CP TFRs required 160 milliseconds. The uniqueness analysis for all CP TFRs took 330 milliseconds. SD TFR searching cost 740 milliseconds over three genes. The TFO design for all SD TFRs took 210 milliseconds. 250 milliseconds were spent for both multiple continuous SD TFR and uniqueness analysis. The time spent for *CaMI*, *CaMII*, and *CaMIII* input process is 150, 80, and 100 milliseconds, respectively. According to this, the overall performance of the system is good enough for gene level analysis. When genome level analysis is available and necessary, it might be an issue to improve the performance.

4.7 Summary

This chapter has described and analyzed the results of the triplex DNA analysis system. This system successfully determines the TFRs in the testing sequences. The system also analyzes CP and SD TFRs, and designs corresponding TFOs in the *CaM* genes. It is observed that there are a lot TFRs in each gene, and many of them may have the potential for good TFR/TFO interactions. Runtime analysis indicates that the system is suitable for mid-sized (about 10 kb) gene triplex DNA analysis.

CHAPTER 5. DISCUSSION

Logic grammars and logic programming have been used in natural language analysis for decades. It is natural to apply logic grammars and logic programming to analyzing biological sequences. Triplex DNA is a promising technique for gene targeting and gene therapy. However, triplex DNA analysis has not been extensively studied. The goal of this research has been to set up a triplex DNA analysis system using logic grammars and logic programming techniques. In this chapter, system extensibility, research contribution, and future work are discussed.

5.1 Extendable system

Triplex DNA research in the field of molecular biology will generate more and more knowledge about the formation of triplex DNA. Identification of two types of triplex DNA, CP and SD, has been implemented in this system. However, the grammar rules are very easily modified for additional types of triplex DNA identification and analysis. Although CLP has been used in this system only in several points, the power of CLP will contribute to the extensibility of the system for chromosome- or even genome-sized analysis.

The system has been designed based on the three *CaM* genes. However the system can be used for any other interesting genes. For example, in collaboration with an outside group, this system has already been applied to a highly conserved and widespread family of eukaryotic proteins – 14-3-3 proteins (Baldin, 2000). 14-3-3

proteins are believed to function as regulators in signal transduction/phosphorylation mechanisms. There are 9 genes coding the same amino acid sequence in humans. Effort has been made to design TFOs to inhibit one or several 14-3-3 gene transcriptions (Cardenas-Garcia et al, 1999). The triplex DNA analysis system described here was used to help in that effort. To modify the code for 14-3-3 gene analysis, only three lines of code were changed to take 14-3-3 DNA sequences as input.

It would be interesting to apply the system to a particular chromosome or an entire genome. However, the human genome is not complete yet. According to The Sanger Centre (The Sanger Centre, 2000), DNA sequencing has been completed on about 10% (9.5 Mb out of 93 Mb) of chromosome 14 (containing the *CaMI* gene), 5.7% (14.5 Mb out of 255 Mb) of chromosome 2 (containing the *CaMII* gene), and 27% (18 Mb out of 67 Mb) of chromosome 19 (containing the *CaMIII* gene). These partially completed DNA sequence data are not available to the public yet. If it is necessary to compare a particular TFR/TFO sequence with the partially completed human genome DNA sequences, a BLAST search can be carried out in the same Web site. When the human genome is complete in future, the system described in this thesis will be modifiable to search a particular chromosome or the whole genome.

5.2 Contributions

Over the years, constraint-based programs have been developed for sequence pattern searching in biosequences. As discussed in Section 2.4.3, these programs are usually designed to search for general secondary structural patterns, such as tandem repeat, stem loop, and palindrome patterns. They can not be directly (without a lot of

work) employed for the analysis of triplex DNA. No existing computational software has been found for triplex DNA analysis and TFO designing. The intention of this research was to generate one system for such tasks.

This research developed a triplex DNA/TFO-specific CLG-based program to identify potential TFRs, analyze and rank them, and design TFOs. These TFOs will be used for transcription inhibition study of the *CaM* genes. When that happens, there will be feedback on the validity of the various biological rules referred to, and on the validity of the various design decisions made when rendering those rules in a precise, logical formalism. It is possible to introduce more rules or modules into the system for ranking TFRs and TFOs. Systems developed from this research can be used for other genes with little modification. Although the program is triplex DNA/TFO-specific, the intention is to make it modifiable for other high-level biological sequence analysis.

This research also represents a new computer science and molecular biology collaboration. The explicit rules for ranking TFO specificity and binding strength will give biologists more specific information to design TFOs. The feedback from biologists will help to improve the programs. These generate-feedback steps may take several iterations, but will eventually enhance triplex DNA research and (thus) help to treat genetic diseases.

5.3 Future work

The triplex DNA analysis system can also be regarded as a string-matching problem. The Boyer-Moore algorithm (Boyer and Moore, 1977) and the Knuth-Morris-Pratt algorithm (Knuth et al., 1977) are good examples of string matching algorithms

(Gusfield, 1997). It is believed that the above algorithms are not necessary in our triplex DNA analysis system because CP and SD TFR identifications are simple string matching problems and the size of this problem is relative small. It might be useful to employ one of the above algorithms for more complex string matching problems when the system is attempting to identify other types of TFRs and/or is extended to genome-sized searches.

Knowledge rules used in this thesis are abstracted from molecular biology observations (as reported in the literature), and implemented in the system. A future work is to apply inductive logic programming technique to exam knowledge factors and process inductive reasoning.

TFR/TFO candidates are discussed in Section 4.5. A future work is to design and develop a program to determine "good"/"bad" TFR/TFO candidates. This program will be able to extract "good" TFR/TFO candidates according to the position, length and G content. One such "good" candidate is *CaM1* 7204 – 7221 having a length of 18 bases and contains 55% G (Table 4.6). The same program will also isolate some "bad" TFR/TFO candidates. Example of these "bad" candidates are *CaM1* 7258 – 7280, *CaM1* 8041 – 8060, and *CaM2* 3253 – 3281 because of their low G contents (Table 4.6).

As mentioned in Section 4.5, a SD TFR may contain a "good" CP TFR. For example, *CaM3* 153 – 174, a SD4 by itself, contains a CP9: *CaM3* 158 – 174 (Tables 4.6 and 4.7). It is valuable to report this feature to allow molecular biologist to obtain more detailed knowledge about TFR/TFO candidates. Hence, a future work is to design and develop a program to report these SD/CP overlapping TFRs.

Triplex DNA can also form inside a gene or a chromosome. This form of triplex DNA is called H-DNA (Frank-Kamenetskii, 1995). A future work is to develop a system to analyze the H-DNA formation in genes, such as in *Calmodulin* genes.

The yeast genome has been completed (SGD, 2000). A future work is to use this software to determine TFRs in yeast. Further, a system can be developed to analyze both TFR/TFO and H-DNA in the whole yeast genome.

These generated TFOs will be supplied to a molecular biology lab for *in vitro* or *in vivo* trials. The feedback from the molecular biology results will be used to improve the knowledge rules and the program and (thus) generate new sets of TFOs. The last two steps will be repeated until either a satisfactory molecular biology result is obtained or the program can not generate any better set of TFOs with further modification.

REFERENCES

Abramson, H., and Dahl, V. (1989) Part I Grammars For Formal Languages And Linguistic Research. In: Logic Grammars. Springer-Verlag, New York, 5-24.

Baldi, P., Chauvin, Y., Hunkapiller, T., and McClure, M. A. (1994) Hidden Markov models of biological primary sequence information. Proc. Natl. Acad. Sci. USA., 91, 1059-1063.

Baldin, V. (2000) 14-3-3 proteins and growth control. Prog Cell Cycle Res, 4:49-60.

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

Boyer, R. S., and Moore, J. S. (1977) A fast string searching algorithm. Communications of the ACM, 20(10), 762-772.

Chan, P. P., and Glazer, P. M. (1997) Triplex DNA: fundamentals, advances, and potential applications for gene therapy. Journal of Molecular Medicine, 75, 267-282.

Clark, D. A., Rawlings, C. J., Shirazi, J. A., Veron, A. A., Reeve, M. (1993) Protein topology prediction through parallel constraint logic programming. ISMB-93, 83-91.

Clocksin, W. F., and Mellish, C. S. (1994) Programming in Prolog (Fourth Edition), Springer-Verlag, New York.

Deransart, P., and Maluszynski, J. (1993) A grammatical view of logic programming. The MIT Press, Cambridge, Massachusetts.

Dong, S., and Searls, D. B. (1996) Gene structure prediction by linguistic methods. Genomics, 23, 540-551.

Eidhammer, I., Gilbert, D. R., Jonassen, I., and Ratnayake, M. (1997) A constraint based structure description language for biosequences. Technical Report 1997/04, Department of Computer Science, City University, UK and Department of Informatics, University of Bergen, Norway.

Frank-Kamenetskii, M. D. (1995) Triplex DNA structures. Annu. Rev Biochem., 64, 65-95.

Gardenas-Garcia, M., Armas, M. A., and Otero, J. L. (1999) Search of therapeutic targets using a model of intracellular signals mediated by Ras. In: Posters and Software Demonstrations, ISMB 99, 24.

Gilbert, D., Eidhammer, I., and Jonassen, I. (1996) StructWeb: biosequence structure searching on the Web using clp(FD). Technical report 1997/05, Department of

Computer Science, City University, UK and Department of Informatics, University of Bergen, Norway.

Gowers, D. M., and Fox, K. R. (1999) Towards mixed sequence recognition by triple helix formation. Nucleic Acids Research, 27(7), 1569-1577.

Gusfield, D. (1997) Algorithms on strings, trees, and sequences: computer science and computational biology. Cambridge University Press.

Horne, D. A., and Dervan, P. B. (1991) Effects of an abasic site on triple helix formation characterized by affinity cleaving. Nucleic Acids Research, 19(18), 4963-4965.

Jaffar, J., and Maher, M. J. (1994) Constraint logic programming: A survey. J. Logic Programming, 19(20), 503-581.

Knuth, D. E., Morris, J. H. JR., and Pratt, V. R. (1977) Fast pattern matching in strings. SIAM J. Computer., 6(2), 323-350.

Koller, M., Schnyder, B., and Strehler, E. E. (1990) Structural organization of the human *CaMIII* calmodulin gene. Biochimica et Biophysica Acta, 1087, 180-189.

Maher, L. J III (1992) DNA triple-helix formation: an approach to artificial gene repressors? BioEssays, 14(12), 807-815.

Maher, L. J. III (1996) Prospects for the therapeutic use of antigene oligonucleotides. Cancer Investigation, 14(1), 66-82.

NCBI: National Center for Biotechnology Information. (2000) http://www2.ncbi.nlm.nih.gov/genbank/query_form.html

Nilsson, U., and Matuszynski, J. (1990) Logic, programming and Prolog. John Wiley & Sons, New York.

Pereira, F. C., and Warren, D. H. D. (1980) Definite Clause Grammars for Language Analysis - A survey of the Formalism and a Comparison with Augmented Transition Networks. Artificial Intelligence, 13, 231-278.

Poole, D., Mackworth, A., Goebel, R. (1998) Computational Intelligence: A logic approach. Oxford University Press, New York.

Rhyner, J. A., Ottiger, M., Wicki, R., Greenwood, T. M., and Strehler, E. E. (1994) Structure of the human *CALM1* calmodulin gene and identification of two *CALM1*-related pseudogenes *CALM1P1* and *CALM1P2*. Eur. J. Biochem. 255, 71-82.

Ritchie, S., and Bonham, K. (1998) The human *c-Src* proto-oncogene promoter contains multiple targets for triplex-forming oligonucleotides. Antisense & Nucleic Acid Drug Development, 8, 391-400.

Saint-Dizier, P. (1994) Advanced logic programming for language processing. Academic Press, New York.

Schulze-Kremer, S. (1996) Molecular bioinformatics. Walter de Gruyter, New York.

Searls, D. B. (1993) The computational linguistics of biological sequences. In: Artificial Intelligence and Molecular biology. The MIT Press, Cambridge, Massachusetts,

47-120.

SGD: *Saccharomyces* Genome Database (2000) http://genome-www.stanford.edu/Saccharomyces/

SICS: Swedish Institute of Computer Science. (2000) http://www.cling.gu.se/datorinfo/sicstus/

Sombe, L. (1990) Reasoning under incomplete information in artificial intelligence. Wiley, Toronto.

Soyfer, V. N., and Potaman, V. N. (1996) Triple-helical nucleic acids. Springer, New York.

Steeg, E. W. (1993) Neural networks, adaptive optimization and RNA secondary structure prediction. In: Artificial Intelligence and Molecular biology. The MIT Press, Cambridge, Massachusetts, 121-160.

Strobel S. A., Doucette-Stamm, L. A., Riba, L., Housman, D. E., Dervan, P. B. (1991) Site-specific cleavage of human chromosome 4 mediated by triple-helix formation. Science 254: 1639-1642.

The Sanger Centre. (2000) http:// www.sanger.ac.uk/

Toutenhoofd, S. L., Foletti, D., Wicki, R., Rhyner, J. A., Garcia, F., Tolon, R., and Strehler, E. E. (1998) Characterization of the human CALM2 calmodulin gene and comparison of the transcriptional activity of CALM1, CALM2 and CALM3. Cell Calcium, 23(5), 3232-338.

Vandonselaar, M., Hickie, R. A., Quail, J. W., and Delbaere, L. T. J. (1994) Trifluoperazine-induced conformational change in Ca2+-calmodulin. Nature (Structural Biology), 1(11): 795-801.

Van Hentenryck, P. (1986) Constraint satisfaction in logic programming. The MIT Press, Cambridge, Massachusetts.

Wolfe, S. L. (1993) Molecular and Cellular Biology. Wadsworth Publishing Company, California.

APPENDIX I: TERMINOLOGY

ATN

Augmented transition network.

Base

In the context of molecular biology, a base refers to a single nucleotide (DNA or RNA). The DNA bases are abbreviated A, T, C,and G. The RNA bases are abbreviated A, U, C, and G.

Base pair

Two bases (nucleotides) that bond to form complementary pairs. In DNA these are A and T, C and G. In RNA they are A and U, C and G. These pairs are also called Watson-Crick pairs.

CFG

Context free grammars.

CLG

Constraint logic grammars.

CLP

Constraint logic programming.

CP

Constraint programming.

CP9

Continuous purine or pyrimidine with mininum length of 9.

CSP

Constraint satisfaction problems.

DCG

Definite clause grammars.

DCTG

Definite clause translation grammars.

Exon

In eukaryotes, genes are typically broken up into alternating regions of exons and introns. The exons contain information that is represented in the messenger RNA (mRNA) and ultimately used to produce a protein or sometimes RNA.

Gene

A contiguous interval of DNA that contains the information needed to code for a protein, or less often, for some RNA. Genes form the basic units of heredity.

HMMs

Hidden Makov Models.

Intron

See exon. Introns are spliced out in the process of gene expression.

LP

Logic programming.

MOSTs

Two or more SD regions can overlap each other, which is called multiple overlapping SD TFRs.

NNs

Neural Networks.

Nucleic acid

Essentially a chain of nucleotides; an RNA or DNA molecule.

Nucleotide

Adenine, thymine, cytosine, or guanine in DNA; adenine, uricil, cytosine, or guanine in RNA. Also see base.

Oligonucleotide

A short nucleic acid chain (DNA or RNA).

Open reading frame (ORF)

A substring in DNA that contains no stop codons (UAA, UAG, or UGA) when read through. Eukaryotic genes are sometimes described as containing several ORFs, meaning several exons.

Promoter

A substring in DNA upstream of a gene, where the RNA polymerase (which helps transcribe the DNA to RNA during transcription) binds to the DNA.

Purine

A purine is either adenine (A) or guanine (G).

Pyrimidine

A pyrimidine is either cytosine (C) or thymine (T).

SD

Single discontinuity

TFOs

Triplex- forming oligonucleotides

TFRs

Triplex formation regions

APPENDIX II: *CaM I* GENE

This gene sequence is composed of two fragments: U11886, with basepairs 1-1776 (in both global and local enumeration); U12022, with basepairs 1777-8357 (in global enumeration) or 1-6581 (in local enumeration). The arrow '↓' position indicates the end position of U11886 fragment. The end base of U11886 fragment is in bold for easy identification. Exons are underlined and labeled above with exon numbers, global positions, and fragment positions in parentheses. The translation start point is indicated with '⇓'.

```
   1  cccgggccct gtaaaacaga agatcaaatt ctgtgttctg ccttggcaag aaaagaaact
  61  gcagcatttt gtctagattt ttacaagtcc tggggttcac attactggct gaaattgttc
 121  tttctctact ttacagaaaa atggaaaaca ctagtaaact taaagattta aatattattt
 181  taaaaggaat ataataaaaa atggaaaaca ctagtatatt taaagattta aatagtcttt
 241  taaaaggaat ataatcaaaa ctgtagttta aaatacaatc tagctccata agagaggcaa
 301  ttggctgtgt gttccacttg taatgcagag gatttgaagc atctattttt tcccttgttt
 361  ctatggattt atgaataaag actctgaccc ttctcaggat caggaaaatt acgaaaaatt
 421  taaagcctgg gtttaaggtc tgtagaagct gcacaggtac actaatttta gtaagacggg
 481  cgccaggaaa aagaacaaaa tagtagggga gaaatattca ggcatcctaa aaaatattca
 541  gtggaaacgt aaaaacatta aagactgatt aaacatcgca gcatgacaca gatttagcaa
 601  ctgagcataa ataatttgac tcggatactg ctccaaaatc cgaagaggac caatttcttc
 661  caggaggaca actacctcgt cctctgcaga cccctctcct cggcagctga aggagtgtgg
 721  ccaatctgcc tccacctccc cgcggacccc ctactctcag gacctcctgc agcaccccaa
 781  actggaagtg gccgctgcag acccaaggac gaggggcacg cgggagccgg cagccctagt
 841  ggagcggttg gagatgttga ggtgggaggg tcacccaggt ggggtgaggc tggggtaggt
 901  agcggagtga acggcttccg aagctctggg ccgcccccag gttggactaa gcaggcgctc
 961  tgtcttcgcc cccgcccagg gtgggcgtct cctgaggact ccccgccaca cctgacccga
1021  gaccgcgcgc ccagcctaga acgcttcccc gacccagcgt agggccgccg cgactggcgc
1081  gcagggggcg gcgggaggcc tggcgaaccc gggggcggga ccaggcgggc aaggcccggc
1141  tgccgcagcg ccgctctgcg cgaggcggct ccgccgcggc ggagggatac ggcgcaccat
                                           Exon 1: 1235-1436(1235-1436)
1201  atatatatcg cggggtgcag actcgcgctc cggcagtggt gctgggagtg tcgtggacgc
1261  cgtgccgtta ctcgtagtca ggcggcggcg caggcggcgg cggcggcata gcgcacagcg
1321  cgccttagca gcagcagcag cagcagcagc atcggaggta cccccgccgt cgcagccccc
                                                      Translation start ⇓
1381  gcgctggtgc agccaccctc gctccctctg ctcttcctcc cttcgctcgc accatggtag
1441  gtcgggagtg gcaaatgccg gcgtagcagc tgcccgagat ttcttcccag atttctagtt
1501  gttttgtttg ttttttgttt gtttttggtt cttggaggtt tttcttttct gagtgttacg
1561  cagcagctgc gcttaaagga ggttgcattt tggatttgca tctcggcgac ctctgccagg
1621  gagcttcatt tattggttcc ccttggagct ggacttggtc gtaggccgtc cacgggcagg
1681  ggctccggcc gcaactgcag cgggggtttc tgcatccaat cccccctgctc cccgcccagc
                          U11886: 1-1776(1-1776) ↓ U12022: 1777-8357(1-6581)
1741  cccgcaccca ctgcatccac tagcgccgca cccggggta ccattttttc ctggcttctg
1801  aatcatagga tagtttgacc cagggcatta gccattgtaa tggtagcctt ttaacaaata
1861  actgcctaat ttaaaggatt ggaaagcatt tgttacatgg aaatgaagtt ggtggcgtac
1921  ccagttgctg tatctttatt ttttctactt aattatttct cataaaatgg atataaaagc
1981  ctgttaatcc aacccaatgc cattatgtaa cgccagtttg gagatttcga gggcctggag
2041  cagtgcgcaa ggtgcgctga aagcctgccc ctggatgaga tccttatcct ggctgtgatg
```

2101 gcagtggcag tgggctgggt cccttgttga gtggaaaggg ggactgcggt gtccatggtg
2161 cagtaggtgg cgctcttctg tcttagagcc tgccgccact gcagctggtg ccaaggggcc
2221 ttctgccact agaggtgcca tttttcacat gatgaactta gcctagttag atcgcagagc
2281 aagctgtaag ccatgggccc agaaaagaaa acttgaagtg agcagatgtt gtcacttcct
2341 tgtaatcctt tgttaaaata gcataaggag ttttctttat tctatttact ttcattaaat
2401 gaccgtgcta caggtttcaa aggattttaa gattgatttt tgaaagatca caatattaaa
2461 agtataactg gaaaatctat gttgaaatca accaaacatg tcgtggactg aatgataacc

Exon 2: 2539-2569(763-793)

2521 ttttctttct tcatataggc tgatcagctg accgaagaac agattgctgg taagttgaca
2581 actccaagga gtccccagaa ggccagaact aggcactgac tcagttttgg tgactcctct
2641 gttcctcccc gctacagtct gggcagtttt ctaagaattt atttaaataa gaacagtaag
2701 cagaaacact gaggtcagat gttattcttg ccagtacttt atagatgagg tgaaaggaag
2761 taaaactaag gatgcccaca tgttaaactc tggagaattt gaccatgttt cacaatgtgc
2821 aaagtttgcg tatgattaat tgtactgagc ctgctactca gcggtttagt ttacaattct
2881 tatgccatgg gtctttcagt aatctgccac gaaagcttgt gctcgctatc ctaaaataaa
2941 tggaaatggg tgaatatgag tgttaggacc actgtagtaa ttgggaagaa agttacatta
3001 gttaaactct gttgcccagg ctggtctcta actcctgggc tcaagcaatc ctcctgcctc
3061 agcctcctga gtagctggga ctacaggcat gtgccaccac gtctggcaga ttttagcttt
3121 ttaatattcc tggaggactt gttttgagac tgtttctcgt taggaaacca ggaatgcttc
3181 tgaaatattc taaaagtcat gtggagagag tttacctggg aatgtacatt tctagtaacc
3241 attttatttg ttatgaaaca agggattctt atggctttag aaatgtaaca ggaagggatt
3301 tgaagggggc acatggacca atcttgtcag attggattta gtcccttgaa cctgggaggc
3361 aggggttgta gtgagctgag attgcaccac tgcactccaa tctcggtgac agagcgagac
3421 tccattgttt aaaaaaaaaa aaagattgga tttaggacta atttaagcat gttccagctt
3481 agccgccttg aaacctttgg gaatattgtg gtgtgtggca ctgtttattg ggagcagtgt
3541 ttgctttatg ggctgctgta tgaaggccag tccaacagga ctattgtggt cattatttca
3601 gtagataaag accagacttc tgatacgttg cacaacttga atggctggct ttggcaagcc
3661 cccggcaagt gtgtattgtg actgggttgg ataaagacat tgattctaac gggtcaactt

Exon 3: 3731-3874(1955-2098)

3721 ttgttttcag aattcaagga agccttctcc ctatttgata aagatggcga tggcaccatc
3781 acaacaaagg aacttggaac tgtcatgagg tcactgggtc agaacccaac agaagctgaa
3841 ttgcaggata tgatcaatga agtggatgct gatggtaaga gctttaaaac catgaatgag
3901 ggccattgtt gtgtaattca agttcagaca tgttacagga ttgtctttca ggtccccaga
3961 gcaaagcaaa tgtgcaaaga tcctttctgt ggttgcccca gggccattga caattaaaat
4021 agaagatgat gggccttgcg tccatcctgc ttagtgtcta gaatgttttc tgcatgggat
4081 cactattgtt ttcttcctgc ttggtgcgac ctagagctca aatctatttt tttttttttt
4141 tttggagacg gagtctcgcc ctgtcgccca ggctggagtg gcactggcgc gatctcggct
4201 cactgcaacc tctgcctctt gggttccagc gattctcctg cgtcagcctt ctgagtagct
4261 ggaattacag gcgtgtgtcg ccacgcccag ttagtgtttt gtatctttag tagagatggg
4321 gtttcaccat gttggccagg ctggtctcaa actcctgacc tcgtgatccg ccctccccgg
4381 cctcccaaag tgctgggatt acaggcgtga accactgctc ctggccgagc tcaaagcttt
4441 tatcaactgg cccatgagtc tgcactgagt cttgaggggg gaggtgaaat taaatagcca
4501 tagaaagtgc tttttaacaa acttactgtg tttaaagagg aggaggaacc cccagatgaa
4561 gtaggtgacg agcactctta gaagttacca taaaagtgag tacagtgtga gctgtagatg
4621 tgtttgctgc agaggagcat gtgaggtttg gaggcggatg tgtggtgact ccaggggata
4681 gatttgcaga acctaacgga aagggaagct gtaaggtgca gggccagagg gaaccagcag
4741 taaccctgat agcggtctgt catctgttcc tctcgactct acagcagcgg acaacagaac
4801 tttgattgct gatttccatc agtaagcagg ctttgaagca cacttcccca cccctaaaaa
4861 aaaaccacgt attttggtaa atcctatata tattctaatg tactgtatga cagtatagaa
4921 catgattttt aaaagatgag ttgggaggag aaaaggataa aagaaaaaat aaaagaagca
4981 ttaagaataa acaattcgga tctagatttt actttctaga tgattgactc gagggtggtg
5041 tagtaaaatc gcttgtctgg tcacaaacat ttggcagcag agcttttgat taggttcttt
5101 gacaaagcct tcagcacgtt agagtggttt tcactaatag tgttttggaa agaaaaggtt
5161 gtccatagtt ctctagtttg ctaagatgat cagctaccca ggaacgtgga gtaacttcct
5221 cttgtttgtg ggagccccgg gaatctgtgc ctggggaggg gagaagtctg ttaggctctt
5281 ggattgtgtg gaagaaggag aagttgtgcc aggctacaga atcctgtgtt tgcactgaga
5341 aaacaggatg gtacctgacc ttctctgcat ggctgtgaga tagcttaaaa taatttcttt

5401 tgtttttgat gaatatgaac aatatcttaa aatttttgag gctaaaaaag tcttgaaggg
5461 atccctgagg tattttcttt gaaaggtact ggtgaaaatg agtaacttaa cctaagggtt
5521 tttctttcta attttatttc catttagttc aatgacactg ttagtctgga gtgcttgtct
5581 ttgggggtat tcatctctta gttttaaaga ggagttgttt ggagtactgg cggtagaaca
5641 gattgttctg acagttccct aagtgttact agtctgagct gtgagaatgc tcctgagctt
5701 ttcccttaat gggaaataaa gatactgagt tggaagaaaa caggtggcta accatcatag
5761 cgtggccaag aaatgatcct ggagaagact tggtaagact tcatggccca tgcatggcat
5821 aacagaatca atgttcctct ctcataatct tttctcctct gaaacacttt atacacttaa
5881 cctgcagctc agttctaggc ctttttgtg ttactgctgt cactaaccaa ggcagagtga
5941 gacctgagtg atttccctaa ctcagggatg gcagtcgggg gcgctttctt ccctcggagt
6001 ggaaagattc agcctgcgga gtggtgtatg ctatttttct tttgaactgt acagcccttc
6061 atgacccttc catgggcttg aatccagatg tgcagtttcc tttgtataat taaatactat
6121 cctgggcact gatgatgagt ttgaaattat gtgaaattgc cctgtgaagt gtttgaacgt
6181 tttagacctg cagatgattg aacctagtaa gatagtctgc ccctttgtcc tagtacatgt
6241 ttaccgttcc gtacagtggt tctgaaatga ttactgcaga gcagcgttaa tggagtgctt

Exon 4: 6334-6440(4558-4664)

6301 actttacatg agctttttgt tttttaattc gaggtaatgg caccattgac ttccccgaat
6361 ttttgactat gatggctaga aaaatgaaag atacagatag tgaagaagaa atccgtgagg
6421 cattccgagt ctttgacaag gtaatccagc atctacatag cagatggtac ttaagtatgg
6481 cttcttccgc tttcacttct aaaagctata ataatgttat agacagaaga cttaaatcta
6541 actgcctgag cctctgatct cactttcaaa aatcctcctt atggtaaccg tatcagggga
6601 gggtaggcat aataaatagg aattttggac catgtttctt gactgttact ttgaattgtt
6661 gtgagctttt gcaaatcctg ttttctgcat tagctgtttg catgtattta gtaggttaga
6721 ggtgggaact agagatcaga gaattgttta tggcagcaga gttagcagta acttgagagg
6781 gcatagctaa gtcaaagacc tacttcccca cactacatca ttagcaataa caattgctga

Exon 5: 6851-6986(5075-5210)

6841 atgttcacag gatggcaatg gttatatcag tgcagcagaa ctacgtcacg tcatgacaaa
6901 cttaggagaa aaactaacag atgaagaagt agatgaaatg atcagagaag cagatattga
6961 tggagacgga caagtcaact atgaaggtaa aactaaattc tctgagctca gtgtttcata
7021 gtcttacctt tagatctgta agcaagccaa ctgcttcact agacagcctt tgactttatt
7081 ttatgtacag taaagatgtt gtgttcatta aagctgtttt caaagataac caaaagttac

Exon 6: 7161-8066(5385-6290)

7141 tattatattt gtcttttcag aattcgtaca gatgatgact gcaaaatgaa gacctacttt
7201 caactccttt ttcccccctc tagaagaatc aaattgaatc ttttacttac ctcttgcaaa
7261 aaaaagaaaa aagaaaaaag ttcatttatt cattctgttt ctatatagca aaactgaatg
7321 tcaaaagtac cttctgtcca cacacacaaa atctgcatgt attggttggt ggtcctgtcc
7381 cctaaagatc aagctacaca tcagttttac aatataaata cttgtactac cttaatgata
7441 aggactcctt aaagttccat ttgctaatga ttaatacact gtttgggctg gccagttttt
7501 catgcatgca gcttgacgat tgagcacagt caggcctttg tattaaaaat gaaaaatgaa
7561 aaaacaaatt caaaacctat tcaaatgggt tctagttcaa tttgtttagt ataaattgtc
7621 atagctggtt tactgaaaac aaacacattt aaaattggtt tacctcagga tgacgtgcag
7681 aaaaatgggt gaaggataaa ccgttgagac gtggccccac tggtaggatg gtcctcttgt
7741 acttcgtgtg ctccgaccca tggtgacgat gacacaccct ggtggcatgc ccgtgtatgt
7801 tggtttagcg ttgtctgcat tgttctagag tgaaacaggt gtcaggctgt cactgttcac
7861 acaaattttt aataagaaac atttaccaag ggagcatctt tggactctct gtttttaaaa
7921 ccttctgaac catgacttgg agccggcaga gtaggctgtg gctgtggact tcagcacaac
7981 catcaacatt gctgttcaaa gaaattacag tttacgtcca ttccaagttg taaatgctag
8041 tctttttttt tttttttcc aataaaaaga ccattaactt aaagtggtgt taaatgcttt
8101 gtaaagctga gatctaaatg gggacaaggc aggtggaggg gaggccagtg tacatgtaaa
8161 tgcccacagc ccagcattgg gtttccctcc caaggcccca gcaccaacct ctgagcccaa
8221 gaccttgcct gaaaacaagc agataccgat tgcttcatcc tatttatgga catgtaggtc
8281 tagttgcatt ttcactgggg ggaggggga aggtgaatta tggtaacttt taatgatcta
8341 ttcaggcagt agagctc

APPENDIX III: *CaM II* GENE

This gene sequence is composed of four fragments: U94725, with basepairs1-1607 (in both global and local enumeration); U94726, with basepairs 1608-2026 (in global enumeration) or 1-419 (in local enumeration); U94727, with 2027-2540 for global and 1-514 for local fragment; and U94728, with 2541-5245 for global and 1-2705 for local fragment. Arrows '↓' are used to indicate the end/start positions of fragments. The end base of a fragment is also in bold. Exons are underlined and labeled above with exon number, global position, and fragment position in parentheses. The translation start point is indicated with '⇓'.

```
   1  tttaaatctg tggagttaga aatccaaaag ggcaaatttc ctatgaaggg gaaggcatat
  61  tatagtgacc aattgttgga gtcctaccct gtgccaggta ctttaacact ttgttaattc
 121  taatctttgc ataattctaa tttcctcaat attctttcga gataggtctt attgcctgta
 181  ttttacaagt gaggctaaaa attaaatgac tcacccaagg tcttgcacct gagagaggga
 241  cccaactaac tcatgcccag ctctgtccaa tccagaaccc tttaacccca cgataacaag
 301  gggccatttc gagatactaa tcccactgaa gctcggatac tgtgactttt gagttacttt
 361  tcttttctgg taaattaccc ttggcaaaga acgcgaagag gaagtataaa cgaaataccc
 421  tttgagatta gcctgtcagc tgaacagcta catttgcccg tgtccagatt tacatgggtt
 481  ttgtgagggc gtgaaatgaa gaacgtgcgg cttccatgag gccaaaacac taggttaaaa
 541  agctgagttc agggtacgcg tttatcggaa tctttcagct tcattcttta tcggcttaag
 601  gtgttttcgt ctacccctca ataaggggcg ggaaagtggg tgagtgtggc agggaaggat
 661  acgtggaaga ggaaggttct ccagccaagc gcaaagcagg cgactctgcc gcctggcgca
 721  gttgccagga cagcgcggcc cgagcagcaa cggcagaagc cgcaccgcgt cccggcggcg
 781  gctgcgagtc ccagtgacgc attgcaggta caaaccagcc aatcagaggc aggctcaacg
 841  accaacgggg gacgcatcgc cgccacgatg ctcctccgcg ccgcaaagta gcgcccctca
 901  ggggccgctc caccggcctt ttaacggctc gctggaaatg aaacccactt acgtcatccg
 961  ttcgcccact ttccttgatc tcttgcctca tcgaccaaat aactgtgaag aaaggggtcc
1021  cgagaagagt ttgaggggag ggggttggcc gggactcgtt tgcgatgttc cgttatctgg
                                       Exon 1: 1139-1215(1139-1215)
1081  atgcggcgga gggatctggc ggagggaggt gtttatgagg cgctgggggc ggaggaggcg
1141  aattagtccg agtggagaga gcgagctgag tggttgtgtg gtcgcgtctc ggaaaccggt
      Translation start ⇓
1201  agcgcttgca gcatggtgag tgcatcgctg agctgccggc gctgggcctg tgggggaag
1261  agactggagc gaactgacta aacaaaggaa actcgcctcc cttacacctt caaggagtgg
1321  tttctgccag agggatggcg cgaaagaggc cagggatgcc ggcgacggtt gtgtcgcagg
1381  tgctccccca cccccattct cttcaggggg gcttcctcta gtgagttgag ggccggacgc
1441  acagttcccg aggtggtgaa gctcctcggg ggcttggctg gaggaggatg ggtccgggat
1501  gagggacagc ccagtgcacg ggaaggagag aaagtgaagg aagggagaga gggagcgtag
                                             U94726: 1608-2026 (1-419)
                                U94725: 1-1607(1-1607) ↓
1561  agagccttct gtcgggcgtc tggaaaggtc aagcaggggg cgaattcttt aaacttgttt
1621  ctgtaactgc taatctacat actctcaagt cactaacctt cctctttgat ctctttgtag
      Exon 2: 1681-1711(74-104)
1681  gctgaccaac tgactgaaga gcagattgca ggtgagaaat actcagctag attgtaccca
1741  ttagattctt attataaatt gaatagccaa cgtcagaata agagacttgt atgaaataat
1801  tgactttggt atatgtcatg gatactaaca tatgggtata atataataca gtaattcaac
```

1861 ctgctttggc aagtgggatt gtaaacttgc cgatggaaga tgtgcagggc taattgtagt
1921 taaagacatg tattttagtc tgcgtggata tactttggag ccttgtggtg tagtggtggt

U94727: 2027-2540 (1-514)

U94726: 1608-2026(1-419) ↓

1981 ctttgttttg ttttctgaag actaccttag tataaaacag gaatt**c**gaat tcattttcat
2041 atgttttgct tgtcatcagc caggggagct atatcctgta gacagccaga aataaagaca
2101 tttattgagc gagatgggta gactaaagat atcttcgttg taagtgggta tagatgaagt
2161 ggcaaggagt ggaaaaaaag gataaaattg gtcacatcta gacataagag acaggagtaa
2221 gagcctttgg cagtatatgg taggaggaaa atcaaagtac tgttacgaag gccaagaaaa
2281 attttgagag tttttccagt gcagttttgg atgctgcaaa aaaatgagat agtctttatc
2341 ttgacattta ttaagtatgt atttgttgga catagttcta agtgtttgac acatactaat
2401 tttttatgg ggggtgacca ggtgaatacc ccttgtgggc atggtatact gtgtcacccc
2461 acttactggg cacatttatg tcatttcaca gaaatgagga agctgaggca gaaaaggtta

U94727: 2027-2540(1-514) ↓ U94728: 2541-5245(1-2705)

2521 gtagcttgtc catattgca**t** ctataaagat tggttaatct tttaggagtc aaagtcacat
2581 tcaattgcta cctccattgg tatttttccg ttaaaagttt ttgggttttt ctgagacgtg
2641 gggtcttaca gtttggccag gctgaagtgc agtggctgtt cactggcacg atcccaccac
2701 tgagtaggct ttaagagtta atacgtttca tttaggttac attttattat taatctgtgt

Exon 3: 2790-2933(250-393)

2761 atttggattg tggtctttct ttcaaacaga attcaaagaa gctttttcac tatttgacaa
2821 agatggtgat ggaactataa caacaaagga attgggaact gtaatgagat ctcttgggca
2881 gaatcccaca gaagcagagt tacaggacat gattaatgaa gtagatgctg atggtaagtc
2941 ttcaattttg tggggctggg aggggttgaa gaagtgatac ttagatgttt ccactaaacc

Exon 4: 3060-3166(520-626)

3001 tttgctgttt gcttatatgt ttccactaaa gctttgatgt ttccactaaa ccatttcagg
3061 taatggcaca attgacttcc ctgaatttct gacaatgatg gcaagaaaaa tgaaagacac
3121 agacagtgaa gaagaaatta gagaagcatt ccgtgtgttt gataaggtac gtcaaggact
3181 ggatatgtta aattttgagg atgaaatgaa gataccagga ctcattatga agacttgtgt
3241 ttttttggt tgttttttt ttttttttt ttttttttt tgatgattta ccaacatatt
3301 ttaattacca agcaaaagct ttttaagaat tcagtataga ctttctaata tatgtattgc
3361 gttcatcttg tcaaaggagg tgtcacttga ctttgggact aggaaataat gatttcaatg
3421 taggacattt tgcttggctc ttaattacat ttgtaggtca actgtctaga tttgtatgca
3481 tttctaaagt atgtttttcc cccaatgtga aatttagtag taaacttaat aaatttccat

Exon 5: 3596-3730(1056-1190)

3541 tccaactctc aagtctgtaa tttgactcag aaactatact aaattttttg ctaggatggc
3601 aatggctata ttagtgctgc agaacttcgc catgtgatga caaaccttgg agagaagtta
3661 acagatgaag aagttgatga aatgatcagg gaagcagata ttgatggtga tggtcaagta
3721 aactatgaag gtaaatgttt cggtctcacg cctcattctt cactttgatt tttaagataa
3781 gaggctcttt gttagggaac tgatcgaaat tagtgaccct tcccccgaaa ttaggttgct
3841 aaaacagtta cttacattta accttcttaa aattcagagt gttgtctgca gactaggtaa
3901 cattgctctg acttgagaac tcaatttaga aatatttact atcaggttct atccctacac
3961 ctggaatcag cattttaata ggattcttgg gtgattcatg tataaagttt gagaagcact
4021 gttcagtgtt tgtgtgattg tggaacaatg tggaaagcaa acttattcca agtttcctaa
4081 aatgaaatta atgtgacctt ctttccccgc actgtattga ctttttttctt taattggctt
4141 tagtctggaa ctgaagttta catgtgcttc aaaagtatcg tatctactaa cttttaagaa
4201 ggccatatca ggaagactgg catatgaaac tgaaggattt tgagtaaaac tctgtatttt
4261 tttcatgatg taactactta acttagaaaa gagcataaaa tgtaatgtta taaaaattaa
4321 acctagccca tagcattgaa tttacaatat tgtcttgtaa tccagcagac ttaaaattca
4381 cgtctaaact aagagccaag aatgctcttt tttatattgg cctatctaaa gcgttatttg
4441 gctcttacta ataaacctga atagttatgg ttaatgaagt tccagcctgg gcaattaaac
4501 tagcttccct ctgtacttaa tggtaattct gaggaaagag aggaatggaa agagtattca
4561 gcaattgagt aaagtgtatt tttttcttgg tcttttgcta ttgacttact cttttgtgta

Exon 6: 4636-5245(2096-2705)

4621 cttcttcttt ttcagagttt gtacaaatga tgacagcaaa gtgaagacct tgtacagaat
4681 gtgttaaatt tcttgtacaa aattgtttat ttgccttttc tttgtttgta acttatctgt
4741 aaaaggtttc tccctactgt caaaaaaata tgcatgtata gtaattagga cttcattcct

4801 ccatgttttc ttcccttatc ttactgtcat tgtcctaaaa ccttatttta gaaaattgat
4861 caagtaacat gttgcatgtg gcttactctg gatatatcta agcccttctg cacatctaaa
4921 cttagatgga gttggtcaaa tgagggaaca tctgggttat gcctttttta aagtagtttt
4981 ctttaggaac tgtcagcatg ttgttgttga agtgtggagt tgtaactctg cgtggactat
5041 ggacagtcaa caatatgtac ttaaaagttg cactattgca aaacgggtgt attatccagg
5101 tactcgtaca ctattttttt gtactgctgg tcctgtacca gaaacatttt cttttattgt
5161 tacttgcttt ttaaactttg tttagccact taaaatctgc ttatggcaca atttgcctca
5221 aaatccattc caagttgtat atttg

APPENDIX IV: *CaM III* GENE

This gene sequence is composed of three fragments: X52606, with basepairs 1-1863 (in both global and local enumeration); X52607, with basepairs 1864-2201 (in global enumeration) or 1-338 (in local enumeration); and X52608, with 2202-6000 for global and 1-3799 for local fragment. Arrows '↓' are used to indicate the end/start positions of fragments. The end base of a fragment is also in bold. Exons are underlined and labeled above with exon number, global position, and fragment position in parentheses. The translation start point is indicated with '⇓'.

1 caccccctatg acccaaacac ctcccaccag gccccacctc caacattggt gatcatattt
61 taacatgaga tttgagactg agtaacaagt gttaggatta aaggcatgag ccaccgcgcc
121 cggccatctt ctttacctct ccactgtgtc agttttattt ttctcttctt ctctagacac
181 tttgttcctc aagatagggg gcagagtggg aaaatgacca tagtcagaat ctatgggaag
241 gacccaccca aaaggaaaag gcttcttgtg cccaggctgc agatccaaac cccagggaag
301 gactctcatt ggttcagtct ccattctgtt ccaggtggag cataatgatt ggtccagcct
361 gactcgtttg cccatctctt aaccatcacc atggtgggga ggggacacgc aggagactgt
421 cagctcctat cctaaacggg ggagggggca ggggtgatca tttttctgag ctcctataaa
481 caaagtttca gttctttttc ctcctatggc tggaacactg gaataaggaa agaggtcgtc
541 tgtctggaga ggctggttgg ggagtatggt ctccccagcc cagccaagct caggtccctg
601 catacagcag gcgcctaatt gatgcttata gcacgaatga ctgaatgaat accccgttcc
661 cacaggctgt cctgggatgg tcctaatttc acagaccctg agcggatcaa gaatgaagcg
721 ggcccagggg tcccgcccac cggctgagtg cgtcttagga cgcagagaca agaccacgga
781 gggcggaacc tctgagcgtt aggtggcacc ttggagaccc cgccacctcc aaaggcccag
841 ttagggggcgg agcttctaaa ccctagcatt ttggattgaa ccatcgtccg tcggttaaag
901 gggagagccg agaaagccct tctgcatttg tcgccagcct ttttaccacc tcctggccac
961 agaaacgacg tatttgcttc tctaacggcg acgccgtttc cccttcgcc tcgttactcc
1021 tcaaagcccc acgcggccac agagttcaga cttacccacc accaccttgc agtggctcgc
1081 cccagtttga cgcattgcag ggaaggtcgg accaatgggc gccaagctct catttccgcc
1141 atgggccagg cccaatagtg acgcggctca cagagggacg cgtcacgacc gccgagctgt
1201 gcccctccgc cgaagaaagt gaagtgggcc cgggaggcgg ggcgcgcggc gagggaaagt
1261 agtccggcga cgggagcgag cgcgcgcgcg cccggcggca aacccaattc ctgtgcaggg
1321 tggggacgag agattcgcgg gcccgtaggt gtggagcggg gcgcggaggg atccgtggga
1381 gccgcagtgc ggcggcgcgc gggccgggtg gggcggggcc gagcgaggcg gggcgcgcgc
1441 ggcggccgtt gagggaccgt tggggcggga ggcggcggcg gcggcggcgc gcgctgcggg
1501 cagtgagtgt ggaggcgcgg acgcgcggcg gagctggaac tgctgcagct gctgccgccg

Exon 1: 1607-1609(1607-1609)

Translation start ⇓

1561 ccggaggaac cttgatcccc gtgctccgga caccccgggc ctcgccatgg tgagtgaggc
1621 tggggggtcg ccgaggctgc gggctctgag gcgggcttaa cggggcagga cccctgaggg
1681 ggcgacagag cccagagtgg ggggcgtccg ggcccggcga gagcctcggg accctttttct
1741 acccgcgttg tcgggggctg ttgaacccag agcgggacgt ctggatcacc agaggtttcc
1801 agaagcgact ttagcaccaa atgggatgtt taagtccaca aatgggtatc tgagcccccta

X52606: 1-1863(1-1863)

↓ X52607: 1864-2201(1-338)

1861 aa**g**ctctggc atgctcctcc ctggcccggc gggaatggca cgtggagctg gcctcactgg
1921 ggccgagggc ctgggctgag tgaggaagat gcgtccgtgc tgtccggcct ggtgactgac

```
                         Exon 2: 2003-2033(140-170)
1981  tttccccttc atgcttttac aggctgacca gctgactgag gagcagattg caggtgagtc
2041  tgctatcccc ctcatcttag ctcaggaaag cctccacatc cccagccaaa tcttagccac
X52607: 1864-2201(1-338)
      ↓ X52608: 2202-6000(1-3799)
2101  tgaggagtga catctgatgg gtgaacctgt gtatcccttg tgtcacctaa cactatgcct
2161  tgtgcctaga atgctgataa atgtgtgtaa gacacaccag tatgacccac tcactacctg
2221  ccctccccaa gaacaacaag gcctcagcac agggaggtgc cagcctccca aagagactct
2281  tgccatgtcc ccagggccca gaggctcagg gcacagcttc ccagctgtgc atgtccctcc
2341  caatgcacct gccacccatc tcccaactct ggccttgtgc aacataaacg gtctgcgttt
2401  tcaccctgtc cccattagca aactgtccac cgcctcaccc cagcccacac ctctgggaat
2461  aggctgccgg acatctgttt ttggagggct tgggggtaga ggacgaccag gtgccaaaat
2521  gggcccagtg ggagggcaag agccatgtgc ttttctccct gcccctgatg cggcccagct
2581  gtggccttcc cactcagcac tgctagctcc tcatcctcct ggcttggcag ccgtcccctc
2641  gctttccatc cccccatcaa ccccacgcac tgtccgtcag ccatgttctg ctgccgacca
2701  cccccacgct caccacacgc agtggctcag ctcctttgga agctggtagc acccacggca
2761  aggaacaggg ctgccagtca gaagggagtg gatgcctgca tttcctcttc aggccgcgca
2821  gacggattct gcagcctccc ctcctcgtcc cagccagcca accctggcta acacactcag
2881  acaaagaatc ctttcggctc tcattcactc catgactttg gttcatttca tcatttcaag
2941  ggagaagtca aagccaggga ccaaacagaa atgtagaatc atctaaggac agctgtcaga
3001  ccatttttccc cactgtcttc actggctggc cagaaaaata gcttcaccta gcacagtgtt
3061  gatgttcaat aactgttgaa tgaaggttgg aagggctttg cctgagcact gaggagagag
3121  agctggttgc gtgggacttg aagtctgtct cagtttcttt aggtgggact gatgcccttg
3181  gccttcctcc agggaaggca tccagcatcc agaggtaagg attctcctgt ggaccttgtg
                                       Exon 3: 3271-3414(1070-1213)
3241  acctctgact cctccccctt cttcccccag agttcaagga ggccttctcc ctctttgaca
3301  aggatggaga tggcactatc accaccaagg agttggggac agtgatgaga tccctgggac
3361  agaaccccac tgaagcagag ctgcaggata tgatcaatga ggtggatgca gatggtgagc
3421  cccacagagc gcgtgggcag ccctgctcct ggtcaccccg agtgactgca gggagcctct
3481  ctcagggtga tggatgagcc cgtgtctctc agggcccagg caagagcatt ctccatcctt
                    Exon 4: 3555-3661(1354-1460)
3541  tccccacctt ccagggaacg ggaccattga cttcccggag ttcctgacca tgatggccag
3601  aaagatgaag gacacagaca gtgaggagga gatccgagag gcgttccgtg tctttgacaa
3661  ggtaagcagc cctctccagg ggcggctctg agactgacgc cagccttcag gcagacaggc
3721  ggaactggag ccacggagct accacttcca ggaggtccgg gtcccggtgc cagcctcatt
3781  gccaacctgc tctgccacct caggcagcct ccctcactgt gctcactgct gagggatggt
3841  gatgacagcc acccctctca ctgcctctct ccccaccggg agaagtgccc agtgaaaggc
                         Exon 5: 3919-4054(1718-1853)
3901  tttatcccca acccccagga tgggaatggc tacatcagcg ccgcagagct gcgtcacgta
3961  atgacgaacc tgggggagaa gctgaccgat gaggaggtgg atgagatgat cagggaggct
4021  gacatcgatg gagatggcca ggtcaattat gaaggtgagt caaggccagg cttgatctct
4081  ggaacaagaa gagaatcgca gcttcaggaa caagccccca gatccctctt gttggggagg
                                            Exon 6: 4198-5848(1997-3647)
4141  gccgctcgcc acttagcctg cccgcctgac ctcctctctc tctgcttcac tccacagagt
4201  ttgtacagat gatgactgca aagtgaaggc cccccgggca gctggcgatg cccgttctct
4261  tgatctctct cttctcgcgc gcgcactctc tcttcaacac tcccctgcgt accccggttc
4321  tagcaaacac caattgattg actgagaatc tgataaagca acaaaagatt tgtcccaagc
4381  tgcatgattg ctctttctcc ttcttccctg agtctctctc catgcccctc atctcttcct
4441  tttgccctcg cctcttccat ccacgtcttc caaggcctga tgcattcata agttgaagcc
4501  ctccccagat ccccttggag cctctgccct cctccagccc ggatggctct cctccatttt
4561  ggtttgtttc ctcttgtttg tcatcttatt ttgggtgctg gggtggctgc cagcctgtcc
4621  cgggacctgc tgggagggac aagaggccct ccccaggcag aagagcatgc cctttgccgt
4681  tgcatgcaac cagccctgtg attccacgtg cagatcccag cagcctgttg ggcaggggt
4741  gccaagagag gcattccaga aggactgagg gggcgttgag gaattgtggc gttgactgga
4801  tgtggcccag gaggggtcga gggggccaac tcacagaagg ggactgacag tgggcaacac
4861  tcacatccca ctggctgctg ttctgaaacc atctgattgg ctttctgagg tttagctggg
```

4921 tggggactgc tcatttggcc actctgcaga ttggacttgc ccgcgttcct gaagcgctct
4981 cgagctgttc tgtaaatacc tggtgctaac atcccatgcc gctccctcct cacgatgcac
5041 ccaccgccct gagggcccgt cctaggaatg gatgtgggga tggtcgcttt gtaatgtgct
5101 ggttctcttt ttttttcttt cccctctatg gcccttaaga ctttcatttt gttcagaacc
5161 atgctgggct agctaaaggg tggggagagg gaagatgggc cccaccagct ctcaagagaa
5221 acgcacctgc aataaaacag tcttgtcggc cagctgccca gggacggcag ctacagcagc
5281 ctctgcgtcc tggtccgcca gcacctcccg cttctccgtg gtgacttggc gccgcttcct
5341 cacatctgtg ctccgtgccc tcttccctgc ctcttccctc gcccacctgc ctgcccccat
5401 actccccagc ggagagcatg atccgtgccc ttgcttctga ctttcgcctc tgggacaagt
5461 aagtcaatgt gggcagttca gtcgtctggg tttttttcccc ttttctgttc atttcatctg
5521 gctcccccca ccacctcccc accccacccc ccaccccctg cttcccctca ctgcccaggt
5581 cgatcaagtg gcttttcctg ggacctgccc agctttgaga atctcttctc atccaccctc
5641 tggcacccag cctctgaggg aaggagggat ggggcatagt gggagaccca gccaagagct
5701 gagggtaagg gcaggtaggc gtgaggctgt ggacattttc ggaatgtttt ggttttgttt
5761 tttttaaacc gggcaatatt gtgttcagtt caagctgtga agaaaaatat atatcaatgt
5821 tttccaataa aatacagtga ctacctgatt tggctcctct cctcctgtct cagtttctgg
5881 cctgcccccc tccaacacac atacacacca accccagcac tgactctttg gcccagaggt
5941 ggaagagctt ctaaagctgg ctcttgaagg ctgtgtccag tcagtgcagc ccagtggacg

APPENDIX V: IDENTIFIED CP9 TFRs FROM *CaM* GENES

Table V.(A. to C.) Identified CP9 TFRs from *CaM* genes. The non-obvious column headings are as follows. Gene: the whole gene sequence is used for base sequence number. Fragment: the sequence order is based on the fragments. Num: fragment number according to its GenBank accession number; an exon in a fragment (if any) is labeled by its order number followed by the region of the exon fragment-wide. Le: length; Co: stamp for pyrimidine to purine antiparallel conversion. 'no' indicates no conversion, 'py->pu' indicates that the sequence is conversed. Gc: G content percentage in the TFR sequence. Lo: location of an identified region (for example, "E4" means the identified region is inside exon 4; +E1, the identified regions wholly upstream of exon 1; -E1, the identified region is wholly downstream of exon 1; -E1/+E2, which is used when a region is after the last exon of one fragment, the identified region is wholly downstream of exon 1 and wholly upstream of exon 2).

Table V.A. Identified CP9 triplex DNA formation regions from *CaMI* gene.

Gene		Fragment			Le	Co	Gc	Lo	Sequence
Base No.			Base No.						
Start	End	Num	Start	End					
48	58	**U11886**	48	58	11	no	18	+E1	aagaaaagaaa
118	127	1-1776	118	127	10	py->pu	30	+E1	agagaaagaa
346	356	Exon:	346	356	11	py->pu	27	+E1	aagggaaaaaa
485	495		485	495	11	no	27	+E1	aggaaaaagaa
505	514	1:	505	514	10	no	50	+E1	aggggagaaa
691	701	1235-	691	701	11	py->pu	72	+E1	gaggagagggg
1411	1424	1436	1411	1424	14	py->pu	57	E1	gaagggaggaagag
1480	1488		1480	1488	9	py->pu	44	-E1 /+E2	gggaagaaa
1539	1550		1539	1550	12	py->pu	16	-E1 /+E2	agaaaagaaaaa
1784	1792	**U12022**	9	17	9	py->pu	22	+E2	aggaaaaaa
2133	2143	1776-	357	367	11	no	63	+E2	ggaaaggggga
2301	2311	6581	525	535	11	no	18	+E2	agaaaagaaaa
2519	2532		743	756	14	py->pu	35	+E2	gaagaaagaaaagg
2642	2650		897	904	9	py->pu	66	+E2	ggggaggaa
2752	2760		977	985	9	no	44	+E2	gaaaggaag
2983	2992	Exon:	1207	1216	10	no	50	+E3	gggaagaaag
3290	3298		1515	1523	9	no	55	+E3	aggaaggga
3431	3445	2:	1655	1669	15	no	6	+E3	aaaaaaaaaaaaaga
3743	3752	763-	1967	1976	10	py->pu	60	E3	agggagaagg
4089	4098	793	2313	2322	10	py->pu	30	+E4	aggaagaaaa
4127	4143		2351	2367	17	py->pu	0	+E4	aaaaaaaaaaaaaaaaa
4474	4484	3:	2698	2708	11	no	81	+E4	gaggggggagg
4534	4548	1955-	2758	2772	15	no	46	+E4	aaagaggaggaggaa
4698	4708	2098	2922	2932	11	no	54	+E4	ggaaagggaag
4856	4864		3081	3089	9	no	0	+E4	aaaaaaaaa
4943	4957		3167	3181	15	no	53	+E4	gggaggagaaaagga
4959	4969	4:	3183	3193	11	no	9	+E4	aaaagaaaaaa
5147	5158	4558-	3371	3382	12	no	41	+E4	ggaaagaaaagg
5215	5223		3440	3448	9	py->pu	44	+E4	aagaggaag
5253	5266	4664	3477	3490	14	no	71	+E4	ggggaggggagaag
5290	5303		3514	3527	14	no	50	+E4	ggaagaaggagaag
5519	5529	5:	3743	3753	11	py->pu	18	+E4	agaaagaaaaa
5606	5614		3831	3839	9	no	44	+E4	aaagaggag
5698	5707	5075-	3922	3931	10	py->pu	40	+E4	aagggaaaag
5732	5740		3957	3965	9	no	33	+E4	ggaagaaaa
5834	5843	5210	4058	4067	10	py->pu	50	+E4	gagagaggaa
5848	5860		4072	4084	13	py->pu	38	+E4	agaggagaaaaga
5984	5995	6:	4208	4219	12	py->pu	50	+E4	gagggaagaaag
6034	6043	5385-	4258	4267	10	py->pu	10	+E4	aaaagaaaaa
6402	6411	6290	4626	4635	10	no	30	E4	gaagaagaaa
6595	6603		4820	4828	9	no	77	-E4 /+E5	aggggaggg
6904	6913		5128	5137	10	no	30	E5	aggagaaaaa
7204	7221		5428	5445	18	py->pu	55	E6	agagggggggaaaaaggag
7258	7280		5482	5504	23	no	13	E6	aaaaaaaagaaaaaagaaaaaag
8041	8060		6265	6284	20	py->pu	15	E6	ggaaaaaaaaaaaaaaaaga
8135	8144		6359	6368	10	no	80	E6	ggaggggagg
8182	8191		6406	6415	10	py->pu	60	E6	gggagggaaa
8297	8313		6521	6537	17	no	82	E6	ggggggagggggggaagg

Table V.B. Identified CP9 triplex DNA formation regions from *CaMII* gene.

Gene		Fragment			Le	Co	Gc	Lo	Sequence
Base No.			Base No.						
Start	End	Num	Start	End					
45	55	**U94725**	45	55	11	no	63	+E1	gaaggggaagg
231	240	1-1607	231	240	10	no	60	+E1	gagagaggga
357	368	Exon:	357	368	12	py->pu	25	+E1	agaaaagaaaag
395	404		395	404	10	no	50	+E1	gaagaggaag
651	659		651	659	9	no	55	+E1	agggaagga
665	676	1:	665	676	12	no	58	+E1	ggaagaggaagg
1007	1017	1139-	1007	1017	11	no	54	+E1	gaagaaagggg
1033	1044	1215	1033	1044	12	no	83	+E1	gaggggaggggg
1101	1109		1101	1109	9	no	77	+E1	ggagggagg
1252	1263		1252	1263	12	no	66	-E1 /+E2	ggggggaagaga
1480	1488		1480	1488	9	no	66	-E1 /+E2	ggaggagga
1520	1534		1520	1534	15	no	53	-E1 /+E2	gggaaggagagaaag
1536	1555		1536	1555	20	no	60	-E1 /+E2	gaaggaagggagagagggag
1657	1667	**U94726** 1608-2026 Exon 2: 74-104	50	60	11	py->pu	45	+E2	aaagaggaagg
2171	2182	**U94727** 2027-	145	156	12	no	33	-E2 /+E3	ggaaaaaaagga
2242	2251	2540 Intron	216	225	10	no	40	-E2 /+E3	aggaggaaaa
2774	2783	**U94728**	234	243	10	py->pu	30	+E3	gaagaagaaa
3128	3137	2541-	588	597	10	no	30	E4	gaagaagaaa
3253	3281	5245	713	741	29	py->pu	0	+E5	aaaaaaaaaaaaaaaaaaaaaaaaaaaaa
3494	3503	Exon:	954	963	10	py->pu	50	+E5	tttttccccc
3649	3657	3: 250-	1109	1117	9	no	55	E5	ggagagaag
3817	3826	393	1277	1286	10	py->pu	80	+E6	gggggaaggg
4097	4108		1557	1568	12	py->pu	58	+E6	ggggaaagaagg
4121	4131	4:	1581	1591	11	py->pu	18	+E6	aaagaaaaaag
4285	4293	520-	1745	1753	9	no	33	+E6	agaaaagag
4405	4413	626	1865	1873	9	py->pu	22	+E6	aaaaaagag
4504	4512		1964	1972	9	py->pu	55	+E6	agagggaag
4531	4545	5:	1991	2005	15	no	46	+E6	gaggaaagagaggaa
4579	4588	1056-	2039	2048	10	py->pu	10	+E6	aagaaaaaaa
4621	4633	1190	2081	2093	13	py->pu	30	+E6	gaaaaagaagaag
4714	4723		2174	2183	10	py->pu	30	E6	aaagaaaagg
4747	4755	6:	2207	2215	9	py->pu	44	E6	agggagaaa
4806	4817	2096-	2266	2277	12	py->pu	33	E6	aagggaagaaaa
5147	5155	2705	2607	2615	9	py->pu	11	E6	aaaagaaaa

Table V.C. Identified CP9 triplex DNA formation regions from *CaMIII* gene.

Gene		Fragment			Le	Co	Gc	Lo	Sequence
Base No.			Base No.						
Start	End	Num	Start	End					
158	174	**X52606**	158	174	17	py->pu	29	+E1	agagaagaagagaaaaa
250	261	1-1863	250	261	12	no	33	+E1	aaaaggaaaagg
294	302		294	302	9	no	55	+E1	agggaagga
396	405	Exon:	396	405	10	no	80	+E1	ggggaggggа
438	448	1:	438	448	11	no	90	+E1	gggggaggggg
492	505	1607-	492	505	14	py->pu	35	+E1	aggaggaaaaagaa
525	535	1609	525	535	11	no	45	+E1	aaggaaagagg
897	907		897	907	11	no	54	+E1	aaaggggagag
997	1007		997	1007	11	py->pu	54	+E1	gaagggggaaa
1251	1259		1251	1259	9	no	55	+E1	gagggaaag
1732	1740		1732	1740	9	py->pu	44	-E1 /+E2	agaaaaggg
1874	1882	**X52607**	11	19	9	py->pu	66	+E2	agggaggag
1980	1990	1864- 2201 Exon 2: 140-170	117	127	11	py->pu	54	+E2	gaaggggaaag
2550	2560	**X52608**	349	259	11	py->pu	45	+E3	agggagaaaag
2780	2788	2202-	579	587	9	no	55	+E3	agaagggag
2801	2810	6000	600	609	10	py->pu	40	+E3	gaagaggaaa
2835	2846		634	645	12	py->pu	75	+E3	gaggaggggagg
2938	2947	Exon:	737	746	10	no	50	+E3	aagggagaag
3111	3122	3:	910	921	12	no	58	+E3	gaggagagagag
3182	3190		981	989	9	py->pu	66	+E3	ggaggaagg
3249	3268	1070-	1048	1067	20	py->pu	70	+E3	gggggaagaagggggaggag
3283	3296	1213	1082	1095	14	py->pu	50	E3	aaagagggagaagg
3536	3545		1335	1344	10	py->pu	60	+E4	ggggaaagga
3623	3632	4:	1422	1431	10	no	60	E4	gaggaggaga
3864	3874	1354-	1663	1671	11	py->pu	72	+E5	ggggagagagg
3972	3981	1460	1771	1780	10	no	70	E5	gggggagaag
4086	4095		1885	1894	10	no	30	+E6	aagaagagaa
4170	4183	5:	1969	1982	14	py->pu	57	+E6	agagagagaggagg
4264	4276	1718-	2063	2075	13	py->pu	46	E6	gagaagagagaga
4286	4295	1853	2085	2094	10	py->pu	50	E6	gaagagagag
4391	4409		2190	2208	19	py->pu	47	E6	agggaagaaggagaaagag
4413	4421		2212	2220	9	py->pu	55	E6	ggagagaga
4432	4443		2231	2242	12	py->pu	33	E6	aaaaggaagaga
4527	4535		2326	2334	9	py->pu	77	E6	ggaggaggg
4547	4555		2346	2354	9	py->pu	66	E6	ggaggagag
4567	4575		2366	2374	9	py->pu	33	E6	aagaggaaa
4835	4843	6:	2634	2642	9	no	55	E6	agaaggggа
5022	5031	1997-	2821	2830	10	py->pu	70	E6	gaggagggag
5103	5127	3647	2902	2926	25	py->pu	32	E6	agaggggaaagaaaaaaaaagagaa
5182	5195		2981	2994	14	no	64	E6	ggggagagggaaga
5358	5368		3157	3167	11	py->pu	63	E6	gagggaagagg
5370	5380		3169	3179	11	py->pu	63	E6	gagggaagagg
5491	5506		3290	3305	16	py->pu	31	E6	agaaaagggga aaaaa
5561	5569		3360	3368	9	py->pu	66	E6	gaggggaag
5622	5630		3421	3429	9	py->pu	44	E6	gagaagaga
5656	5669		3455	3468	14	no	64	E6	gagggaaggaggga
5799	5807		3598	3606	9	no	22	E6	gaagaaaaa
5854	5866		3653	3665	13	py->pu	61	-E6	aggaggagaggag
5885	5893		3684	3692	9	py->pu	88	-E6	ggaggggg

APPENDIX VI: UNIQUE CP9 TFRs FROM *CaM* GENES

Table VI. Unique CP9 TFRs from *CaM* genes. The TFRs are sorted by position. The total number of TFRs is 58.

Gene	Start	End	Length	Conversion	G ratio	Sequence
CaM2	45	55	11	no	63	gaaggggaagg
CaM1	48	58	11	no	18	aagaaaagaaa
CaM1	118	127	10	py->pu	30	agagaaagaa
CaM3	250	261	12	no	33	aaaaggaaaagg
CaM1	346	356	11	py->pu	27	aagggaaaaaa
CaM1	505	514	10	no	50	aggggagaaa
CaM3	525	535	11	no	45	aaggaaagagg
CaM1	691	701	11	py->pu	72	gaggagagggg
CaM3	897	907	11	no	54	aaaggggagag
CaM3	997	1007	11	py->pu	54	gaaggggaaa
CaM2	1007	1017	11	no	54	gaagaaagggg
CaM2	1101	1109	9	no	77	ggagggagg
CaM3	1251	1259	9	no	55	gagggaaag
CaM2	1252	1263	12	no	66	ggggggaagaga
CaM1	1411	1424	14	py->pu	57	gaagggaggaagag
CaM2	1520	1534	15	no	53	gggaaggagagaaag
CaM1	1784	1792	9	py->pu	22	aggaaaaaa
CaM3	1874	1882	9	py->pu	66	agggaggag
CaM3	1980	1990	11	py->pu	54	gaaggggaaag
CaM1	2133	2143	11	no	63	ggaaaggggga
CaM2	2171	2182	12	no	33	ggaaaaaaagga
CaM1	2642	2650	9	py->pu	66	ggggaggaa
CaM1	2752	2760	9	no	44	gaaaggaag
CaM2	2774	2783	10	py->pu	30	gaaagaaaga
CaM3	2780	2788	9	no	55	agaagggag
CaM3	2938	2947	10	no	50	aagggagaag
CaM3	3182	3190	9	py->pu	66	ggaggaagg
CaM3	3249	3268	20	py->pu	70	gggggaagaaggggggaggag
CaM3	3536	3545	10	py->pu	60	ggggaaagga
CaM3	3623	3632	10	no	60	gaggaggaga
CaM2	3649	3657	9	no	55	ggagagaag
CaM2	3817	3826	10	py->pu	80	gggggaaggg
CaM3	3864	3874	11	py->pu	72	ggggagagagg
CaM3	3972	3981	10	no	70	gggggagaag
CaM2	4097	4108	12	py->pu	58	ggggaaagaagg
CaM3	4170	4183	14	py->pu	57	agagagagaggagg
CaM2	4285	4293	9	no	33	agaaaagag
CaM3	4391	4409	19	py->pu	47	agggaagaaggagaaagag
CaM3	4432	4443	12	py->pu	33	aaaaggaagaga
CaM1	4474	4484	11	no	81	gaggggggagg
CaM3	4527	4535	9	py->pu	77	ggaggaggg

Table VI – continued.

Gene	Start	End	Length	Conversion	G ratio	Sequence
CaM2	4531	4545	15	no	46	gaggaaagagaggaa
CaM2	4621	4633	13	py->pu	30	gaaaaagaagaag
CaM1	4698	4708	11	no	54	ggaaagggaag
CaM3	4835	4843	9	no	55	agaagggga
CaM1	4943	4957	15	no	53	gggaggagaaaagga
CaM3	5022	5031	10	py->pu	70	gaggagggag
CaM1	5290	5303	14	no	50	ggaagaaggagaag
CaM3	5358	5368	11	py->pu	63	agggaagaggg
CaM3	5370	5380	11	py->pu	63	gagggaagagg
CaM1	5519	5529	11	py->pu	18	agaaagaaaaa
CaM3	5561	5569	9	py->pu	66	gaggggaag
CaM1	5698	5707	10	py->pu	40	aagggaaaag
CaM3	5799	5807	9	no	22	gaagaaaaa
CaM1	5834	5843	10	py->pu	50	gagagaggaa
CaM1	5848	5860	13	py->pu	38	agaggagaaaaga
CaM1	6904	6913	10	no	30	aggagaaaaa
CaM1	8182	8191	10	py->pu	60	gggagggaaa

APPENDIX VII: IDENTIFIED GENE SUBLIST CP TFR GROUPS FROM THE *CaM* GENES

Table VII. Identified gene sublist CP TFR groups from the *CaM* genes. The TFRs are sorted by query position. A query is the last TFR in a group. Some groups contain query and parent TFRs from the same gene only. The gene names of these groups are bolded. There are a total of 20 groups.

Gene	Start	End	Length	Conversion	G ratio	Sequence
CaM2	1536	1555	20	no	60	gaaggaagggagagagggag
CaM2	231	240	10	no	60	gagagaggga
CaM3	5656	5669	14	no	64	gagggaaggaggga
CaM3	294	302	9	no	55	agggaagga
CaM2	665	676	12	no	58	ggaagaggaagg
CaM2	395	404	10	no	50	gaagaggaag
CaM1	2983	2992	10	no	50	gggaagaaag
CaM1	5984	5995	12	py->pu	50	gagggaagaaag
CaM2	4806	4817	12	py->pu	33	aagggaagaaaa
CaM1	1480	1488	9	py->pu	44	gggaagaaa
CaM3	5491	5506	16	py->pu	31	agaaaaggggaaaaaa
CaM3	1732	1740	9	py->pu	44	agaaaaggg
CaM1	1539	1550	12	py->pu	16	agaaaagaaaaa
CaM2	357	368	12	py->pu	25	agaaaagaaaag
CaM1	2301	2311	11	no	18	agaaaagaaaa
CaM1	5984	5995	12	py->pu	50	gagggaagaaag
CaM1	2983	2992	10	no	50	gggaagaaag
CaM1	8041	8060	20	py->pu	15	ggaaaaaaaaaaaaaaaaga
CaM1	3431	3445	15	no	6	aaaaaaaaaaaaaga
CaM3	158	174	17	py->pu	29	agagaagaagagaaaaa
CaM3	4086	4095	10	no	30	aagaagagaa
CaM3	4264	4276	13	py->pu	46	gagaagagagaga
CaM3	4286	4295	10	py->pu	50	gaagagagag
CaM3	3111	3122	12	no	58	gaggagagagag
CaM3	4413	4421	9	py->pu	55	ggagagaga
CaM3	5854	5866	13	py->pu	61	aggaggagaggag
CaM3	4547	4555	9	py->pu	66	ggaggagag
CaM3	2801	2810	10	py->pu	40	gaagaggaaa
CaM3	4567	4575	9	py->pu	33	aagaggaaa
CaM1	3431	3445	15	no	6	aaaaaaaaaaaaaga
CaM1	4127	4143	17	py->pu	0	aaaaaaaaaaaaaaaaa
CaM1	8041	8060	20	py->pu	15	ggaaaaaaaaaaaaaaaaga
CaM3	5103	5127	25	py->pu	32	agaggggaaagaaaaaaaaa gagaa
CaM2	3253	3281	29	py->pu	0	aaaaaaaaaaaaaaaaaaaa aaaaaaaaa
CaM1	4856	4864	9	no	0	aaaaaaaaa
CaM1	7258	7280	23	no	13	aaaaaaaagaaaaaagaaaa aag
CaM1	4959	4969	11	no	9	aaaagaaaaaa

Table VII – continued.

Gene	Start	End	Length	Conversion	G ratio	Sequence
CaM1	6034	6043	10	py->pu	10	aaaagaaaaa
CaM1	2301	2311	11	no	18	agaaaagaaaa
CaM1	4959	4969	11	no	9	aaaagaaaaaa
CaM1	1539	1550	12	py->pu	16	agaaaagaaaaa
CaM2	357	368	12	py->pu	25	agaaaagaaaag
CaM1	7258	7280	23	no	13	aaaaaaaagaaaaaagaaaaaag
CaM2	5147	5155	9	py->pu	11	aaaagaaaa
CaM1	4534	4548	15	no	46	aaagaggaggaggaa
CaM1	5606	5614	9	no	44	aaagaggag
CaM3	4264	4276	13	py->pu	46	gagaagagagaga
CaM3	5622	5630	9	py->pu	44	gagaagaga
CaM1	4089	4098	10	py->pu	30	aggaagaaaa
CaM2	4806	4817	12	py->pu	33	aagggaagaaaa
CaM1	5732	5740	9	no	33	ggaagaaaa
CaM1	4959	4969	11	no	9	aaaagaaaaaa
CaM1	1539	1550	12	py->pu	16	agaaaagaaaaa
CaM1	7258	7280	23	no	13	aaaaaaaagaaaaaagaaaaaag
CaM1	6034	6043	10	py->pu	10	aaaagaaaaa

APPENDIX VIII: IDENTIFIED SD4 TFRs FROM THE *CaM* GENES

Table VIII (A. to C.) Identified SD4 TFRs from the *CaM* genes. The non-obvious column headings are as follows. Gene: the whole gene sequence is used for base sequence number. Fragment: the sequence order is based on the fragments. Num: fragment number according to its GenBank accession number; an exon in a fragment (if any) is labeled by its order number followed by the region of the exon fragment-wide. Le: length. Co: stamp for pyrimidine to purine antiparallel conversion. no indicates no conversion, py->pu indicates that the sequence is conversed. Gc: G content percentage in the TFR sequence. Lo: location of an identified region (for example, E4, the identified region is inside exon 4; +E1, the identified regions wholly upstream of exon 1; -E1, the identified region is wholly downstream of exon 1; -E1/+E2, which is used when a region is after the last exon of one fragment, the identified region is wholly downstream of exon 1 and wholly upstream of exon 2; I, the identified region is in an intron.).

Table VIII.A. Identified SD4 triplex DNA formation regions from *CaMI* gene.

Gene		Fragment			Le	Co	Gc	Lo	Sequence
Base No.			Base No.						
Start	End	Num	Start	End					
13	23	**U11886**	13	23	11	no	18	+E1	aaaacagaaga
36	44	1-1776	36	44	9	py->pu	33	+E1	aaggcagaa
118	132	Exon:	118	132	15	py->pu	26	+E1	aaagtagagaaagaa
135	148		135	148	14	no	21	+E1	agaaaaatggaaaa
196	208	1:	196	208	13	no	15	+E1	aaaaaatggaaaa
346	362	1235-	346	362	17	py->pu	23	+E1	agaaacaagggaaaaaa
485	500	1436	485	500	16	no	18	+E1	aggaaaaagaacaaaa
675	685		675	685	11	py->pu	54	+E1	agaggacgagg
729	741		729	741	13	py->pu	76	+E1	ggggaggtggagg
747	758		747	758	12	py->pu	66	+E1	gagagtaggggg
805	816		805	816	12	no	58	+E1	aaggacgagggg
859	870		859	870	12	no	75	+E1	gaggtgggaggg
881	889		881	889	9	no	77	+E1	ggggtgagg
958	967		958	967	10	py->pu	40	+E1	gaagacagag
963	973		963	973	11	py->pu	63	+E1	gggggcgaaga
1111	1120		1111	1120	10	no	80	+E1	gggggcggga
1396	1409		1396	1409	14	py->pu	64	E1	agagggagcgaggg
1402	1424		1402	1424	23	py->pu	56	E1	gaagggaggaagagcagaggg ag
1720	1733		1720	1733	14	py->pu	71	-E1 /+E2	ggggagcaggggga
1899	1908	**U12022**	124	133	10	no	40	-E1 /+E2	ggaaatgaag
1933	1946	1776-	158	171	14	py->pu	14	-E1 /+E2	agaaaaaataaaga
2081	2090	6581	306	315	10	py->pu	40	-E1 /+E2	aggataagga
2173	2184	Exon:	398	409	12	py->pu	33	-E1 /+E2	aagacagaagag
2371	2383		596	608	13	py->pu	15	-E1 /+E2	agaataaagaaaa
2634	2650	2:	859	875	17	py->pu	58	-E2 /+E3	ggggaggaacagaggag
2747	2760	763-	972	985	14	no	50	-E2 /+E3	gaggtgaaaggaag
2752	2765	793	977	990	14	no	28	-E2 /+E3	gaaaggaagtaaaa
3049	3060		1274	1285	12	py->pu	58	-E2 /+E3	gaggcaggagga
3255	3265		1480	1490	11	no	36	-E2 /+E3	gaaacaaggga
3354	3365		1579	1590	12	no	75	-E2 /+E3	gggaggcagggg
3411	3419		1636	1644	9	no	44	-E2 /+E3	agagcgaga
3718	3728		1943	1953	11	py->pu	18	-E2 /+E3	gaaaacaaaag
3958	3966	3:	2183	2191	9	no	33	-E3 /+E4	agagcaaag
4016	4026	1955-	2241	2251	11	no	18	-E3 /+E4	aaaatagaaga
4144	4153	2098	2369	2378	10	no	60	-E3 /+E4	ggagacggag
4154	4162		2379	2387	9	py->pu	55	-E3 /+E4	agggcgaga
4209	4220		2434	2445	12	py->pu	50	-E3 /+E4	aagaggcagagg
4312	4321		2537	2546	10	no	60	-E3 /+E4	agagatgggg
4474	4489		2699	2714	16	no	62	-E3 /+E4	gaggggggaggtgaaa

Table VIII.A. – continued.

Gene Base No. Start	Gene Base No. End	Fragment Num	Fragment Base No. Start	Fragment Base No. End	Le	Co	Gc	Lo	Sequence
4843	4855		3068	3080	13	py->pu	69	-E3 /+E4	aggggtggggaag
4943	4969		3168	3194	27	no	33	-E3 /+E4	gggaggagaaaaggataaaagaaaaaa
4959	4978		3184	3203	20	no	15	-E3 /+E4	aaaagaaaaaataaaagaag
5007	5017		3232	3242	11	py->pu	18	-E3 /+E4	agaaagtaaaa
5337	5348		3562	3573	12	no	33	-E3 /+E4	gagaaaacagga
5394	5407		3619	3632	14	py->pu	7	-E3 /+E4	aaaaacaaaagaaa
5532	5541		3757	3766	10	py->pu	20	-E3 /+E4	ggaaataaaa
5711	5722		3936	3947	12	no	33	-E3 /+E4	gggaaataaaga
5929	5937		4154	5162	9	no	44	-E3 /+E4	aaggcagag
5934	5942		4159	4167	9	no	44	-E3 /+E4	agagtgaga
5996	6007		4221	4232	12	no	50	-E3 /+E4	ggagtggaaaga
6220	6231		4445	4456	12	py->pu	50	-E3 /+E4	aggacaaagggg
6313	6325	4:	4538	4500	13	py->pu	7	-E3 /+E4	aaaaaacaaaaag
6378	6391	4558-	4603	4616	14	no	21	+E4	agaaaaatgaaaga
6481	6494		4706	4719	14	py->pu	42	-E4 /E5	gaaagcggaagaag
6490	6500	4664	4715	4725	11	py->pu	36	-E4 /E5	agaagtgaaag
6558	6567		4783	4792	10	py->pu	40	-E4 /E5	gaaagtgaga
6676	6686		4901	4911	11	py->pu	27	-E4 /E5	agaaaacagga
6718	6728	5:	4943	4953	11	no	54	-E4 /E5	agaggtgggaa
6983	6992	5075-	5208	5217	10	no	30	E5 /-E5	gaaggtaaaa
7022	7031	5210	5247	5256	10	py->pu	30	-E5 /E6	aaaggtaaga
7074	7082		5299	5307	9	py->pu	11	-E5 /E6	aaaataaag
7183	7192	6:	5408	5417	10	no	20	E6	aaaatgaaga
7293	7302	5385-	5518	5527	10	py->pu	20	E6	agaaacagaa
7373	7383	6290	5598	5608	11	py->pu	54	E6	aggggacagga
7545	7556		5770	5781	12	no	8	E6	aaaaatgaaaaa
7551	7564		5776	5789	14	no	14	E6	gaaaaatgaaaaaa
7827	7835		6052	6060	9	no	33	E6	agagtgaaa
7905	7916		6130	6141	12	py->pu	25	E6	aaaaacagagag
8120	8129		6345	6354	10	no	60	-E6	ggggacaagg
8253	8261		6478	6486	9	py->pu	44	-E6	aggatgaag

Table VIII.B. Identified SD4 triplex DNA formation regions from *CaMII* gene.

Gene		Fragment			Le	Co	Gc	Lo	Sequence
Base No.			Base No.						
Start	End	Num	Start	End					
82	90	**U94725**	82	90	9	py->pu	55	+E1	agggtagga
493	503	1-1607	493	503	11	no	27	+E1	gaaatgaagaa
579	589	Exon:	579	589	11	py->pu	27	+E1	aaagaatgaag
623	636		623	636	14	no	57	+E1	aaggggcgggaaag
934	943		934	943	10	no	30	+E1	ggaaatgaaa
979	989	1:	979	989	11	py->pu	45	+E1	gaggcaagaga
1125	1138	1139-	1125	1138	14	no	78	+E1	gggggcggaggagg
1383	1395	1215	1383	1395	13	py->pu	84	-E1 /E2	gggggtgggggag
1391	1404		1391	1404	14	py->pu	57	-E1 /E2	gaagagaatggggg
1496	1506		1496	1506	11	no	63	-E1 /E2	gggatgaggga
1520	1555		1520	1555	36	no	55	-E1 /E2	gggaaggagagaaagtgaaggaagggagagagggag
1775	1785	**U94726**	168	178	11	no	27	-E2 /+I	agaataagaga
1980	1989	1608-	373	382	10	py->pu	10	-E2 /+I	aaaacaaaga
1986	1996	2026 Exon 2:	378	389	11	py->pu	9	-E2 /+I	agaaaacaaaa
2014	2023	74-104	407	416	10	no	20	-E2 /+I	aaaacaggaa
2088	2098	**U94727**	62	72	11	no	18	I	agaaataaaga
2164	2182	2027-	138	156	19	no	36	I	aaggagtggaaaaaaagga
2171	2187	2540	145	161	17	no	23	I	ggaaaaaaaggataaaa
2206	2217	Intron	180	191	12	no	41	I	aagagacaggag
2213	2223		187	197	11	no	45	I	aggagtaagag
2318	2329		292	303	12	no	16	I	aaaaaaatgaga
2333	2342		307	316	10	py->pu	20	I	aagataaaga
2490	2502		464	476	13	no	38	I	agaaatgaggaag
2505	2517		479	491	13	no	46	I	gaggcagaaaagg
2890	2899	**U94728** 2541- 5245	350	359	10	no	40	E3	agaagcagag
3103	3117	Exon:	563	577	15	no	20	E4	aagaaaaatgaaaga
3197	3206	3: 250-	657	666	10	no	40	-E4 /+E5	gaggatgaaa
3203	3212	393	663	672	10	no	30	-E4 /+E5	gaaatgaaga
3534	3543		994	1003	10	py->pu	40	-E4 /+E5	ggaatggaaa
3751	3761	4:	1211	1221	11	py->pu	45	-E5 /+E6	gaagaatgagg
3756	3766	520-	1216	1226	11	py->pu	27	-E5 /+E6	aaagtgaagaa
3774	3784	626	1234	1244	11	no	36	-E5 /+E6	aagataagagg
3947	3956		1407	1416	10	py->pu	40	-E5 /+E6	agggatagaa
4079	4087	5:	1539	1547	9	no	11	-E5 /+E6	aaaatgaaa
4531	4554	1056- 1190	1991	2014	24	no	45	-E5 /+E6	gaggaaagagaggaatggaaagag
4658	4667	6:	2118	2127	10	no	30	E6	aaagtgaaga
4791	4802	2096-	2251	2262	12	py->pu	50	E6	ggaggaatgaag
4806	4822	2705	2266	2282	17	py->pu	29	E6	aagataaggaagaaaa
4841	4849		2301	2309	9	py->pu	22	E6	aaaataagg

Table VIII.C. Identified SD4 triplex DNA formation regions from *CaMIII* gene.

Gene		Fragment			Le	Co	Gc	Lo	Sequence
Base No.			Base No.						
Start	End	Num	Start	End					
32	41	**X52606**	32	41	10	py->pu	80	+E1	ggaggtgggg
127	142	1-1863	127	142	16	py->pu	43	+E1	ggagaggtaaagaaga
153	174		153	174	22	py->pu	22	+E1	agagaagaagagaaaaataa aa
180	190		180	190	11	py->pu	36	+E1	gaggaacaaag
191	201		191	201	11	no	54	+E1	aagataggggg
196	206		196	206	11	no	63	+E1	agggggcagag
203	214		203	214	12	no	41	+E1	agagtgggaaaa
318	327		318	327	10	py->pu	40	+E1	agaatggaga
324	332		324	332	9	py->pu	33	+E1	ggaacagaa
424	433		424	433	10	py->pu	50	+E1	aggataggag
438	454		438	454	17	no	82	+E1	gggggagggggcagggg
520	535	Exon:	520	535	16	no	43	+E1	ggaataaggaaagagg
652	661	1:	652	661	10	py->pu	70	+E1	gggaacgggg
709	718	1607-	709	718	10	no	30	+E1	aagaatgaag
764	773	1609	764	773	10	no	30	+E1	agagacaaga
778	788		778	788	11	no	63	+E1	ggagggcggaa
843	852		843	852	10	no	70	+E1	aggggcggag
997	1012		997	1012	16	py->pu	56	+E1	gaggcgaagggggaaa
1127	1137		1127	1137	11	py->pu	45	+E1	ggaaatgagag
1212	1224		1212	1224	13	no	38	+E1	gaagaaagtgaag
1232	1242		1232	1242	11	no	81	+E1	gggaggcgggg
1317	1326		1317	1326	10	no	70	+E1	agggtgggga
1322	1333		1322	1333	12	no	58	+E1	ggggacgagaga
1353	1361		1353	1361	9	no	77	+E1	ggagcgggg
1410	1418		1410	1418	9	no	88	+E1	ggggcgggg
1425	1433		1425	1433	9	no	77	+E1	gaggcgggg
1462	1472		1462	1472	11	no	81	+E1	ggggcgggagg
1662	1670		1662	1670	9	no	66	-E1 /E2	ggggcagga
1694	1704		1694	1704	11	no	72	-E1 /E2	agagtgggggg
1769	1777		1769	1777	9	no	55	-E1 /E2	agagcggga
2046	2058	**X52607** 1864- 2201 Exon 2: 140-70	183	195	13	py->pu	53	-E2 /+E3	aagatgaggggga
2398	2407	**X52608**	197	206	10	py->pu	40	-E2 /+E3	agggtgaaaa
2404	2413	2202-	203	212	10	py->pu	70	-E2 /+E3	ggggacaggg
2433	2441	6000	232	240	9	py->pu	77	-E2 /+E3	ggggtgagg
2492	2503		291	302	12	no	66	-E2 /+E3	gggggtagagga
2530	2542		329	341	13	no	61	-E2 /+E3	gggagggcaagag

Table VIII.C. – continued.

Gene		Fragment			Le	Co	Gc	Lo	Sequence
Base No.			Base No.						
Start	End	Num	Start	End					
2550	2566		349	365	17	py->pu	52	-E2 /+E3	aggggcagggagaaaag
2607	2620		406	419	14	py->pu	57	-E2 /+E3	aggaggatgaggag
2634	2647		433	446	14	py->pu	57	-E2 /+E3	ggaaagcgagggga
2642	2655		441	454	14	py->pu	64	-E2 /+E3	ggggggatggaaag
2760	2770	Exon:	559	569	11	no	45	-E2 /+E3	aaggaacaggg
2835	2851	3:	634	650	17	py->pu	70	-E2 /+E3	gggacgaggaggggagg
3211	3221		1010	1020	11	no	45	-E2 /+E3	agaggtaagga
3300	3310	1070-	1099	1109	11	no	45	E3	aaggatggaga
3356	3364	1213	1155	1163	9	no	44	E3	gggacagaa
3372	3380		1171	1179	9	no	44	E3	gaagcagag
3441	3450	4:	1240	1249	10	py->pu	60	-E3 /+E4	aggagcaggg
3529	3545	1354-	1328	1344	17	py->pu	52	-E3 /+E4	ggggaaaggatggagaa
3536	3552	1460	1335	1351	17	py->pu	58	E4	ggaaggtggggaaagga
3553	3563		1352	1362	11	no	54	E4	agggaacggga
3599	3612	5:	1398	1411	14	no	35	E4	agaaagatgaagga
3900	3909	1718-	1699	1708	10	py->pu	50	-E4 /+E5	ggggataaag
3917	3926	1853	1716	1725	10	no	50	+E5 /E5	aggatgggaa
4081	4095		1880	1894	15	no	33	-E5 /+E6	ggaacaagaagagaa
4170	4188		1969	1987	19	py->pu	52	-E5 /+E6	gaagcagagagagaggagg
4185	4193		1984	1992	9	py->pu	55	-E5 /+E6	ggagtgaag
4220	4229	6:	2019	2028	10	no	40	E6	aaagtgaagg
4425	4443	1997-	2224	2242	19	py->pu	47	E6	aaaaggaagagatgagggg
4432	4449	3647	2261	2248	18	py->pu	44	E6	gagggcaaaaggaagaga

Table VIII.C. – continued.

Gene Base No. Start	Gene Base No. End	Fragment Num	Fragment Base No. Start	Fragment Base No. End	Le	Co	Gc	Lo	Sequence
4445	4458		2244	2257	14	py->pu	64	E6	ggaagaggcgaggg
4521	4535		2320	2334	15	py->pu	66	E6	ggaggagggcagagg
4547	4560		2346	2359	14	py->pu	42	E6	aaaatggaggagag
4584	4592		2383	2391	9	py->pu	11	E6	aaaataaga
4632	4646		2431	2445	15	no	60	E6	gggagggacaagagg
4730	4739		2529	2538	10	no	80	E6	ggggcagggg
5141	5150		2940	2949	10	py->pu	20	E6	aaaatgaaag
5175	5195		2974	2994	21	no	57	E6	aaagggtggggagagggaag a
5304	5317		3103	3116	14	py->pu	64	E6	ggagaagcgggagg
5358	5380		3157	3179	23	py->pu	60	E6	gagggaagaggcagggaaga ggg
5428	5438		3227	3239	11	py->pu	45	E6	agaagcaaggg
5441	5451		3240	3250	11	py->pu	45	E6	agaggcgaaag
5534	5545		3333	3344	12	py->pu	83	E6	ggggtggggagg
5542	5552		3341	3351	11	py->pu	90	E6	ggggggtgggg
5547	5559		3346	3358	13	py->pu	84	E6	agggggtgggggg
5554	5569		3363	3368	16	py->pu	68	E6	gaggggaagcaggggg
5656	5674		3455	3473	19	no	68	E6	gagggaaggagggatgggg
5701	5711		3500	3510	11	no	63	E6	gagggtaaggg
5753	5765		3552	3564	13	py->pu	0	E6	aaaaaaaacaaaa
5854	5871		3653	3670	18	py->pu	55	E6	gagacaggaggagaggag
5935	5947		3734	3746	13	no	53	-E6	agaggtggaagag

APPENDIX IX: IDENTIFIED SD GENE SUBLIST TFR GROUPS FROM *CaM* GENES

Table IX: Identified SD gene sublist TFR groups from C*aM* genes. The TFRs are sorted by query position. A query is the last TFR in a group. There are a total of 13 groups.

Gene No.	Start	End	Length	Conversion	G ratio	Sequence
CaM3	5175	5195	21	no	57	aaagggtggggagagggaag a
CaM3	1317	1326	10	no	70	agggtgggga
CaM3	1232	1242	11	no	81	gggaggcgggg
CaM3	1425	1433	9	no	77	gaggcgggg
CaM2	493	503	11	no	27	gaaatgaagaa
CaM2	3203	3212	10	no	30	gaaatgaaga
CaM1	346	362	17	py->pu	23	agaaacaagggaaaaaa
CaM1	3255	3265	11	no	36	gaaacaaggga
CaM3	3529	3545	17	py->pu	52	ggggaaaggatggagaa
CaM3	3300	3310	11	no	45	aaggatggaga
CaM2	2890	2899	10	no	40	agaagcagag
CaM3	4170	4188	19	py->pu	52	gaagcagagagagaggagg
CaM3	3372	3380	9	no	44	gaagcagag
CaM2	4531	4554	24	no	45	gaggaaagagaggaatggaa agag
CaM2	3534	3543	10	py->pu	40	ggaatggaaa
CaM3	5141	5150	10	py->pu	20	aaaatgaaag
CaM1	7545	7556	12	no	8	aaaaatgaaaaa
CaM1	6378	6391	14	no	21	agaaaaatgaaaga
CaM1	7551	7564	14	no	14	gaaaaatgaaaaaa
CaM2	3103	3117	15	no	20	aagaaaaatgaaaga
CaM2	4079	4087	9	no	11	aaaatgaaa
CaM2	3756	3766	11	py->pu	27	aaagtgaagaa
CaM2	4658	4667	10	no	30	aaagtgaaga
CaM3	438	454	17	no	82	gggggagggggcagggg
CaM3	4730	4739	10	no	80	ggggcagggg
CaM1	5337	5348	12	no	33	gagaaaacagga
CaM1	6676	6686	11	py->pu	27	agaaaacagga
CaM1	1933	1946	14	py->pu	14	agaaaaaataaaga
CaM1	7074	7082	9	py->pu	11	aaaataaag
CaM1	7551	7564	14	no	14	gaaaaatgaaaaaa
CaM1	7545	7556	12	no	8	aaaaatgaaaaa

APPENDIX X: IDENTIFIED MULTIPLE OVERLAPPING SD4 TFRs FROM *CaMI* GENE

Table X: Identified multiple overlapping SD4 TFRs from *CaMI* gene. The TFRs are sorted by the first TFR's position of a group. There are a total of 7 groups.

Gene No.	Start	End	Length	Conversion	G ratio	Sequence
CaM1	958	967	10	py->pu	40	gaagacagag
CaM1	963	973	11	py->pu	63	gggggcgaaga
CaM1	1396	1409	14	py->pu	64	agagggagcgaggg
CaM1	1402	1424	23	py->pu	56	gaagggaggaagagcagagg gag
CaM1	2747	2760	14	no	50	gaggtgaaaggaag
CaM1	2752	2765	14	no	28	gaaaggaagtaaaa
CaM1	4943	4969	27	no	33	gggaggagaaaaggataaaa gaaaaaa
CaM1	4959	4978	20	no	15	aaaagaaaaaataaaagaag
CaM1	5929	5937	9	no	44	aaggcagag
CaM1	5934	5942	9	no	44	agagtgaga
CaM1	6481	6494	14	py->pu	42	gaaagcggaagaag
CaM1	6490	6500	11	py->pu	36	agaagtgaaag
CaM1	7545	7556	12	no	8	aaaaatgaaaaa
CaM1	7551	7564	14	no	14	gaaaaatgaaaaaa

APPENDIX XI: IDENTIFIED MULTIPLE OVERLAPPING SD4 TFRs FROM *CaMII* GENE

Table XI: Identified multiple overlapping SD4 TFRs from *CaMII* gene. The TFRs are sorted by the first TFR's position of a group. There are a total of 6 groups.

Gene No.	Start	End	Length	Conversion	G ratio	Sequence
CaM2	1383	1395	13	py->pu	84	gggggtgggggag
CaM2	1391	1404	14	py->pu	57	gaagagaatggggg
CaM2	1980	1989	10	py->pu	10	aaaacaaaga
CaM2	1986	1996	11	py->pu	9	agaaaacaaaa
CaM2	2164	2182	19	no	36	aaggagtggaaaaaaagga
CaM2	2171	2187	17	no	23	ggaaaaaaaggataaaa
CaM2	2206	2217	12	no	41	aagagacaggag
CaM2	2213	2223	11	no	45	aggagtaagag
CaM2	3197	3206	10	no	40	gaggatgaaa
CaM2	3203	3212	10	no	30	gaaatgaaga
CaM2	3751	3761	11	py->pu	45	gaagaatgagg
CaM2	3756	3766	11	py->pu	27	aaagtgaagaa

APPENDIX XII: IDENTIFIED MULTIPLE OVERLAPPING SD4 TFRs FROM *CaMIII* GENE

Table XII: Identified multiple overlapping SD4 TFRs from *CaMIII* gene. The TFRs are sorted by the first TFR's position of a group. The total group number is 9.

Gene No.	Start	End	Length	Conversion	G ratio	Sequence
CaM3	191	201	11	no	54	aagataggggg
CaM3	196	206	11	no	63	agggggcagag
CaM3	203	214	12	no	41	agagtgggaaaa
CaM3	318	327	10	py->pu	40	agaatggaga
CaM3	324	332	9	py->pu	33	ggaacagaa
CaM3	1317	1326	10	no	70	agggtgggga
CaM3	1322	1333	12	no	58	ggggacgagaga
CaM3	2398	2407	10	py->pu	40	agggtgaaaa
CaM3	2404	2413	10	py->pu	70	ggggacaggg
CaM3	2634	2647	14	py->pu	57	ggaaagcgagggga
CaM3	2642	2655	14	py->pu	64	ggggggatggaaag
CaM3	3529	3545	17	py->pu	52	ggggaaaggatggagaa
CaM3	3536	3552	17	py->pu	58	ggaaggtggggaaagga
CaM3	4170	4188	19	py->pu	52	gaagcagagagagaggagg
CaM3	4185	4193	9	py->pu	55	ggagtgaag
CaM3	4425	4443	19	py->pu	47	aaaaggaagagatgagggg
CaM3	4432	4449	18	py->pu	44	gagggcaaaaggaagaga
CaM3	4445	4458	14	py->pu	64	ggaagaggcgaggg
CaM3	5534	5545	12	py->pu	83	ggggtggggagg
CaM3	5542	5552	11	py->pu	90	gggggggtgggg
CaM3	5547	5559	13	py->pu	84	agggggtgggggg
CaM3	5554	5569	16	py->pu	68	gaggggaagcaggggg

APPENDIX XIII: TFRs IN UPSTEAM OF THE EXON 1 OF THE *CaM III* GENE

CP9 TFRs upstream of the exon 1 are double-underlined. SD4 TFRs upstream of the exon 1 are underlined and italicized. The legend is the same as described as in Appendix IV.

```
   1  cacccctatg acccaaacac ctcccaccag gccccacctc caacattggt gatcatattt
  61  taacatgaga tttgagactg agtaacaagt gttaggatta aaggcatgag ccaccgcgcc
 121  cggccatctt ctttacctct ccactgtgtc agttttattt ttctcttctt ctctagacac
 181  tttgttcctc aagatagggg gcagagtggg aaaatgacca tagtcagaat ctatgggaag
 241  gacccaccca aaaggaaaag gcttcttgtg cccaggctgc agatccaaac cccagggaag
 301  gactctcatt ggttcagtct ccattctgtt ccaggtggag cataatgatt ggtccagcct
 361  gactcgtttg cccatctctt aaccatcacc atggtgggga ggggacacgc aggagactgt
 421  cagctcctat cctaaacggg ggaggggg ca ggggtgatca tttttctgag ctcctataaa
 481  caaagtttca gttcttttc  ctcctatggc tggaacactg gaataaggaa agaggtcgtc
 541  tgtctggaga ggctggttgg ggagtatggt ctccccagcc cagccaagct caggtccctg
 601  catacagcag gcgcctaatt gatgcttata gcacgaatga ctgaatgaat accccgttcc
 661  cacaggctgt cctgggatgg tcctaatttc acagaccctg agcggatcaa gaatgaagcg
 721  ggcccagggg tcccgcccac cggctgagtg cgtcttagga cgcagagaca agaccacgga
 781  gggcggaacc tctgagcgtt aggtggcacc ttggagaccc cgccacctcc aaaggcccag
 841  ttaggggcgg agcttctaaa ccctagcatt ttggattgaa ccatcgtccg tcggttaaag
 901  gggagagccg agaaagccct tctgcatttg tcgccagcct ttttaccacc tcctggccac
 961  agaaacgacg tatttgcttc tctaacggcg acgccgtttc cccttcgcc tcgttactcc
1021  tcaaagcccc acgcggccac agagttcaga cttacccacc accaccttgc agtggctcgc
1081  cccagtttga cgcattgcag ggaaggtcgg accaatgggc gccaagctct catttccgcc
1141  atgggccagg cccaatagtg acgcggctca cagagggacg cgtcacgacc gccgagctgt
1201  gcccctccgc cgaagaaagt gaagtgggcc cgggaggcgg ggcgcgcggc gagggaaagt
1261  agtccggcga cgggagcgag cgcgcgcgcg cccggcggca aacccaattc ctgtgcaggg
1321  tggggacgag agattcgcgg gcccgtaggt gtggagcggg gcgcggaggg atccgtggga
1381  gccgcagtgc ggcggcgcgc gggccgggtg gggcggggcc gagcgaggcg gggcgcgcgc
1441  ggcggccgtt gagggaccgt tggggcggga ggcggcggcg gcggcggcgc gcgctgcggg
1501  cagtgagtgt ggaggcgcgg acgcgcggcg gagctggaac tgctgcagct gctgccgccg
```

Exon 1: 1607-1609(1607-1609)

Translation start ⇓

```
1561  ccggaggaac cttgatcccc gtgctccgga caccccgggc ctcgccatgg tgagtgaggc
1621  tggggggtcg ccgaggctgc gggctctgag gcgggcttaa cggggcagga cccctgaggg
1681  ggcgacagag cccagagtgg ggggcgtccg ggcccggcga gagcctcggg acccttttct
1741  acccgcgttg tcgggggctg ttgaacccag agcgggacgt ctggatcacc agaggtttcc
1801  agaagcgact ttagcaccaa atgggatgtt taagtccaca aatgggtatc tgagccccta
X52606: 1-1863(1-1863)
       ↓ X52607: 1864-2201(1-338)
1861  aagctctggc atgctcctcc ctggcccggc gggaatggca cgtggagctg gcctcactgg
1921  ggccgagggc ctgggctgag tgaggaagat gcgtccgtgc tgtccggcct ggtgactgac
```

APPENDIX XIV: SOURCE CODE OF THE PROGRAM

```
/*********************************************************************/
/* Program for identifying                                           */
/*    continuous pu/py (CP): 9 or more purines/pyrimidines           */
/*    single discontinuity (SD):                                     */
/*                  4 or more pus, py, 4 or more pus                 */
/*                  4 or more pys, pu, 4 or more pys                 */
/*    design:                                                        */
/*    read in file (to the end of file), put char codes in a list    */
/*       no space, no number will be recorded,                       */
/*       only a-z, A-Z are recorded                                  */
/*    convert char codes list into                                   */
/*       input( GATC, PuPy )                                         */
/*          GATC = [a,c,c,t,c,t,g,t,t,t,t,t,t,t,t,t,t, ...],         */
/*          PuPy = [pu,py,py,py,py,py,pu,py,py,py,py,py, ...]        */
/*    group two or more pu/py together                               */
/*          PuPy2 = [pu, pys(2,5), pu, pys(8,10), ...]               */
/*             pus( PosiStart, Length )                              */
/*             pys( PosiStart, Length )                              */
/*    search 9 or more purines/pyrimidines                           */
/*    search single discontinuity region                             */
/*       results are stored in OutPutL:                              */
/*          [region( [filename, no/py-pu],                           */
/*                   NumberOfBases, PosiStart, PosiEnd,              */
/*                   G_Ratio,                                        */
/*                   CP/SD_GATC, PuPy ), ...]                        */
/*          PuPy of CP: [pu(8, 10), 0, 0]                            */
/*          PuPy of SD: [pu(8, 10), py, pu(20, 9)]                   */
/*    write output                                                   */
/*********************************************************************/
:- use_module( library( lists ) ).
:- use_module( library( clpfd ) ).

/*****************************************************
read_input( Stream, CharCodesL )
read the next line of characters in stream Stream
return them as a list of character codes, in CharCodesL
test:
    open( 'CaM1', read, Stream ),
    read_input( Stream, CharCodesL ),
    char_to_letter( InputSL, CharCodesL, []).
*/
read_input( Stream, CharCodesL ) :-
     peek_char( Stream, PeekChar ),
     read_rest( Stream, PeekChar, CharCodesL ).

/*  end of file: PeekChar = -1
 *  stop reading
 */
read_rest( Stream, -1, [] ) :- !, close( Stream ).

/* not end of file
 * go on to read the rest
 */
read_rest( Stream, _PeekChar, Chars ) :-
     get0( Stream, Char ),
     read_rest2( Char, Stream, Chars ).

/* recursive case,
 * numbers & spaces are removed,
 * only A-Z, a-z are kept
 */
read_rest2( Char, Stream, Chars ) :-
     (( Char >= 65 , Char < 123 ) ->    % letter chars: A-Z, a-z
          Chars = [Char|RestOfChars]    % are retained
```

```
     ;
          Chars = RestOfChars ),        % nonprintables are ignored
     peek_char( Stream, Peek ),
     read_rest( Stream, Peek, RestOfChars ).


/* char_to_letter( InputL, CharCodesL, [] )
 * convert char code to a,g,c,t, and pu,py
 */
char_to_letter( input( GATCs, PuPys ) ) -->
     base_codes( GATCs, PuPys ), !.

% "eager" base_codes()
base_codes( [GATC|GATCs], [PuPy|PuPys] ) -->
     base_code( GATC, PuPy ),
     base_codes( GATCs, PuPys ), !.
% base case
base_codes( [], [] ) --> [].

% convert code to a/g/c/t and pu/py
base_code( c, py ) --> [0'c], !.
base_code( g, pu ) --> [0'g], !.
base_code( t, py ) --> [0't], !.
base_code( a, pu ) --> [0'a], !.
base_code( c, py ) --> [0'C], !.
base_code( g, pu ) --> [0'G], !.
base_code( t, py ) --> [0'T], !.
base_code( a, pu ) --> [0'A], !.


/* group_puspys( InputSL, InputL ),
 * convert [ pu, pu, py, ... ] to
 *         [ pus( 1, 2 ), py, ... ]
 * test:
 * group_puspys( input( [a,c,c,t,c,t,g,t,t,t,t,t,t,t,t,t,t,a],
 *     [pu,py,py,py,py,py,pu,py,py,py,py,py,py,py,py,py,py,pu] ),
 *     InputL ).
 * initialize the index to 1
 */
group_puspys( input( GATCs, PuPys ),
              input( GATCs, PuSPyS ) ) :-
     groups( 1, _PosiOut, PuSPyS, PuPys, [ ] ), !.

/* lazy groups
 * regular expression:
 *   [ ( (pu,pu)pu* | pu | [] ) ( (py,py)py* | py | [] ) ]*
 */
/* end of list, stop */
groups( PosiIn, PosiIn, [] ) --> [].
/* look for pu or pus, py or pys
 * then recurively look for rest
 */
groups( PosiIn, PosiOut, PuSPyS ) -->
     pu_or_pus( PosiIn, PosiOut1, Front ),
     py_or_pys( PosiOut1, PosiOut2, Rest ),
     { append( Front, Rest, FrontPuSPyS ) },
     groups( PosiOut2, PosiOut, PuSPyS2 ),
     { append( FrontPuSPyS, PuSPyS2, PuSPyS ) }.

/* look for pus */
pu_or_pus( PosiIn, PosiOut, [pus( N, L )] ) -->
     { N = PosiIn },
     con_pus( PosiIn, PosiOut, L ).
/* look for single pu */
pu_or_pus( PosiIn, PosiOut, [pu] ) -->
     [pu],
     { PosiOut #= PosiIn + 1 }.
/* no pu exit */
pu_or_pus(PosiIn, PosiOut, []) -->
     [],!,
     { PosiOut = PosiIn }.
```

```
/* if two pus found */
con_pus( PosiIn, PosiOut, L ) -->
     two_pus,
     { PosiOut1 #=PosiIn + 2 },
     more_pus( PosiOut1, PosiOut, L1 ),
     { L #= L1 + 2 }.

two_pus -->
     [pu, pu].

/* eager more pus */
more_pus( PosiIn, PosiOut, L ) -->
   [pu],
   { PosiOut1 #= PosiIn + 1 },
   more_pus( PosiOut1, PosiOut, L1 ),
   { L #= L1 + 1 }.
more_pus( PosiIn, PosiIn, 0 ) --> [].


/* look for py or pys */
py_or_pys( PosiIn, PosiOut, [pys( N, L )] ) -->
     { N = PosiIn },
     con_pys( PosiIn, PosiOut, L ).
/* look for single py */
py_or_pys( PosiIn, PosiOut, [py] ) -->
     [py],
     { PosiOut #= PosiIn + 1 }.
/* no py exit */
py_or_pys(PosiIn, PosiOut, []) -->
     [],!,
     { PosiOut = PosiIn }.

/* if two pys found */
con_pys( PosiIn, PosiOut, L ) -->
     two_pys,
     { PosiOut1 #= PosiIn + 2 },
     more_pys( PosiOut1, PosiOut, L1 ),
     { L #= L1 + 2 }.

two_pys -->
     [py, py].

/* eager more pys */
more_pys( PosiIn, PosiOut, L ) -->
   [py],
   { PosiOut1 #= PosiIn + 1 },
   more_pys( PosiOut1, PosiOut, L1 ),
   { L #= L1 + 1 }.
more_pys( PosiIn, PosiIn, 0 ) --> [].


/****************************************************************
search_cp( InputFile, InputL, Mcp, OutPutL )
      InputL: line( GATC, PuPy )
      OutPutL: [region( NumberOfBases, PosiStart, PosiEnd,
                        HomoGATC, HomoPuPy ), ...]
 test:
   search( InputFile,
           input( [a,c,c,t,c,t,g,t,t,t,t,t,t,t,t,t,a],
              [pu, pys(2,5), pu, pys(8,10), pu] ),
           OutPutL ),!.
*/
search_cp( InputFile, input( GATC, PuPy ), Mcp, OutPutL ) :-
     search_puspys( InputFile, shortL(1, GATC), Mcp, OutPutL, PuPy, []),!.

/* grammar rules for 9 or more pus/pys
 * eager for more regions
 * input:
 *      InputFile, input DNA file name
 *      ShortL, containing structure shortL(N, GATC)
```

```
 *              N starts at 1
 *      Mcp, the threshold of CP TFR of CP TFR
 * output:
 *      Region1 and MoreRegions, containing one CP TFR
 */
search_puspys( InputFile, ShortL, Mcp,
               [ Region1 | MoreRegions ]  ) -->
     bases,
     one_pus_or_pys( InputFile, ShortL, Mcp, ShorterL, Region1 ),
     search_puspys( InputFile, ShorterL, Mcp, MoreRegions ).
%no triplex regions, red cuts
search_puspys( _InputFile, _ShortL, _Mcp, [ ] ) --> bases, !.
search_puspys( _InputFile, _ShortL, _Mcp, [ ] ) --> [ ], !.

%lazy_bases
bases --> [ ].
bases --> base, bases.

%green cuts
base --> [ pu ], !.
base --> [ py ], !.
base --> [ pus( _P, _L ) ], !.
base --> [ pys( _P, _L ) ], !.

/* a pus(N,L) found, L >= 9
 * then get the region and store in region
 * input:  InputFile, shortL( Position, GATC_Posi), Mcp
 * output: shortL( PosiStart, GATC_Rest)
 *         region( [InputFile, 'no'], Length,
 *                  PosiStart, PosiEnd,
 *                  HomoGATC, HomoPuPy )
 * InputFile: the text file name, e.g., Cam1
 * Position: the point of input sequence
 * GATC_Posi: list of g,a,t,c
 * Mcp: the threshold of CP TFR of continuous pu/py
 * PosiStart: start position pus(PosiStrat, Length1)
 * GATC_Rest: list of g,a,t,c of the above pus(PosiStrat, Length1)
 * PosiEnd: end position of the pus(PosiStrat, Length1)
 * HomoGATC: list of g,a,t,c of pus(PosiStrat, Length1)
 * HomoPuPy: list of the identified pus(PosiStrat, Length1)
 */
one_pus_or_pys( InputFile, shortL( Position, GATC_Posi ),
                Mcp, shortL( PosiStart, GATC_Rest ),
                region( [InputFile, 'no'], Length,
                        PosiStart, PosiEnd,
                        HomoGATC, HomoPuPy ) ) -->
     [ pus( PosiStart, Length1 ) ],
     { Length1 >= Mcp,
       PosiEnd #= PosiStart - 1 + Length1,
       Length = Length1,
       HomoPuPy = [pus( PosiStart, Length ), 0, 0],
       PosiStart2 #= PosiStart - Position + 1,
       PosiEnd2 #= PosiEnd - Position + 1,
       /* two-step slice:
        * slice1( GATC_Posi, PosiStart2,
        *         PosiEnd3, GATC_Rest )
        * GATC_Posi: input list
        * PosiStart2: the slicing start point in
        *    the list GATC_Posi
        * PosiEnd3: the slicing end point in the list GATC_Posi
        * GATC_Rest: the resulting list containing
        *    elements from PosiStart2 to the end of GATC_Posi
        *    GATC_Rest is used for further search
        * slice2( GATC_Rest, PosiEnd4, HomoGATC )
        * GATC_Rest: input list
        * PosiEnd4: end slicing position
        * HomoGATC: return list with elements from
        *    start to PosiEnd4 of GATC_Rest
        */
       slice1( GATC_Posi, PosiStart2,
               PosiEnd3, GATC_Rest ),
```

```
       PosiEnd4 #= PosiEnd2 - PosiEnd3,
       slice2( GATC_Rest, PosiEnd4, HomoGATC ) }.

/* a pys(N,L) found, L >= 9
 * then get the region and store in region
 */
one_pus_or_pys( InputFile, shortL( Position, GATC_Posi ),
                Mcp, shortL( PosiStart, GATC_Rest ),
                region( [InputFile, 'no'], Length,
                        PosiStart, PosiEnd,
                        HomoGATC, HomoPuPy ) ) -->
     [ pys( PosiStart, Length1 ) ],
     { Length1 >= Mcp,
       PosiEnd #= PosiStart - 1 + Length1,
       Length = Length1,
       HomoPuPy = [pys( PosiStart, Length ), 0, 0],
       PosiStart2 #= PosiStart - Position + 1,
       PosiEnd2 #= PosiEnd - Position + 1,
       slice1( GATC_Posi, PosiStart2,
               PosiEnd3, GATC_Rest ),
       PosiEnd4 #= PosiEnd2 - PosiEnd3,
       slice2( GATC_Rest, PosiEnd4, HomoGATC ) }.

/*************************************************************
search SD regions
 test:
   search_sd( input( [a,c,c,t,c,t,g,t,t,t,t,t,t,t,t,t,t,a],
                     [pu, pys(2,5), pu, pys(8,10), pu] ),
              input( [a,c,c,t,c,t,g,t,t,t,t,t,t,t,t,t,t,a],
                     [pu, pys(2,5), pu, pys(8,10), pu] ),
              OutPutL ), !, writeoutput( OutPutL ).
*/
search_sd( InputFile, input( GATC, PuPy ), Nsd, OutPutL ) :-
     search_sdiscon( InputFile, shortL( 1, GATC ),
                     Nsd, OutPutL, PuPy, []).

/* grammar rules to find SD regions
 * eager to find more SD regions
 */
% two or more SD TFRs (pus,py,pus,py,pus)
search_sdiscon( InputFile, ShortL, Nsd,
               [ Region1 | Regions ] ) -->
    bases,
    sdiscon( InputFile, ShortL, Nsd, ShorterL, Region1, PuPy ),
    % PuPy: pus(PosiStart2, Length2 ) of
    % identified SD: [pus(PosiStart1, Length1),py,pus(PosiStart2, Length2)]
    lookfurther( InputFile, ShorterL, Nsd,
                 _ShorterL2, Regions, PuPy ).
% one SD TFR (pus,py,pus)
search_sdiscon( InputFile, ShortL, Nsd, Regions ) -->
    base,
    search_sdiscon( InputFile, ShortL, Nsd, Regions ).
/* no SD regions are found */
search_sdiscon( _InputFile, _ShortL, _Nsd, [] ) --> bases, !.
search_sdiscon( _InputFile, _ShortL, _Nsd, []) --> [],!.


/* look for pus(N,L), py, pus(N2,L2)
 * L & L2 >= Nsd
 */
sdiscon( InputFile, shortL( Position, GATC_Posi ),
         Nsd, shortL( PosiStart, GATC_Rest ),
         region( [InputFile, 'no'], Length,
                 PosiStart, PosiEnd,
                 HomoGATC, HomoPuPy ),
         pus( PosiStart2, Length2 ) ) -->
      [ pus( PosiStart1, Length1 ) ],
      { Length1 >= Nsd },
      single_py,
      [ pus( PosiStart2, Length2 ) ],
      { Length2 >= Nsd,
```

```
        PosiStart = PosiStart1,
        PosiEnd #= PosiStart2 - 1 + Length2,
        Length #= Length1 + Length2 + 1,
        HomoPuPy = [ pus( PosiStart1, Length1 ), py,
                     pus( PosiStart2, Length2 ) ],
        PosiStart3 #= PosiStart1 - Position + 1,
        PosiEnd2 #= PosiEnd - Position + 1,
        slice1( GATC_Posi, PosiStart3,
                PosiEnd3, GATC_Rest ),
        PosiEnd4 #= PosiEnd2 - PosiEnd3,
        slice2( GATC_Rest, PosiEnd4, HomoGATC ) }.

/* look for pys(N,L), pu, pys(N2,L2)
 * L & L2 >= Nsd
 */
sdiscon( InputFile, shortL( Position, GATC_Posi ),
         Nsd, shortL( PosiStart, GATC_Rest ),
         region( [InputFile, 'no'], Length,
                 PosiStart, PosiEnd,
                 HomoGATC, HomoPuPy ),
         pys( PosiStart2, Length2 ) ) -->
      [ pys( PosiStart1, Length1 ) ],
      { Length1 >= Nsd },
      single_pu,
      [ pys( PosiStart2, Length2 ) ],
      { Length2 >= Nsd,
        PosiStart = PosiStart1,
        PosiEnd #= PosiStart2 - 1 + Length2,
        Length #= Length1 + Length2 + 1,
        HomoPuPy = [ pys( PosiStart1, Length1 ), pu,
                     pys( PosiStart2, Length2 ) ],
        PosiStart3 #= PosiStart - Position + 1,
        PosiEnd2 #= PosiEnd - Position + 1,
        slice1( GATC_Posi, PosiStart3,
                PosiEnd3, GATC_Rest ),
        PosiEnd4 #= PosiEnd2 - PosiEnd3,
        slice2( GATC_Rest, PosiEnd4, HomoGATC ) }.

single_py --> [py].
single_pu --> [pu].

/***************************************************
 * lookfurther: when a SD TFR found, look further to find
 * overlapping SD TFRs
 * lookfurther( InputFile, shortL( Position, GATC_Posi ),
 *              Nsd, shortL( PosiStart, GATC_Rest ),
 *              [region( [InputFile, 'no'], Length,
 *                     PosiStart, PosiEnd,
 *                     HomoGATC, HomoPuPy )|R],
 *              pus( PosiStartSM, LengthSM ) )
 * InputFile: the text file name, e.g., Cam1
 * Position: the point of input sequence
 * GATC_Posi: list of g,a,t,c
 * Nsd: the boundary of continuous pu/py in SD region
 * PosiStart: start position pus(PosiStrat, Length1)
 * GATC_Rest: list of g,a,t,c of the above pus(PosiStrat, Length1)
 * PosiEnd: end position of the pus(PosiStrat, Length1)
 * HomoGATC: list of g,a,t,c of pus(PosiStrat, Length1)
 * HomoPuPy: list of the identified pus(PosiStrat, Length1)
 * PosiStartSM, LengthSM: start position and length in
 *    pus(PosiStratSM, LengthSM),
 *    this is the 2nd pu region in  previous SD region as
 *    pus(P2,L2) in [pus(P1, L1),py,pus(P2,L2)]
 */
% single py
lookfurther( InputFile, shortL( Position, GATC_Posi ),
             Nsd, shortL( PosiStart, GATC_Rest ),
             [region( [InputFile, 'no'], Length,
                      PosiStart, PosiEnd,
                      HomoGATC, HomoPuPy )|R],
             pus( PosiStartSM, LengthSM ) ) -->
```

```
     single_py,
     [ pus( PosiStart1, Length1 ) ],
     { Length1 >= Nsd,
     PosiStart = PosiStartSM,
     PosiEnd #= PosiStart1 - 1 + Length1,
     Length #= LengthSM + Length1 + 1,
     HomoPuPy = [pus( PosiStartSM, LengthSM ), py,
                 pus( PosiStart1, Length1 ) ],
     PosiStart3 #= PosiStartSM - Position + 1,
     PosiEnd2 #= PosiEnd - Position + 1,
     slice1( GATC_Posi, PosiStart3,
             PosiEnd3, GATC_Rest ),
     PosiEnd4 #= PosiEnd2 - PosiEnd3,
     slice2( GATC_Rest, PosiEnd4, HomoGATC ) },
     lookfurther( InputFile, shortL( PosiStart, GATC_Rest ),
                  Nsd, _ShorterL, R, pus( PosiStart1, Length1 ) ).
% single pu
lookfurther( InputFile, shortL( Position, GATC_Posi ),
             Nsd, shortL( PosiStart, GATC_Rest ),
             [region( [InputFile, 'no'], Length,
                      PosiStart, PosiEnd,
                      HomoGATC, HomoPuPy )|R],
             pys( PosiStartSM, LengthSM ) ) -->
     single_pu,
     [ pys( PosiStart1, Length1 ) ],
     { Length1 >= Nsd,
     PosiStart = PosiStartSM,
     PosiEnd #= PosiStart1 - 1 + Length1,
     Length #= LengthSM + Length1 + 1,
     HomoPuPy = [pys( PosiStartSM, LengthSM ), pu,
                 pys( PosiStart1, Length1 ) ],
     PosiStart3 #= PosiStartSM - Position + 1,
     PosiEnd2 #= PosiEnd - Position + 1,
     /* two step slice:
      * the first gets I to 1, then gets the rest list
      * the second gets K to 1, then gets the sliced region
      * slice1( List1, I, K, List2 )
      * List1: input list
      * List2: sliced/output list
      * I: position slice starts in the List1
      * K: position slice stops in the List1
      */
     slice1( GATC_Posi, PosiStart3,
             PosiEnd3, GATC_Rest ),
     PosiEnd4 #= PosiEnd2 - PosiEnd3,
     slice2( GATC_Rest, PosiEnd4, HomoGATC ) },
     lookfurther( InputFile, shortL( PosiStart, GATC_Rest ),
                  Nsd, _ShorterL, R, pys( PosiStart1, Length1 ) ).
% no further SM region found.
lookfurther( InputFile, ShortL,
             Nsd, _ShorterL,
             Regions,
             _PuPy ) -->
     search_sdiscon( InputFile, ShortL, Nsd, Regions ).

/* two step slice:
 * the first gets I to 1, then gets the rest list
 * the second gets K to 1, then gets the sliced region
 * slice1( List1, I, K, List2 )
 * List1: input list
 * List2: sliced/output list
 * I: position slice starts in the List1
 * K: position slice stops in the List1
 * test:
 * slice1([a,b,c,d,e,f,g], 3, K, L),
 *  K1 #= 5-K,
 *  slice2(L, K1, Lout).
 */

slice1( Xs, I, K, Xs ) :-
     I #= 1, K #= 0,!.
```

```
slice1( [_I01,_I02,_I03,_I04,_I05,_I06,_I07,_I08,_I09,_I10,
        _I11,_I12,_I13,_I14,_I15,_I16,_I17,_I18,_I19,_I20,
        _I21,_I22,_I23,_I24,_I25,_I26,_I27,_I28,_I29,_I30,
        _I31,_I32,_I33,_I34,_I35,_I36,_I37,_I38,_I39,_I40,
        _I41,_I42,_I43,_I44,_I45,_I46,_I47,_I48,_I49,_I50,
        _I51,_I52,_I53,_I54,_I55,_I56,_I57,_I58,_I59,_I60,
        _I61,_I62,_I63,_I64,_I65,_I66,_I67,_I68,_I69,_I70,
        _I71,_I72,_I73,_I74,_I75,_I76,_I77,_I78,_I79,_I80,
        _I81,_I82,_I83,_I84,_I85,_I86,_I87,_I88,_I89,_I90,
        _I91,_I92,_I93,_I94,_I95,_I96,_I97,_I98,_I99,_I100 | Xs],
       I, K, Ys ) :-
     I #> 100,
     I1 #= I - 100,
     K1 #= K - 100,
     slice1( Xs, I1, K1, Ys ).
slice1( [_I01,_I02,_I03,_I04,_I05,_I06,_I07,_I08,_I09,_I10 | Xs],
        I, K, Ys ) :-
     I #> 10,
     I1 #= I - 10,
     K1 #= K - 10,
     slice1( Xs, I1, K1, Ys ).
slice1( [_|Xs], I, K, Ys ) :-
     I #> 1,
     I1 #= I - 1,
     K1 #= K - 1,
     slice1( Xs, I1, K1, Ys ).

slice2( [], K, [] ) :-
     K #= 1.
slice2( [X|_], K, [X] ) :-
     K #= 1.
slice2( [I01,I02,I03,I04,I05,I06,I07,I08,I09,I10 | Xs],
        K,
       [I01,I02,I03,I04,I05,I06,I07,I08,I09,I10 | Ys] ) :-
     K #> 10,
     K1 #= K - 10,
     slice2( Xs, K1, Ys ).
slice2( [X|Xs], K, [X|Ys] ) :-
     K #> 1,
     K1 #= K - 1,
     slice2( Xs, K1, Ys ).


/* output a list one by one
 */
writeoutput( [] ).
writeoutput( [region( [Gene, Convert], N, S, E, Bases, _PuorPy ) | L] ) :-
   format( "~t~a~5+~t~a~8+~t~d~6+~t~d~10+~t~d~7+~t~40|",
           [Gene, Convert, N, S, E] ),
   writelist( Bases ),
   nl,
   writeoutput( L ).

/* writelist(Y) prints the contains of the list to the screen */
writelist( [] ).
writelist( [X|L] ) :-
     write( X ),
     writelist( L ).

/* writesublist
 * used for sublist output
 */
writesublist( [] ).
writesublist( [H | L] ) :-
   rwriteoutput( H ),
   nl,
   writesublist( L ).

/* output a list one by one
 */
rwriteoutput( [] ).
```

```
rwriteoutput( [region( [Gene, Convert], N, S, E, R, Bases, _PuorPy ) | L] ) :-
   format( "~t~a~5+~t~a~8+~t~d~6+~t~d~10+~t~d~7+~t~d~5+~t~46|",
           [Gene, Convert, N, S, E, R] ),
   writelist( Bases ),
   nl,
   rwriteoutput( L ).

/*************************************************
 * check sublist: search the sorted (by length) list with
 * the first element, if the first element is a sublist
 * of a TFR, they are grouped together;
 * then search with the second element …
 * through the rest of the list
 * check_sublist( SortList, SubList ):
 * Input: SortList - sorted (by length, ascending) TFR list
 * Output: SubList - grouped TFR list
 */
% nothing to do if list is empty
check_sublist( [], [] ).
% don't search the last one
check_sublist( [H|R], [SubList2|SubList] ) :-
     base_sublist( [H|R], SubList2 ),
     check_sublist( R, SubList ).

/* base_sublist( SortList, OneGroupList):
 *   Input: SortList - sorted (by length, ascending) TFR list
 *   Output: OneGroupList - grouped list containing the first
 *                         element and its parent TFRs
 * An accumulator technique is used for building up the
 * output argument
 */
% add the query to the end list
base_sublist( [H], [H] ).
% if the query is a sublist of the second item
% output the second item in the Output list
base_sublist( [region( Gene1, N1, S1, E1, R1, Base1, PuPy1 ),
               region( Gene2, N2, S2, E2, R2, Base2, PuPy2 )|R],
           [region( Gene2, N2, S2, E2, R2, Base2, PuPy2 )|SubList] ):-
     /* my_sublist determines if Base1 is a sublist of Base2
     my_sublist( Base1, Base2 ),
     base_sublist( [region( Gene1, N1, S1, E1, R1, Base1, PuPy1 )|R],
                    SubList ).
% skip one if not a sublist
base_sublist( [H1, _H2|R], SubList ) :-
      base_sublist( [H1|R], SubList ).

/* when a specific TFO
 * used as a query for sublist search
 */
single_sublist( _Q, [], [] ).
single_sublist( Q,
                [region( Gene1, N1, S1, E1, R1,
                         Base1, PuPy1 )|R],
                [region( Gene1, N1, S1, E1, R1,
                         Base1, PuPy1 )|SubList] ) :-
     my_sublist( Q, Base1 ),
     single_sublist( Q, R, SubList ).
single_sublist( Q,
                [_H|R],
                SubList ) :-
     single_sublist( Q, R, SubList ).

/* my_sublist
 * call prefix and
 * check whether L1 #= sublist of L2
 */
my_sublist( [X|L], [X|M] ) :-
     prefix( L, M ), !.
my_sublist( L, [_|M] ) :- my_sublist( L, M ).
```

```
/* complement bases */
pypu_complement( [], [] ).
% pu: when a pus group is found, don't change anything
%     about it, go on with the rest
pypu_complement( [region( Gene, N, S, E,
                          AAGAG, [pus( Start, Length ),N1,N2] )|Res],
                 [region( Gene, N, S, E,
                          AAGAG, [pus( Start, Length ),N1,N2] )|Complement] ) :-
     pypu_complement( Res, Complement ).
% py to pu
pypu_complement( [region( [Gene, _Con], N, S, E, CTCTT,
                          [pys( Start, Length ),N1,N2] )|Res],
            [region( [Gene, 'py->pu'], N, S, E, AAGAG,
                     [pys( Start, Length ),N1,N2] )|Complement] ) :-
     reverse( CTCTT, TTCTC ),
     ta_cg_convert( AAGAG, TTCTC, [] ),
     pypu_complement( Res, Complement ).

/* eager t,c to a,g; a,g to t,c */
ta_cg_convert( [H|Res] ) -->
     complement( H ),
     ta_cg_convert( Res ).
ta_cg_convert( [] ) --> [],!.

complement( a ) --> [t],!.
complement( g ) --> [c],!.
complement( c ) --> [g],!.
complement( t ) --> [a],!.

/**********************************************
search, sort, and check sublist
for homopurine/homopyrimidine,
then look for TFRs which are sublists of longer
TFRs in the same gene.
e.g.

 CaM2      no    20      1536   1555    gaaggaagggagagagggag
 CaM2      no    10       231    240    gagagaggga

and

 CaM1  py->pu    12      1539   1550    agaaaagaaaaa
 CaM2  py->pu    12       357    368    agaaaagaaaag
 CaM1      no    11      2301   2311    agaaaagaaaa

will be collected and reported.

Search stretigy:
for each input item (list), if length >= 2,
use the last item's gene name (e.g. 'CaM1')
search from the 1st item to the last second item,
compare gene name, if matched, put into result list
*/
/**********************************************
check_ingene( Input, Output )
if find 2 or more in the same gene,
collect the list,
if not, search the next
*/
check_ingene( [], [] ).
check_ingene( [H|R], [H|Res] ) :-
     length( H, Len ),
     Len >= 2,
     ingene( H ),
     check_ingene( R, Res ).
check_ingene( [_H|R], Res ) :-
     check_ingene( R, Res ).

/**********************************************
ingene( Input )
find the gene name of the last item
```

```
compare the name with that of the rest of the list
if matched, true, otherwise false
*/
ingene( Input ) :-
     last( Input,
           region( [GeneName, _Con], _N, _S, _E, _R,
                    _GATC, _PuPys ) ),
     same_name( Input, GeneName ).

% cut: don't compare the last one, the last one is the query
same_name( [_H], _GeneName ) :- !, fail.
same_name( [region( [GeneName1, _Con], _N, _S, _E, _R,
                    _GATC, _PuPys ) |_Res], GeneName ) :-
     GeneName1 == GeneName.
same_name( [_H|R], GeneName ) :-
     same_name( R, GeneName ).

/* to identify unique TFRs
 * unique( Input, Output ):
 * input: Input, a list contains contains TFR sublists
 * output: Output, a list contains lists with only
 *         one element (single TFR)
 */
unique( [], [] ).
unique( [H|R1], [H1|R2] ) :-
     length( H, Length ),
     Length == 1,
     H = [H1],
     unique( R1, R2 ).
unique( [H|R1], R2 ) :-
     length( H, Length ),
     Length > 1,
     my_select( H, R1, R3 ),
     unique( R3, R2 ).

/* to remove (longer) single sublists from the rest of list
 * because these (longer) single sublists are a part of shorter
 * sublist.  E.g.,
 *   [ [[a,g,t,c], [a,a,g,g,a,g,t,c]], [a,a,g,g,a,g,t,c] ]
 * the single sublist [a,a,g,g,a,g,t,c] is removed because
 * it is not really unique (it's substring matches [a,g,t,c]
 * my_select( H, R1, R3 )
 * Input:  H  - query sublist, each element will be used to search
 *              the input list R1
 *         R1 - list containing TFR sublists
 * Output: R3 - rest of R1 list with un-unique single sublist removed
 */
my_select( [_], R, R ).
my_select( [H|R1], R2, R3 ) :-
     select2( [H], R2, R2a ),
     my_select( R1, R2a, R3 ).

select2( _H, [], [] ).
select2( H, [H|R1], R ) :-
     select2( H, R1, R).
select2( H, [H1|R1], [H1|R2] ) :-
     H \== H1,
     select2( H, R1, R2 ).

/******************************************************
 * look for multiple overlapping SM TFRs
 * pack them into sublist
 * look_furtherSD( X1, [X2|Xs], [X2|Xs], X1 )
 * X1: the 1st item - region( [GeneName1, Con1], N1, S1, E1, R1,
                        GATC1, PuPys1 ),
 * X2: the 2nd item - region( [GeneName2, Con2], N2, S2, E2, R2,
                        GATC2, PuPys2 )
 * Xs: the rest of the list
 * the program compares the start position (S2) of the 2nd item
 * with the end position (E1) of the 1st item
 * if S2 > E1, a multiple overlapping SD identified
```

```
*/
look_conSD( [], [] ).
look_conSD( [X|Xs], [Z|Zs] ) :-
     look_furtherSD( X, Xs, Ys, Z ),
     look_conSD( Ys, Zs ).

look_furtherSD( X, [], [], [X] ).
look_furtherSD( region( [GeneName1, Con1], N1, S1, E1, R1,
                        GATC1, PuPys1 ),
               [region( [GeneName2, Con2], N2, S2, E2, R2,
                        GATC2, PuPys2 ) | Res],
               [region( [GeneName2, Con2], N2, S2, E2, R2,
                        GATC2, PuPys2 ) | Res],
               [region( [GeneName1, Con1], N1, S1, E1, R1,
                        GATC1, PuPys1 )] ) :-
     S2 > E1.
look_furtherSD( region( [GeneName1, Con1], N1, S1, E1, R1,
                        GATC1, PuPys1 ),
               [region( [GeneName2, Con2], N2, S2, E2, R2,
                        GATC2, PuPys2 ) | Res],
               Rcon, [region( [GeneName1, Con1], N1, S1, E1, R1,
                        GATC1, PuPys1 ) | Zs] ) :-
     S2 < E1,
     look_furtherSD( region( [GeneName2, Con2], N2, S2, E2, R2,
                        GATC2, PuPys2 ), Res, Rcon, Zs ).

/*********************************************************
remove single item list
keep continuous SM TFRs
Input:
[[region([CaM3,no],11,5542,5552,[c,c,c,c,a,c,c,c,c,c,c],[pys(5542,4),pu,pys(5547,6)]),

region([CaM3,no],13,5547,5559,[c,c,c,c,c,c,a,c,c,c,c,c,t],[pys(5547,6),pu,pys(5554,6)],
       [region(...)]]
Out:
[[region([CaM3,no],11,5542,5552,[c,c,c,c,a,c,c,c,c,c,c],[pys(5542,4),pu,pys(5547,6)]),

region([CaM3,no],13,5547,5559,[c,c,c,c,c,c,a,c,c,c,c,c,t],[pys(5547,6),pu,pys(5554,6)]]
*/
remove_single( [], [] ).
remove_single( [H|R], [H|Res] ) :-
     length( H, Len ),
     Len > 1,
     remove_single( R, Res ).
remove_single( [_H|R], Res ) :-
     remove_single( R, Res ).

/*******************************************************************
Base ratio: 75% A for pyrimidine motif TFO
75% G for purine motif TFO
ratio([region(['CaM1',no],11,1,11,[a,a,a,g,g,a,a,a,g,a,a], [pus,pys]),
       region(['CaM1',no],15,13,28,[a,a,g,g,a,g,g,a,g,g,g,g,a,a,g,g], [pus,pys])],
R),
rmerge_sort(R, Sort).
*/
ratio( [], [] ).
ratio( [region( [Gene, Convert], N, S, E, Bases, PuorPy ) | L1],
       [region( [Gene, Convert], N, S, E, R, Bases, PuorPy ) | L2] ) :-
     count( Bases, R ),
     ratio( L1, L2 ).

count( Bases, R ) :-
     count_g( Bases, Len, G_num ),
     R is G_num * 100 // Len.

count_g( [], 0, 0 ).
count_g( [H|R], Len, G_num ) :-
     H == 'g',
     G_num #= G_num2 + 1,
     Len #= Len2 + 1,
     count_g( R, Len2, G_num2 ).
```

```
count_g( [_H|R], Len, G_num ) :-
     %H == 'a',
     Len #= Len2 + 1,
     count_g( R, Len2, G_num ).

/*************************************************
                  mergesort part
**************************************************/
order( region( _Gene1, N1, _S1, _E1, _R1, _Base1, _Pu1 ),
       region( _Gene2, N2, _S2, _E2, _R2, _Base2, _Pu2 ) ) :-
       N1 =< N2, !.

/* merge_sort( L1, L2 ): L1 contains unsorted list,
 *     L2 contains stable merge sorted list
 * (list, list)
 */
merge_sort( [], [] ).
merge_sort( [X], [X] ).
merge_sort( List, Sorted ) :-
    divide( List, L1, L2 ),
    merge_sort( L1, Sorted1 ),
    merge_sort( L2, Sorted2 ),
    merge( Sorted1, Sorted2, Sorted), !.

/* merge( L1, L2, L ): L1 and L2 are input lists
 *     L contains the list with merged L1 and L2
 * (list, list, list)
 */
merge( [], L, L ).
merge( L, [], L ) :- L \== [], !.
merge( [X|T1], [Y|T2], [X|T] ) :-
     order( X, Y),
     merge( T1, [Y|T2], T ).
merge( [X|T1], [Y|T2], [Y|T] ) :-
     \+( order( X, Y) ),
     merge( [X|T1], T2, T ).

/********************************************************
                 re-order merge sort
*********************************************************/
/**************************************
rmerge_sort( Flag, Flag2, L1, L2 ):
Flag: indicate the item sorting based on
L1 contains unsorted list,
L2 contains stable merge sorted list
Flag2: 1 (ascending) or 2 (descending)
*/
rmerge_sort( _Flag, _Flag2, [], [] ).
rmerge_sort( _Flag, _Flag2, [X], [X] ).
rmerge_sort( Flag, Flag2, List, Sorted ) :-
    divide( List, L1, L2 ),
    rmerge_sort( Flag, Flag2, L1, Sorted1 ),
    rmerge_sort( Flag, Flag2, L2, Sorted2 ),
    rmerge( Flag, Flag2, Sorted1, Sorted2, Sorted), !.

/**********************************************
rmerge( Flag, Flag2, L1, L2, L ):
Flag: indicate the item sorting based on
L1 and L2 are input lists
L contains the list with merged L1 and L2
Flag2: 1 (ascending) or 2 (descending)
*/
rmerge( _Flag, _Flag2, [], L, L ).
rmerge( _Flag, _Flag2, L, [], L ) :- L \== [], !.
rmerge( Flag, Flag2, [X|T1], [Y|T2], [X|T] ) :-
     rorder( Flag, Flag2, X, Y),
     rmerge( Flag, Flag2, T1, [Y|T2], T ).
rmerge( Flag, Flag2, [X|T1], [Y|T2], [Y|T] ) :-
     \+( rorder( Flag, Flag2, X, Y) ),
     rmerge( Flag, Flag2, [X|T1], T2, T ).
```

```
/************************************************
msd_merge_sort:
for multiple continuous SD TFRs
*/
msd_merge_sort( _Flag, _Flag2, [], [] ).
msd_merge_sort( _Flag, _Flag2, [X], [X] ).
msd_merge_sort( Flag, Flag2, List, Sorted ) :-
    divide( List, L1, L2 ),
    msd_merge_sort( Flag, Flag2, L1, Sorted1 ),
    msd_merge_sort( Flag, Flag2, L2, Sorted2 ),
    msd_merge( Flag, Flag2, Sorted1, Sorted2, Sorted), !.

/************************************************
msd_merge( Flag, Flag2, L1, L2, L ):
Flag: indicate the item sorting based on
L1 and L2 are input lists
L contains the list with merged L1 and L2
*/
msd_merge( _Flag, _Flag2, [], L, L ).
msd_merge( _Flag, _Flag2, L, [], L ) :- L \== [], !.
msd_merge( Flag, Flag2, [X|T1], [Y|T2], [X|T] ) :-
     msd_order( Flag, Flag2, X, Y),
     msd_merge( Flag, Flag2, T1, [Y|T2], T ).
msd_merge( Flag, Flag2, [X|T1], [Y|T2], [Y|T] ) :-
     \+( msd_order( Flag, Flag2, X, Y) ),
     msd_merge( Flag, Flag2, [X|T1], T2, T ).

/************************************************
uni_merge_sort:
for sublist TFRs
*/
uni_merge_sort( _Flag, _Flag2, [], [] ).
uni_merge_sort( _Flag, _Flag2, [X], [X] ).
uni_merge_sort( Flag, Flag2, List, Sorted ) :-
    divide( List, L1, L2 ),
    uni_merge_sort( Flag, Flag2, L1, Sorted1 ),
    uni_merge_sort( Flag, Flag2, L2, Sorted2 ),
    uni_merge( Flag, Flag2, Sorted1, Sorted2, Sorted), !.

/************************************************
uni_merge( Flag, L1, L2, L ):
Flag: indicate the item sorting based on
L1 and L2 are input lists
L contains the list with merged L1 and L2
*/
uni_merge( _Flag, _Flag2, [], L, L ).
uni_merge( _Flag, _Flag2, L, [], L ) :- L \== [], !.
uni_merge( Flag,  Flag2,[X|T1], [Y|T2], [X|T] ) :-
     uni_order( Flag,  Flag2,X, Y),
     uni_merge( Flag,  Flag2,T1, [Y|T2], T ).
uni_merge( Flag, Flag2, [X|T1], [Y|T2], [Y|T] ) :-
     \+( uni_order( Flag, Flag2, X, Y) ),
     uni_merge( Flag, Flag2, [X|T1], T2, T ).

/************************************************
divide( L, L1, L2 ):
divide list L into two lists L1 and L2
L1 contains half items (n/2) of L
     if there is even number (n) items in L;
     (n-1)/2 items if there is odd number (n) items in L.
L2 contains the rest of items in L.
*/
divide( L, L1, L2 ):-
     length( L, Len ),
     Len mod 2 =:= 0,     % even number of items in the list
     LenHalf is Len // 2,
     divide2( L, LenHalf, L1, L2 ).
divide( L, L1, L2 ) :-
     length( L, Len ),
     Len mod 2 =\= 0,     % odd number of items in the list
     LenHalf is ( Len - 1 ) // 2,
```

```
     divide2( L, LenHalf, L1, L2 ).

/**********************************************
divide2(L,N,L1,L2):
the list L1 contains the first N elements
of the list L, the list L2 contains the remaining elements.
*/
divide2( L, 0, [], L ).
divide2( [X|Xs], N, [X|Ys], Zs ) :-
     N > 0,
     N1 #= N - 1,
     divide2( Xs, N1, Ys, Zs ).

/**********************************************
rorder( Flag, Flag2, X, Y )
Flag = p: order by position in gene, default
Flag = l: order by length
Flag = r: order by G ratio
Flag2 = 1 (ascending) or 2 (descending)
*/
rorder( p, Flag2, region( _Gene1, _N1, S1, _E1, _R1, _Base1, _Pu1 ),
       region( _Gene2, _N2, S2, _E2, _R2, _Base2, _Pu2 ) ) :-
       sort_type( Flag2, S1, S2 ).
rorder( l, Flag2, region( _Gene1, N1, _S1, _E1, _R1, _Base1, _Pu1 ),
       region( _Gene2, N2, _S2, _E2, _R2, _Base2, _Pu2 ) ) :-
       sort_type( Flag2, N1, N2 ).
rorder( r, Flag2, region( _Gene1, _N1, _S1, _E1, R1, _Base1, _Pu1 ),
       region( _Gene2, _N2, _S2, _E2, R2, _Base2, _Pu2 ) ) :-
       sort_type( Flag2, R1, R2 ).

/**********************************************
msd_order: msd ordering based on different data
structure
*/
msd_order( p, Flag2, [region( _Gene1, _N1, S1, _E1, _R1, _Base1, _Pu1 )|_Res1],
       [region( _Gene2, _N2, S2, _E2, _R2, _Base2, _Pu2 )|_Res2] ) :-
       sort_type( Flag2, S1, S2 ).
msd_order( l, Flag2, [region( _Gene1, N1, _S1, _E1, _R1, _Base1, _Pu1 )|_Res1],
       [region( _Gene2, N2, _S2, _E2, _R2, _Base2, _Pu2 )|_Res2]  ) :-
       sort_type( Flag2, N1, N2 ).
msd_order( r, Flag2, [region( _Gene1, _N1, _S1, _E1, R1, _Base1, _Pu1 )|_Res1],
       [region( _Gene2, _N2, _S2, _E2, R2, _Base2, _Pu2 )|_Res2] ) :-
       sort_type( Flag2, R1, R2 ).

/**********************************************
uni_order:
different structure
*/
uni_order( p, Flag2, region( _Gene1, _N1, S1, _E1, _R1, _Base1, _Pu1 ),
       region( _Gene2, _N2, S2, _E2, _R2, _Base2, _Pu2 ) ) :-
       sort_type( Flag2, S1, S2 ).
uni_order( p, Flag2, region( _Gene1, _N1, S1, _E1, _R1, _Base1, _Pu1 ),
       ListItem ) :-
       is_list( ListItem ),
       last( ListItem,
             region( _Gene2, _N2, S2, _E2, _R2, _Base2, _Pu2 ) ),
       sort_type( Flag2, S1, S2 ).
uni_order( p, Flag2, ListItem,
       region( _Gene2, _N2, S2, _E2, _R2, _Base2, _Pu2 ) ) :-
       is_list( ListItem ),
       last( ListItem,
             region( _Gene1, _N1, S1, _E1, _R1, _Base1, _Pu1 ) ),
       sort_type( Flag2, S1, S2 ).
uni_order( p, Flag2, ListItem1, ListItem2 ) :-
       is_list( ListItem1 ), is_list( ListItem2 ),
       last( ListItem1, region( _Gene1, _N1, S1, _E1, _R1, _Base1, _Pu1 ) ),
       last( ListItem2, region( _Gene2, _N2, S2, _E2, _R2, _Base2, _Pu2 ) ),
       sort_type( Flag2, S1, S2 ).
%order by length
uni_order( l, Flag2, region( _Gene1, N1, _S1, _E1, _R1, _Base1, _Pu1 ),
       region( _Gene2, N2, _S2, _E2, _R2, _Base2, _Pu2 ) ) :-
```

```
        sort_type( Flag2, N1, N2 ).
uni_order( l, Flag2, region( _Gene1, N1, _S1, _E1, _R1, _Base1, _Pu1 ),
        ListItem ) :-
        is_list( ListItem ),
        last( ListItem, region( _Gene2, N2, _S2, _E2, _R2, _Base2, _Pu2 ) ),
        sort_type( Flag2, N1, N2 ).
uni_order( l, Flag2, ListItem,
        region( _Gene2, N2, _S2, _E2, _R2, _Base2, _Pu2 )  ) :-
        is_list( ListItem ),
        last( ListItem, region( _Gene1, N1, _S1, _E1, _R1, _Base1, _Pu1 ) ),
        sort_type( Flag2, N1, N2 ).
uni_order( l, Flag2, ListItem1, ListItem2 ) :-
        is_list( ListItem1 ), is_list( ListItem2 ),
        last( ListItem1, region( _Gene1, N1, _S1, _E1, _R1, _Base1, _Pu1 ) ),
        last( ListItem2, region( _Gene2, N2, _S2, _E2, _R2, _Base2, _Pu2 ) ),
        sort_type( Flag2, N1, N2 ).
%order by G ratio
uni_order( r, Flag2, region( _Gene1, _N1, _S1, _E1, R1, _Base1, _Pu1 ),
        region( _Gene2, _N2, _S2, _E2, R2, _Base2, _Pu2 ) ) :-
        sort_type( Flag2, R1, R2 ).
uni_order( r, Flag2, region( _Gene1, _N1, _S1, _E1, R1, _Base1, _Pu1 ),
        ListItem ) :-
        is_list( ListItem ),
        last( ListItem, region( _Gene2, _N2, _S2, _E2, R2, _Base2, _Pu2 ) ),
        sort_type( Flag2, R1, R2 ).
uni_order( r, Flag2, ListItem,
        region( _Gene2, _N2, _S2, _E2, R2, _Base2, _Pu2 ) ) :-
        is_list( ListItem ),
        last( ListItem, region( _Gene1, _N1, _S1, _E1, R1, _Base1, _Pu1 ) ),
        sort_type( Flag2, R1, R2 ).
uni_order( r, Flag2, ListItem1, ListItem2 ) :-
        is_list( ListItem1 ), is_list( ListItem2 ),
        last( ListItem1, region( _Gene1, _N1, _S1, _E1, R1, _Base1, _Pu1 ) ),
        last( ListItem2, region( _Gene2, _N2, _S2, _E2, R2, _Base2, _Pu2 ) ),
        sort_type( Flag2, R1, R2 ).

/* sort_type( Flag2, X1, X2 )
 * Flag2: 1 (ascending) or 2 (descending)
 * X1 =< X2 for ascending
 * X1 >= X2 for descending
 */
sort_type( 1, X1, X2 ) :-
     X1 =< X2, !.
sort_type( 2, X1, X2 ) :-
     X1 >= X2, !.

/****************************************************************************
                                  Interface
****************************************************************************/
menu :-
     write( 'Welcome to use Threesome: Triplex DNA Analysis Program' ), nl,
     write( 'Copyright 1999-2010, Zhuan Mike Chen' ), nl,
     nl,
     repeat, % when input is not invalid
     write( 'Identify Triplex DNA Forming Regions (TFRs) with:' ), nl,
     write( '(1) continuous purine/pyrimidine (CP9)' ), nl,
     write( '(2) single discontinuity (SD4)' ), nl,
     write( '(3) both (1) and (2)' ), nl,
     write( '(4) CPm (9 < m < 50)' ), nl,
     write( '(5) SDn (4 < n < 50)' ), nl,
     write( '(6) both (4) and (5)' ), nl,
     nl,
     read( Num ),
     verify_1( Num ), % verify if the input is between 1-6
     !, % don't backtrack up.
     feedback_1( Num, Mcp, Nsd ),
     evaluate_all_in( Num, Mcp, Nsd ).

/* evaluate TFRs
 * Num1: 1-6, user input from the main menu
 * Mcp: 9-50 (default 9), user input from feedback_1/3
```

```
 * Nsd: 4-50 (default 4), user input from feedback_1/3
 * Ask the user how to order the TFRs
 */
evaluate_all_in( Num1, Mcp, Nsd ) :-
     repeat,
     write( 'A - Evaluate TFRs:' ), nl,
     write( '(0) bypass' ), nl,
     write( '(1) sorting by position in a gene' ), nl,
     write( '(2) sorting by length' ), nl,
     write( '(3) sorting by guanine (G) contents' ), nl,
     read( Num2 ),
     verify_2( Num2 ), % verify if the input is between 0-3
     !, % don't backtrack up
     feedback_2( Num2 ),
     evaluate_msd_in( Num1, Mcp, Nsd, Num2 ).

/* multiple SD
 * Num1, Mcp, and Nsd are passed in from evaluate_all_in/3
 * Num2: 0-3, user input for the order to sort MSD TFRs
 */
evaluate_msd_in( Num1, Mcp, Nsd, Num2 ) :-
     ( Num1 == 1; Num1 == 4 ) ->
       ( write( 'Bypass: User does not want search SD TFRs' ), nl, nl,
         Num3 = 0,
         evaluate_uni_in( Num1, Mcp, Nsd, Num2, Num3 )
       );
       ( repeat,
         write( 'B - Evaluate multiple single discontinuity (MSD) TFRs:' ),
         nl,
         write( '(0) bypass' ), nl,
         write( '(1) sorting by position in a gene' ), nl,
         write( '(2) sorting by length' ), nl,
         write( '(3) sorting by guanine (G) contents' ), nl,
         read( Num3 ),
         verify_2( Num3 ),
         !, % don't backtrack up
         feedback_2( Num3 ),
         evaluate_uni_in( Num1, Mcp, Nsd, Num2, Num3 )
       ).

/* uniqueness
 * Num1, Mcp, Nsd and Num2 are passed in from evaluate_msd_in/4
 * Num3: 0-3, user input for uniqueness analysis
 *       and the order to sort the results
 */
evaluate_uni_in( Num1, Mcp, Nsd, Num2, Num3 ) :-
     repeat,
     write( 'C - Uniqueness analysis of TFRs:' ), nl,
     write( '(0) bypass' ), nl,
     write( '(1) sorting by position in a gene' ), nl,
     write( '(2) sorting by length' ), nl,
     write( '(3) sorting by guanine (G) contents' ), nl,
     read( Num4 ),
     verify_2( Num4 ),
     !, % don't backtrack up
     feedback_2( Num4 ),
     design_TFO_in( Num1, Mcp, Nsd, Num2, Num3, Num4 ).

/* design TFO interface
 * Num1, Mcp, Nsd, Num2 and Num3 are passed from evaluate_uni_in/5
 * Num4: 0-3, user input: how to design TFOs
 */
design_TFO_in( Num1, Mcp, Nsd, Num2, Num3, Num4 ) :-
     repeat,
     write( 'Design TFOs:' ), nl,
     write( '(0) bypass' ), nl,
     write( '(1) purine motif' ), nl,
     write( '(2) pyrimidine motif' ), nl,
     write( '(3) both (1) and (2)' ), nl,
     read( Num5 ),
     verify_2( Num5 ),
```

```
     !, % don't backtrack up
     feedback_3( Num5 ),
     a_de_scend( Num1, Mcp, Nsd, Num2, Num3, Num4, Num5 ).

a_de_scend( Num1, Mcp, Nsd, Num2, Num3, Num4, Num5 ) :-
     repeat,
     write( 'Sort all output by:' ), nl,
     write( '(1) ascending' ), nl,
     write( '(2) descending' ), nl,
     read( Num6 ),
     verify_3( Num6 ),
     !, % don't backtrack up
     feedback_4( Num6 ),
     main_prog( Num1, Mcp, Nsd, Num2, Num3, Num4, Num5, Num6 ).

/* verify whether the input is a right integer */
verify_1( X ) :-
     integer( X ),
     X > 0,
     X < 7.
verify_1( _X ) :-
     write( 'Error: invalid input!' ), nl,
     write( 'Please enter a number between 1 to 6' ),nl,
     nl,
     fail.

/* verify CP bounders when user wants a longer length than
 * the default length (9)
 */
verify_11( Mcp ) :-
     integer( Mcp ),
     Mcp > 9,
     Mcp < 51.
verify_11( _Mcp ) :-
     write( 'Error: invalid input!' ), nl,
     write( 'Please enter a number between 10 to 50' ),nl,
     nl,
     fail.

/* verify SD bounders when user wants a longer length than
 * the deault length (4)
 */
verify_12( Nsd ) :-
     integer( Nsd ),
     Nsd > 4,
     Nsd < 51.
verify_12( _Nsd ) :-
     write( 'Error: invalid input!' ), nl,
     write( 'Please enter a number between 5 to 50' ),nl,
     nl,
     fail.

/* verify evaluation input: 0-3 */
verify_2( X ) :-
     integer( X ),
     X >= 0,
     X < 4.
verify_2( _X ) :-
     write( 'Error: invalid input!' ), nl,
     write( 'Please enter a number between 0 to 3' ),nl,
     nl,
     fail.

/* verify sort type input: 1 or 2 */
verify_3( X ) :-
     integer( X ),
     X >= 1,
     X =< 2.
verify_3( _X ) :-
     write( 'Error: invalid input!' ), nl,
     write( 'Please enter a number between 1 or 2' ),nl,
```

```
     nl,
     fail.

/* when a right input is entered
 * give a appropriate feedback
 * and assign Mcp - length of CP TFRs
 *            Nsd - length of SD TFRs
 *     corresponding values
 */
feedback_1( 1, Mcp, Nsd ) :-
     Mcp = 9,
     Nsd = 4,
     write( 'You want to identify TFRs with:' ), nl,
     write( '(1) continuous purine/pyrimidine (CP9)' ), nl,
     nl.
feedback_1( 2, Mcp, Nsd ) :-
     Mcp = 9,
     Nsd = 4,
     write( 'You want to identify TFRs with:' ), nl,
     write( '(2) single discontinuity (SD4)' ), nl,
     nl.
feedback_1( 3, Mcp, Nsd ) :-
     Mcp = 9,
     Nsd = 4,
     write( 'You want to identify TFRs with:' ), nl,
     write( '(3) both (1) and (2)' ), nl,
     nl.
/* when the user wand a longer CP length than the default
 * length (9):
 */
feedback_1( 4, Mcp, Nsd ) :-
     Nsd = 4,
     repeat,
     write( 'You want to identify TFRs with:' ), nl,
     write( '(4) CPm' ), nl,
     nl,
     repeat,
     write( 'Please input the lower bounder (m) of CP TFR search (10 - 50):' ), nl,
     read( Mcp ),
     verify_11( Mcp ), % verify if the input is between 10 - 50
     !, % don't backtrack up
     write( 'You will search CP TFR with lower bounder as: ' ),
     write( Mcp ), nl.
/* when the user wand a longer SD length than the default
 * length (4):
 */
feedback_1( 5, Mcp, Nsd ) :-
     Mcp = 9,
     write( 'You want to identify TFRs with:' ), nl,
     write( '(5) SDn' ), nl,
     nl,
     repeat,
     write( 'Please input the lower bounder (n) of SD TFR search (5 - 50):' ), nl,
     read( Nsd ),
     verify_12( Nsd ), % verify if the input is between 5 - 50
     !, % don't backtrack up
     write( 'You will search SD TFR with lower bounder as: ' ),
     write( Nsd ), nl.
/* when the user wand a longer CP and SD lengths than the default
 * length (9 or 4):
 */
feedback_1( 6, Mcp, Nsd ) :-
     write( 'You want to identify TFRs with:' ), nl,
     write( '(6) both (4) and (5)' ), nl,
     nl,
     repeat,
     write( 'Please input the lower bounder (m) of CP TFR search (10 - 50):' ), nl,
     read( Mcp ),
     verify_11( Mcp ), !,
     write( 'You will search CP TFR with lower bounder as: ' ),
     write( Mcp ), nl, nl,
```

```
     repeat,
     write( 'Please input the lower bounder (n) of SD TFR search (5 - 50):' ), nl,
     read( Nsd ),
     verify_12( Nsd ), !,
     write( 'You will search SD TFR with lower bounder as: ' ),
     write( Nsd ), nl.

% feedback for right evaluation input
feedback_2( 0 ) :-
     write( 'You want to bypass this part' ), nl,
     nl.
feedback_2( 1 ) :-
     write( 'You want to evaluate all TFRs and sort by position' ),
     nl, nl.
feedback_2( 2 ) :-
     write( 'You want to evaluate all TFRs and sort by length' ),
     nl, nl.
feedback_2( 3 ) :-
     write( 'You want to evaluate all TFRs and sort by G content' ),
     nl, nl.

/* feedback_3( X )
   feedback for the valid input
 */
feedback_3( 0 ) :-
     write( 'You want to bypass this part' ), nl,
     nl.
feedback_3( 1 ) :-
     write( 'You want to design purine motif TFOs' ),
     nl, nl.
feedback_3( 2 ) :-
     write( 'You want to design pyrimidine motif TFOs' ),
     nl, nl.
feedback_3( 3 ) :-
     write( 'You want to design both purine and pyrimidine motif TFOs' ),
     nl, nl.

/* feedback_4( X )
   feedback for a valid input of sort type
 */
feedback_4( 1 ) :-
     write( 'You want to sort output by ascending' ),
     nl, nl.
feedback_4( 2 ) :-
     write( 'You want to sort output by descending' ),
     nl, nl.

/******************************** Interface *******************************/
/*************************************************************************
 * main_prog( Num1, Mcp, Nsd, Num2, Num3, Num4, Num5, Num6 )
 * all input variables are passed from design_TFO_in/7
 *  Num1: 1-6 (search for CP or SD, or both)
 *  Mcp: 9-50 (CP length)
 *  Nsd: 4-50 (SD length)
 *  Num2-Num5: 0-3
 *  Num6: 1 (ascending) or 2 (descending)
 */
main_prog( Num1, Mcp, Nsd, Num2, Num3, Num4, Num5, Num6 ) :-
     % process output name according to input
     % e.g., if the user choose
     %  Num1=1, identify CP TFR
     %  Mcp=9,  analyze 9 or more continuous CP
     %  Nsd=4,  analyze 4 or more SD
     %  Num2=1, sorting by location
     %  Num3=0, bypass MSD evaluation
     %  Num4=3, process uniqueness analysis, sort by G contents
     %  Num5=0, do not design TFO
     %  Num6=2, sort by descending
     % this group of choice will have a text with name
     %  1941030.txt
     name( Num1, Num1L ), name( Mcp, McpL ),
```

```
     name( Nsd, NsdL ), name( Num2, Num2L ),
     name( Num3, Num3L ), name( Num4, Num4L ),
     name( Num5, Num5L ), name( Num6, Num6L ),
     append( Num1L, McpL, Name1 ),
     append( Name1, NsdL, Name2 ),
     append( Name2, Num2L, Name3 ),
     append( Name3, Num3L, Name4 ),
     append( Name4, Num4L, Name5 ),
     append( Name5, Num5L, Name6 ),
     append( Name6, Num6L, Name7 ),
     append( Name7, ".txt", OutFileL ),
     name( OutFile, OutFileL ),
     tell( OutFile ),
     evaluate_all_pro( Num1, Mcp, Nsd, Num2, CPTFRs, SDTFRs, Num5, Num6 ),
     evaluate_msd_pro( Nsd, Num3, Num5, Num6 ),
     evaluate_uni_pro( Num4, CPTFRs, SDTFRs, Num5, Num6 ),
     told.

/*********************************************************************
 * identify all CP or SD TFRs, or both
 * all variables are passed from main_prog/7
 *   Num1 = 1 or 4, identify CP TFRs only,
 *   Num1 = 2 or 5, identify SD TFRs only,
 *   Num1 = 3 or 6, identify both CP and SD TFRs
 *   Num2 = 0, don't report/save
 *   Num2 = 1, sort by position, and report/save the results
 *   Num2 = 2, sort by length, and report/save the results
 *   Num2 = 3, sort by G content, and report/save the results
 */
evaluate_all_pro( Num1, Mcp, _Nsd, Num2, CPTFRs, SDTFRs, Num5, Num6 ) :-
     ( Num1 == 1; Num1 == 4 ),
     InputFile1 = 'CaM1',
     open_file( InputFile1, InputL1 ),
     cp_pro( InputFile1, InputL1, Mcp, Num2, CPTFRs1, Num5, Num6 ),
     InputFile2 = 'CaM2',
     open_file( InputFile2, InputL2 ),
     cp_pro( InputFile2, InputL2, Mcp, Num2, CPTFRs2, Num5, Num6 ),
     InputFile3 = 'CaM3',
     open_file( InputFile3, InputL3 ),
     cp_pro( InputFile3, InputL3, Mcp, Num2, CPTFRs3, Num5, Num6 ),
     append( CPTFRs1, CPTFRs2, CPTFRtemp1 ),
     append( CPTFRtemp1, CPTFRs3, CPTFRs ),
     nl, nl,
     write( 'User does not want to search SD TFRs' ), nl,
     SDTFRs = [].
evaluate_all_pro( Num1, _Mcp, Nsd, Num2, CPTFRs, SDTFRs, Num5, Num6 ) :-
     ( Num1 == 2; Num1 == 5 ),
     write( 'User does not want to search CP TFRs' ), nl,
     CPTFRs = [],
     InputFile1 = 'CaM1',
     open_file( InputFile1, InputL1 ),
     sd_pro( InputFile1, InputL1, Nsd, Num2, SDTFRs1, Num5, Num6 ),
     InputFile2 = 'CaM2',
     open_file( InputFile2, InputL2 ),
     sd_pro( InputFile2, InputL2, Nsd, Num2, SDTFRs2, Num5, Num6 ),
     InputFile3 = 'CaM3',
     open_file( InputFile3, InputL3 ),
     sd_pro( InputFile3, InputL3, Nsd, Num2, SDTFRs3, Num5, Num6 ),
     append( SDTFRs1, SDTFRs2, SDTFRtemp1 ),
     append( SDTFRtemp1, SDTFRs3, SDTFRs ),
     nl, nl.
evaluate_all_pro( Num1, Mcp, Nsd, Num2, CPTFRs, SDTFRs, Num5, Num6 ) :-
     ( Num1 == 3; Num1 == 6 ),
     InputFile1 = 'CaM1',
     open_file( InputFile1, InputL1 ),
     cp_pro( InputFile1, InputL1, Mcp, Num2, CPTFRs1, Num5, Num6 ),
     sd_pro( InputFile1, InputL1, Nsd, Num2, SDTFRs1, Num5, Num6 ),
     InputFile2 = 'CaM2',
     open_file( InputFile2, InputL2 ),
     cp_pro( InputFile2, InputL2, Mcp, Num2, CPTFRs2, Num5, Num6 ),
     sd_pro( InputFile2, InputL2, Nsd, Num2, SDTFRs2, Num5, Num6 ),
```

```
     InputFile3 = 'CaM3',
     open_file( InputFile3, InputL3 ),
     cp_pro( InputFile3, InputL3, Mcp, Num2, CPTFRs3, Num5, Num6 ),
     sd_pro( InputFile3, InputL3, Nsd, Num2, SDTFRs3, Num5, Num6 ),
     append( CPTFRs1, CPTFRs2, CPTFRtemp1 ),
     append( CPTFRtemp1, CPTFRs3, CPTFRs ),
     append( SDTFRs1, SDTFRs2, SDTFRtemp1 ),
     append( SDTFRtemp1, SDTFRs3, SDTFRs ).

/* open gene sequence text file
 * return grouped pu/py list in InputL
 * e.g., open_file( 'CaM1', InputL )
 */
open_file( InputFile, InputL ) :-
     open( InputFile, read, Stream ),
     read_input( Stream, CharCodesL ),
     char_to_letter( InputSL, CharCodesL, [] ),
     group_puspys( InputSL, InputL ).

/********************* evaluate_all_pro ********************************/

/****************************************************************
 * cp_pro( InputFile, InputL, Mcp, Num2, CPTFRs, Num5, Num6 )
 * all input variables are passed from evaluate_all_pro/8
 *   Mcp: the lower bounder of CP TFR search
 *   Num2 = 0, don't report/save
 *   Num2 = 1, sort by position, and report/save the results
 *   Num2 = 2, sort by length, and report/save the results
 *   Num2 = 3, sort by G content, and report/save the results
 *   Num5: 0-3, TFO designing options
 *   Num6: 1 (ascending) or 2 (descending)
 * output: CPTFRs, identified CP TFRs
 */
cp_pro( InputFile, InputL, Mcp, 0, CPTFRs, _Num5, _Num6 ) :-
     % search CP
     search_cp_pro( InputFile, InputL, Mcp, CPTFRs ),
     nl, nl,
     write( InputFile ),
     write( ' all CP' ), write( Mcp ),
     write( ' TFRs are not required by user' ),
     nl, nl.

% sorted by position
cp_pro( InputFile, InputL, Mcp, 1, CPTFRs, Num5, Num6 ) :-
     % search CP
     search_cp_pro( InputFile, InputL, Mcp, CPTFRs ),
     % sort ascending or descending
     rmerge_sort( 'p', Num6, CPTFRs, CPTFRsSort ),
     nl, nl,
     write( InputFile ),
     write( ' all CP' ), write( Mcp ),
     write( ' TFRs sorted by position are: ' ),
     nl, nl,
     length( CPTFRsSort, Leg ),
     write( '   Total number of regions:' ),
     write( Leg ), nl, nl,
     format("~t~s~5+~t~s~8+~t~s~9+~t~s~7+~t~s~7+~t~s~7+~t~s~8+~t~20|~n",
            ["Gene", "py->pu", "Len(bp)","Start","End","Ratio","Bases"]),
     rwriteoutput( CPTFRsSort ),
     evaluate_cp_tfo( Num5, CPTFRsSort ).

% sorted by length
cp_pro( InputFile, InputL, Mcp, 2, CPTFRs, Num5, Num6 ) :-
     % search CP
     search_cp_pro( InputFile, InputL, Mcp, CPTFRs ),
     rmerge_sort( 'l', Num6, CPTFRs, SortLen ),
     nl, nl,
     write( InputFile ),
     write( ' all CP' ), write( Mcp ),
     write( ' TFRs sorted by length are: ' ),
     nl, nl,
```

```
     length( SortLen, Leg ),
     write( '   Total number of regions:' ),
     write( Leg ), nl, nl,
     format("~t~s~5+~t~s~8+~t~s~9+~t~s~7+~t~s~7+~t~s~7+~t~s~8+~t~20|~n",
            ["Gene", "py->pu", "Len(bp)","Start","End","Ratio","Bases"]),
     rwriteoutput( SortLen ),
     evaluate_cp_tfo( Num5, SortLen ).

% sorted by G content
cp_pro( InputFile, InputL, Mcp, 3, CPTFRs, Num5, Num6 ) :-
     % search CP
     search_cp_pro( InputFile, InputL, Mcp, CPTFRs ),
     rmerge_sort( 'r', Num6, CPTFRs, SortRatio ),
     nl, nl,
     write( InputFile ),
     write( ' all CP' ), write( Mcp ),
     write( ' TFRs sorted by G content are: ' ),
     nl, nl,
     length( SortRatio, Leg ),
     write( '   Total number of regions:' ),
     write( Leg ), nl, nl,
     format("~t~s~5+~t~s~8+~t~s~9+~t~s~7+~t~s~7+~t~s~7+~t~s~8+~t~20|~n",
            ["Gene", "py->pu", "Len(bp)","Start","End","Ratio","Bases"]),
     rwriteoutput( SortRatio ),
     evaluate_cp_tfo( Num5, SortRatio ).

/*******************************************************************/

/*******************************************************************
 * sd_pro( InputFile, InputL, Nsd, Num2, SDTFRs, Num5, Num6 )
 * all input variables are passed from evaluate_all_pro/8
 *   Nsd: the lower bounder of SD TFR search
 *   Num2 = 0, don't write out
 *   Num2 = 1, sort by position
 *   Num2 = 2, sort by length
 *   Num2 = 3, sort by G content
 *   Num6: 1 (ascending) or 2 (descending)
 * output: SDTFRs identified SD TFRs
 */
sd_pro( InputFile, InputL, Nsd, 0, SDTFRs, _Num5, _Num6 ) :-
     % search SD
     search_sd_pro( InputFile, InputL, Nsd, SDTFRs ),
     nl, nl,
     write( InputFile ),
     write( ' all SD' ), write( Nsd ),
     write( ' TFRs are not required by user' ),
     nl, nl.

% sorted by position
sd_pro( InputFile, InputL, Nsd, 1, SDTFRs, Num5, Num6 ) :-
     % search SD
     search_sd_pro( InputFile, InputL, Nsd, SDTFRs ),
     rmerge_sort( 'p', Num6, SDTFRs, SDTFRsSort ),
     nl, nl,
     write( InputFile ),
     write( ' all SD' ), write( Nsd ),
     write( ' TFRs sorted by position are:' ),
     nl, nl,
     length( SDTFRsSort, Leg2 ),
     write( '   Total number of regions: ' ),
     write( Leg2 ), nl, nl,
     format("~t~s~5+~t~s~8+~t~s~9+~t~s~7+~t~s~7+~t~s~7+~t~s~8+~t~20|~n",
            ["Gene", "py->pu", "Len(bp)","Start","End","Ratio","Bases"]),
     rwriteoutput( SDTFRsSort ),
     evaluate_sd_tfo( Num5, SDTFRsSort ).

% sorted by length
sd_pro( InputFile, InputL, Nsd, 2, SDTFRs, Num5, Num6 ) :-
     % search SD
     search_sd_pro( InputFile, InputL, Nsd, SDTFRs ),
     rmerge_sort( 'l', Num6, SDTFRs, SortLen ),
```

```
     nl, nl,
     write( InputFile ),
     write( ' all SD' ), write( Nsd ),
     write( ' TFRs sorted by length are:' ),
     nl, nl,
     length( SDTFRs, Leg2 ),
     write( '   Total number of regions: ' ),
     write( Leg2 ), nl, nl,
     format("~t~s~5+~t~s~8+~t~s~9+~t~s~7+~t~s~7+~t~s~7+~t~s~8+~t~20|~n",
            ["Gene", "py->pu", "Len(bp)","Start","End","Ratio","Bases"]),
     rwriteoutput( SortLen ),
     evaluate_sd_tfo( Num5, SortLen ).

% sorted by G content
sd_pro( InputFile, InputL, Nsd, 3, SDTFRs, Num5, Num6 ) :-
     % search SD
     search_sd_pro( InputFile, InputL, Nsd, SDTFRs ),
     rmerge_sort( 'r', Num6, SDTFRs, SortRatio ),
     nl, nl,
     write( InputFile ),
     write( ' all SD' ), write( Nsd ),
     write( ' TFRs sorted by G content are:' ),
     nl, nl,
     length( SDTFRs, Leg2 ),
     write( '   Total number of regions: ' ),
     write( Leg2 ), nl, nl,
     format("~t~s~5+~t~s~8+~t~s~9+~t~s~7+~t~s~7+~t~s~7+~t~s~8+~t~20|~n",
            ["Gene", "py->pu", "Len(bp)","Start","End","Ratio","Bases"]),
     rwriteoutput( SortRatio ),
     evaluate_sd_tfo( Num5, SortRatio ).

/************************** sd_pro **********************************************/

/****************************************************
 * group program for TFR search, complement (py-pu mark)
 * and count G ratio
 * all input variables are passed from cp_pro/6
 * InputFile: sequence file name
 * InputL: grouped TFRs
 * Mcp: the length of CP
 * CPTFRs: results for output
 */
search_cp_pro( InputFile, InputL, Mcp, CPTFRs ) :-
     search_cp( InputFile, InputL, Mcp, CPori ),
     % mark py-pu stamp
     pypu_complement( CPori, CPcom ),
     % count G ratio
     ratio( CPcom, CPTFRs ).
% all input variables are passed from sd_pro/6
search_sd_pro( InputFile, InputL, Nsd, SDTFRs ) :-
     search_sd( InputFile, InputL, Nsd, SDori ),
     % mark py-pu stamp
     pypu_complement( SDori, SDcom ),
     % count G ratio
     ratio( SDcom, SDTFRs ).
/****************************************************/

/****************************************************************************
 * evaluate_msd_pro( Nsd, Num3, Num5, Num6 )
 * all input variables are passed from main_prog/8
 *   Nsd: 4-50
 *   Num3: 0-3
 */
evaluate_msd_pro( _Nsd, 0, _Num5, _Num6 ) :-
     nl, write( 'User bypass multiple SD TFR search' ), nl,!.
evaluate_msd_pro( Nsd, 1, Num5, Num6 ) :-
     InputFile1 = 'CaM1',
     open_file( InputFile1, InputL1 ),
     msd_p( InputFile1, InputL1, Nsd, Num5, Num6 ),
     InputFile2 = 'CaM2',
     open_file( InputFile2, InputL2 ),
```

```
     msd_p( InputFile2, InputL2, Nsd, Num5, Num6 ),
     InputFile3 = 'CaM3',
     open_file( InputFile3, InputL3 ),
     msd_p( InputFile3, InputL3, Nsd, Num5, Num6 ),!.
evaluate_msd_pro( Nsd, 2, Num5, Num6 ) :-
     InputFile1 = 'CaM1',
     open_file( InputFile1, InputL1 ),
     msd_l( InputFile1, InputL1, Nsd, Num5, Num6 ),
     InputFile2 = 'CaM2',
     open_file( InputFile2, InputL2 ),
     msd_l( InputFile2, InputL2, Nsd, Num5, Num6 ),
     InputFile3 = 'CaM3',
     open_file( InputFile3, InputL3 ),
     msd_l( InputFile3, InputL3, Nsd, Num5, Num6 ),!.
evaluate_msd_pro( Nsd, 3, Num5, Num6 ) :-
     InputFile1 = 'CaM1',
     open_file( InputFile1, InputL1 ),
     msd_r( InputFile1, InputL1, Nsd, Num5, Num6 ),
     InputFile2 = 'CaM2',
     open_file( InputFile2, InputL2 ),
     msd_r( InputFile2, InputL2, Nsd, Num5, Num6 ),
     InputFile3 = 'CaM3',
     open_file( InputFile3, InputL3 ),
     msd_r( InputFile3, InputL3, Nsd, Num5, Num6 ),!.

/**************************************************
 * msd_?( InputFile, InputL, Nsd, Num5, Num6 ),
 * ? = p (position), l (length), r (G ratio)
 * all input variables are passed from evaluate_msd_pro/4
 *    Nsd: the length of SD TFRs
 *    Num5: 0-3, TFO designing options
 */
msd_p( InputFile, InputL, Nsd, Num5, Num6 ) :-
     search_sd_pro( InputFile, InputL, Nsd, SDTFRs ),
     look_conSD( SDTFRs, OutconSD ),
     remove_single( OutconSD, OutMSD ),
     nl, nl,
     write( InputFile ),
     write( ' SD' ), write( Nsd ),
     write( ' multiple continuous SD sorted by position:' ),
     nl, nl,
     msd_merge_sort( 'p', Num6, OutMSD, SortPos ),
     format("~t~s~5+~t~s~8+~t~s~9+~t~s~7+~t~s~7+~t~s~7+~t~s~8+~t~20|~n",
            ["Gene", "py->pu", "Len(bp)","Start","End","Ratio","Bases"]),
     writesublist( SortPos ), nl, nl,
     msd_tfo_pro( Num5, SortPos ).
msd_l( InputFile, InputL, Nsd, Num5, Num6 ) :-
     search_sd_pro( InputFile, InputL, Nsd, SDTFRs ),
     look_conSD( SDTFRs, OutconSD ),
     remove_single( OutconSD, OutMSD ),
     nl, nl,
     write( InputFile ),
     write( ' SD' ), write( Nsd ),
     write( ' multiple continuous SD sorted by 1st TFR length:' ),
     nl, nl,
     msd_merge_sort( 'l', Num6, OutMSD, SortLen ),
     format("~t~s~5+~t~s~8+~t~s~9+~t~s~7+~t~s~7+~t~s~7+~t~s~8+~t~20|~n",
            ["Gene", "py->pu", "Len(bp)","Start","End","Ratio","Bases"]),
     writesublist( SortLen ), nl, nl,
     msd_tfo_pro( Num5, SortLen ).
msd_r( InputFile, InputL, Nsd, Num5, Num6 ) :-
     search_sd_pro( InputFile, InputL, Nsd, SDTFRs ),
     look_conSD( SDTFRs, OutconSD ),
     remove_single( OutconSD, OutMSD ),
     write( InputFile ),
     write( ' SD' ), write( Nsd ),
     write( ' multiple continuous SD sorted by 1st TFR G content:' ),
     nl, nl,
     msd_merge_sort( 'r', Num6, OutMSD, SortRatio ),
     format("~t~s~5+~t~s~8+~t~s~9+~t~s~7+~t~s~7+~t~s~7+~t~s~8+~t~20|~n",
            ["Gene", "py->pu", "Len(bp)","Start","End","Ratio","Bases"]),
```

```
     writesublist( SortRatio ), nl, nl,
     msd_tfo_pro( Num5, SortRatio ).

/*************************************************************
 * evaluate_uni_pro( Num4, CPTFRs, SDTFRs, Num5, Num6 )
 * called by main_prog/8
 * Num4: 0-3, uniqueness analysis options
 * CPTFRs: all CP TFRs from three genes
 * SDTFRs: all SD TFRs from three genes
 * Num5: 0-3, TFO designing options
 * Num6: 1 (ascending) or 2 (descending)
 */
evaluate_uni_pro( 0, _CPTFRs, _SDTFRs, _Num5, _Num6 ) :-
     write( 'User bypasses uniqueness search' ), nl, !.
% sorted by position
evaluate_uni_pro( 1, CPTFRs, SDTFRs, Num5, Num6 ) :-
     % CP part
     (
     ( CPTFRs == [] ) ->
       ( write( 'Unique CP TFRs: not required by user or none' ), nl)
       ;
       ( % sort all TFRs ascending by length
         rmerge_sort( 'l', '1', CPTFRs, CPSortList ),
         % output all sorted CP in three genes
         write( 'Uniqueness analysis: all CP in three genes sorted by position' ), nl,
         nl,
         uni_output( 'p', Num6, CPTFRs, _Out1cp), nl,
         % output all sublists
         check_sublist( CPSortList, CPSubList ),
         write( 'Uniqueness analysis: all CP sublists in three genes sorted by position'
), nl,
         nl,
         uni_output_sub( 'p', Num6, CPSubList, _Out2cp ),
         % output real unique TFRs
         unique( CPSubList, CPUniSubList ),
         write( 'Uniqueness analysis: unique CP sorted by position' ), nl,
         nl,
         uni_output( 'p', Num6, CPUniSubList, OutCPUniSubList ),
         % CP TFO
         evaluate_cp_tfo( Num5, OutCPUniSubList ),
         % output TFR-in-gene
         check_ingene( CPSubList, InGeneList ),
         write( 'Uniqueness analysis: CP TFR-in-gene sorted by position' ), nl,
         nl,
         uni_output_sub( 'p', Num6, InGeneList, OutInGeneList ),
         % in-gene TFO
         evaluate_cp_tfo_sub( Num5, OutInGeneList )
       )
     ),
     % SD part
     (
     ( SDTFRs == [] ) ->
       (write( 'Unique SD TFRs: not required by user or none' ),nl)
       ;
       ( rmerge_sort( 'l', '1', SDTFRs, SDSortList ),
         % output all sorted SD in three genes
         write( 'Uniqueness analysis: all SD in three genes sorted by position' ), nl,
         nl,
         uni_output( 'p', Num6, SDTFRs, _Out1sd ),
         % output all sublists
         check_sublist( SDSortList, SDSubList ),
         write( 'Uniqueness analysis: all SD sublists in three genes sorted by position'
), nl,
         nl,
         uni_output_sub( 'p', Num6, SDSubList, _Out2sd ),
         % output real unique TFRs
         unique( SDSubList, SDUniSubList ),
         write( 'Uniqueness analysis: unique SD sorted by position' ), nl,
         nl,
         uni_output( 'p', Num6, SDUniSubList, OutSDUniSubList ),
         % SD TFO
```

```
        evaluate_sd_tfo( Num5, OutSDUniSubList ),
        % output TFR-in-gene
        check_ingene( SDSubList, SDInGeneList ),
        write( 'Uniqueness analysis: SD TFR-in-gene sorted by position' ), nl,
        nl,
        uni_output_sub( 'p', Num6, SDInGeneList, OutSDInGeneList ),
        % in-gene TFO
        msd_tfo_pro( Num5, OutSDInGeneList )
      )
      ), !.

% sorted by length
evaluate_uni_pro( 2, CPTFRs, SDTFRs, Num5, Num6 ) :-
     % CP part
     (( CPTFRs == [] ) ->
       ( write( 'Unique CP TFRs: not required by user or none' ), nl)
       ;
       ( rmerge_sort( 'l', 'l', CPTFRs, CPSortList ),
         % output all sorted CP in three genes
         write( 'Uniqueness analysis: all CP in three genes sorted by length' ), nl,
         nl,
         uni_output( 'l', Num6, CPTFRs, _Out1cp ),
         % output all sublists
         check_sublist( CPSortList, CPSubList ),
         write( 'Uniqueness analysis: all CP sublists in three genes sorted by length'
), nl,
         nl,
         uni_output_sub( 'l', Num6, CPSubList, _Out2cp ),
         % output real unique TFRs
         unique( CPSubList, CPUniSubList ),
         write( 'Uniqueness analysis: unique CP sorted by length' ), nl,
         nl,
         uni_output( 'l', Num6, CPUniSubList, OutCPUniSubList ),
         % CP TFO
         evaluate_cp_tfo( Num5, OutCPUniSubList ),
         % output TFR-in-gene
         check_ingene( CPSubList, InGeneList ),
         write( 'Uniqueness analysis: CP TFR-in-gene sorted by length' ), nl,
         nl,
         uni_output_sub( 'l', Num6, InGeneList, OutInGeneList ),
         % in-gene TFO
         evaluate_cp_tfo_sub( Num5, OutInGeneList )
       )),
     % SD part
     (( SDTFRs == [] ) ->
       ( write( 'Unique SD TFRs: not required by user or none' ), nl )
       ;
       ( rmerge_sort( 'l', 'l', SDTFRs, SDSortList ),
         % output all sorted SD in three genes
         write( 'Uniqueness analysis: all SD in three genes sorted by length' ), nl,
         nl,
         uni_output( 'l', Num6, SDTFRs, _Out1sd ),
         % output all sublists
         check_sublist( SDSortList, SDSubList ),
         write( 'Uniqueness analysis: all SD sublists in three genes sorted by length'
), nl,
         nl,
         uni_output_sub( 'l', Num6, SDSubList, _Out2sd ),
         % output real unique TFRs
         unique( SDSubList, SDUniSubList ),
         write( 'Uniqueness analysis: unique SD sorted by length' ), nl,
         nl,
         uni_output( 'l', Num6, SDUniSubList, OutSDUniSubList ),
         % SD TFO
         evaluate_sd_tfo( Num5, OutSDUniSubList ),
         % output TFR-in-gene
         check_ingene( SDSubList, SDInGeneList ),
         write( 'Uniqueness analysis: SD TFR-in-gene sorted by length' ), nl,
         nl,
         uni_output_sub( 'l', Num6, SDInGeneList, OutSDInGeneList ),
         % in-gene TFO
```

```
        msd_tfo_pro( Num5, OutSDInGeneList )
      )), !.

% sorted by G content
evaluate_uni_pro( 3, CPTFRs, SDTFRs, Num5, Num6 ) :-
     % CP part
     (( CPTFRs == [] ) ->
       (write( 'Unique CP TFRs: not required by user or none' ), nl)
       ;
       ( rmerge_sort( 'l', 'l', CPTFRs, CPSortList ),
         % output all sorted CP in three genes
         write( 'Uniqueness analysis: all CP in three genes sorted by G content' ), nl,
         nl,
         uni_output( 'r', Num6, CPTFRs, _Out1cp ),
         % output all sublists
         check_sublist( CPSortList, CPSubList ),
         write( 'Uniqueness analysis: all CP sublists in three genes sorted by G
content' ), nl,
         nl,
         uni_output_sub( 'r', Num6, CPSubList, _Out2cp ),
         % output real unique TFRs
         unique( CPSubList, CPUniSubList ),
         write( 'Uniqueness analysis: unique CP sorted by G content' ), nl,
         nl,
         uni_output( 'r', Num6, CPUniSubList, OutCPUniSubList ),
         % CP TFO
         evaluate_cp_tfo( Num5, OutCPUniSubList ),
         % output TFR-in-gene
         check_ingene( CPSubList, InGeneList ),
         write( 'Uniqueness analysis: CP TFR-in-gene sorted by G content' ), nl,
         nl,
         uni_output_sub( 'r', Num6, InGeneList, OutInGeneList ),
         % in-gene TFO
         evaluate_cp_tfo_sub( Num5, OutInGeneList )
       )),
     % SD part
     (( SDTFRs == [] ) ->
       ( nl, write( 'Unique SD TFRs: not required by user or none' ), nl)
       ;
       ( rmerge_sort( 'l', 'l', SDTFRs, SDSortList ),
         % output all sorted SD in three genes
         write( 'Uniqueness analysis: all SD in three genes sorted by G content' ), nl,
         nl,
         uni_output( 'r', Num6, SDTFRs, _Out1sd ),
         % output all sublists
         check_sublist( SDSortList, SDSubList ),
         write( 'Uniqueness analysis: all SD sublists in three genes sorted by G
content' ), nl,
         nl,
         uni_output_sub( 'r', Num6, SDSubList, _Out2sd ),
         % output real unique TFRs
         unique( SDSubList, SDUniSubList ),
         write( 'Uniqueness analysis: unique SD sorted by G content' ), nl,
         nl,
         uni_output( 'r', Num6, SDUniSubList, OutSDUniSubList ),
         % SD TFO
         evaluate_sd_tfo( Num5, OutSDUniSubList ),
         % output TFR-in-gene
         check_ingene( SDSubList, SDInGeneList ),
         write( 'Uniqueness analysis: SD TFR-in-gene sorted by G content' ), nl,
         nl,
         uni_output_sub( 'r', Num6, SDInGeneList, OutSDInGeneList ),
         % in-gene TFO
         msd_tfo_pro( Num5, OutSDInGeneList )
      )), !.


/* output sorted sublists
 * called by evaluate_uni_pro/4
 * Flag: p (positon), l (length), or r (G ratio)
 * InList: uniqueness analyzed TFR list
```

```
 * ListSort: output, sorted list
 */
uni_output( Flag, Flag2, InList, ListSort ) :-
     uni_merge_sort( Flag, Flag2, InList, ListSort ),
     length( ListSort, Len ),
     write( 'Total number of TFRS: ' ),
     write( Len ), nl,nl,
     format("~t~s~5+~t~s~8+~t~s~9+~t~s~7+~t~s~7+~t~s~7+~t~s~8+~t~20|~n",
     ["Gene", "py->pu", "Len(bp)","Start","End","Ratio","Bases"]),
     rwriteoutput( ListSort ).

/* output with sublist i.e. sublist data structure*/
uni_output_sub( Flag, Flag2, InList, ListSort ) :-
     uni_merge_sort( Flag, Flag2, InList, ListSort ),
     length( ListSort, Len ),
     write( 'Total number of group TFRS: ' ),
     write( Len ), nl,nl,
     format("~t~s~5+~t~s~8+~t~s~9+~t~s~7+~t~s~7+~t~s~7+~t~s~8+~t~20|~n",
     ["Gene", "py->pu", "Len(bp)","Start","End","Ratio","Bases"]),
     writesublist( ListSort ).

/*******************************************************
 * TFO design for CP TFRs
 * evaluate_cp_tfo( Num5, SortRatio )
 * called by evaluate_uni_pro/4
 *   Num5 = 0: bypass
 *   Num5 = 1: purine motif (R) TFO
 *   Num5 = 2: pyrimidine motif (Y) TFO
 *   Num5 = 3: both R and Y TFO
 */
evaluate_cp_tfo( 0, _InTFRs ) :-
     write( 'User doen not design any TFOs' ),nl,!.
evaluate_cp_tfo( 1, InTFRs ) :-
     cp_r_tfo( InTFRs, OutTFOs ),
     write( 'CP purine motif TFOs' ), nl,
     format("~t~s~5+~t~s~8+~t~s~9+~t~s~7+~t~s~7+~t~s~7+~t~s~8+~t~20|~n",
            ["Gene", "py->pu", "Len(bp)","Start","End","Ratio","Bases"]),
     rwriteoutput( OutTFOs ), nl.
evaluate_cp_tfo( 2, InTFRs ) :-
     cp_y_tfo( InTFRs, OutTFOs ),
     write( 'CP pyrimidine motif TFOs' ), nl,
     format("~t~s~5+~t~s~8+~t~s~9+~t~s~7+~t~s~7+~t~s~7+~t~s~8+~t~20|~n",
            ["Gene", "py->pu", "Len(bp)","Start","End","Ratio","Bases"]),
     rwriteoutput( OutTFOs ), nl.
evaluate_cp_tfo( 3, InTFRs ) :-
     cp_r_tfo( InTFRs, OutTFOsR ),
     write( 'CP purine motif TFOs' ), nl,
     format("~t~s~5+~t~s~8+~t~s~9+~t~s~7+~t~s~7+~t~s~7+~t~s~8+~t~20|~n",
            ["Gene", "py->pu", "Len(bp)","Start","End","Ratio","Bases"]),
     rwriteoutput( OutTFOsR ), nl,
     cp_y_tfo( InTFRs, OutTFOsY ),
     write( 'CP pyrimidine motif TFOs' ), nl,
     format("~t~s~5+~t~s~8+~t~s~9+~t~s~7+~t~s~7+~t~s~7+~t~s~8+~t~20|~n",
            ["Gene", "py->pu", "Len(bp)","Start","End","Ratio","Bases"]),
     rwriteoutput( OutTFOsY ),nl.

/*********************************************
 * evaluate_cp_tfo_sub( Num5, InTFRs )
 * with sublist data structure,
 *    because uniqueness analysis brings in an inner list as sublist
 * called by evaluate_uni_pro/4
 *    Num5 = 0, bypass
 *    Num5 = 1, purine motif TFOs
 *    Num5 = 2, pyrimidine motif TFOs
 *    Num5 = 3, both
 */
evaluate_cp_tfo_sub( 0, _InTFRs ).
evaluate_cp_tfo_sub( 1, InTFRs ) :-
     cp_r_tfo_sub( InTFRs, OutTFOs ),
     write( 'CP purine motif TFOs' ), nl,
     format("~t~s~5+~t~s~8+~t~s~9+~t~s~7+~t~s~7+~t~s~7+~t~s~8+~t~20|~n",
```

```
            ["Gene", "py->pu", "Len(bp)","Start","End","Ratio","Bases"]),
     writesublist( OutTFOs ), nl.
evaluate_cp_tfo_sub( 2, InTFRs ) :-
     cp_y_tfo_sub( InTFRs, OutTFOs ),
     write( 'CP pyrimidine motif TFOs' ), nl,
     format("~t~s~5+~t~s~8+~t~s~9+~t~s~7+~t~s~7+~t~s~7+~t~s~8+~t~20|~n",
            ["Gene", "py->pu", "Len(bp)","Start","End","Ratio","Bases"]),
     writesublist( OutTFOs ), nl.
evaluate_cp_tfo_sub( 3, InTFRs ) :-
     cp_r_tfo_sub( InTFRs, OutTFOsR ),
     write( 'CP purine motif TFOs' ), nl,
     format("~t~s~5+~t~s~8+~t~s~9+~t~s~7+~t~s~7+~t~s~7+~t~s~8+~t~20|~n",
            ["Gene", "py->pu", "Len(bp)","Start","End","Ratio","Bases"]),
     writesublist( OutTFOsR ), nl,
     cp_y_tfo_sub( InTFRs, OutTFOsY ),
     write( 'CP pyrimidine motif TFOs' ), nl,
     format("~t~s~5+~t~s~8+~t~s~9+~t~s~7+~t~s~7+~t~s~7+~t~s~8+~t~20|~n",
            ["Gene", "py->pu", "Len(bp)","Start","End","Ratio","Bases"]),
     writesublist( OutTFOsY ), nl.

cp_r_tfo_sub( [], [] ).
cp_r_tfo_sub( [H|R], [H1|R1] ) :-
     sd_r_tfo( H, H1 ),
     cp_r_tfo_sub( R, R1 ).

cp_y_tfo_sub( [], [] ).
cp_y_tfo_sub( [H|R], [H1|R1] ) :-
     sd_y_tfo( H, H1 ),
     cp_y_tfo_sub( R, R1 ).

/*********************************************
 * evaluate_sd_tfo( Num5, InTFRs ).
 * called by evaluate_uni_pro/4
 *   Num5 = 0, bypass
 *   Num5 = 1, purine motif TFOs
 *   Num5 = 2, pyrimidine motif TFOs
 *   Num5 = 3, boty
*/
evaluate_sd_tfo( 0, _InTFRs ) :-
     write( 'User doen not design any TFOs' ),nl,!.
evaluate_sd_tfo( 1, InTFRs ) :-
     sd_r_tfo( InTFRs, OutTFOs ),
     write( 'SD purine motif TFOs' ), nl,
     format("~t~s~5+~t~s~8+~t~s~9+~t~s~7+~t~s~7+~t~s~7+~t~s~8+~t~20|~n",
            ["Gene", "py->pu", "Len(bp)","Start","End","Ratio","Bases"]),
     rwriteoutput( OutTFOs ), nl.
evaluate_sd_tfo( 2, InTFRs ) :-
     sd_y_tfo( InTFRs, OutTFOs ),
     write( 'CP pyrimidine motif TFOs' ), nl,
     format("~t~s~5+~t~s~8+~t~s~9+~t~s~7+~t~s~7+~t~s~7+~t~s~8+~t~20|~n",
            ["Gene", "py->pu", "Len(bp)","Start","End","Ratio","Bases"]),
     rwriteoutput( OutTFOs ), nl.
evaluate_sd_tfo( 3, InTFRs ) :-
     sd_r_tfo( InTFRs, OutTFOsR ),
     write( 'CP purine motif TFOs' ), nl,
     format("~t~s~5+~t~s~8+~t~s~9+~t~s~7+~t~s~7+~t~s~7+~t~s~8+~t~20|~n",
            ["Gene", "py->pu", "Len(bp)","Start","End","Ratio","Bases"]),
     rwriteoutput( OutTFOsR ), nl,
     sd_y_tfo( InTFRs, OutTFOsY ),
     write( 'CP pyrimidine motif TFOs' ), nl,
     format("~t~s~5+~t~s~8+~t~s~9+~t~s~7+~t~s~7+~t~s~7+~t~s~8+~t~20|~n",
            ["Gene", "py->pu", "Len(bp)","Start","End","Ratio","Bases"]),
     rwriteoutput( OutTFOsY ),nl.

/* purine motif design */
cp_r_tfo( [], [] ).
% pu
cp_r_tfo( [region( Gene, N, S, E, R,
                          TFRs, PuPy )|Res],
          [region( Gene, N, S, E, R,
                      TFRs, PuPy ),
```

```
           region( Gene, N, S, E, R,
                          TFOs, PuPy)|Complement] ) :-
     r_tfo( TFRs, TFOs, [] ),
     cp_r_tfo( Res, Complement ).

/* eager t,c to a,g; a,g to t,c */
r_tfo( [H|Res] ) -->
     r_tfo_match( H ),
     r_tfo( Res ).
r_tfo( [] ) --> [],!.

/* CP TFO design */
r_tfo_match( g ) --> [g].
r_tfo_match( a ) --> [t].
%r_tfo_match( a ) --> [a]. //only a-t match is used


/* pyrimidine motif design */
cp_y_tfo( [], [] ).
% pu
cp_y_tfo( [region( Gene, N, S, E, R,
                          TFRs, PuPy )|Res],
          [region( Gene, N, S, E, R,
                   TFRs, PuPy ),
           region( Gene, N, S, E, R,
                   TFOs, PuPy)|Complement] ) :-
     y_tfo( TFRs, TFOs, [] ),
     cp_y_tfo( Res, Complement ).

/* eager t,c to a,g; a,g to t,c */
y_tfo( [H|Res] ) -->
     y_tfo_match( H ),
     y_tfo( Res ).
y_tfo( [] ) --> [],!.

/* CP TFO design */
y_tfo_match( a ) --> [t].
y_tfo_match( g ) --> [c].

/* SD purine motif design */
sd_r_tfo( [], [] ).
sd_r_tfo( [region( Gene, N, S, E, R,
                          TFRs, PuPy )|Res],
          [region( Gene, N, S, E, R,
                    TFRs, PuPy ),
           region( Gene, N, S, E, R,
                          TFOs, PuPy)|Complement] ) :-
     %reverse( TFRs, RevTFRs ),
     r_tfosd( TFRs, TFOs, [] ),
     sd_r_tfo( Res, Complement ).

/* eager t,c to a,g; a,g to t,c */
r_tfosd( [H|Res] ) -->
     r_tfosd_match( H ),
     r_tfosd( Res ).
r_tfosd( [] ) --> [],!.

/* SD TFO design */
r_tfosd_match( g ) --> [g].
r_tfosd_match( a ) --> [t].
r_tfosd_match( c ) --> [t].
/* There is no match for a T discontinuity,
 * "nnnNOT_AVAIABLEnnn" is used to indicate
 * this fact and set as a marker in the result.
 */
r_tfosd_match( t ) --> [nnnNOT_AVAIABLEnnn],!.


/* SD pyrimidine motif design */
sd_y_tfo( [], [] ).
sd_y_tfo( [region( Gene, N, S, E, R,
```

```
                          TFRs, PupY )|Res],
          [region( Gene, N, S, E, R,
                   TFRs, PuPy ),
           region( Gene, N, S, E, R,
                   TFOs, PuPy)|Complement] ) :-
     y_tfosd( TFRs, TFOs, [] ),
     sd_y_tfo( Res, Complement ).

/* eager t,c to a,g; a,g to t,c */
y_tfosd( [H|Res] ) -->
     y_tfosd_match( H ),
     y_tfosd( Res ).
y_tfosd( [] ) --> [],!.

/* SD TFO design */
y_tfosd_match( a ) --> [t].
y_tfosd_match( g ) --> [c].
y_tfosd_match( c ) --> [c].
y_tfosd_match( t ) --> [g].

/*****************************************************************
 * msd_tfo_pro( Num5, InTFRs )
 * called by evaluate_uni_pro/4
 *   Num5 = 0, bypass
 *   Num5 = 1, purine motif TFOs
 *   Num5 = 2, pyrimidine motif TFOs
 *   Num5 = 3, boty
 */
msd_tfo_pro( 0, _InTFRs ) :-
    write( 'User does not want design any TFOs' ), nl.
msd_tfo_pro( 1, InTFRs ) :-
     msd_r_tfo( InTFRs, OutTFOs ),
     write( 'SD purine motif TFOs' ), nl,
     format("~t~s~5+~t~s~8+~t~s~9+~t~s~7+~t~s~7+~t~s~7+~t~s~8+~t~20|~n",
            ["Gene", "py->pu", "Len(bp)","Start","End","Ratio","Bases"]),
     writesublist( OutTFOs ), nl.
msd_tfo_pro( 2, InTFRs ) :-
     msd_y_tfo( InTFRs, OutTFOs ),
     write( 'SD pyrimidine motif TFOs' ), nl,
     format("~t~s~5+~t~s~8+~t~s~9+~t~s~7+~t~s~7+~t~s~7+~t~s~8+~t~20|~n",
            ["Gene", "py->pu", "Len(bp)","Start","End","Ratio","Bases"]),
     writesublist( OutTFOs ), nl.
msd_tfo_pro( 3, InTFRs ) :-
     msd_r_tfo( InTFRs, OutTFOsR ),
     write( 'SD purine motif TFOs' ), nl,
     format("~t~s~5+~t~s~8+~t~s~9+~t~s~7+~t~s~7+~t~s~7+~t~s~8+~t~20|~n",
            ["Gene", "py->pu", "Len(bp)","Start","End","Ratio","Bases"]),
     writesublist( OutTFOsR ), nl,
     msd_y_tfo( InTFRs, OutTFOsY ),
     write( 'SD pyrimidine motif TFOs' ), nl,
     format("~t~s~5+~t~s~8+~t~s~9+~t~s~7+~t~s~7+~t~s~7+~t~s~8+~t~20|~n",
            ["Gene", "py->pu", "Len(bp)","Start","End","Ratio","Bases"]),
     writesublist( OutTFOsY ), nl.

msd_r_tfo( [], [] ).
msd_r_tfo( [H|R], [H1|R1] ) :-
     sd_r_tfo( H, H1 ),
     msd_r_tfo( R, R1 ).

msd_y_tfo( [], [] ).
msd_y_tfo( [H|R], [H1|R1] ) :-
     sd_y_tfo( H, H1 ),
     msd_y_tfo( R, R1 ).
```