12-1-2014

DNA Chemical Reaction Network Design Synthesis and Compilation

M. Leigh Fanning

Margaret Leigh Fanning

Candidate

Computer Science

Department

This dissertation is approved, and it is acceptable in quality and form for publication:

Approved by the Dissertation Committee:

Darko Stefanovic , Chairperson

Shuang Luan

George Lugar

Christof Teuscher

DNA Chemical Reaction Network Design Synthesis and Compilation

by

M. Leigh Fanning

B.S., Engineering Physics, University of Colorado
M.S., Computer Science, University of New Mexico

DISSERTATION

Submitted in Partial Fulfillment of the
Requirements for the Degree of

Doctor of Philosophy
Computer Science

The University of New Mexico

Albuquerque, New Mexico

December, 2014

Acknowledgments

Advisor and Dissertation Supervisor Darko Stefanovic,
Associate Professor of Computer Science, University of New Mexico

Committee Member Shuang Luan,
Associate Professor of Computer Science, University of New Mexico

Committee Member George Luger,
Professor of Computer Science, University of New Mexico

Committee Member Christof Teuscher,
Professor of Computer Science, Portland State University

This work was supported by National Science Foundation grants EMT-0829881, CCF-0829793, 1027877, and 1028238, scholarships from Intel Corporation, and the UNM Charlotte and William Kraft graduate fellowship.

DNA Chemical Reaction Network Design Synthesis and Compilation

by

M. Leigh Fanning

B.S., Engineering Physics, University of Colorado
M.S., Computer Science, University of New Mexico

Ph.D., Computer Science, University of New Mexico

Abstract

The advantages of biomolecular computing include 1) the ability to interface with, monitor, and intelligently protect and maintain the functionality of living systems, 2) the ability to create computational devices with minimal energy needs and hazardous waste production during manufacture and lifecycle, 3) the ability to store large amounts of information for extremely long time periods, and 4) the ability to create computation analogous to human brain function. To realize these advantages over electronics, biomolecular computing is at a watershed moment in its evolution. Computing with entire molecules presents different challenges and requirements than computing just with electric charge. These challenges have led to ad-hoc design and programming methods with high development costs and limited device performance. At the present time, device building entails complete low-level detail immersion. We address these shortcomings by creation of a systems engineering process for building and programming DNA-based computing devices.

Contributions of this thesis include numeric abstractions for nucleic acid sequence and secondary structure, and a set of algorithms which employ these abstractions. The abstractions and algorithms have been implemented into three artifacts: DNADL, a design description language; Pyxis, a molecular compiler and design toolset; and KCA, a simulation of DNA kinetics using a cellular automaton discretization. Our methods are applicable to other DNA nanotechnology constructions and may serve in the development of a full DNA computing model.

Contents

Chapter 1

Thesis Introduction and Central Question

Engineering is the creative application of scientific principles to design or develop structures, machines, apparatus, or manufacturing processes, or works utilizing them singly or in combination; or to construct or operate the same with full cognizance of their design; or to forecast their behavior under specific operating conditions; all as respects an intended function, economics or operation and safety to life and property. – American Engineer's Council for Professional Development

Whereas the physical sciences have benefitted from the reductionist approach in understanding phenomena, the life sciences are distinctly more complex and challenging to distill. It is therefore not surprising that computing technologies built using reductionist physical principles occurred first. Babbage's Difference and Analytical Engines were advanced in the age of the steam engine [24]. Turing and Newman's Colossus and ENIAC were advanced at the inception of the age of electronics, an age we have not yet left. We are just now arriving at the biological age, where we can manipulate the building blocks of life to compute. This age, anticipated by Turing as unorganized systems and Feynman as

infinitesimal machines, will irrevocably integrate computing into the fabric of humanity.

Molecular computing as a field has achieved demonstration of solving small instances of computationally hard problems. For a final design, materials costs are low yet up-front development and testing costs are high, and worse, the spectacular advantage held by electronic computing suggests that performance will never catch up and justify current research dollar expenditures. At the current level of technology, computing with nucleic acids DNA or RNA, involves low-level detail iteration over long timelines. Designing individual components and then getting them to work together cohesively defies engineering approaches that have sufficed for electronic computers. Additionally, although various architectures have been shown to work for certain problems, problem coverage remains sparse. Designers of necessity have chosen problems that fit their specific architectures, and the field lacks the general ability to solve an arbitrary problem on a molecular substrate. Missing are design standards, benchmarks, commonly recognized formalisms, and well-understood abstractions in the manner of the stored program computer model put forth by Eckert, Mauchly, and von Neumann [55].

These observations, however, are not made dissuasively. Truly new technology always suffers in comparisons, yet these comparisons dampen extreme responses. Neither wildly unrealistic promises on the part of proponents, nor instant dismissal on the part of opponents who may fear change and eventual loss in market dominance, serve purpose in engineering. Within the current context, comparison to electronic computing and critical observation of existing development processes help us practically identify the main difficulties, and what challenges can bear fruit if solved.

1.1 The Central Challenge for Molecular Computing

Electronic computing achieved dominance through iterative device performance gains, where each new plateau enabled solving larger problems. Scaling was achieved by process engineering, a science unto itself, that succeeded by narrowing the search of all possible component configurations and interactions into testable development cycles. These cycles engendered standards, benchmarks, models, and formalisms which in turn engendered more powerful devices. Molecular computing is at the starting gate of a similar evolution. To elevate the field beyond demonstration requires generalizing device building and programming away from niche problems, and providing real computational capability guarantees over theoretical simulations. The central challenge, at this time, is to initiate similar scaling to electronics.

At the current level of technology, DNA base components are composable into architectures, each of which support construction and programming for a variety of applications as shown in Figure 1.1.

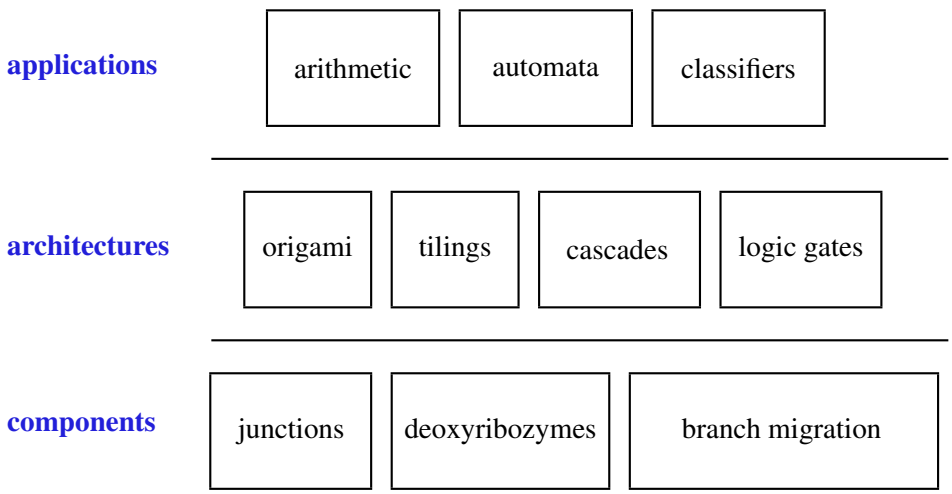


Figure 1.1: Nucleic acid components can be ultimately organized into programmed constructions that execute target applications.

Programmed constructions are physically located in reaction vessels and operate as chemical reaction networks. Networks carry their own energy, and lack central control or a system clock. Reactions are asynchronous, parallel, and possess a large number of variant pathways. The bootstrap scaling challenge is therefore rooted both in the hardware—the chemistry of nucleic acids in reaction networks, and in the software—how the chemistry can be abstracted to build programmable devices. The end goal is the capability to transform algorithms written in some appropriate programming language into low-level chemical instructions ready for laboratory execution. A measure of success is reliable and repeatable laboratory results that support generalization and standards development.

1.2 Thesis Question

Our thesis research question addresses the bootstrap scaling challenge for molecular computing: **What abstraction stack can be built that will facilitate programmable nucleic acid device construction amenable to standards development and performance guarantees?**

We take our cue from the history of electronic computing where abstractions are the unifying concept that enabled scaling and standards development. At the present time, we can easily direct a computer to execute intricate data manipulation to solve arbitrary problems. We can describe these problems in mathematically inflected human-like languages and minimally interact with the operational characteristics of the computing machine. Yet early electronic computers, built with switches, relays, vacuum tubes and magnetic drums all initially necessitated use of machine languages and hardware manipulation. Over time, the hardware became more sophisticated and powerful, but equally importantly, people built up the stack of commensurately sophisticated abstractions to alleviate need of direct hardware interaction. Further, these abstractions have become so well understood that the

hardware is now largely discussed in time and space performance terms. The electronic computing abstraction stack achieved generalization and scaling that supported creation of high-level language families, all able to describe arbitrary problems, where each problem is executable on arbitrary hardware configurations.

Expressive high-level programming languages aimed at molecular computing platforms, in contrast, are not yet available. Development undertaken by various groups instead has produced disparate low-level meta-languages [20, 90, 115] that are tightly molded around their particular and distinctive approaches to molecular computation. As these meta-languages mature and the field evolves to include more participants, a later heterogeneous group of full-fledged languages will presumably be the norm. Our taken approach acknowledges this context, and instead proceeds from the bottom up. The "bare metal" for molecular computing are the molecules themselves. Molecules are data, and reactions are operations on this data. In the physical world, reactions are also probabilistic events that shift electrons from one configuration to another [126], yet in fantastically large numbers. Molecular computing means successful orchestration of DNA and RNA physical activity because both device realization and program execution are functions of this activity.

Our approach is the development of nucleic acid abstractions that capture the relevant properties and physical effects which are principally responsible for their behaviors. The abstractions are incorporated into a molecular compiler Pyxis to accomplish systems engineering of nucleic acid chemical reaction networks. Adoption of compilation brings together both the low-level chemistry, and its programmability, into an abstraction stack. Each level of the stack organizes previous ad-hoc methods into a pipeline process. To determine the abstraction set, and its logical pipeline organization, a careful study of existing approaches was made with a focus on deoxyribozymes as the basis components. However, our abstaction pipeline is not limited to deoxyribozyme architectures. To emphasize this point, a DNA description language, DNADL, was created to serve as the top level of the

abstraction stack. DNADL is similar in flavor to electronic computing assembly code and is itself designed to serve as a target of a higher-level programming language.

We begin in this chapter with background on how deoxyribozyme-based computing works in order to provide sufficient foundation for the dissertation. We discuss where the present state of the art lies and what technical difficulties are present. The artifacts of the dissertation are applicable to any form of DNA computing, or DNA device building. Chapters 2 and 3 cover the abstractions and related algorithms. Chapter 4 covers implementation into the Pyxis compiler, the KCA simulation, and the DNADL language. Chapter 5 presents a summary list of contributions.

1.3 Platform, Programming, Achievable Computation

In this section we outline in detail the context for our questions within the confines of the present state of technological development of deoxyribozyme-based molecular computing. We describe platform characteristics as principally developed by Stojanovic and Stefanovic [69, 73, 88, 89, 119, 120, 121], programming steps, and what limitations are present. Additional supporting deoxyribozyme technology development was reported by Stojanovic, Margolin, and Kolpashchikov [67, 76, 122, 123].

1.3.1 Deoxyribozyme Computing Platform

The basis of all molecular computing, including deoxyribozyme-based architectures, is chemistry, therefore a new platform is constructed for each computation execution. There is no notion of a fixed, general purpose assembly. Instead, a well plate provides a matrix of reaction compartments where one or more wells contain species mixtures (Figure 1.2).

Each holds a small volume of buffer liquid and DNA, and there is no flow between them. Although different computations may be simultaneously occurring in each of the wells, they are independent and are unable to communicate. This technology limitation precludes flow of information across wells, and requires some duplication of logic and inputs in each participating well as a work-around when needed.

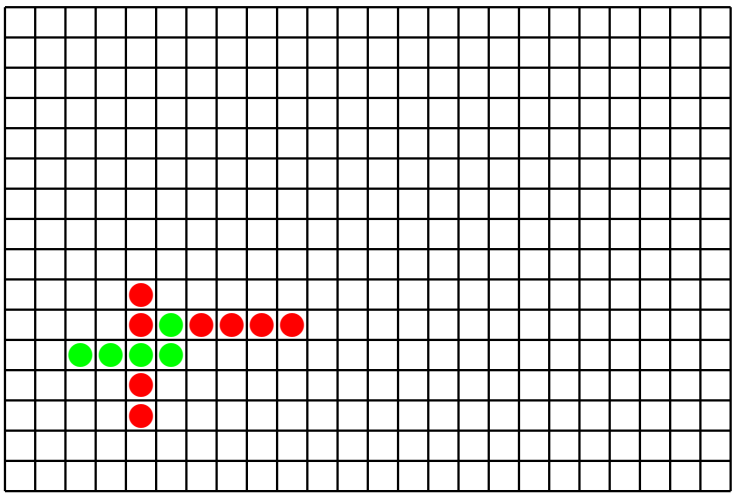


Figure 1.2: 384-well plate using a two color output scheme to signal results of a computation. Although finite, the combinatorial space of wells and colors leads to significantly large numbers of interpretable input/output/work patterns. For example, 384 wells limited to only two marking colors yields 2^{384} distinct patterns, while increasing to five marking colors yields 5^{384} distinct patterns.

DNA species are short single-stranded oligonucleotides and are usually no more than 60-130 bases long. Each species is introduced in nanomolar to micromolar concentrations into specially prepared buffers, identically constituted for each participating well. In most constructions we do not consider the ordering of species introduction, except for species that are part of the read-out reaction. Depending on viewpoint, this is either a limitation or a benefit. Oligonucleotides are named and classified according to the function they are destined to carry out. At present, there are three classes: gates, inputs, and substrates.

Gates act as basic units, similar to electronic logic gates used in combinational and sequential circuits in digital electronic integrated devices. Gates and inputs work together as switches to execute Boolean calculations, and gates and substrates work together to signal the calculation results. All calculations are performed autonomously, without feedback or guidance from a control system.

The basic gate set (Table 1.1) encompasses sufficient functionality to encode any Boolean formula. There are two single input gates, YES and NOT, two double input gates, AND and ANDNOT, and one three input gate, ANDANDNOT. The YES and NOT act analogously to buffer (I_a) and inverter ($\neg I_a$) gates respectively in electronic circuits. The AND acts as a 2-input Boolean *and* ($I_a \wedge I_b$), whereas the ANDNOT acts as a 2-input Boolean *and* with one input inverted ($I_a \wedge \neg I_b$) and the ANDANDNOT acts as a 3-input Boolean *and* with one input inverted ($I_a \wedge I_b \wedge \neg I_c$).

Basic Deoxyribozyme Logic Gate Components										
Single			**Double**				**Triple**			
Input	Output		Input		Output		Input			Output
I_a	YES	NOT	I_a	I_b	AND	ANDNOT	I_a	I_b	I_c	ANDANDNOT
0	0	1	0	0	0	0	0	0	0	0
1	1	0	0	1	0	0	0	0	1	0
			1	0	0	1	0	1	0	0
			1	1	1	0	0	1	1	0
							1	0	0	0
							1	0	1	0
							1	1	0	1
							1	1	1	0

Table 1.1: Logic gates YES (I_a), NOT ($\neg I_a$), AND ($I_a \wedge I_b$), ANDNOT ($I_a \wedge \neg I_b$), and ANDANDNOT ($I_a \wedge I_b \wedge \neg I_c$) used in encoding single-layer Boolean formulas.

1.3.2 Deoxyribozyme Computing Components

Basis Components

Deoxyribozyme logic gates are constructed with DNA enzymes. In both natural and synthetic systems, enzymes serve to catalyze reactions. Most natural enzymes are made of proteins, some are made of RNA (ribozymes), yet none have yet been found to be made of DNA (deoxyribozymes). In each case, the role of the biological enzyme is to speed up a transformative reaction involving nucleic acids such as splicing or cleaving. These transformations use a variety of mechanisms, yet all of them rely on the enzyme's flexibility to take on different structural and spatial orientations. This allows the enzyme, typically a short molecular chain, to properly align with a target substrate molecular chain and maximally promote intended substrate modification, with rate increases up to a billion-fold over the uncatalyzed rate as reported by Breaker [15]. Since DNA, like other biomolecules, is able to fold into a variety of secondary and tertiary structures, chemists have succeeded in using DNA as a raw material to devise DNA-based enyzmes. These synthetic enzymes possess functional characteristics similar to their natural protein and RNA-based counterparts [16, 105].

DNA enzymes have been designed and tested together with specific short oligonucleotide substrate molecules to determine the particular sequences capable of cleaving a substrate. Substrates are composed of either all RNA, or DNA with a single embedded RNA nucleotide. The two enzymes most commonly used in constructing deoxyribozyme logic gates are the E6 and 8-17 enzymes (Figure 1.3). Each enzyme has an operational dependency that requires introduction of a metal ion to produce catalytic rate gains. This ion can be incompatible with a living system such as Zn^{2+}, or able to mimic real biological system conditions such as Mg^{2+}. After enzyme-substrate binding, substrate cleavage occurs immediately to the left of the single RNA nucleotide marked in red in both diagrams. The

enzyme unbinds, allowing the separated substrate pieces to move apart, and is then ready to repeat the cycle with additional uncleaved substrate molecules.

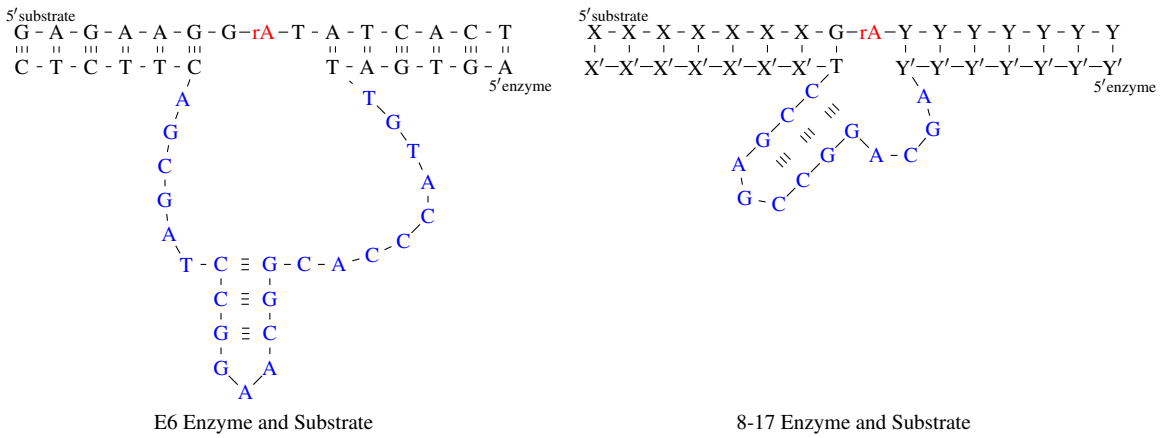


Figure 1.3: Sequence and structure of the E6 [16] and 8-17 [105] enzymes and substrates, where each enzyme has a catalytic core (blue) built with a small stem-loop. The cores are flanked by two binding arms which allow base pairing to the DNA-RNA substrate. The arm sequences for the E6 are fixed, but are variable for the 8-17 as long as arm-substrate complementarity is preserved for each domain, and sequence assignments are unique. The 8-17 arm-substrate bonds are placeholders since bond strength is not determined until subsequence assignment for domains X and Y are made. Each substrate has a central RNA adenine nucleotide shown in red.

Platform Readout

The E6 and 8-17 enzymes form the basis of deoxyribozyme logic gates. Substrates are labeled with a *fluorophore* at the 5' end and a *quencher* at the 3' end to exploit the effect of Förster Resonance Energy Transfer (FRET) as a means of signaling logical output state (Figure 1.4).

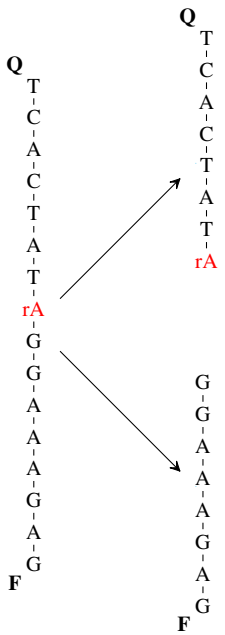


Figure 1.4: Two common donor (F) and acceptor (Q) pairs are tetramethylrhodamine (TAMRA) and Black Hole 2 (BH_2), and fluorescein and Black Hole 1 (BH_1). TAMRA is excited with light at wavelength $\lambda_{excitation}$ = 530 nm, and upon separation from the quencher generates a red light signal at wavelength $\lambda_{emission}$ = 580 nm. Fluorescein uses input light at wavelength $\lambda_{excitation}$ = 480 nm, and generates upon separation a green light signal at wavelength $\lambda_{emission}$ = 530 nm.

FRET was first theorized classically by Jean Perrin in 1909 and later given quantum mechanical treatment by Förster in 1948 as a non-radiative dipole to dipole electronic energy transfer mechanism. In 1978 Stryer showed experimental results [125] both proving the theory and pointing out applicability as a biomolecule spectroscopic ruler since the distance scales are consistent. Under UV light exposure, fluorophores act as energy donors and quenchers as non-emitting acceptors. Close proximity and overlap of the donor emission and acceptor absorption spectra allow sufficient transference of donor energy such that it appears "quenched" even though there is still some residual donor fluorescence. In the case of deoxyribozymes [67, 76, 122, 123], the effect provides both an accurate method to detect the physical state of the substrate and a natural mapping to digital interpretation. An intact substrate has most fluorophore output absorbed and serves as an *off* signal. Alternatively, following substrate cleavage, the two halves diffuse away and allow fluorescence build-up to serve as an *on* signal. Different fluorophores and appropriate quenchers can be successfully attached to substrates, and since each fluorophore emits in a different part of the spectrum, there is greater flexibility in output expression and the possibility of synthesizing and programming a system employing multi-valued logics.

Applications typically signal one color in each well for the duration, yet the output signal capacity can be expanded beyond base-2 digits used in electronic computing. Experimental results have shown that the signalling logic can reliably employ up to five colors in each individual well used in two different ways. The first way allows for simultaneous presence of multiple substrates, each with different fluorophore-quenchers, such that the signal can be one of five different colors. This arrangement yields a possible base-5 digit per well. The second way is an extension of the first and uses additional programming of the plate reader to allow multicolor output where the main technical issue becomes visual color disambiguation. If all possible combinations are considered, a base-31 digit is conceivable since there are $\sum_{i=1}^{5} \binom{5}{i} = 5+10+10+5+1 = 31$ separate ways to combine

colors as singles, doubles, triples, quadruples, and the final quintuple. If absence of a color is designated as a signal, then the base capacity of each digit scheme increases by one.

For all of the variations, output signals are persistent since gates do not yet possess reliable reset functionality, a limiting characteristic which has obvious implications for arranging computation. Figure 1.5 shows an example of fluorescence emissions over time, where observations are taken every fifteen minutes until saturation. Estimated product yield as the concentration of cleaved substrate with the attached fluorophore may be computed at each time point. This is accomplished using the initial substrate concentration, the series of fluorescence readings over the reaction period, and a baseline set of readings at known different product concentrations. Overall reaction times may extend to three hours.

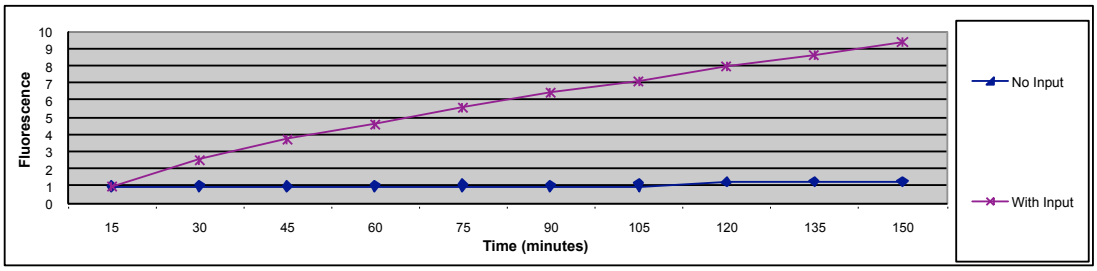


Figure 1.5: Rate of fluorescence gain, $\Delta f / \Delta t$, normalized to arbitrary units vs. time t. Signal strength varies with FRET fluorophore/quencher characteristics and does not achieve consistent maxima, therefore low and high thresholds for each FRET pair must be defined to distinguish 0 and 1 signals from intermediate values indicative of an error status.

Mapping to Two State Logic and Boolean Formulas

Treating the physical condition of substrate molecules as a two state system, with FRET as a read-out method, is not unlike the mapping of 0 and 1 states to high and low voltages in electronic circuits. A variety of methods [67, 119] have been developed to control access to the substrate binding regions by the enzyme, yielding the ability to transduce inputs

into outputs corresponding to various Boolean logic connectives. The simplest method is to add one or two stem loops to the $5'$ and $3'$ ends of the enzyme molecular chain, such that for each involved stem, one of its single strands possesses the same substrate subsequence. The reverse complementarity between stems and enzyme effectively sequesters these regions, termed *critical regions*, and precludes the enzyme to substrate binding reaction as long as the stem is intact. Hybridization between complementary subsequences again plays a pivotal role as a control mechanism. If one (YES gates) or two (AND gates) input oligonucleotides which are the reverse complements to the loop regions are introduced they will bind to the loops, pull the stems apart, and reverse the situation. The secondary enzyme-substrate binding reaction may then proceed, and further, we see that the 0-1 mapping applies to both the substrate and stem-loop physical state. When a gate is *activated*, the appropriate inputs have been introduced in solution and subsequent substrate cleavage generates an output signal. When a gate is not activated, stems remain intact and no cleavage reactions occur. Each input acts as a logical input, therefore the single input YES gate requires only one stem-loop, while the double input AND gate requires two stem-loops. The expected hybridization bindings, input-loop first, followed by enzyme-substrate, must have a majority response for all participating strands in solution in order to generate an unequivocal FRET output signal. AND-gates with only one input introduced are not able to bind to both substrate ends, therefore cleavage and signaling are prevented.

A different method [119] is used to alter the enzyme-substrate binding reaction to create negation used in the NOT gate, ANDNOT gate, and ANDANDNOT gate. These gates behave as input inverting rather than output inverting since negation of an input is recognized. Input negation is created by use of the E6 enzyme (the 8-17 is unable to support this function), where a much larger stem-loop structure is substituted for the small inner loop of the enzyme. This provides a similar recognition mechanism for gate activation as in the

YES gate and AND gate architectures. Without introduction of input complementary to the inner enzyme loop region, the gate is already activated and generates an output signal. Conversely when input is added, the enzyme core deforms, leading to gate deactivation and loss of output signal. Therefore absence of input maps to an *on* state, and presence of input maps to an *off* state. The ANDNOT and ANDANDNOT gate designs combine the enzyme core modification to handle negation of one of the inputs with one or two additional stem-loops added to the enzyme binding arms to handle the single or dual positive inputs respectively.

A third class of gates [69] includes *pre-complexed* gates where loop regions and partial stem sections are hybridized to additional complementary oligonucleotides prior to introduction of inputs. This negates activating stem-loops attached to enzyme ends, or the single inhibiting stem-loop attached to the enzyme core region. Starting with the ANDANDNOT gate equivalent to $I_a \wedge I_b \wedge \neg I_c$, a covering complement containing the $\overline{I_c}$ subsequence deforms the enzyme core region, rendering it deactivated. Upon introduction of the shorter I_c input, the I_c:$\overline{I_c}$ binding pulls off the deformative blocking oligonucleotide to allow enzyme core refolding back to its active state. Overall, the ANDANDNOT ($I_a \wedge I_b \wedge \neg I_c$) gate becomes an ANDAND ($I_a \wedge I_b \wedge I_c$) gate, to yield full three-input logic. Similarly, blocking oligonucleotides may be introduced to pre-bind to one of the activating stem-loops to modify positive activation into negative inhibition. In this variation the ANDANDNOT ($I_a \wedge I_b \wedge \neg I_c$) gate becomes an ANDNOTANDNOT ($I_a \wedge \neg I_b \wedge \neg I_c$) gate where two inputs rather than one are negated.

Component Modularity

All deoxyribozyme gate components are largely modular and may be switched out to optimize functionality. YES and AND gates may use either DNA enzyme, while the rest must

use the E6. Additionally, there is flexibility in designing the YES gates. Because only one stem-loop is required it is possible to place it on either side of the enzyme: when it occurs before the enzyme at the 5' end, it is unofficially termed a LEFT-YES gate, and when it occurs after the enzyme at the 3' end it is termed a RIGHT-YES gate. Positive input stem-loops for the AND gate may be switched so that a gate encoding $I_a \wedge I_b$ becomes $I_b \wedge I_a$. The same holds true for the ANDANDNOT gate where $I_a \wedge I_b \wedge \neg I_c$ is logically equivalent to $I_b \wedge I_a \wedge \neg I_c$. The full suite of the basic designs is shown in Appendix A.1.

The largest source of modularity arises from the loop recognition regions of each gate stem-loop. Tremendous freedom appears possible in making up the loops since the number of subsequence choices for a single oligonucleotide is exponential in the length n of its string of bases ($\Sigma = \{A, C, G, T\}$, 4^n strings). Typically these loops are fifteen nucleotides long, therefore each stem-loop may be parameterized with up to 4^{15} different subsequence assignments. Selection of these subsequences from such a large space is a central design problem for architecting individual gates and specifying entire systems.

Platform Kinetics

The basic reaction sequences for all Table 1.1 gates are shown in Appendix A.2. In practice, several important experimental observations have been noted. First, the rate limiting step for all deoxyribozyme reactions is the catalytic gate-input-substrate complex reaction which cleaves the substrate [82]. Second, reactions are sensitive to species initial concentrations and it is not the case that selectively increasing concentrations yields more products or faster results. On the contrary, the desired general three-step procedure initiated with gate activation, and completed with substrate signalling, can sometimes be fatally perturbed in unknown ways and not generate the desired reactions. This is not a limitation of the technology per se, rather it is an indicator of how difficult it is to capture

and examine nanometer-scale behaviors in order to finely tune them. Silverman [117] has bluntly pointed out the need for "structural and mechanistic investigations of DNA catalysts," while Halász [5] has noted the existence of experimentally inaccessible reaction mechanisms. All reactions are as well part of a dynamic liquid system and subject to diffusion. The correct gate, input and substrate molecules must find each other in solution while undergoing collisions with all other well species and water molecules, therefore there is constant competition for binding opportunities, as well as dissolution of bound regions.

Platform kinetics are equal in importance to gate design when devising a system. The speed of the slowest reaction dictates the overall speed of the entire computation for single layer logic programs, and will be the execution time bottleneck for multi-layer logic programs. Various approaches have been undertaken which *simulate* chemical system evolution in terms of likely reactions, initial species concentrations, and rates where they are known. Simulation codes such as COPASI [60], DSD [91], and ENZO [11] are valuable if they can be provided with good input data. We have equally treated kinetics using a simulation approach and incorporated a kinetics pass for estimation of intermediate product yields within the compiler pipeline.

1.3.3 Platform Programming and Instantiation

Using the catalog of deoxyribozyme gate architectures, platform construction and programming tasks are divisible into two groups roughly equivalent to the compiler front and back ends used in electronic computing. Front-end type tasks involve determining the total number of required wells, determining the logic formulas for each well, converting the formulas into disjunctive normal form (DNF), and then finally converting each DNF clause as needed into an equivalent expression involving only the logic connectives available in the gate design catalog. Previous applications have all been single-layer logic such

that the series of disjuncts act as an *or* over all clauses. Back-end type tasks involve making instantiation choices to effect each intended Boolean formula. These include selecting an enzyme, choosing one or more fluorophore-quencher pairs, and making an assignment of DNA subsequences for all inputs and therefore also their corresponding reverse complements within the loop regions of each gate stem-loop. Because there is potential for unwanted hybridization reactions happening along with the directed ones, the assignment problem is non-trivial.

There is a certain level of freedom in specifying a system. Unlike electronic computing, where there is a fixed number of bits per word per architecture, and a large but finite memory, we can assume there is no practical limit to deoxyribozyme platform size. There are several ways a platform may be measured, and with each there is the possibility of scaling. We place no limit on the size of the Boolean expressions in each well, nor on the number of participating wells, nor on the interpretation of well-plate signal patterns. Although now execution is typically worked out by a human technician, we further assert that any arbitrary assignment of substrate, gate and input oligonucleotides, into any number of wells, can be adequately executed as needed instead by use of robotics or other automated means.

To show the range of practical computing capability, and exemplify the high-level programming steps, we briefly outline previously built systems, and discuss one in development. In each of these successes, the engineering effort to solve the required low-level design and optimization tasks was challenging, yet each computation is inspiring and thought-provoking, and quickly suggests other interesting and practical problems which could be similarly worked out.

Digital Logic Programming

A deoxyribozyme-based half-adder was reported by Stojanovic and Stefanovic [120], and a full adder was reported by Lederman [69]. The half-adder handled two single bit inputs, while the full adder handled three single bit inputs, each producing correct sum and carry digits interpreted from signaling the results of input summation encoded into simple Boolean formulas. In these platforms all computation took place in a single well, using the truth tables shown in Figure 1.2.

Adder Logic Truth Tables						
Device	**Input**			**Output**		**Interpretation**
	i_1	i_2	i_3	Carry	Sum	Base 2
Half Adder	0	0	-	0	0	00
	0	1	-	0	1	01
	1	0	-	0	1	01
	1	1	-	1	0	10
Full Adder	0	0	0	0	0	00
	0	0	1	0	1	01
	0	1	0	0	1	01
	0	1	1	1	0	10
	1	0	0	0	1	01
	1	0	1	1	0	10
	1	1	0	1	0	10
	1	1	1	1	1	11

Table 1.2: The half adder processed two inputs to produce two outputs, while the full adder processed three inputs to produce two outputs. For each, the sum output was signaled in red through use of fluorophore TAMRA and quencher BH_2, while the carry output was signaled in green through using fluorescein and BH_1. For the full adder, the well plate reader was programmed to allow dual signal acquisition of simultaneous fluorescence of both fluorophores.

The half adder used two ANDNOT gates to produce an XOR for the Sum output and a single AND gate for the Carry output. The full adder used three pre-complexed ANDNOTANDNOT

gates, and one pre-complexed ANDAND gate to compute the Sum output. Three AND gates were used to compute the Carry output. The full formulas logically encoding the truth tables are shown in Appendix A.3.1.

Game Programming

Three different game playing automata have been demonstrated, all termed Molecular Array of Yes and And Gates (MAYA). MAYA1 (Stojanovic and Stefanovic [121]), and MAYA2 (Macdonald [73]), played tic-tac-toe without losing for all possible game variations by transforming a non-losing strategy into Boolean formulas. The third automaton, MAYA3 (Pei [89]), improved the demonstration of synthetic intelligence by exhibiting the ability to learn from a human guided training period to play a game of tit-for-tat. Each of the MAYAs used multiple wells, with different formulas encoded per well. The choice of games reflects the limitation of persistent signaling, since game play involved claiming board squares only once. In contrast to the adders, the programming inception point was not a truth table. Instead, the natural beginning was the game decision tree, from the perspective of the automaton. We focus on the second generation tic-tac-toe automaton MAYA2 to show the leap in complexity.

MAYA2 made a single initial assumption of the automaton playing first in the middle square. From this point on, each possible human move and the optimal strategic response was determined and diagrammed as a set of decisions. The decision tree was captured as a Mealy machine where nodes represented the historical set of moves already played, and edges represented the next possible human/automaton move pairs as Mealy machine input and outputs. We recall that Mealy machines act as deterministic finite transducers which convert strings in one language into strings of either the same or a different language. For the MAYA2 game, input and output strings ($\Sigma = \{1,\ldots,9\} \setminus \{5\}$) represented the

concatenated set of human and automaton moves respectively, and the labeling of nodes with substring pairs served as a form of memory: it was encoded into the automaton logic to always "know" what had been already played. Each well signaled a single move by either player, therefore two output colors were required to distinguish moves. Using the input to output transduction encoded in the Mealy diagram, automaton moves were mapped onto Boolean formulas, and then simplified to a form expressible using only the YES, AND, and ANDNOT gates. Human moves were appended as single variable clauses expressed using YES gates. The final formulas for each well are shown in Appendix A.3.2.

Sensor Platform Programming

Our third and final example is a project in development where recognition of inputs and signaling using only YES gates is the basis for a DNA-level virus detection platform. The platform will act as a molecular quorum sensing device able to make decentralized decisions based on achieving sensing thresholds. The platform is intended to be used in austere environments and rely little on up-front sample preparation or sophisticated operator training. Conceptually, the logic is straightforward: $I_{strain1} \vee I_{strain2} \vee \cdots \vee I_{strainN}$ for N viral strains in a single well for the simplest possible implementation. Since YES gates signal *on* in the presence of activating input, the gates can act as sensors to indicate presence or absence of pathogens. The platform requires a bioinformatics effort to determine representative reverse complement 15-mer subsequences to place in the loop region of the single attached stem-loop in each gate. Depending on the genomic separation of strains the detector covers, multiple signature subsequences might be required for unequivocal discrimination. With a single well approach, signalling requires N different substrates labelled with non-overlapping FRET fluorophore/quencher pairs. Yet this would be prone to interpretation error on the part of the human user, hence we instead exploit the flexibility of arranging gates within multiple wells. With careful programming we can distribute gates

such that positive identification is displayed in a dot-matrix readout as shown in Figure 1.6. This illustrates a unique platform characteristic — by shifting more programming effort onto the platform, a computation output can be immediately readable in any language or number system. Formulas for the figure example are shown in Appendix A.3.3.

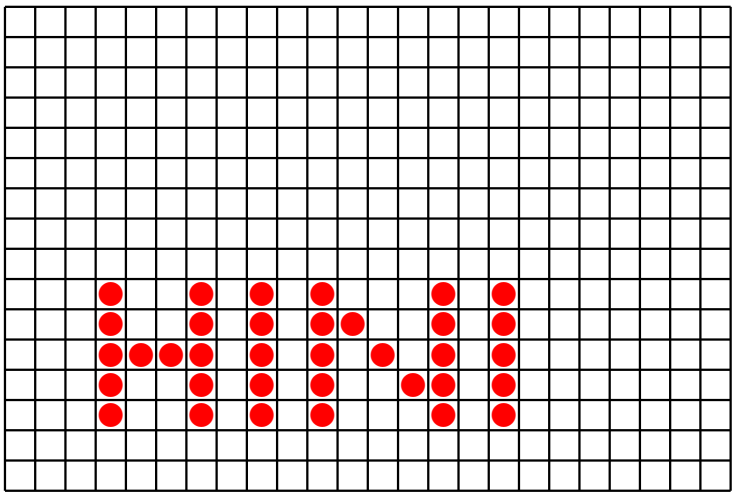


Figure 1.6: Example output for positive viral strain detection. Programming involves distributing YES gates for each strain covered by the detector platform such that simultaneous signaling in multiple wells creates a human readable identification string in a dot-matrix display.

1.3.4 Contrasts with electronic computing

The short history of molecular computing shows essentially the same challenges facing engineers 60 years ago when computer science started branching from logic and computability questions into making real machines and devising practical applications. How can the execution of algorithms be worked out on a device consisting of molecules in solution? The state of the art in molecular computing, now, is really all about designing and building circuits. To understand this claim, we find similar status in the early days of electronic computing when we recall that in 1947, the US Army had a very new and exciting tool at its disposal. The Electronic Numerical Integrator and Computer, ENIAC, was operated by programmatic control of memory. A program input signal *stimulated the unit to perform* [23]. A single function, as selected from a table of functions, required presetting of switches to prime circuits responsible for executing the function. Output, as the result of applying the function to the input, was emitted by signal, or the machine could be instructed to retain this information for later use. Prior to ENIAC deployment, a function and input written on a piece of paper could have been handed to a human computer with the result handed back on a different piece of paper, and perhaps checked by an additional person. Indeed, these actions were captured exactly as ENIAC was modeled conceptually by Turing and von Neumann upon human procedures and protocols. Use of electronics as a computing substrate prevailed over neurons by virtue of scaling where far larger numbers and more complicated operations on numbers could be worked out reliably at a much faster rate. The point is not that people were easily overwhelmed or stupid. Their actions were effectively copied and recreated in painstaking fashion. The remarkable and unprecendented achievement was that interesting problems could be cast onto an electronic platform. The mapping of executing mathematics in a device required methods to represent numbers and their operators, carry out operations in a precise order, store values, and report them. Since these methods were generalized and not restricted to

Molecular Platform	**Silicon Platform**
no clock: reactions initiate and proceed	commonly uses a clock signal tied to an internal crystal resonant frequency
elements carry their own required energy	outside power source required
write once read only memory	uses an updatable memory
single pass feed-forward only logic	multiple pass hierarchical logic levels
no reset	can reset
up to base-5 native number representation	base-2 numbers only

Table 1.3: Hardware comparison between molecular and electronic silicon based computing platforms.

small numbers or small problems, scaling could occur.

When we compare the two regimes we see the scaling differential immediately, both in the basic architecture, as well as the maturity of the technology. This comparison, shown in Table 1.3, strives to highlight what niche is available to molecular computers in light of current overwhelming electronic platform superiority, and serves to temper the terminology we are tempted to employ when discussing molecular computing. No group can realistically suggest future competitiveness with electonic computing since we are still firmly within the phase of recapitulating electronic design and organizational motifs. A better alternative will be development of an entirely different problem encoding and solution execution paradigm, and in the interim, we already have the potential capability of natural system interface since the molecules involved are biologically based.

1.4 Related Work

The first successful experiment proving the viability of using nucleic acid chemistry as a computational substrate was accomplished in 1994 by Adleman [6]. Adleman showed a

method to solve a small instance of the NP-complete Hamiltonian path problem which asks if there is a path for visiting each vertex of a directed graph without repetition. Edge and vertex relationships were encoded into DNA sequences, without the benefit of any high-level language, or design tool set for generation of the sequences. Seeman [109] predated this work in 1981 with the first nucleic acid based synthetic constructions, and also wrote one of the earliest sequence design programs, SEQUIN [110, 111]. At this time, the most mature non-enzyme based DNA computing technology and process toolset all emanate from the substantial work accomplished by Pierce and Winfree's groups at Caltech. There are two different molecular computing architectures which have been well developed from these groups: DNA tiling systems (Rothemund [102]) and strand displacement systems (Seelig [107]).

Only recently has the field started using the mainstream terminology associated with automated circuit design, high-level computing languages, program compilation, and program execution on standard silicon electronic computing platforms. Design methodologies includes works separately accomplished by Dirks and Zhang [27, 28, 29, 140], a combinatorial optimization approach by Kai [63], molecular recognition as signal detection by Savir [106], and DNA motif, crystal, and origami construction techniques by Seeman [108]. The fundamental DNA word problem is intrinsically part of any design approach, and attracted significant attention early on, notably with contributions by Andronescu [8] and Marathe [74]. Recent theoretical formalization and language development work includes interrelated approaches by Cardelli, Phillips, Qian, and Winfree ([20, 90, 95]). Using elements of all these works, groups headed by Winfree (Shin [116]), and Riedel (Shea [115]), have conceived of and built molecular compilers aimed at their respective architectures. Of note is their conceptual mapping of programs to solve the sequence assignment problem as assemblers. An alternative DNA molecular compiler has been in long-term development by Feldkamp [36, 37], while Way and colleagues [131]

have explored non-nucleic acid molecular compiler and platform building. Experimentally validated systems, arguably the strongest results, include results from Benenson [9], Elbaz [33], Rothemund [102], Qian [94, 96], Strack [124], and Wu [137].

Chapter 2

Abstractions Development

Everybody who has analyzed the logical theory of computers has come to the conclusion that the possibilities of computers are very interesting – if they could be made to be more complicated by several orders of magnitude. – Richard P. Feynman [39]

2.1 DNA Computing Model Foundations

Developing abstractions for deoxyribozyme computing facilitates the design-build process towards reliability guarantees, and the understanding of the computer science aspects of molecular computing. The implicit statistical physics and chemistry questions which arise can easily overwhelm the computational questions or lead to just applied computer science approaches with little recognition of the deeper implications. Moret [81] observes that "since much of complexity theory is about modeling computation in order to understand it, we naturally want to study new devices such as DNA computing, develop models of its mode of computation, and compare the results with current models."

Historically, there are two modes of thought in devising a model: focus on the logic level or focus on the implementation level. Finite state automata, stack machines, grammars, cellular automata, tiling systems, recursive functions, lambda calculus, and Turing machines are all abstract computing models at the logic level. The logic level addresses the computability of transforming inputs into outputs, the study of problems, and creation of the complexity hierarchy. At the smallest indivisible abstraction, it is enough to discuss operations. Operations undertaken in solving a problem instance will execute in unit time or space, without our knowledge of exactly how. We simply count the total number of units and use the overall resource demand to rank problem hardness relative to others and therefore classify the complexity. This yields a way to write and weigh the utility of algorithms invented to solve problems. In a sense, despite the fact that it appears very far away from the machine under the microscope, logical model creation is practical. Blelloch [12] echoes this sentiment, noting "the ultimate purpose of an abstract model is not to directly model a real machine but to help the algorithm designer produce efficient algorithms."

Implementation-level models, which mimic the system under study, serve the hardware designer and allow design trade decisions. This level uses component abstractions combinationally to devise architectures or sub-architectures. It is often assumed that components are free of dependencies and contextual requirements, and are thus able to be arbitrarily combined. DNA chemical reaction networks, in contrast, do not admit dependency-free operating conditions. Component operation is sensitive to design and build conditions such as the sequence of oligonucleotide bases, their concentrations, and the chemical environment in solution. Molecular computing implementation-level models are therefore physical-level models. Physical attributes and processes are describable for single entities, or populations of entities. The populational aspect is a critical distinguishing factor that is not present for modelers abstracting electronic circuits. For DNA systems, all elements are present as multitudes and hence form multisets rather than sets. Population interactions

occur in any one of an exponential number of non-centralized timelines, hence a different approach is warranted to determine suitable abstractions to serve as atomic operations in a complexity measure.

2.2 Physical-Level Entities and Interactions

Knowledge of nucleic acid composition, organization, and potential interaction is critical to understanding their behavior and use. In natural systems DNA reflects evolutionary state by encoding genetic information. RNA serves as an intermediary between DNA and proteins for coding genes, acts as an enzyme [21], regulates gene expression [86], and orchestrates embryonic organogenesis [25]. In addition to primitive computing platforms, DNA has also been used in the construction of synthetic biology devices such as data storage systems [22, 51], theranostic sense-delivery instruments [53, 75, 100], logic circuits [89, 93, 96, 124], and molecular scale machines [72, 129, 134, 138].

In general, transitioning from the science of nucleic acids to the engineering of nucleic acids requires a smart control system to promulgate specific behavioral responses. This is, in effect, no different than directing the discharge of electrons in a transistor. Understanding of the physics of electricity and semiconductors through detailed experimentation enabled the engineering of digital computers. Understanding of the physics behind the chemistry of nucleic acids will in time achieve parity. DNA and RNA properties tightly depend on environmental conditions, and interactions occur within multiple time and spatial scales. In natural systems, the result is robust, fault tolerant processes that we would like to emulate. At present, the extent to which native properties and interactions are fully rendered into working models largely dictates the complexity and sophistication of what can be engineered.

2.2.1 Entities

DNA and RNA are chains of nucleotides, where each nucleotide consists of a nitrogenous Lewis base bonded to a five-carbon sugar, and polymerized together by sugar-attached phosphate groups. The bases guanine (G), adenine (A), thymine (T), and cytosine (C) in DNA, with uracil (U) substituted for thymine in RNA, additionally interact through hydrogen bonding to hybridize single-stranded regions into double-stranded helices. Natively in the cell, DNA is a double-stranded helix primarily formed through G-C and A-T base-pairing, while single-stranded RNA folds into various functional configurations through G-C and A-U base-pairing. For molecular computing or alternative synthetic biology constructions, DNA is used as a raw material with single strands engineered to self-assemble into designed helices and folding shapes.

At a fine grain, each DNA or RNA nucleotide is composed of atoms, and they interact by way of atomic collisions. Because the energies involved in each collision are not enough to disturb the atomic nuclei, it is sufficient to understand the overall interactions on the basis of electromagnetic forces and energy exchange, principally hydrogen bonding that is responsible for base-pairing. Hydrogen bonding in biological-based systems is an association between two electronegative atoms, linked by an intermediate electropositive hydrogen atom, to create two proximal dipoles. Bond length denotes the distance between atomic centers whereas bond energy is the amount of energy necessary to break the bond and produce neutral fragments. The formation of an ionic bond between atoms of biological systems is not energetically possible because the atoms are moving too slowly to allow for creation of cation/anion pairs that require a large activation energy input as the ionization energy. Instead nucleic acid behavior, principally Watson-Crick base pairing, is governed by hydrogen bonding. Hydrogen bonding is a type of covalent bonding where the positively charged electron of a single hydrogen atom allows for formation of two

dipoles via attraction to two different negatively charged electrons in nitrogen and oxygen atoms flanking the opposing pentose rings found in each opposing nucleotide. Cytosine to guanine Watson-Crick bond formation is nothing more than three dipole pairs, one as N-H-N, another O-H-N, and the third N-H-O, while thymine to adenine pairing is two dipole pairs, one as N-H-N and the other as O-H-N.

Oligonucleotide sequences are termed *primary* structure, their folding patterns which result from stretches of base-pairing are the *secondary* structure, and the actual physical orientation in 3D space is the *tertiary* structure. While tertiary structure data is the most valuable, it is also the hardest to obtain experimentally. For both DNA and RNA, the folding of a single strand into helices occurs as a multi-step process that follows multiple parallel pathways [135]. Pathways generally exhibit collapse to an intermediate compact form followed by a slower diffusive conformational search to a lower thermodynamically stable state [103, 135], or to possibly kinetically trapped misfolded states [136]. The energy driven search for an optimal arrangement of hydrogen bonds is true for single strands, or many strands interacting together. All principal entity-level physical properties are outlined in Table 2.1.

2.2.2 Interactions

Interactions can be reactions involving DNA or RNA oligonucleotides, or the rearrangement of base-pairing hydrogen bonds in a single oligonucleotide. The result in either case is production of new molecular species and reduction of existing species. The exact mechanisms responsible for these transformations are not comprehensively known and multiple competing mechanisms are possible. The set of all known and unknown interactions constitutes the chemical reaction network. Network behavior is also subject to change under varying environmental conditions. For example, hybridization is sensitive to

Classification	Property
gross aspects	molecular weight physical dimensions base sequence base count base-pairing pattern
mechanical	groove type bend and twist angles molecular stress and strain vibrational frequencies
thermal	melting temperature stacking energy base-pairing binding energies
electrical	charge distribution

Table 2.1: Physical properties pertaining to a single DNA or RNA oligonucleotide. The associated cost of obtaining accurate, high quality property descriptions varies with the difficulty of direct measurement or modeling of each property.

different buffer conditions, principally through the use of salts such as Mg^{2+}. Magnesium, manganese, and cobalt divalent cations, or a number of monovalent cations such as Na^{+}, have the effect of shielding negative charge on the phosphate groups that are a part of each nucleotide molecule. An elevated concentration of one of these salts will stabilize bond formation by opposing the phosphate group negative charge repulsion [128]. Overall, the high degree of variability and probabilistic nature of competing mechanisms prohibit exact characterization of the network, yet directing the network to serve as a computational substrate, and allow execution of a program, is the goal of molecular computing.

If it were possible to know the position, momentum, and therefore velocity of each atom within each molecular chain, then system interactions would be in principle amenable to state change computations for exact system modeling [98]. This information is not readily available, however, in a many-bodied system comprising an uncountable number of participants. Statistical mechanics treatment is an alternative where systems are characterized

as ensembles. Aggregate behavior can be deduced by starting with a number of identical systems that are composed of the same constituent components but in different configurations initially. On the way towards achieving equilibrium, there is nothing preventing the occurrence of multiple systems visiting the same state at some point along their trajectory of states. Because the state space is large, the probability of state parameters taking on particular values is computed, rather than focusing on characterizing any one individual system. There are other smaller and more familiar scenarios that benefit from a similar probabilistic treatment. Consider measuring the two-state coin flip system, the eleven state sum of two dice throw system, or the $100,000$ state five digit lotto number system. To understand the nature of their respective state spaces and see if a system is fair or rigged, progressively larger and larger flips, throws and dollars must leave the hand. In this way, each test is an independent system in the ensemble.

The ensemble of oligonucleotide systems can be modeled as a many-particle chemical system, such that their accessible states follow a Boltzmann distribution [98]:

$$p(s) = \frac{e^{-\beta E_s}}{\sum_{s \in \Omega} e^{-\beta E_s}}$$

where E_s is the energy of state s, Ω is the space of all states, and the divisor $\sum_{s \in \Omega} e^{-\beta E_s}$ is the partition function Z (from the German, Zustandsumme, "sum over states"), which normalizes the total probability to 1. Prior to equilibrium, each member system of the ensemble will constantly make transitions between states. When the distribution over states remains uniform, equilibrium has been achieved. Some of these states will be advantageous and reflect oligonucleotides with specifically required properties while others will be deleterious and instead hinder some part of an overall interaction event chain.

This model makes several assumptions, and is not completely consistent with the real systems it is intended to capture. The first assumption pertains to the state of matter. Chemical

reaction networks of interacting oligonucleotides are well mixed liquids. Liquids are both dense and disordered, and therefore more difficult to formulate mechanics for than gases or solids [13]. In contrast, gases are so sparse that the time between particle collisions is much longer than the time of a collision itself, and solids possess regular structure. Liquids possess neither of these aspects and there is no easy way to generalize the gas state Boltzmann distribution nor solid-state lattice approaches towards their behavior. The adoption of the Boltzmann distribution assumes that statistical mechanics methods developed for gases are adequate for well-mixed liquids. Additionally it is assumed that all other liquid properties such as viscosity and thermal gradient induced motion can be neglected. The most critical assumption, however, is that systems are in equilibrium. This is not actually the case for most molecular computing systems, and in particular for anything built with deoxyribozymes. The interactions that carry out platform construction and program execution occur well before equilibrium, hence although this approach correctly casts DNA systems as many-bodied, it is implicitly introducing error because we really don't know the correct model of the state distribution. Furthermore, there is no simple way to determine better statistical parameters without extensive laboratory work to fully elucidate the correct distribution. The engineering challenge is thus to appropriately adjust analytical results and exploit these results in a search process to determine optimal system specification.

2.3 Abstractions

The conceptual value of the Boltzmann distribution model for oligonucleotides is that it does reflect their transient nature, and the range of structural manifestations that are possible for any one sequence. Through Watson-Crick base-pairing, potentially any double-helical region may form wherever there are complementary stretches of bases. This poten-

tial exists within a single oligonucleotide, or between multiple oligonucleotides, and is the basis for working out any form of computation or device building. Controlling hybridization through design of sequences is therefore the central low-level required task. Furthermore, given the current status of molecular computing technology, design effectively is the sole control system. Different tasks that incorporate sequence design within the process of transforming initial conceptions to successful instances of molecular computing are shown in Figure 2.1. These tasks have evolved over time, through the course of the discovering the viability of DNA as a computational substrate and experimenting with different ways to build circuits and applications. Previous successes suggest the process is easy, yet in reality it is challenging and difficult because of the large number of interrelated variables. Hence our goals in devising abstractions have been twofold: the abstractions must serve to reduce development timelines, and increase reliability. To make this concrete, workflows were studied to determine how abstractions could be organized towards replacing the ad-hoc development style with a rigorous systems engineering approach amenable to testing and evaluation.

Figure 2.1: The current process for engineering a platform from concept to finished product can be divided into three stages, proceeding from top to bottom. These processes are deliberately shown disconnected, rather than in a flow chart, to reflect how the overall design and build engineering effort currently proceeds. There is no central, generally recognized, set of steps to program, build, and verify a deoxyribozyme computing platform.

There are three groups of abstractions that have been developed. What this means in practical terms is that our contribution includes utility methods to represent core physical entities and interactions in compressed formats that retain the full scope of information within a more tractable packaging. The groups cover (1) representation methods and algorithms for nucleic acid sequences, (2) representation methods and algorithms for nucleic acid secondary structures, and (3) representation methods for reactions. They work together in the DNADL language, and the Pyxis compiler, to create a logical and pipelined workflow for conception of new molecular computing ideas to successful laboratory execution. The nature of compiling for a molecular platform is in many ways fundamentally different from mainstream computing. A relatively new concept has been proposed for reconfigurable hardware that seems to fit—PICO—program in, chip out. In this regime, rather than the compiler creating code for a particular architecture and instruction set, the compiler crafts the machine to fit the code. Assigning gates to formulas, designing sequences for gates and inputs, and evaluating gate/input/substrate interactions as a whole system are interrelated tasks. The final product is identical to the PICO concept where platform construction of DNA-based logic gates is not separable from writing the program that will be executed on that platform.

2.3.1 Sequence Abstractions

To describe a particular sequence, we can simply list the bases from the DNA or RNA alphabets ordered in the biological standard as 5' to 3'. Formats for sequence files have been adopted that embed alphabet-based sequences with additional relevant data that may cover other physical properties, translation to amino acids, or data provenance identifiers. The largest collection of natural sequences is the GenBank genomic sequence repository.

GenBank now contains 157,943,793,171 base pairs, as of the February, 2014 release[1]. Designed sequences for synthetic use such as molecular computing do not have any standard format, nor are they particularly large in count because there has not been any concerted effort to build searchable libraries. This need does exists, and will increase in importance as nucleic acid devices transition from backing through government financed research projects to manufacturable and salable product lines.

Rather than listing bases in ASCII in flat text files, it is more efficient to use a binary encoding. Two bits are enough to encode either four DNA or four RNA letter sets, yet if we are clever in assigning bit patterns to each letter we can gain the subtle benefit of determining complements as a short series of logical *xor* and *and* operations which are native and fast on standard computer processors.

Definition DNARNABitStringEncoding. DNARNABitStringEncoding represents DNA or RNA strings as bitstrings. The DNA encoding key is T = 00, C = 01, A = 10, G = 11. For RNA strings, T is replaced with U = 00. Each character of a sequence string is mapped in-place to the bit encoding per the key.

As an example, the DNA sequence TCAG is 00011011. No information is lost, and because ASCII characters nominally use a 7 bit encoding, this is a lossless compression scheme, yielding a reduction of 71% of space required per character. The representation also encodes a fast complementarity check between two sequences through bit shifting and the *xor* operation native to electronic computer processors. The Watson-Crick base-pairs T-A or A-T, and G-C or C-G are identified by comparing the first bit of each 2-bit encoding and verifying they are not equal, along with comparing the second bit of each 2-bit encoding and verifying they are equal. For example, if two DNA sequences ALPHA = AACGT

[1]U.S. National Institute of Health, NCBI-GenBank Flat File Release 200.0, February 15, 2014. ftp://ftp.ncbi.nih.gov/genbank/gbrel.txt

and BETA = ACGTT, we can see that they are complementary as ALPHA and the reverse of BETA, BETAREVERSE = TTGCA, consistent with chemistry of DNA hybridization. We encode these as:

ALPHA is 10 (A) 10 (A) 01 (C) 11 (G) 00 (T) = 1010011100

BETAREVERSE is 00 (T) 00(T) 11(G) 01(C) 10(A) = 0000110110

Then comparing each 2-bit interval:

	Oligonucleotide Name	Base	Encoding
Interval 1	ALPHA	A	10
	BETAREVERSE	T	00
Interval 2	ALPHA	A	10
	BETAREVERSE	T	00
Interval 3	ALPHA	C	01
	BETAREVERSE	G	11
Interval 4	ALPHA	G	11
	BETAREVERSE	C	01
Interval 5	ALPHA	T	00
	BETAREVERSE	A	10

Table 2.2: Complementarity checking between AACGT and TTGCA uses fewer native operations per base.

For each interval, the first bits compare as different and the second bits compare as identical, therefore it is quickly computed that the sequences are complementary and able to form a double helix. This check is expressed as $(Bit1A \bigoplus Bit1B) \wedge \neg(Bit2A \bigoplus Bit2B)$ where A and B abbreviate ALPHA and BETAREVERSE.

An alternative is to pack four two bit sequence representation encodings into appropriately selected characters from the ASCII character set. Since computers nominally address memory as bytes, the 7-bit per character ASCII encoding appends a leading zero for consistency across memory boundaries. For example, the ASCII character J is encoded

as 01001010. This can alternatively encode DNA sequence CTAA, where 01001010 expanded as 01 00 10 10 in terms of four 2 bit groups maps as 01 to base C, 00, to base T, 10 to base A, and 10 to base A. Thus in a flat text file the sequence CTAA is replaced by the single letter J, and the original sequence may be reconstituted as needed, making this also a lossless compression scheme. Each variation can be further compressed by composition with other methods such as the Burrows-Wheeler algorithm [19] that is the basis of the Bzip utilities.

Algorithms FindAllHybrids and Separated employ DNARNABitStringEncoding as shown in Chapter 3. Other uses are possible, such as direct query search of all occurrences of pattern subsequences within larger sequences using compressed encodings directly and bypassing expansion into full alphabet-based representations. Therefore, the encoding acts as a homomorphism.

2.3.2 Structure Abstraction

To describe a particular secondary structure, we can simply list those pairs of bases that have formed hydrogen bonds. However, the exact manner in which we present this list (symbolically, numerically, graphically, etc.) greatly affects how easily the secondary structure can be grasped by the human reader, or processed by computer algorithms. Numerous approaches have been proposed in the literature including schematic pictures, dot-parenthesis strings [59], circle plots [85], dot-plots [61], mountain plots [59], arc diagrams, trees [42, 112, 113, 143], compressed tree-like mappings [47], and graphs [38, 64]. To assess each approach, we should ask (1) how much structure information is conveyed? (2) how much is disregarded? (3) how compact is the representation? and (4) how can the abstraction be used effectively?

Optimal representation properties include maximal information, efficiency of use, and ease

of implementation. Ideally, a representation will disregard very little and ultimately be convertible to the full list of base pairs. It will support satisfactory comprehension by the user, and not require constant reference to the definition [84]. An abstraction that reflects small changes in base-pairing in commensurately small changes in the representation may be aesthetically desired[2]. To a certain extent, desired properties will depend on intended use, and the number of secondary structures that will be handled. Large-scale applications require high-throughput analysis, for example prediction of non-coding RNAs (ncRNA) *de novo* from next-generation sequencing platforms will benefit from efficiently executed structure alignment algorithms [52].

In building a DNA computer, a different high-throughput need exists. Design proceeds as a coupled search through both sequence and structure space. The long development times experienced by designers so far underline that this search has been done by hand. As part of speeding up the process through automation with a compiler, it is useful to consult the full suite of possible secondary structures predicted by a thermodynamic code. This type of modeling, termed suboptimal modeling, returns more than the single minimum free energy structure for each sequence. Instead, an entire range is considered within an energy gap above the minimum. To efficiently use this type of modeling, and be able to automate the search, we saw that the other approaches are not able to retain maximal information and support high-throughput scanning, parsings, and analysis. In many cases, a visual assessment must be done which precludes large-scale characterization and comparison questions, and impedes efficient database storage and retrieval. To address this shortcoming, a contribution of this thesis is the ISO numeric representation (Fanning [35]). A surprising result is that the numeric method allows for algorithmic probing of structures, which can then be easily automated in compiler code.

ISO notation describes nucleic acid secondary structure as a list of triples (***i****ndex*, ***s****tem*,

[2]Robert Giegerich, personal communication, 2011.

***o**pening)*, where each triple defines a distinct hybridization region within a single nucleic acid oligonucleotide, or between multiple oligonucleotides bound together as a complex.

Definition ISO. Let $P = \{p_0, p_1, \ldots\}$ be a set of n nucleotide strings, drawn from $\Sigma = \{A, C, G, T\}$, and let $d \notin \Sigma$ be a neutral spacer symbol. Form concatenated string c by ordering $5'$ to $3'$ all strings $p_i \in P$, separating each p_i by d such that $c = p_0 d p_1 d \ldots d p_{n-1}$. Let t be a list of m triples, $t = [(i,s,o)_0, (i,s,o)_1, \ldots, (i,s,o)_{m-1}]$. t is a unique representation of secondary structure features in c where for each feature:

1. i defines the (zero-based) indexing location relative to the p_0 $5'$ end.
2. s defines the length of binding stem.
3. o defines the opening enclosed by s, equal to the number of bases, paired or unpaired, which are intermediate between the last opening base and first closing base of the feature.

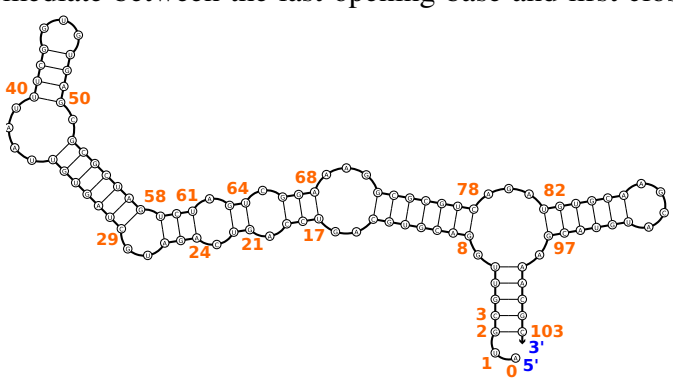


Figure 2.2: 105 base RNA segment with two stem-loops, five internal loops and one three-way multi-branch feature [42]. The ISO representation, starting from the marked $5'$ end, is $[(2,5,92),(8,7,57),(17,3,46),(21,2,40),(24,3,32),(29,7,16),(40,4,3),(82,6,4)]$.

Figure 2.2 exemplifies this notation for a natural RNA oligonucleotide. An equally useful variation of ISO dispenses with separating characters d within the sequence concatenation c, with referral to a vector of oligonucleotide lengths to distinguish one sequence from the next. We next show how use of ISO enables inference of hybridization features.

Nucleic acids may naturally, or as synthetically directed, prefer to fold into a variety of motifs including bulges, internal loops, hairpins, stem-loops and multibranches. ISO expresses all structure information exactly for each of these forms by virtue of relationships between the triples. Consider a structure with m triples, $[(i,s,o)_0,\ldots,(i,s,o)_{m-1}]$. A motif anchored as triple j, $j \in [0,m-1]$, is identified in the following ways. In each of these not only can we recognize existence of a specific motif, and locate it precisely, we can also infer the size exactly via further simple arithmetic over the triples defining the motif.

1. **Bulges.** A bulge is one or more unpaired bases on one side of two stem regions. A bulge is recognized within the list of triples where triples $(i,s,o)_j$ and $(i,s,o)_{j+1}$ satisfy either of the following, but not both:

   - $i_{j+1}-(i_j+s_j)>0$, $(i_j+s_j+o_j)-(i_{j+1}+2s_{j+1}+o_{j+1})=0$
   - $i_{j+1}-(i_j+s_j)=0$, $(i_j+s_j+o_j)-(i_{j+1}+2s_{j+1}+o_{j+1})>0$

   An example of a bulge is shown in Figure 2.2, in triples 4 = $(24,3,32)$ and 5 = $(29,7,16)$, since $i_5-(i_4+s_4)=29-(24+3)>0$ and $(i_4+s_4+o_4)-(i_5+2i_5+o_5)=(24+3+32)-(29+14+16)=0$.

2. **Internal loops.** An internal loop is formed by unpaired open regions surrounded by exactly two stems, where at least one unpaired base must occur on both sides. An internal loop is recognized within the list of triples where triples $(i,s,o)_j$ and $(i,s,o)_{j+1}$ satisfy:

   - $i_{j+1}-(i_j+s_j)>0$, $(i_j+s_j+o_j)-(i_{j+1}+2s_{j+1}+o_{j+1})>0$

   An internal loop example in Figure 2.2 is found between the stems represented by triples 1 = $(8,7,57)$ and 2 = $(17,3,46)$, since $i_5-(i_4+s_4)=17-(8+7)>0$ and $(i_4+s_4+o_4)-(i_5+2s_5+o_5)=(8+7+57)-(17+6+46)>0$.

3. **Hairpins.** Hairpins are terminal stems with no unpaired bases intervening. Hairpins are recognized as a triple with a size zero opening:

   - $(i,s,0)_j$

4. **Stem-loops.** Stem-loops are hairpins with at least one unpaired base between the opening and closing paired bases of the stem. Stem-loops are recognized as a triple:

   - $(i,s,>0)_j$

   An example stem-loop is seen in Figure 2.2 as the last triple $(82,6,4)$, where a binding stem of six base pairs surrounds four unpaired bases.

5. ***R*-way multibranches.** An *r*-way multibranch is an internal loop formed by *r* surrounding stems. An *r*-way multibranch is recognized within the list of triples where exactly $r-1$ triples $(i,s,o)_{j+1},\ldots,(i,s,o)_{j+r-2}$ are enclosed by triple $(i,s,o)_j$, but not enclosed by each other:

   - $\forall i_k, k \geq j+1, i_j+s_j < i_k < i_j+s_j+o_j, i_k+2s_k+o_k > i_{k-1}+2s_{k-1}+o_{k-1}$

   The first constraint tracks binding openings and stipulates that each triple defining a multibranch stem follows the initiating 5'-most stem, and is completely enclosed by the opening and closing bindings of this stem. The second constraint tracks binding closings and stipulates that subsequent stems *not* be enclosed by any previous defining one. For example, in Figure 2.2, triple 0 = $(2,5,92)$ initiates a 3-way branch, and triples 1-7 are enclosed by this triple. However, triples 2-6 are excluded since each of their extents is enclosed by triple 1 = $(8,7,57)$, and therefore they fail to satisfy the second constraint. Hence, the 3-way branch is represented by triples 0, 1, and 7, equal to sublist $[(2,5,92),(8,7,57),(82,6,4)]$. In the case of nested multibranches, this relation is extended further and requires use of an additional computation to in-

fer parent to child relationships for each embedded mulitbranch. Chapter 3 details the expanded algorithm for the generalized case.

6. **Pseudoknots.** A pseudoknot is a region of intercalated hybridizations such that successive binding stems do not occur serially, nor are nested. Triples in list $S = [\ldots,(i,s,o)_j,\ldots(i,s,o)_k,\ldots]$, $k \geq j+1$ where triples $(i,s,o)_j$ and $(i,s,o)_k$ satisfy the following qualify as a pseudoknot structure:

   - $i_k < i_j + s_j + o_j$, $i_k + s_k + o_k > i_j + 2s_j + o_j$

   The first constraint places the opening bases of triple $(i,s,o)_k$ within the unpaired open region of triple $(i,s,o)_j$. The second constraint places the corresponding closing bases of $(i,s,o)_k$ outside triple $(i,s,o)_j$, hence the triples are not nested. The intercalation may occur for successive triples or any two triples separated within the ISO list. Counting all such pairs of triples that satisfy this test yields the degree of knots. An example is shown in Figure 2.3, where the two triples representing the structure satisfy the pseudoknot test.

The most well-known secondary structure representation is the dot-parenthesis notation introduced in 1984 [59]. Thermodynamic design and modeling programs such as Vienna [58], Mfold [77], RNAsoft [7], and NUPACK [139], all use the basic dot-parenthesis notation as either output for predicted conformations, or input for determination of energy parameters associated with a desired conformation. This notation nominally uses a three-character alphabet $\{.,(,)\}$, where full stop ("dot") symbols indicate unpaired bases, and matching parentheses indicate paired bases. Strings with balanced parentheses describe structure patterns in which all hybridization regions are properly nested. The encoding is linear in the number of bases it abstracts, and thus is not a particularly compact representation. Location information for the various folding motifs can only be accomplished by overlaying a numeric index, and the notation is unable to handle pseudoknots without

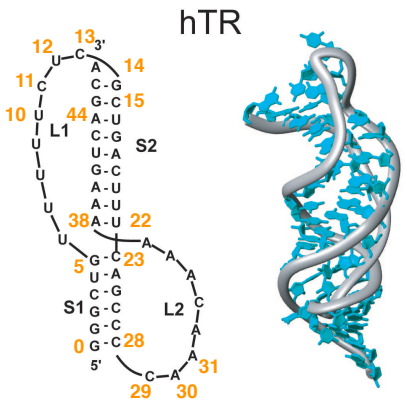


Figure 2.3: Human telomerase (hTR) pseudoknot structure located at the 5' end of the 451 base RNA [118]. The 49 base pseudoknot sequence segment is GGGCUGUUUUUCUCGCUGACUUUCAGCCCCAAACAAAAAAGUC which folds into an H-type motif with two stems and two loop regions. We describe this pseudoknot as $[(0,6,17),(14,9,14)]$.

resorting to additional characters. As an instance of a context-free language, a stack is required to parse dot-parenthesis strings, whereas ISO strings are in the class of regular languages, requiring no more than regular expressions for parsing. Another notation is the "RNA as graphs" (RAG) project [38]. RAG takes the approach of dropping all stem information and only tracks connections between folding shapes such as loops and bulges. Commentary by Leontis [70] has indicated at least one place where the RAG approach fails, and we note as well their difficulty handling pseudoknots. In general, no other representation that we know of can handle pseudoknots natively without some form of special handling, change, or extension.

ISO was motivated by different reasons than development of other representation schemes, yet the importance of secondary structure representation remains consistent. Cataloging and classifying secondary structure in terms of motif connections is an active research area [38, 64, 66, 87]. ISO keeps motif information, location, and size, therefore in addition to connection determination, distinction can be made for motif extents rather than abstracted

away. By virtue of using numbers to describe what is ultimately hydrogen bonding between molecular chains, we have greater ability to cast the understanding of nucleic acid structure space as a pattern recognition problem.

2.3.3 Abstracting Reactions

A molecular computer is implemented through arrangement of DNA oligonucleotide interactions as a chemical reaction network. The final abstraction is for oligonucleotide interactions $R' \in R$ that specifically implement platform and program execution out of R possible interactions, where R varies with the properties of all oligonucleotides in a system and with its chemical conditions. Rather than using stoichiometric formulas as shown in Appendix A.2, reactants and products are represented using a subset of their properties, termed *strands*, and interactions between strands are restricted to *fold*, *bind*, *unbind*, *ligate*, *exchange* or *cleave* operations that take one or more strands as input and create one or more strands as output, each termed *transitions*. Both strands and transitions carry a limited amount of information pertaining to the physical entity they represent. Multiple transitions are ordered as event streams to enable a mapping from build/execute logic to physical instantiation.

Definition Strand. A strand is a single oligonucleotide o, or complex of n oligonucleotides $o_0, o_1, \ldots, o_{n-1}$, characterized by its sequence s_{SEQ}, iso s_{STRC}, and length s_L, $s = (s_{SEQ}, s_{STRC}, s_L)$.

- s_{SEQ} is a sequence entity for oligonucleotide s or a concatenation of sequence entities $s_0, s_1 \ldots s_{n-1}$, $s_{SEQ} = s_0 - s_1 - \cdots - s_{n-1}$, where each s_i is the sequence entity for oligonucleotides o_i ordered $5'$ to $3'$ as part of a complex.

- s_{STRC} is the ISO entity, $s_{STRC} = [(i,s,o)_0,\ldots]$ for one or more oligonucleotides with sequence s_{SEQ} and length s_L. Note that ISO handles single oligonucleotides, or multiple oligonucleotides bound as a complex, hence there is no compositing scheme of individual structure strings required.

- $s_L, s_L \in N$ is the length of the single oligonucleotide s, or the total length of all n oligonucleotides o_0, o_1, …, o_{n-1} lengths, $s_L = l_0 + l_1, + \cdots + l_{n-1}, l_i \in N$ if s is a complex of n oligonucleotides.

Definition Transition. A reaction step happening in a vessel or well. Reactions may be one strand s_a taking on a new structural form s_b through an alternative hybridization pattern (*fold*), strand s_a and strand s_b becoming a complex s_c through hybridization (*bind*), complex s_c dissociating into strands s_a, s_b (*unbind*), strands s_a, s_b becoming a longer strand s_c through backbone linkage (*ligate*), a complex and a single oligonucleotide undergoing strand exchange (*exchange*), or a complex or single oligonucleotide s_a splitting along a backbone linkage into parts (*cleave*). Transitions are described through use of the fold, bind, ligate, exchange or cleave operators $t = (a, s_a, s_b, s_c, s_d, op)$, where inclusion of strand entities s_a, s_b, s_c, or s_d depends on choice of operator op, and a denotes the vessel or well addressable location.

- ***fold*** is an unary operation on a strand s_a to produce s_b with the same sequence and length, but with a different structure, $s_b = fold(s_a)$.

- ***bind*** is a binary operation on two strand entities s_a, s_b to produce a new complexed strand entity s_c, $s_c = bind(s_a, s_b)$.

- ***unbind*** is a unary operation on one strand s_c to produce two strand entities s_a and s_b, $s_a, s_b = unbind(s_c)$.

- ***ligate*** is a binary operation on two strand entities s_a and s_b to produce a new ligated strand s_c, $s_c = ligate(s_a, s_b)$.

- ***exchange*** is a binary operation on two strand entities s_a and s_b to swap single oligonucleotides and form s_c and s_d. The operator implicitly expects that at least s_a or s_b is a complex of at least two oligonucleotides and that at least one of s_c or s_d is a complex of at least two oligonucleotides, $s_c, s_d = exchange(s_a, s_b)$.

- ***cleave*** is an unary operation on one strand s_a to create two strand entities s_b, s_c, $s_b, s_c = cleave(s_a)$.

Transitions are inspired by Temkin [126] who was motivated to represent enzymatic catalysts in the presence of metal ions and employed graphs to explain reaction mechanisms. The use of graphs aided the subsequent formulation of reaction mechanisms to identify multiple routes, intermediate products, and elementary steps comprising the actions of enzyme catalysts. In DNA-based devices there are also reaction mechanisms, and to build a computing device and execute a program on it these mechanisms must occur in a fairly precise and ordered manner. Thus transitions define the implementation in behavioral terms as a series of critical reaction steps. These steps are atomic: no further division into smaller reaction substeps is defined.

Transitions, rather than species population counts, can also define system state because they can be clearly identified, named, and ordered. They are supported by computational

assessment tools, from a time and cost perspective, that provide a realistic degree of physical understanding. In contrast, a straight physics approach would dictate that state be defined as exact molecule counts based on known stoichiometrically balanced chemical equations, species concentrations, and reaction rates. Yet this treatment then insists on use of molecular dynamics simulations because of Boltzmann model limitations, and lack of comprehensive kinetics for arbitrary DNA or RNA oligonucleotide interactions. It is infeasible to engineer molecular computing platforms with this level of assessment because the cost and data collection time are prohibitive. Furthermore, there is a fundamental system characteristic that would be ignored: there is never just a *single* literal oligonucleotide molecule involved in any one transition. Reaction endpoints are entire populations where there is no capability to correctly infer an accurate census, and there is always a plethora of alternative species and reactions concurrently active. Because nucleic acids are promiscuous and constantly undergoing physical change, branching reaction mechanisms exist up until the point of equilibrium at which time even if individual strands are continuing to transition, the population statistics remain invariant. Systems such as an origami construction, say for the purpose of implementing a tiling, are equilibrium systems, but the ones we are interested in here implement a computation before reaching equilibrium. Instead, the computation is worked out on the way through direction of a subset of all possible reaction pathways. Every other pathway not in the subset, and every intermediate or final product not in the design set, is noise.

The unknowable populational dynamics as a construct admits some contrasts to electronic computing. Traditional computing systems and their abstractions rely on bit state, which is generally a known datum. Flipped bits representing error may be checked in hardware or software. But even if a bit is in the wrong state and goes undetected, the associated error within a program execution is a function of far fewer system states. Yet biological dynamics with vastly more system states and many more error states yield redundant, robust,

self-healing systems that in some measures out-perform all human engineered computing machines. Hence, the attendant uncertainty of working with biologically-based materials is a feature, not a flaw, even if we don't know how to fully exploit this inherent system-level property. Another direct contrast is that any constructed device or platform has no clock or program counter. There is ample parallelism, yet no central control. The implication is that while moving from one desired population state to another can be identified and defined as a requirement, the responsible transition occurs within an indeterminate number of time units. Physically some transitions may represent very quick transformations, such as input-loop binding between a gate and input pair, or slow ones, such as the one-way transition from intact substrate molecules to FRET tagged cleavage products. The transitions will additionally overlap in time, since each occupies a range and a succeeding transition step can start executing as soon as a preceding one has produced even a single molecule of the next required species. Because we are endeavoring to control the system by design principles, the tactic is to create dominant transition trajectories that will act to shift the mass of intended product creations in a particular ordering which will happen to end with the correct signaling state.

Rosen [101] from the context of computationally influenced theoretical biology introduced the concept of instantaneous state of a system as a *specification of its* ***structure*** *at an instant of time* [101]. In addition to defining state in terms of structure at a particular time, he also asked when can function be inferred from structure, and when can a designer create a desired function with a given set of structurally specified pieces? Structure is closely aligned with what something is, and function is associated with what something does. To create reliable computation, and scale it towards increasing performance, the effort is the inverse of the canonical biology "function from structure" problem. That is, *we use structure to direct function*.

In this vein, we can define system state using the sequence, structure, strand and transition abstractions. Together, they work directly with matching granularity modeling and simulation analytical tools. Figure 2.4 illustrates the use of structure to show ordered transitions as platform states. The depicted platform runs a one line logic program implemented using the reaction pathway shown in Table A.2.1.1. Three oligonucleotides are used to build the platform and run the program: a yes gate $G_{\overline{a}}$, an input I_a, and a substrate S. The preferential structures to attain this pathway are shown in each state (Tables A.12, A.15), for the active oligonucleotides present. Lack of structure is indicated as "[ ]" since for optimal binding to the gate, the input and substrate oligonucleotides should be structure-free and therefore available to bind on cue.

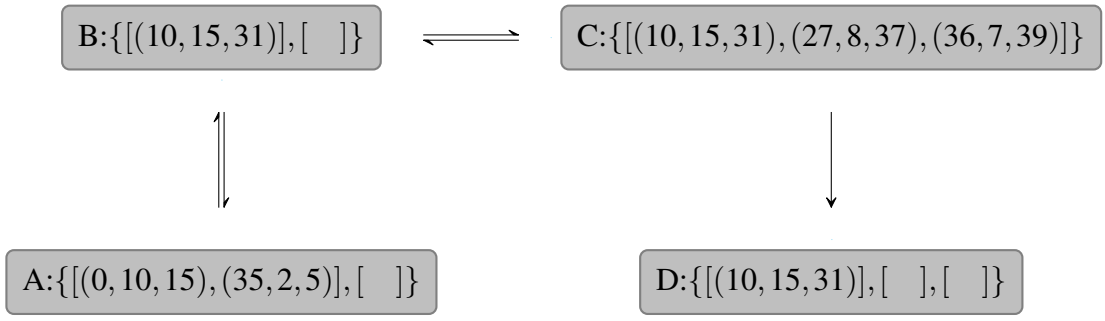


Figure 2.4: Deoxyribozyme transition steps for the YES gate, input, and substrate reactions, where the expected secondary structure is substituted in place of oligonucleotide identifier names for each state.

Chapter 3

Algorithms

The problems of chemistry and biology can be greatly helped if our ability to see what we are doing, and to do things on an atomic level, is ultimately developed–a development which I think cannot be avoided." – Richard P. Feynman [39]

The sequence and secondary structure abstractions facilitate algorithmic methods to solve various problems that are part of searching sequence and structure space for specfication of a nucleic acid based chemical reaction network. In this chapter, we cover these problems and a suite of algorithms all based on the sequence and structure abstractions that provide solutions. Each algorithm is encoded in the Pyxis compiler.

3.1 Sequence Algorithms

3.1.1 FindAllHybrids

The problem of finding all longest-possible oligonucleotide pair-binding regions, for an arbitrary number of oligonucleotides of varying lengths, is an implied need in building DNA computing systems but typically is not explicitly considered. Instead, secondary structure prediction programs such as MFold [142], Vienna [58] and NUPACK [139] solve an energy-based variant of this problem using dynamic programming algorithms originated by early researchers Nussinov [85] and Sankoff [143]. These algorithms energetically fit secondary structures using the Nearest Neighbor Model [104] to a Boltzmann distribution to return structures at and above the minimum free energy for a given sequence or sequences. Since any one secondary structure may contain more than one pair-binding region, prediction codes must not only identify bindings, but also formulate combinations of bindings. The time complexity of these algorithms, for a length n sequence, scales as $O(n^3)$ without pseudoknots [114].

The approach here finds all longest possible pair-binding regions, with a threshold of at least three base-pairs. The algorithm is used within the Kinetic Cellular Automaton (KCA) simulation to find possible hybridization reactions under the assumption that they will fire one at a time. Thus, the job of the algorithm is to return individual viable triples since other mechanisms within the simulation handle combinations. Formally, let P and Q be two sequence strings over $\Sigma = \{A, C, G, T\}$ with lengths $n, m > 2$, determine bindings set B where for each triple $(i, s, o)_b \in B$, index i_b addresses a unique length s_b pair-binding region, $2 < s_b \leq min(n, m)$, in concatenated string PQ, and for any two triples $(i, s, o)_j$ and $(i, s, o)_k$, $i_k > i_j + s_j$. For example, if P = Q = CATATG, then $B = \{(0, 6, 0)\}$.

With the example, we observe that sequence string P is composed of four sequence sub-

strings (0) CAT, (1) ATA, (2) TAT, and (3) ATG. Since we are interested in determining binding regions, we consider the reverse of sequence string Q composed of sequence substrings (0) TAC, (1) ATA, (2) TAT, and (3) GTA. Because the sequences are hypothetically part of two DNA strands that may encounter each other physically anywhere along their respective extents, checking for complementarity between the sequence substrings of P and the sequence substrings of Q reverse is done by checking all combinations. We can achieve looking at these combinations by conceiving of a P vs. Q reverse sequence substring table such that each cell i, j represents the complementarity check between sequence substring i of P and sequence substring j of Q reverse (Table 3.1). To determine the longest possible binding regions, each diagonal is individually examined. Contiguous checks occurring on the upper left to lower right diagonals correspond to overlapping complementary subsequences, hence counting these checked cells along the diagonals is used to compute binding lengths. Where the contiguity ends, the identified binding locations hold the longest possible one, hence through manipulation of P and Q indices a complete ISO triple can be formed and added to the overall list of identified binding regions. It is not necessary to construct the full table in memory since the orientation variations represented by the diagonals are independent. To see this, Table 3.2 shows the correspondence between cells along the diagonals and orientation variations that example strands P and Q may encounter assuming the binding length three threshold.

	(0) CAT ≡ 011000	(1) ATA ≡ 100010	(2) TAT ≡ 001000	(3) ATG ≡ 100011
(3) GTA ≡ 110010	√			
(2) TAT ≡ 001000		√		
(1) ATA ≡ 100010			√	
(0) TAC ≡ 001001				√

Table 3.1: Sequence strings P=Q=CATATG is Watson-Crick palindromic, thus the single maximal binding is found corresponding to placing the strands end-to-end.

diagonal	orientation								
0,0	G	T	A	C T	A A	T C	A	T	G
1,0 0,1	G	T	C A	A T	T A	A C	T	G	
2,0 1,1 0,2	G	C T	A A	T T	A A	T C	G		
3,0 2,1 1,2 0,3	C G	A T	T A	A T	T A	G C			
3,1 2,2 1,3	C	A G	T T	A A	T T	G A	C		
3,2 2,3	C	A	T G	A T	T A	G T	A	C	
3,3	C	A	T	A G	T T	G A	T	A	C

Table 3.2: Correspondence between table diagonals and relative orientations of strands P and Q that may yield binding regions.

Support Algorithms

The algorithm relies on support algorithms **EncodeSequence**, **SectionNucleotides**, **ReverseSequence**, **IsComp**, and **ExamineDiagonal**. Together, they work to arrange the details of creating the table diagonals, counting complements along the diagonals, and computing ISO triples.

EncodeSequence converts a length n sequence string drawn from the A-C-G-T alphabet into its corresponding DNARNABitStringEncoding binary format. The algorithm has $O(n)$ running time and space usage for a length n oligonucleotide.

Algorithm EncodeSequence

Input: `DNA sequence string d, sequence length n.`
Output: `Numeric encoded sequence string d'.`

```
\*  Key T: 00, C: 01, A: 10, G: 11  *\
1: d' ← 0
2: i ← 0
3: while i < n do
```

```
 4:    if d[i] == 'T' then
 5:      d' ← (d'≪2)^0b00
 6:    else if d[i] == 'C' then
 7:      d' ← (d'≪2)^0b01
 8:    else if d[i] == 'A' then
 9:      d' ← (d'≪2)^0b10
10:    else if d[i] == 'G' then
11:      d' ← (d'≪2)^0b11
12:    i ← i+1
13: return d'
```

SectionNucleotides slices a length n sequence string into $n-k+1$ length k sequence substrings using a right-to-left scan. For example, length 6 sequence string P=CATATG yields 6-3+1=4 length 3 subsequence strings CAT, ATA, TAT and ATG. The algorithm uses O(n) bits in running time and space to scan and formulate subsequence strings through use of the numeric representation format and bit operations.

Algorithm SectionNucleotides

Input: `DNA numeric encoded sequence string d, sequence length n, section length k.`
Output: `List of DNA numeric encoded length k sequence substrings kmers.`

```
 \*  Key T: 00, C: 01, A: 10, G: 11  *\
 1: kmers ← [  ]
 2: stamp ← 0
 3: i ← 0
 4: for i < k do
 5:   stamp ← (stamp≪2)^0b11
 6:   i ← i+1
 7: j ← 0
 8: while j < n-k+1 do
 9:   kmers ← append(kmers, d&stamp)
10:   d ← d≫2
11:   j ← j+1
```

```
12: return kmers
```

ReverseSequence reverses a length n sequence string using $O(n)$ bits running time and space.

Algorithm ReverseSequence

Input: DNA numeric encoded sequence string d, sequence length n.
Output: Reverse sequence string drev of d.

```
 \*  Key T: 00, C: 01, A: 10, G: 11  *\
 1: drev ← 0
 2: i ← n-1
 3: while i >= 0 do
 4:    drev ← drev^(d&0b11)≪2i
 5:    d ← d≫2
 6:    i ← i-1
 7: return drev
```

IsComp checks complementarity between length 3 sequence substrings (3-mers) using $O(1)$ bits running time and 18 bits of space.

Algorithm IsComp

Input: DNA numeric encoded length 3 sequence substrings d, d'.
Output: True or False boolean.

```
 \*  Key T: 00, C: 01, A: 10, G: 11                          *\
 \*  010101 selects alternate bits for 3-mer complementarity  *\
 \*  check.                                                   *\
 1: return !((d&0b010101)^(d'&0b010101))
```

ExamineDiagonal determines maximal bindings between two strands A and B by examining a list of binding index pairs. Each pair defines a length 3 complementarity region

between A and B. The first index a refers to the address of a participating sequence substring on A, and the second index b refers to the address of a participating sequence substring on B. The algorithm iterates a single time through all (a,b) pairs to build up binding stems where successive addresses a_i, a_{i+1} on A differ only by one. A break in addresses $(a_{i+1} - a_i > 1)$ indicates a new binding region. Manipulation of information stored in the index pairs yields complete ISO triples, with each ISO binding address i corresponding to the initiating binding location on the A sequence string for concatenated AB sequence string. The algorithm has O(m) running time and space usage where m counts the number of diagonal cells which at most are the length of the shortest input sequence string.

Algorithm ExamineDiagonal

Input: `List of index pairs (a,b) diag, representing table diagonal cells for A vs. B reverse sequence strings.`
Output: `List of triples (i,s,o) binds.`

```
 \*  Each (a,b) pair refers to a 3 nt complement between two  *\
 \*  sequences A,B where a is the subsequence address in A    *\
 \*  and b is the subsequence address in B, both relative to  *\
 \*  their respective 5' ends; (i,s,o) is longest possible    *\
 \*  binding at address i within AB.                          *\
 1: binds ← [  ]
 2: iter ← 0
 3: while iter < |diag| do
 4:   i ← diag[iter].a
 5:   s ← 3
 6:   lasta ← i
 7:   next ← iter+1
 8:   while next < |diag| do
 9:     if diag[next].a - lasta == 1 then
10:       s ← s+1
11:       lasta ← diag[next].a
12:       next ← next+1
13:     else
14:       break
15:   o ← |A|+diag[iter].b-2*s-i+3
```

```
16:    binds ← append(binds,(i,s,o))
17:    iter ← iter+s-2
18: return binds
```

Algorithm **FindAllHybrids** first starts with two sequence strings A and B, encodes them into bitstrings, and then splits the bitstrings into shorter ones representing all possible length 3 sequence substrings. This preparation step has $O(n)$ running time and space usage, driven by conversion from the A-C-G-T alphabet into bitstring encodings for length n sequence strings. The algorithm then creates a table of encoded A vs. B sequence substrings, one diagonal at a time. Diagonals $(0,0)$ through the main diagonal, $(n-1,0),(n-2,1),\ldots,(0,m-1)$ are formulated, and then diagonals $(1,m-1),\ldots,(n-1,1)$ through $(n-1,m-1)$ are formulated (Table 3.1). Processing on each diagonal proceeds by checking for bindability, and then creating ISO triples through review of the examined diagonal cells that showed complementarity. Complete binding regions may be as short as three base-pairs, corresponding to a single cell, or m base-pairs corresponding to the entire length m of the shortest oligonucleotide. A diagonal may also yield multiple triples since any one orientation of two strands can generate more than one binding region. Forming and processing information on the diagonals examines each cell once. If A has length n, and B has length m, then a total of nm operations are used leading to running time and space usage of $O(n^2)$. Use of bitstrings and bit-level manipulation limit resource usage in the inner processing iterations in lines 12 and 25 to $O(n)$ bits. Assuming a RAM model of computation with 64-bit words, running time and space increase to $O(n^3)$ at a threshold oligonucleotide strand length of 32.

Algorithm FindAllHybrids

Input: `Sequence strings A, B.`
Output: `List of longest possible hybridizations triples.`

```
\*  Address i, for each list triple, refers to concatenated  *\
\*  sequence string AB.                                       *\
triples ← [ ]
Anumeric ← EncodeSequence(A,|A|)
A3mers ← SectionNucleotides(Anumeric,|A|,3)
Bnumeric ← EncodeSequence(B,|B|)
B3mers ← SectionNucleotides(Bnumeric,|B|,3)
betaiter ← 0
while betaiter < |B|-2 do
  astart ← 0
  astop ← betaiter
  diaglist ← 0
  idx ← astart
  while idx <= astop do
    if IsComp(A3mers[idx],ReverseSequence(B3mers[astop-idx],3)) then
      diaglist ← append(diaglist,(idx,astop-idx)
    idx ← idx+1
  if |diaglist| > 0 then
    triples ← append(triples,ExamineDiagonal(diaglist))
  betaiter ← betaiter+1
betaiter ← 1
while betaiter < |B|-2 do
  astart ← betaiter
  astop ← |B|-3
  diaglist ← 0
  idx ← astart
  while idx <= astop do
    if IsComp(A3mers[idx],
    ReverseSequence(B3mers[|B|-idx-astart-3])) then
      diaglist ← append(diaglist,(idx,|B|-idx-astart-3)
    idx ← idx+1
  if |diaglist| > 0 then
    triples ← append(triples,ExamineDiagonal(diaglist))
  betaiter ← betaiter+1
return triples
```

3.1.2 Generate Separated DNA Oligonucleotides

For some applications, avoiding bond formation is imperative and instead what is required is a set of oligonucleotides that can be used together without undue cross-reactivity or self-folding, which could otherwise compromise their functionality. The problem is to determine base assignments such that minimal Watson-Crick base-pairs will form leading to very few or no double helical regions. Formally, the generate separated DNA oligonucleotides problem considers a count c, a length n, and an avoidance length k, and searches 4^n sequence space to find a satisfying set $S = \{seq_0, seq_1, \ldots, seq_{c-1}\}$ such that for all $\binom{c}{2}$ pairs (seq_i, seq_j), and all c^2 pairs (seq_i, seq_i), there are no binding regions with length greater than $k-1$.

Examples where it is important to avoid bond formation between co-located oligonucleotides include primer and aptamer design. DNA primers can be used to test the presence or absence of a particular DNA sequence, such as in a lateral flow device where a small amount of a biological sample is absorbed and allowed to chemically interact with carefully chosen sensor DNA oligonucleotides impregnated within the device. Aptamers are small DNA or RNA oligonucleotides that can bind to proteins and peptides through adoption of specific three dimensional shapes, and are being marketed as an alternative to antibodies for treatment of some disorders. In yet another application area, a designer of a synthetic biology system such as the DNA bricks as shown by Wei [132] will need to design many strands, some of them very long, that will allow Watson-Crick base-pairs to form at only a few particular locations and nowhere else. Building logic gates with deoxyribozymes entails solving the same problem in wild-card regions of the gates, and in selecting gate inputs.

The algorithm considers each sequence string as a vertex in an undirected graph $G = (V, E)$. An edge $e_{i,j} \in E$ indicates a binding region of length at least k between seq_i and

seq_j. Multiedges and self edges are allowed since any two sequence strings may have multiple independent binding regions or a sequence may have complementary regions within itself. The search for satisfying sequence strings starts with randomly generated strings and iteratively replaces the string with the highest edge count with a new random string until the edge set is empty. Accurate maintainance of the edge set during the search is aided by two dictionaries, one to hold the current working set of sequence strings, and the other to track length k substrings that compose the strings.

Support Algorithms

The algorithm relies on support algorithms **RandomNucleotides**, **SectionNucleotides**, **RevComp**, and **DecodeNT**. The support algorithms handle low-level tasks associated with creating and using bitstrings to represent DNA sequence strings.

RandomNucleotides generates bitstring encoded random sequence strings in $O(n)$ bits running time and space usage for a length n sequence string, where we assume a constant time uniform random generator function.

Algorithm RandomNucleotides

Input: `Input length n.`
Output: `A length n random DNA sequence encoded with key as shown.`

```
\*  Key T: 00, C: 01, A: 10, G: 11                              *\
\*  URNG performs uniform random selection from argument list.   *\
1: ntstring ← 0
2: i ← 0
3: for i < n do
4:    ntstring ← (ntstring≪2)^URNG(0b00,0b01,0b10,0b11)
5:    i ← i+1
6: return ntstring
```

RevComp employs bit-level manipulations to produce reverse-complemented DNA sequence strings in the DNARNABitStringEncoding binary format in O(1) running time and O(n) bit space usage for a length n sequence string.

Algorithm RevComp

Input: `DNA numeric encoded sequence d, sequence length n.`
Output: `Reverse complement DNA numeric encoded sequence d'.`

```
\*  Key T: 00, C: 01, A: 10, G: 11                              *\
\*  In place add 2 mod 4 complements each encoded nucleotide.   *\
d' ← 0
i ← 0
for i < n do
    d' ← (d'≪ 2)^(((d&0b11) + 2) % 4)
    d ← d≫2
    i ← i+1
return d'
```

DecodeNT converts DNA sequence strings from the DNARNABitstringEncoding binary format into A-C-G-T sequence string format in O(n) running time and space usage.

Algorithm DecodeNT

Input: `DNA numeric encoded sequence d, sequence length n.`
Output: `DNA A-C-G-T sequence string d'.`

```
key ← [T,C,A,G]
d' ← ""
i ← 0
while i < n do
    d' ← append(d', key(d&0b11))
    d ← d≫2
    i ← i+1
return d'
```

Algorithm Non-Intersecting Sequence Set proceeds through several steps. The algorithm maintains the current set of sequence strings in vector `workingAvoidanceSet`. A key-value dictionary `blackSubseq` is employed where keys are all avoidance length subsequence strings and values are the owner sequence strings identified by their `workingAvoidanceSet` indices. Each sequence string in `workingAvoidanceSet` is additionally represented as a graph vertex by use of a second key-value dictionary `edges` that tracks complementary subsequence string information between vertex pairs. Keys and value lists in `edges` are sequence strings also identified by their `workingAvoidanceSet` indices. Multiple edges between vertex pairs reflect multiple occurrences of complementary subsequence strings. Circular edges from and to the same vertex are also possible and reflect occurrences of complementary subsequence strings within a sequence string. The algorithm computes counts of complementary length k subsequence strings, hence the value lists in `edges` take the form of multisets with duplicate indices allowed.

Step 1 populates `workingAvoidanceSet` with c randomly generated length n sequence strings. Each of these sequence strings is then sliced into length k subsequence strings and dictionary `blackSubseq` is populated by using the subsequence strings as keys and the owner sequence strings indices as values. Since the same length k subsequence string may occur within a single sequence string, the `blackSubseq` value lists may contain duplicates.

Step 2 iterates through dictionary `blackSubseq` and for each key α, the reverse complement α' is computed and checked for existence within `blackSubseq`. If α' is found, α and α' within `blackSubseq` are Watson-Crick base-pairable, and therefore in conflict. This conflict is recorded within dictionary `edges`. First, all the sequence strings that have α as a subsequence string are found as the value list of key α within `blackSubseq`. The value list contains a list of indices, where each index serves as a name of the owner sequence string. The value list of key α' within `blackSubseq` similarly contains a list of indices, where each index names the owner sequence of α'. The α owner indices are iterated to

populate dictionary `edges` keys, and the α' owner indices are appended as `edges` value lists, for each separate key.

Step 3 uses data structures `workingAvoidanceSet`, `blackSubseq`, and `edges` to orchestrate the search for a satisfying set of sequence strings. A loop is executed to regenerate new sequence strings and check their cross-reactivity against existing ones until a zero length k conflict count amongst all sequence strings in `workingAvoidanceSet` is achieved. At the top of the loop, the sequence string with the highest number of conflict subsequence strings is tallied by examining the total number of conflict indices in the value lists for each vertex key in dictionary `edges`. The key with the maximum count, `maxowner`, is selected for replacement as a greedy decision. Prior to deletion and regeneration, the information associated with `maxowner` is excised from dictionaries `blackSubseq` and `edges`, and finally from within `workingAvoidanceSet`. Loop substeps are as follows.

Step 3a cleans up dictionary `blackSubseq`. First, all subsequence strings associated with `maxowner` are recomputed since they have been merged into `blackSubseq` and a separate data structure to hold them was not used in Step 1. Next, any duplicate subsequence strings from this set are removed to form set `killsubseqs` as the set of all unique subsequence strings found in the sequence string named by `maxowner`. Iteration through `killsubseqs` proceeds to examine each unique subsequence string and look up its associated owner list in dictionary `blackSubseq`, followed by a nested iteration through each owner list to look for and delete instances of `maxowner`. It may be the case that either no other sequence strings contained the subsequence string under examination other than `maxowner`, or that the list of owner sequence strings is reduced. As such, the key under review is deleted outright from `blackSubseq`, or the `blackSubseq` value list for the key is replaced with the reduced owner list.

Steb 3b cleans up dictionary `edges`. Prior to removing the `maxowner` key completely from

`edges`, references to `maxowner` must be removed from the value lists of the remaining keys. We recall that `edges` is tracking conflict sequence strings that contain Watson-Crick pairable subsequence strings. First, the value list of key `maxowner` in `edges` is copied into a temporary variable `killnodes` so that each of these vertices may have `maxowner` removed from their corresponding value lists. Iteration through `killnodes` accomplishes this goal by examination of each individual vertex and deletion of all `maxowner` instances from the associated value list. The cleaned-up lists of each key other than `maxowner` in `edges` then replace the existing value lists, and finally the entire `maxowner` key is reset within `edges` so that it can be reused in the next substep.

Step 3c revisits Steps 1 and 2 for a single sequence string. First, `workingAvoidanceSet` is updated by assigning a new randomly generated length n sequence string to the old `maxowner` index. Next, all the subsequence strings of the new sequence string are determined and searched for within dictionary `blackSubseq`. Any new subsequence strings are added as new keys, with a value list holding only the old `maxowner` index. If a subsequence string is found as already existing in `blackSubseq`, its value list is simply augmented to include the `maxowner` instance. Reverse-complements of all the new subsequence strings are computed and used to additionally update `blackSubseq` and also dictionary `edges`.

Step 4 is reached after iterating through Step 3 substeps until no more conflict subsequence strings are present in any of the sequence strings of `workingAvoidanceSet`. Once this status is achieved, all sequence strings are decoded back into A-C-G-T format to produce a satisfying set of sequence strings that possess no Watson-Crick subsequence complements larger than length $k-1$.

Time and resource usage is based on the expected number of iterations through the main loop. The space of all length n sequence string c-sets is 4^{nc}. There is always a satisfying solution, despite the exponentially sized space that is searched. Trivially, for any com-

bination of c, n, and k, sequence strings can consist of all "A" nucleotides, or all "C" nucleotides, or be composed from only "A" and "C" nucleotides. There are additional symmetric cases that similarly satisfy the problem statement. There are many more satisfying string subsets beyond the trivial cases as well that we identify as *safe sets* of the total solution space. Safe sets are composed from length k subsequence strings that are not pairable. For example, if $k = 2$ and $n = 3$, one possible safe set is:

$$\{TAA, AAA, AAC, ACA, CCC, ACC, CCA, CTC, TCA, GCA, CAC, CCT, TCT, TCC, CAA\}.$$

A variant safe set replaces element TAA with TAC. Further safe sets result from additional single element switch-outs, and from considering different subsets of the $4^3 = 64$ space of length 3 sequence strings. Safe sets are heavily intersected, since each n-mer string that is non-self complementary at and below the avoidance length k will belong to multiple safe sets.

The action of the algorithm is to find one of the safe sets over the course of execution, and return sequence strings from this set. It does so by virtue of the greedy selection step, which at each iteration replaces the sequence string with the highest number of conflicts. The remaining sequence strings each have fewer conflicts unless there is a tie for *maxowner*; in general the smaller conflict count means each remaining string is closer to the conflict-free ideal, and closer to each other in terms of which safe set they might be in. The *maxowner* will continually be reselected for replacement until its conflict count reduces to lower than that of the next highest neighbor, which serves to push it closer to the locus of the safe subsets, and eventually cluster all elements into only one safe set. The algorithm ultimately chooses the safe set through random selection, because the greedy strategy is only counting conflicts and not insisting on membership into any one safe set. This means the safe set will most likely change over the course of the algorithm until one starts to dominate and exert an attractive pull on the remaining n-mers.

Algorithm Non-Intersecting Sequence Set
Input: Count c, sequence length n, avoidance length k.
Output: A set of c length n strands with no Watson-Crick pairable subsequences greater than length k-1.

```
 \*  Key T: 00, C: 01, A: 10, G: 11                                  *\
 \*  Standard list append function used to build up vectors of       *\
 \*  indices; standard list duplicate removal function used to       *\
 \*  eliminate repeats; standard list sort function used used        *\
 \*  to numerically sort low to high index lists; standard list      *\
 \*  delete function used to remove entries.                         *\
 1: workingAvoidanceSet ← [  ]
 2: blackSubseq ← [  ]
 3: edges ← [  ]
    {Step 1.}
 4: i ← 0
 5: for i < c do
 6:   rndmseq ← RandomNucleotides(n)
 7:   workingAvoidanceSet[i] ← rndmseq
 8:   subseqs ← SectionNucleotides(rndmseq,n,k)
 9:   j ← 0
10:   for j < |subseqs| do
11:     elt ← subseqs[j]
12:     if blackSubseq[elt] then
13:       blackSubseq[elt] ← append(blackSubseq[elt], i)
14:     else
15:       blackSubseq[elt] ← [i]
16:     j ← j+1
17:   i ← i+1
    {Step 2.}
18: i ← 0
19: for i < |blackSubseq| do
20:   conflictSubseq ← RevComp(blackSubseq[i],k)
21:   if blackSubseq[conflictSubseq] then
22:     nodes ← blackSubseq[conflictSubseq]
23:     j ← 0
24:     for j < |nodes| do
25:       edges[j] ←
          append(edges[nodes[j]], blackSubseq[conflictSubseq])
```

```
26:         j ← j+1
27:     i ← i+1
    {Step 3.}
28: while true do
29:     maxcount ← 0
30:     i ← 0
31:     for i < |edges| do
32:       if |edges[i]| > maxcount then
33:         maxcount ← |edges[i]|
34:         maxowner ← i
35:     if maxcount < 1 then
36:       break
    {Step 3a.}
37:     maxsubs ← SectionNucleotides(workingAvoidanceSet[maxowner],n,k)
38:     killsubseqs ← removeDuplicates(maxsubs)
39:     i ← 0
40:     for i < |killsubseqs| do
41:       owners ← sort(blackSubseq[i])
42:       j ← 0
43:       while j < |owners| do
44:         if owners[j] > maxowner then
45:           break
46:         if owners[j] == maxowner then
47:           delete(owners[j])
48:         else
49:           j ← j+1
50:       if |owners| > 0 then
51:         blackSubseq[i] ← owners
52:       else
53:         delete(blackSubseq[i])
54:       i ← i+1
    {Step 3b.}
55:     killnodes ← edges[maxowner]
56:     i ← 0
57:     for i < |killnodes| do
58:       nodes ← sort(edges[i])
59:       j ← 0
60:       while j < |nodes| do
61:         if nodes[j] > maxowner then
```

```
62:           break
63:         if nodes[j] == maxowner then
64:           delete(nodes[j])
65:         else
66:           j ← j+1
67:       edges[i] ← nodes
68:       i ← i+1
69:     edges[maxowner] ← [  ]
      {Step 3c.}
70:     newrndmseq ← RandomNucleotides(n)
71:     workingAvoidanceSet[maxowner] ← newrndmseq
72:     newsubseqs ← SectionNucleotides(newrndmseq,n,k)
73:     newconflicts ← [  ]
74:     i ← 0
75:     for i < |newsubseqs| do
76:       if blackSubseq[i] then
77:         blackSubseq[i] ← append(blackSubseq[i], maxowner)
78:       else
79:         blackSubseq[i] ← [maxowner]
80:       newconflicts ←
          append(newconflicts, RevComp(blackSubseq[i],k))
81:       i ← i+1
82:     i ← 0
83:     for i < |newconflicts| do
84:       if blackSubseq[i] then
85:         newconflictNodes ← blackSubseq[i]
86:         edges[maxowner] ← append(edges[maxowner],newconflictNodes)
87:         j ← 0
88:         for j < |newconflictNodes| do
89:           edges[j] ←
              append(edges[newconflictNodes[j]],maxowner)
90:           j ← j+1
91:       i ← i+1
    {Step 4.}
92: mutualAvoidanceSet ← [  ]
93: i ← 0
94: for i < c do
95:   mutualAvoidanceSet[i] ← decodeNT(workingAvoidanceSet[i],n)
96:   i ← i+1
```

97: **return** `mutualAvoidanceSet`

Example algorithm output using inputs $c = 5$, $n = 100$, and $k = 5$ is as follows. The first 5-mer of Sequence 1 is ATAGT. The complement is TATCA, and the reverse complement is ACTAT, neither of which is found elsewhere within Sequence 1, nor within Sequences 2-5. This property is true for every single 5-mer within all five sequences, and as well for all subsequences with lengths greater than 5.

index	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
Seq 1	A	T	A	G	T	T	G	C	G	G	G	G	G	C	T	T	G	G	G	T
Seq 2	G	T	A	T	A	T	T	T	T	G	G	A	T	G	G	T	T	G	C	A
Seq 3	T	G	G	A	G	A	A	T	T	G	G	T	G	C	G	G	T	T	C	G
Seq 4	C	A	T	A	A	T	A	G	G	T	G	T	T	A	G	G	T	C	C	G
Seq 5	A	G	G	T	T	G	A	G	G	A	C	G	T	G	A	A	T	T	G	C
index	20	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35	36	37	38	39
Seq 1	A	G	C	G	T	C	T	G	T	T	G	A	T	G	T	A	T	A	G	C
Seq 2	T	A	G	G	T	T	C	T	G	T	A	G	T	T	T	C	G	G	G	T
Seq 3	G	C	T	A	A	G	T	T	G	G	G	C	A	G	G	G	G	T	G	G
Seq 4	T	A	T	T	G	T	G	G	G	T	T	G	G	A	C	T	C	T	C	G
Seq 5	T	C	T	G	T	C	C	C	A	G	G	T	G	C	T	T	G	T	G	T
index	40	41	42	43	44	45	46	47	48	49	50	51	52	53	54	55	56	57	58	59
Seq 1	G	T	T	G	A	T	T	G	A	A	G	C	G	G	C	T	T	G	G	A
Seq 2	G	G	C	G	T	G	C	G	T	C	G	C	A	T	A	G	G	T	C	A
Seq 3	T	T	C	A	T	T	T	T	C	G	T	C	T	A	A	G	G	A	T	A
Seq 4	C	A	G	C	G	T	T	T	A	T	T	T	C	C	T	A	C	T	G	A
Seq 5	G	C	C	G	T	A	T	G	G	G	G	G	T	G	G	G	T	G	A	G
index	60	61	62	63	64	65	66	67	68	69	70	71	72	73	74	75	76	77	78	79
Seq 1	T	A	G	T	G	G	G	T	T	A	G	G	T	T	G	G	A	G	T	G
Seq 2	T	A	A	C	G	G	T	G	G	G	G	A	A	T	C	T	A	C	T	T
Seq 3	G	T	C	T	G	G	A	G	C	G	A	A	G	G	T	C	A	T	G	G
Seq 4	T	A	G	C	G	T	G	G	G	G	T	C	C	G	A	G	G	G	G	G
Seq 5	G	G	C	T	C	T	T	G	G	T	C	T	G	C	T	C	G	T	A	T
index	80	81	82	83	84	85	86	87	88	89	90	91	92	93	94	95	96	97	98	99
Seq 1	G	G	C	A	G	G	T	G	C	A	T	G	T	T	T	A	A	T	T	T
Seq 2	C	T	G	G	T	C	A	A	G	T	G	T	T	C	T	G	T	A	G	C
Seq 3	C	C	C	G	G	C	T	G	A	G	T	T	T	G	T	G	T	A	G	C
Seq 4	T	T	T	A	G	T	A	T	G	T	T	T	G	C	A	G	C	G	G	A
Seq 5	T	T	A	A	G	T	C	G	T	G	A	G	T	T	G	T	A	C	T	C

Table 3.3: Five 100-mer DNA sequences with no reverse-complementary subsequences greater than length 4.

3.2 Structure Algorithms

3.2.1 ISO/Dot-Parenthesis Conversion

Due to the heavy use of dot-parenthesis in thermodynamic models, the conversion algorithms are for transforming back and forth between dot-parenthesis and ISO. First, to transform dot-parenthesis notation into ISO, a linear scan of the input string yields two lists of open and close parentheses. Repeating until list `stackclose` is empty, we examine list `stackopen` from the end forward to find the corresponding opening parenthesis. From this location, the largest stem-loop possible is built and saved as a complete ISO triple. Participating `stackopen` and `stackclose` elements are removed as each ISO triple completes.

To transform an ISO list into dot-parenthesis notation, we require additional length information input. We form a list of full stop (dot) characters to the input length, and replace each with either open or close parenthesis symbols based on examination of each ISO structure in the list. We then convert the list into a string as output.

Both algorithms have O(n) running time and space usage for a length n dot-parenthesis string.

Algorithm Dot-Parenthesis to ISO Conversion

Algorithm dp2iso
Input: `Dot-Parenthesis string oligo.`
Output: `List of ISO triples.`

```
\*  Standard list append function used to build up vectors of   *\
\*  indices; list delete function used to remove entries; sort  *\
\*  function used to numerically sort low to high index lists.  *\
1: open ← [  ]
```

```
 2: close ← [  ]
 3: triples ← [  ]
 4: idx ← 0
 5: while idx < len(oligo) do
 6:   if oligo[idx] ≡ "(" then
 7:     open ← append(open,idx)
 8:   if oligo[idx] ≡ ")" then
 9:     close ← append(close,idx)
10:   idx ← idx + 1
11:   if |open| < 1 then
12:     return triples
13: while |close| > 0 do
14:   anchorClose ← close[0]
15:   openIdx ← |open| - 1
16:   while anchorClose < open[openIdx] do
17:     openIdx ← openIdx - 1
18:   i ← open[openIdx]
19:   s ← 1
20:   o ← anchorClose - open[openIdx] - 1
21:   openDel ← [open[openIdx]]
22:   closeDel ← [close[0]]
23:   prevOpen ← open[openIdx]
24:   prevClose ← close[0]
25:   openIdx ← openIdx - 1
26:   closeIdx ← 1
27:   while openIdx > 1 ∧ closeIdx < |close| do
28:     if close[closeIdx] - prevClose ≡ 1 ∧
      prevOpen - open[openIdx] ≡ 1 then
29:       i ← i - 1
30:       s ← s + 1
31:       openDel ← append(openDel, open[openIdx])
32:       closeDel ← append(closeDel, close[closeIdx])
33:       prevOpen ← open[openIdx]
34:       prevClose ← close[closeIdx]
35:       openIdx ← openIdx - 1
36:       closeIdx ← closeIdx + 1
37:     else
38:       break
39:   triples ← append(triples,(i,s,o))
```

```
40:    for all elt ∈ openDel do
41:      open ← delete(open,elt)
42:    for all elt ∈ closeDel do
43:      close ← delete(close,elt)
44: return sort(triples)
```

Algorithm ISO to Dot-Parenthesis Conversion

Algorithm iso2dp
Input: List of triples, length n.
Output: Dot-Parenthesis string dpstring.

```
 \*  Standard list append function     *\
 \*  used to build up vector of dots.  *\
 1: dpstring ← [  ]
 2: i ← 0
 3: while i < n do
 4:   dpstring ← append(dpstring,".")
 5:   i ← i+1
 6: for all iso ∈ triples do
 7:   j ← 0
 8:   while j < iso[1] do
 9:     dpstring[iso[0]+j] ← "("
10:     dpstring[iso[0]+iso[1]+iso[2]+j] ← ")"
11:     j ← j+1
12: return dpstring
```

3.2.2 Shape Inference

Shapes, also termed motifs, are unbound contiguous single-stranded regions enclosed by one or more double helical regions. Shapes can be identified as bulges, stem-loops, internal loops, and as r-way multibranches. Naming reflects the induced forms as shown in Figure

2.2 (Page 42). In that example, there are eight double helical regions that start at addresses 2, 8, 17, 21, 24, 29, 40, and 82. There are two stem-loops, one defined by the stem at address 40, and one defined by the stem at address 82. There is one bulge defined by the two stems at addresses 24 and 29. Relative to the 5' end, there are three internal loops. The first internal loop is defined by the stems at addresses 8 and 17, the second internal loop is defined by the stems at addresses 17 and 21, and the third internal loop is defined by the stems at addreses 21 and 24. Lastly, there is one 3-way multibranch defined by stems at addresses 2, 8, and 82.

Alternative naming and shape classification variants are possible. In this work, a stem-loop means a single stem enclosing a single loop of unpaired bases. Some authors refer to stem-loops as hairpins, however here we reserve the term hairpin to mean a stem with no attached loop. Although this is not a typical natural form, in self-assembled systems there are often examples where the secondary structure can be described as a stem-loop with a size 0 loop. To distinguish this case, we refer to it as a hairpin. Bulges may be further described as being either upper or lower, where the additional terms respectively refer to pucker location as relative to either the 5' or the 3' end of the strand. As a general rule, naming helps to meaningfully differentiate small variations. Bulges and internal loops are indeed nearly the same thing. The key characteristic of a bulge is that the pucker of unpaired bases exists between either the opening bases of the two defining stems, or between the closing bases of the two defining stems, but not both. If the pucker does exist between both the opening and closing bases of the two defining stems, then it is identified as an internal loop.

The shape algorithms infer presence, location, and sizes of defining ISO triples. All instances meeting the shape criteria are returned as property vectors, thus empty vectors indicate absence of a shape within an input structure. Addressing for each inference algorithm is relative to the 5' end of the strand. In many cases, if a pseudoknot is present,

the algorithm will fail, or return partly incorrect results. The pseudoknot inference algorithm can be executed as a required preprocessing step, as needed, with knotted base pairs filtered out before checking for shapes and their properties. All algorithms in this group work for one strand, or multiple strands. If more than one strand is represented by the ISO input then reported shape addresses will change with each strand order permutation.

Algorithm Stem-Loop Inference

Each triple in an ISO list is examined. A check is made (Line 16) to determine reaching the last triple in the list, or if the next triple is completely downstream of the current one. This check ensures that only true stem-loops are reported by looking for the innermost triple within any nested triples that may be present. The algorithm has O(k) running time and space usage for a length k ISO. Example output for a more complicated structure, illustrated in Figure 3.5, is shown in Appendix B.1.

Algorithm Stem-Loop Filter
Input: `List of triples iso.`
Output: `For each unique stem-loop, defining triples list stemloopTriples, stem lengths list stemLengths, loop counts list loopCounts, and addresses list stemloopAddresses.`

```
 \*  Standard list append function used to build output lists.  *\
 1: stemloopTriples ← [  ]
 2: stemLengths ← [  ]
 3: loopCounts ← [  ]
 4: stemloopAddresses ← [  ]
 5: current ← 0
 6: while current < |iso| do
 7:   triple ← iso[current]
 8:   if current+1 < |iso| then
 9:     tripleNext ← iso[current+1]
10:     ijp1 ← tripleNext[0]
11:   else
```

```
12:     ijp1 ← -1
13:   ij ← triple[0]
14:   sj ← triple[1]
15:   oj ← triple[2]
16:   if ijp1 < 0 ∨ ijp1 > ij + 2*sj + oj - 1 then
17:     if oj > 0 then
18:       stemloopTriples ← append(stemloopTriples,triple)
19:       stemLengths ← append(stemLengths,sj)
20:       loopCounts ← append(loopCounts,oj)
21:       stemloopAddresses ← append(stemloopAddresses,ij)
22:   current ← current+1
23: return stemloopTriples,stemLengths,loopCounts,stemloopAddresses
```

Algorithm Hairpin Filter

This algorithm is almost identical to Stem-Loop Filter. A stem with no contained bases is seen in an ISO as a triple with an opening of 0. Although this shape is not that common in natural system structures, it is found in synthetic designs. A differentiating logic at the end of the algorithm checks to see if a hairpin is occurring as a binding using bases from the beginning and ending of the entire oligonucleotide(s) being structurally represented. With this special case there are no overhanging unpaired bases ("sticky ends"). The algorithm has O(k) running time and space usage for a length k ISO. Example output for the structure shown in Figure 3.5 is in Appendix B.1.

Algorithm Hairpin Filter
Input: `List of triples iso, length n.`
Output: `For each unique hairpin, defining triples list hairpinTriples, stem lengths list stemLengths, and addresses list hairpinAddresses.`

```
 \*  Standard list append function used to build output lists.  *\
 1: hairpinTriples ← [  ]
 2: stemLengths ← [  ]
```

```
 3: hairpinAddresses ← [  ]
 4: while current < |iso| do
 5:   triple ← iso[current]
 6:   if current+1 < |iso| then
 7:     tripleNext ← iso[current+1]
 8:     ijp1 ← tripleNext[0]
 9:   else
10:     ijp1 ← -1
11:   ij ← triple[0]
12:   sj ← triple[1]
13:   oj ← triple[2]
14:   if ijp1 < 0 ∨ ijp1 > ij + 2*sj + oj - 1 then
15:     if oj ≡ 0 then
16:       hairTriples ← append(hairpinTriples,triple)
17:       stemLengths ← append(stemLengths,sj)
18:       hairpinAddresses ← append(hairpinAddresses,ij)
19:   if ij ≡ 0 ∧ ij + 2*sj + oj ≡ length then
20:     hairTriples ← append(hairpinTriples,triple)
21:     stemLengths ← append(stemLengths,sj)
22:     hairpinAddresses ← append(hairpinAddresses,ij)
23:   current ← current+1
24: return hairpinTriples,stemLengths,hairpinAddresses
```

Algorithm Bulge Filter

Bulges are puckers of unpaired bases that are present only on one side of a helical region. Relative to the 5' end of one or more represented oligonucleotides, a one-sided pucker closest to the 5' end is an "upper" bulge while one closest to the 3' end is a "lower" bulge. Figures 3.1 and 3.2 show the distinction.

Each triple in an ISO list is examined. Relationships between triples are checked to determine if there is a bulge of unpaired bases between consecutive triples. Relativity to the 5' end of the first oligonucleotide represented structurally by its ISO defines whether a bulge

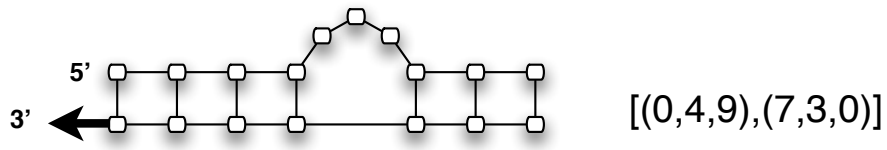


Figure 3.1: An example of an upper bulge, with unpaired bases on the 5'-most side between two double-helical regions.

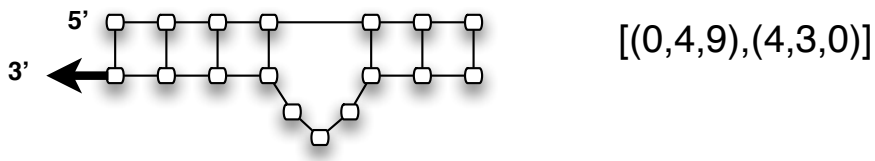


Figure 3.2: An example of an lower bulge, with unpaired bases on the 3'-most side between two double-helical regions.

is upper (5'-most) or lower (3'-most). Extra checks (Lines 24 and 30) ensure that there are no other stems emanating from the bulge region. If more than two stems surround a puckered out region of unpaired bases, the shape is instead defined as a multibranch. The algorithm has O(k) running time and space usage for a length k ISO.

Algorithm Bulge Filter
Input: `List of triples iso.`
Output: `For each unique bulge, defining triples list bulgeTriples, stem lengths list stemLengths, loop counts list loopCounts, and addresses list bulgeAddresses.`

```
\*  Standard list append function used to build output lists.  *\
1: bulgeTriples ← [  ]
2: stemLengths ← [  ]
3: loopCounts ← [  ]
4: bulgeAddresses ← [  ]
5: current ← 0
6: while current < |iso| - 1 do
7:   triple ← iso[current]
```

```
 8:   tripleNext ← iso[current+1]
 9:   ij ← triple[0]
10:   sj ← triple[1]
11:   oj ← triple[2]
12:   ijp1 ← tripleNext[0]
13:   sjp1 ← tripleNext[1]
14:   ojp1 ← tripleNext[2]
15:   if current + 2 < |iso| then
16:     tripleNextNext ← iso[current+2]
17:     ijp2 ← tripleNextNext[0]
18:   else
19:     ijp2 ← -1
20:   upper ← false
21:   lower ← false
22:   if ijp1 - (ij + sj) > 0 then
23:     if (ij + sj + oj) - (ijp1 - 2*sjp1 + ojp1) ≡ 0 then
24:       if (ijp2 < 0) ∨ (ijp2 > (ij + 2*sj + oj)) then
25:         upper ← true
26:         bulgeInit ← ij + sj - 1
27:         bulgeStop ← ijp1
28:   if ijp1 - (ij + sj) ≡ 0 then
29:     if (ij + sj + oj) - (ijp1 + 2*sjp1 + ojp1) > 0 then
30:       if (ijp2 < 0) ∨ (ijp2 > (ij + 2*sj + oj)) then
31:         lower ← true
32:         bulgeInit ← ijp1 + 2*sjp1 + ojp1 - 1
33:         bulgeStop ← ij + sj + oj
34:   if upper ∨ lower then
35:     bulgeTriples ← append(bulgeTriples,[triple,tripleNext])
36:     stemLengths ← append(stemLengths,[sj,sjp1])
37:     loopCounts ← append(loopCounts,bulgeStop - bulgeInit - 1)
38:     bulgeAddresses ← append(bulgeAddresses,[ij,ijp1])
39:   current ← current + 1
40: return bulgeTriples,stemLengths,loopCounts,bulgeAddresses
```

Algorithm Internal Loop Filter

The algorithm works very similarly to bulge inference, but differs in computing loop size and the check for a loop. As before, each triple in an ISO list is examined. Line 22 checks to ensure the unpaired bases between two stems are not part of a multibranch. Where there are unpaired bases on both the 5'-most ("upper") and 3'-most ("lower") regions between two consecutive stems, the shape is an internal loop. The algorithm has $O(k)$ running time and space usage for a length k ISO.

Algorithm Internal Loop Filter
Input: `List of triples iso.`
Output: `For each unique internal loop, defining triples list internalloopTriples, stem lengths list stemLengths, loop counts list loopCounts, and addresses list internalloopAddresses.`

```
 \*  Standard list append function used to build output lists.  *\
 1: internalloopTriples ← [  ]
 2: stemLengths ← [  ]
 3: loopCounts ← [  ]
 4: internalloopAddresses ← [  ]
 5: current ← 0
 6: while current < |iso| - 1 do
 7:   triple ← iso[current]
 8:   tripleNext ← iso[current+1]
 9:   ij ← triple[0]
10:   sj ← triple[1]
11:   oj ← triple[2]
12:   ijp1 ← tripleNext[0]
13:   sjp1 ← tripleNext[1]
14:   ojp1 ← tripleNext[2]
15:   if current + 2 < |iso| then
16:     tripleNextNext ← iso[current+2]
17:     ijp2 ← tripleNextNext[0]
18:   else
19:     ijp2 ← -1
20:   if ijp1 - (ij + sj) > 0 then
```

```
21:     if ((ij + sj + oj) - (ijp1 + 2*sjp1 + ojp1)) > 0 then
22:       if (ijp2 < 0) ∨ (ijp2 > ij + 2*sj + oj) then
23:         internlloopTriples ←
            append(internalloopTriples,[triple,tripleNext])
24:         stemLengths ← append(stemLengths,[sj,sjp1])
25:         internalloopAddresses ←
            append(internalloopAddresses,[ij,ijp1])
26:         upperInit ← ij + sj - 1
27:         upperStop ← ijp1
28:         upperLoopcount ← upperStop - upperInit - 1
29:         lowerInit ← ijp1 + 2*sjp1 + ojp1 - 1
30:         lowerStop ← ij + sj + oj
31:         lowerLoopcount ← lowerStop - lowerInit - 1
32:         loopCounts ←
            append(loopCounts,upperLoopcount + lowerLoopcount)
33:   current ← current + 1
34: return internaloopTriples,stemLengths,loopCounts,
    internalloopAddresses
```

Algorithm Parent-Child Determination

This algorithm is directly required by **Algorithm Multibranch Inference**. Multibranches are internal loops with more than two stems emanating from the loop region. There is no limit on the number of stems. Because the ISO triple list naturally orders triples depth-first, we can start at the end of the list and note that for each triple, if it has a parent, the parent is the tighest numerically enclosing triple that precedes it in the list. Each triple can have more than one child, but can only have a single parent. The algorithm has O(k^2) running time and space usage for a length k ISO and uses precise binding pair locations as shown in Figure 3.3.

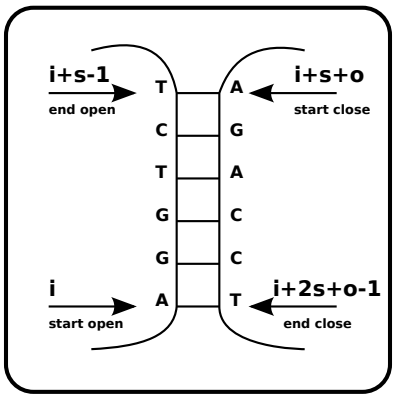


Figure 3.3: Index naming convention used by Parent-Child.

Algorithm Parent-Child
Input: List of triples iso, total length n of represented oligonucleotides.
Output: A list parent, showing for each triple a sublist of other directly adjacent and contained triples, and a list children, showing the containing parent for each triple.

```
\*  Standard list append function used to build output lists.  *\
 1: startOpens ← [  ]
 2: endOpens ← [  ]
 3: startCloses ← [  ]
 4: endCloses ← [  ]
 5: children ← [  ]
 6: i ← 0
 7: for i < |iso| do
 8:   children ← append(children,[])
 9:   i ← i+1
10: current ← 0
11: while current < |iso| do
12:   triple ← iso[current]
13:   startOpens ← append(startOpens,triple[0])
14:   endOpens ← append(endOpens,triple[0] + triple[1] - 1)
15:   startCloses ←
      append(startCloses,triple[0] + triple[1] + triple[2])
16:   endCloses ←
      append(endCloses,triple[0] + 2*triple[1] + triple[2] - 1)
17:   current ← current + 1
18: i ← 0
19: for i < |iso| do
20:   parent ← i
21: if |iso| > 0 then
22:   parent[0] ← -1
23: i ← |iso|-1
24: for i > 0 do
25:   minimumStartDelta ← n
26:   minimumEndDelta ← n
27:   upstream ← i-1
28:   for upstream > -1 do
29:     startDelta ← startOpens[i] - endOpens[upstream]
```

```
30:      endDelta ← startCloses[upstream] - endCloses[i]
31:      if startDelta > 0 ∧ endDelta > 0 then
32:        if startDelta < minimumStartDelta ∧
           endDelta < minimumEndDelta then
33:          minimumStartDelta ← startDelta
34:          minimumEndDelta ← endDelta
35:          parent[i] ← upstream
36:      upstream ← upstream-1
37:    if parent[i] ≡ i then
38:      parent[i] ← -1
39:    i ← i-1
40: i ← 0
41: for i < |iso| do
42:    children[parent[i]] ← append(children[parent[i]],i)
43:    i ← i+1
44: return parent,children
```

Algorithm R-Way Multibranch Inference

With the use of algorithm **Parent-Child**, the inference of multibranches is almost immediate. Recall that each triple represents a separate binding stem, and all triples are ordered based on their address location relative to the 5' end of the first (or only) oligonucleotide structurally described by its ISO. Given a triple α, any other triples β, γ, ... that occur in the ISO list after α by definition have larger-valued addresses for their initiating locations. The entire extent of a binding feature in ISO terms is the total mapped footprint which includes the starting bases, any unpaired bases, and the closing bases. When we combine the initiating location addresses (i), along with the footprints ($i + 2s + o$), it is straightforward to infer presence of a multibranch. An occurrence of three or more stems where one is a left-most triple in the ISO list, and the remaining have initiating addresses within the footprint of the left-most triple, can either be a junction or a multibranch. If there are

unpaired bases between the stems, as in a bulge or internal loop, then the feature qualifies as a multibranch. If there are no open bases between the stems, then the feature qualifies as a junction. Figure 3.4 (structure with a junction only) and Figure 3.5 (structure with multibranches) show this distinction.

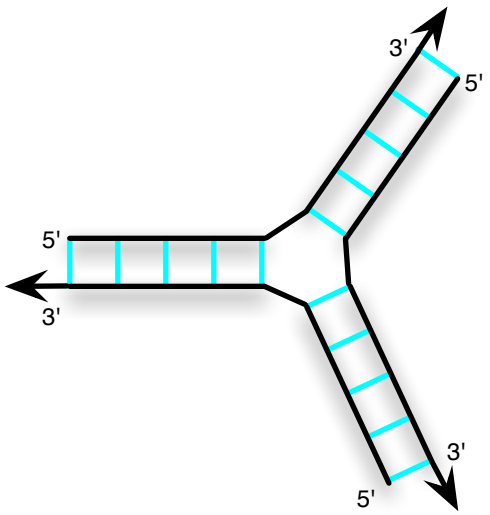
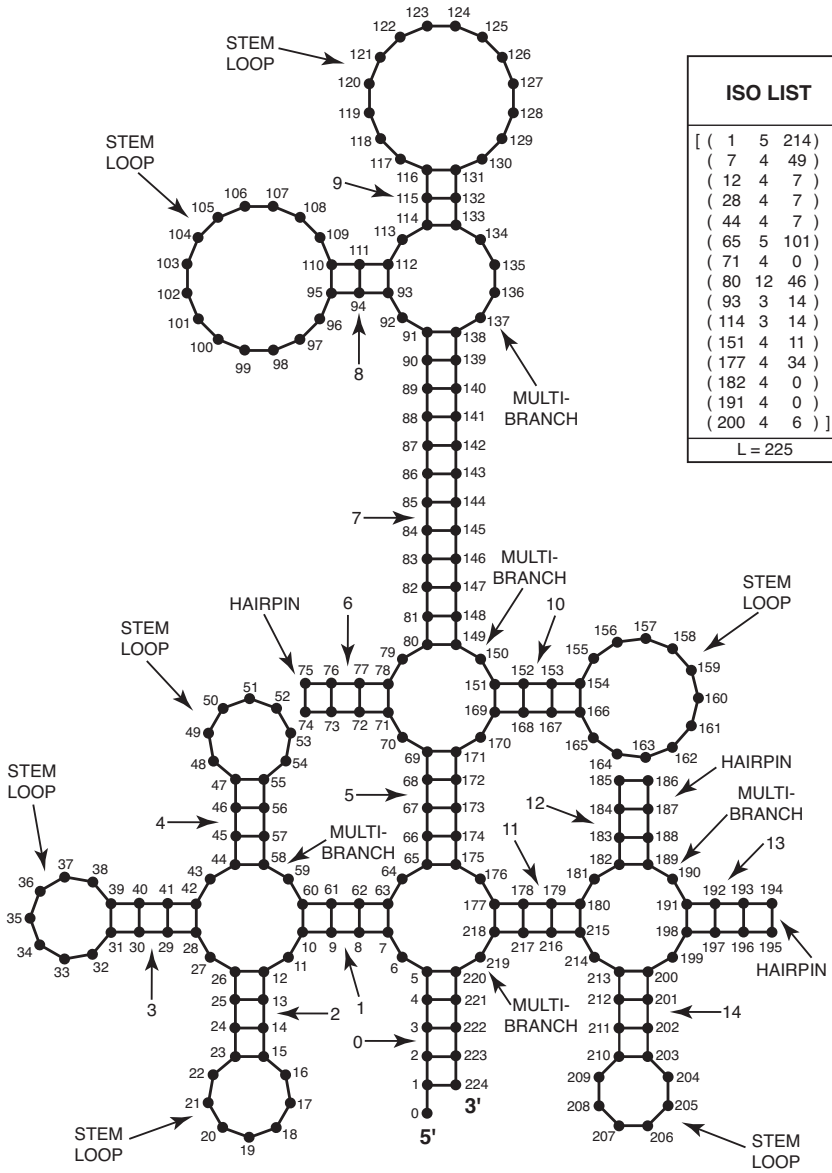


Figure 3.4: Synthetically designed junction structure. The structure is perfectly symmetric thus any ordering of oligonucleotides yields ISO [(0,5,20),(5,5,0),(10,5,0)].



ISO LIST	TRIPLE #
[(1 5 214)	0
(7 4 49)	1
(12 4 7)	2
(28 4 7)	3
(44 4 7)	4
(65 5 101)	5
(71 4 0)	6
(80 12 46)	7
(93 3 14)	8
(114 3 14)	9
(151 4 11)	10
(177 4 34)	11
(182 4 0)	12
(191 4 0)	13
(200 4 6)]	14
L = 225	

Figure 3.5: Synthetically designed structure with nested multibranches, multiple stem-loops, and hairpins.

Since **Parent-Child** determines the relationships between triples, we can read off the child information of each triple in the list to determine if some triple subset represents a multibranch. A leftmost triple (a parent) with at least two fully contained triples (children), and at least one unpaired base within the parent footprint, must exist. Therefore, algorithm **Multibranch Filter** proceeds exactly this way. First, each triple is examined, along with its children. Since the left-most parent triple counts as one of the branches, the algorithm checks to see if there is more than one child to make up the additional branches. Next, footprint information from the parent and children are combined to determine if there are unpaired bases. The running time and space usage of the algorithm is driven by the requirement to call **Parent-Child** for relationship information, and thus is $O(k^2)$ in time and space usage for a length k ISO. Example output for the structure shown in Figure 3.5 is in Appendix B.1.

Algorithm Multibranch Filter
Input: `List of triples iso, total length n of represented oligonucleotides.`
Output: `For each unique multibranch, defining triples list multibranchTriples, stem lengths list stemLengths, loop counts list loopCounts, and addresses list multibranchAddresses.`

```
 \*  Standard list append function used to build output lists.  *\
 1: multibranchTriples ← [  ]
 2: stemLengths ← [  ]
 3: loopCounts ← [  ]
 4: multibranchAddresses ← [  ]
 5: parents,children ← Parent-Child(iso,n)
 6: index ← 0
 7: for index < |iso| do
 8:   if |children[index]| > 1 then
 9:     triples ← [iso[index]]
10:     stems ← [iso[index][1]]
11:     addresses ← [iso[index][0]]
12:     daughter ← 0
13:     for daughter < |children[index]| do
```

```
14:       triples ← append(triples,iso[daughter])
15:       stems ← append(stems,iso[daughter][1])
16:       addresses ← append(addresses,iso[daughter][0])
17:       daughter ← daughter+1
18:     encloserOpening ← triples[0][2]
19:     childFootprints ← 0
20:     child ← 1
21:     for child < |triples| do
22:       childFootprints ←
          childFootprints + 2*triples[child][1] + triples[child][2]
23:       child ← child+1
24:     unpaired ← encloserOpening - childFootprints
25:     if unpaired > 0 then
26:       multibranchTriples ← append(multibranchTriples,triples)
27:       stemLengths ← append(stemLengths,stems)
28:       multibranchAddresses ←
          append(multibranchAddresses,addresses)
29:       loopCounts ← append(loopCounts,unpaired)
30: return multibranchTriples,stemLengths,loopCounts,
    multibranchAddresses
```

Algorithm Pseudoknot Filter

Pseudoknots are non-symmetric. Their bindings are not nested nor linearly separated, and instead cross over each other. Natural RNA folds into pseudoknots [118] as a result of additional stacking plane hydrogen bond opportunities, which yield stability benefits despite the asymmetric and jumbled appearance. RNA has evolved to use pseudoknots for a variety of cellular functions including self-cleavage of ribozymes, frameshifting the coding regions for viruses during translation, processing activity of telomerases, and autoregulation of viral gene expression [17].

The representation of pseudoknots is problematic. Dot-parenthesis notation requires the addition of new symbols to distinguish between pairing regions, typically square brackets

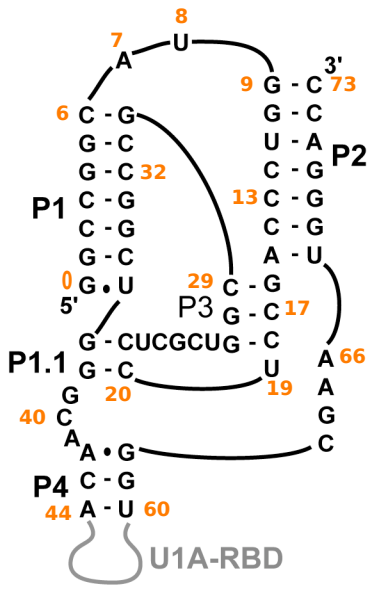


Figure 3.6: Hepatitis delta virus (HDV) ribozyme secondary structure, the fastest natural self-cleaving ribozyme with a cleavage rate greater than 1 per second [118, 127]. The structure is [(0,7,23),(9,7,51),(16,3,8),(20,2,15),(42,3,15)].

([,]) or curly braces ({,}). As an example, **((.....))**, and **()..()** are properly nested, while **(([[..))]]** is pseudoknotted. Each new intercalated binding region requires a new matching symbol. Suggestions have also been made to show each distinct intercalated binding region with a different color[1]. To demonstrate these difficulties, the sequence and full dot-parenthesis representation of the pseudoknot in Figure 3.6 is shown in Table 3.4 where four symbols must be used to cover the recursively knotted structure. Pseudoknots as well cannot be represented by rooted trees, nor planar graphs. With ISO, pseudoknots are handled by list inspection since the numeric representation makes clear start and stop regions for each binding. If a new binding initiates before an old one completes, there is a pseudoknot. Checking for pseudoknots in this example shows the benefit of ISO as a regular language. Despite the complicated form adopted by HDV, its structure in ISO only requires parsing

[1]Luca Cardelli, personal communication, 2011.

index	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
base	**G**	**G**	**C**	**C**	**G**	**G**	**C**	**A**	**U**	**G**	**G**	**U**	**C**	**C**	**C**	**A**	**G**	**C**	**C**	**U**
symbol	(	(	(	(	(	(	(	.	.	[	[	[	[	[	[	[	{	{	{	.
index	20	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35	36	37	38	39
base	**C**	**C**	**U**	**C**	**G**	**C**	**U**	**G**	**G**	**C**	**G**	**C**	**C**	**G**	**G**	**C**	**U**	**G**	**G**	**G**
symbol	<	<	.	.	.	.	.	}	}	}	)	)	)	)	)	)	)	>	>	.
index	40	41	42	43	44	45	46	47	48	49	50	51	52	53	54	55	56	57	58	59
base	**C**	**A**	**A**	**C**	**A**	**X**	**X**	**X**	**X**	**X**	**X**	**X**	**X**	**X**	**X**	**X**	**X**	**X**	**X**	**X**
symbol	.	.	(	(	(	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.
index	60	61	62	63	64	65	66	67	68	69	70	71	72	73	74	75	76	77	78	79
base	**U**	**G**	**G**	**C**	**G**	**A**	**A**	**U**	**G**	**G**	**G**	**A**	**C**	**C**						
symbol	)	)	)	.	.	.	.	]	]	]	]	]	]	]						

Table 3.4: HDV sequence and structural representation using dot-parenthesis, the bases corresponding to the U1A-RBD domain are not shown.

by a regular expression, whereas use of dot-parenthesis requires four separate stacks to track the intercalations, and indication of which symbols are in use.

The algorithm to deduce pseudoknots works by first forming all $\binom{k}{2}$ pair combinations of k triples in an ISO list, starting from the head of the list. Each pair (α, β) is examined to determine if the address β_i occurs before the α start close location $\alpha_i + \alpha_s + \alpha_o$ and the β start close location $\beta_i + \beta_s + \beta_o$ occurs after the α end close location $\alpha_i + 2\alpha_s + \alpha_o - 1$ (Figure 3.3). Any pair achieving these conditions indicates intercalation between the α and β binding regions. The algorithm has $O(k^2)$ running time and space usage for k ISO triples, driven by forming all triple pairs. For the HDV structure (Figure 3.6) there are three pseudoknots reported by the algorithm as (1) [(0, 7, 23), (9, 7, 51)], (2) [(0, 7, 23), (20, 2, 15)], and (3) [(16, 3, 8), (20, 2, 15)].

Algorithm Pseudoknot Filter
Input: `List of triples iso.`
Output: `The defining triples for each pseudoknot, knots.`

```
\*  Standard list append function used to  *\
\*  build lists.                           *\
1: subsets ← [  ]
2: i ← 0
3: for i < |iso|-1 do
4:    j ← i+1
5:    for j < |iso| do
```

```
 6:     subsets ← append(subsets,[iso[i],iso[j]])
 7:     j ← j+1
 8:   i ← i+1
 9: knots ← [  ]
10: triples ← 0
11: for triples < |subsets|-1 do
12:   ij ← subsets[triples][0][0]
13:   sj ← subsets[triples][0][1]
14:   oj ← subsets[triples][0][2]
15:   ik ← subsets[triples][1][0]
16:   sk ← subsets[triples][1][1]
17:   ok ← subsets[triples][1][2]
18:   if ik < (ij + sj + oj) then
19:     if (ik + sk + ok) >= (ij + 2*sj + oj) then
20:       knots ← append(knots,triples)
21:   triples ← triples+1
22: return knots
```

3.3 Binding Characterization

Algorithm To Determine All Bound Regions

The algorithm iteratively scans the triples in an ISO string, examines each triple to extract the bases involved in the represented binding regions, stores these bases into an unsorted list, and ends with a sort to return the bound bases in order. Each triple demarcates the location of a bound region, and each bound region is constituted by a continuous run of base-pairs. For each triple, opening bases initiate at address i, and are continuous through address $i+s$, and closing bases initiate at address $i+s+o$, and are continuous through address $i+2s+o$. The algorithm returns a list of base addresses, where each address is involved in a base-pair.

At best, no bindings are present and an empty list is returned in constant time. At worst, there are two maximal binding cases to consider for a length n oligonucleotide. Case 1 is $n/2$ base pairs arranged one after the other represented by $n/2$ triples each with a stem length of 1, hence requiring $n/2$ constant time opening and closing address range computations. Case 2 is one large binding represented by one triple with a $n/2$ stem length, requiring single $n/2$ time opening and closing range computations. The algorithm bottleneck though is the final sort that correctly orders results for inputs with nested binding regions, therefore algorithm running time and space usage is O($n\ log\ n$). Example output for the structure shown in Figure 3.5 is in Appendix B.1.

Algorithm Bindings
Input: `List of triples iso.`
Output: `List of bound indices bindings.`

```
 \*  Standard list append function used to build up vectors   *\
 \*  of indices; list sort function used to numerically sort  *\
 \*  low to high index lists.                                 *\
 1: bindings ← [  ]
 2: if |iso| > 0 then
 3:   for all triple ∈ iso do
 4:     i ← triple[0]
 5:     s ← triple[1]
 6:     o ← triple[2]
 7:     openrange ← [i,...,i+s]
 8:     closerange ← [i+s+o,...,i+2*s+o]
 9:     bindings ← append(bindings,openrange)
10:     bindings ← append(bindings,closerange)
11: return sort(bindings)
```

Algorithm To Determine All Unbound Regions

The algorithm finds all bases not implicated in a binding region. The algorithm depends on algorithm **Bindings** and performs an index by index comparison with the resulting `BoundList` and the full index range $[0, n-1]$ for a length n oligonucleotide to aggregate the indices that are unbound. An index guard `BoundCountGuard` is employed to jump out of checking the `Boundlist` after its last entry is handled. The remaining bases through index $n-1$, if any, are filled in to complete the list. Since the bound region list is sorted, and indices are iterated in order, the final list of all unbound, open bases is also sorted. The algorithm uses $O(n)$ time and space to visit each index $[0, n-1]$ for a length n oligonucleotide. Example output for the structure shown in Figure 3.5 is in Appendix B.1.

Algorithm Unbound
Input: `List of triples iso, length n.`
Output: `List of unbound base indices.`

```
 \*  Standard list append function used to build up vectors   *\
 \*  of indices.                                               *\
 1: BoundList ← bindings(iso,length)
 2: if |BoundList| ≡ 0 then
 3:   NotBoundList ← [0,...,n-1]
 4:   return NotBoundList
 5: BoundCountGuard ← |BoundList|-1
 6: NotBoundList ← [ ]
 7: idx ← 0
 8: bdx ← 0
 9: while idx < n do
10:   if idx ≡ BoundList(bdx) then
11:     if bdx < BoundCountGuard then
12:       bdx ← bdx+1
13:     else
14:       break
15:   else
16:     NotBoundList ← append(NotBoundList,idx)
17:   idx ← idx+1
```

```
18: RemainingUnbound ← [BoundList(bdx)+1,...,n-1]
19: NotBoundList ← append(NotBoundList,RemainingUnbound)
20: return NotBoundList
```

Algorithm To Determine All Binding Partners

This algorithm performs similarly to algorithm **Bindings**, but instead returns a list of base-pairs for each binding region. As before, the algorithm iteratively scans the triples in an ISO string, and examines each triple to extract the bases involved in the represented binding regions. The matching opening a and closing b base addresses are saved into a list as base-pair (a,b), for each individual base-pair constituted within s, starting at address i, for each triple. In this case rather than working with opening and closing address ranges for each bound region, the algorithm determines the exactly matching opening and closing base addresses. Base-pairs are returned in sorted order using the opening bases as a sort key. The algorithm has $O(n)$ running time and space usage for a length n oligonucleotide. Example output for the structure shown in Figure 3.5 is in Appendix B.1.

Algorithm Partners
Input: `List of triples iso.`
Output: `List of base-pairs.`

```
 \*  Standard list append function   *\
 \*  used to build up vectors of     *\
 \*  indices.                        *\
 1: partners ← [ ]
 2: for all triple ∈ iso do
 3:   i ← triple[0]
 4:   s ← triple[1]
 5:   o ← triple[2]
 6:   index ← 0
 7:   while s > 0 do
```

```
 8:     basepair ← (index+i, index+i+2*s+o-1)
 9:     partners ← append(partners,basepair)
10:     i ← i+1
11:     s ← s-1
12: return partners
```

3.3.1 Exhaustive Generation Method For All Possible Structures

An interesting research problem is how many secondary structures are possible for a given length, where length can represent a single oligonucleotide, or be summed from multiple oligonucleotides concatenated together. If there is more than one oligonucleotide, the ISO of the secondary structure changes based on the ordering since base indices reflect ordering, consistent with all methods of representing structure. Structure enumeration can be posed regardless of oligonucleotide number however, as many oligonucleotides concatenated together are equivalent to a single one with the same sequence. Enumeration of the total number of bonding patterns without pseudoknots was shown by Waterman and Smith [130]. Enumeration of structures with a particular class of pseudoknots using arc-crossing diagrams is found in Jin and Reidys [62].

Use of ISO triples for enumeration and generation.

With ISO, a different approach is possible for the structure enumeration problem that yields a straightforward construction algorithm. The key to this approach is that each triple in an ISO list is complete and self-contained, therefore it can act as a building block. Triples identify points in 3D space and are subject to several constraints. For a length n, the address i must be in $[0, n-2]$, the stem-length s must be in $[1, \lfloor n/2 \rfloor]$, and the opening o must be in $[1, n-2]$, subject to fitting the $i+2s+o$ footprint within $[0, n-1]$.

As an example, consider an oligonucleotide with six bases. We can enumerate a list of satisfying triples in a series of tableaux as shown in Table 3.5 and graphically in Figure 3.7. The numeric symmetries for each dimension reflect satisfying these constraints. Biological realism aside, as a combinatorial object of interest, we note in the top tableau that the first row shows the five different ways a $s = 1$ structure can be located by starting at the 5'-most side and moving one index at a time towards 3'. The extent is always size 2, thus it can occupy base index locations (0-1), (1-2), (2-3), (3-4) and (4-5). The second row expands the opening by one base, now the extent is always size 3, and it can occupy base index locations (0-2), (1-3), (2-4), (3-5). Each subsequent row expands the opening, thereby giving up location space for the address between the 5' and 3' ends. The additional tableaux for $s = 2$ and $s = 3$ similarly show the different ways a triple with these stem-lengths can satisfy the constraints.

o ↓	i →	0	1	2	3	4	s
0		(0,1,0)	(1,1,0)	(2,1,0)	(3,1,0)	(4,1,0)	1
1		(0,1,1)	(1,1,1)	(2,1,1)	(3,1,1)		1
2		(0,1,2)	(1,1,2)	(2,1,2)			1
3		(0,1,3)	(1,1,3)				1
4		(0,1,4)					1
0		(0,2,0)	(1,2,0)	(2,2,0)			2
1		(0,2,1)	(1,2,1)				2
2		(0,2,2)					2
0		(0,3,0)					3

Table 3.5: For a length 6 oligonucleotide, 22 possible triples can fit.

The constraint symmetries lead to a general expression of the total number of legal triples for a length n oligonucleotide. The minimum stem-length s is 1 and the maximum is $\frac{n}{2}$, and for each increase in stem-length in $s = [1, \frac{n}{2}]$, 2 bases are used. Each stem-length, as part of a triple, can be initiated with an address in the range $i = [0, n-2]$ as long as $i + 2s + o < n$. Combining these two facts leads to variant results for oligonucleotide odd

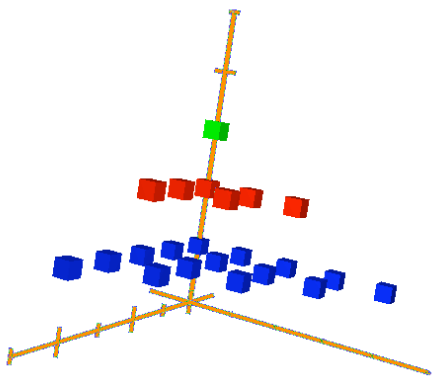


Figure 3.7: 3D view of 22 satisfying triples for a length 6 oligonucleotide.

and even length n values. The maximum possible stem size s is identical, but odd lengths n allow for expansion in the i and o dimensions by one since otherwise there would be space for an entire additional base-pair and thus a greater by one stem-length.

The basis for counting legal triples hinges on stem-length. Given any stem-length s, space must be reserved for the opening and closing bases, exactly $2s$ locations, leaving $n-2s$ remaining locations for placement of the triple (i), or expansion of its opening (o). First consider $s=1$ and place its opening base at address $i=0$, there are now $n-1$ ways to locate the closing base within the remaining indices in $[1,n-1]$. If we move the opening base forward to $i=1$, there are now $n-2$ ways to locate the closing base within the remaining indices $[2,n-1]$. Since there are overall $n-1$ ways to position the opening base within indices $[0,n-2]$, we can sum total number of options to locate a stem-length 1 structure as $(n-1)+(n-2)+\cdots+1=\sum_1^{n-1} i$ as address i ranges between 0 and $n-2$. Increasing the stem-length s to 2 doubles the base count to 4, leaving $n-3$ remaining

locations for the closing base, such that now the total number of placement options is $\sum_1^{n-3}$. Ultimately there are $\frac{n}{2}$ sums possible as the stem-length increases, where each plus 1 increment in stem-length lowers the upper bound on the inner summation by 2. Even lengths n cause the upper bound $n-1$ to be an odd such that the progression of sums is $1,3,5,\ldots,n-1$. Odd lengths n cause the upper bound $n-1$ to be an even making this progression $2,4,6,\ldots,n-1$. The sum of sums for each reduce as follows.

Even-length n.

$$|triples| = \sum_{j=1,j+2}^{n-1} \sum_{1}^{j} i \tag{3.1}$$

Combining summations, counting only the odd increments $j = 1,3,5,\ldots,n-1$ for each stem-length, and using the equation for the sum of the first $j = 2k+1$ integers yields:

$$= 1 + \sum_{k=1}^{\frac{n-1}{2}} \frac{(2k+1)((2k+1)+1)}{2} \tag{3.2}$$

$$= 1 + \sum_{k=1}^{\frac{n-1}{2}} [2k^2+3k+1] \tag{3.3}$$

$$= 1 + 2\sum_{k=1}^{\frac{n-1}{2}} k^2 + 3\sum_{k=1}^{\frac{n-1}{2}} k + \frac{n-1}{2} \tag{3.4}$$

Letting $x = \frac{n-1}{2}$, and substituting equations for sums of squares and sums of integers:

$$= 1 + 2\sum_{1}^{x} k^2 + 3\sum_{1}^{x} k + x \tag{3.5}$$

$$= x + 1 + 2\left[\frac{x(x+1)(2x+1)}{6}\right] + 3\left[\frac{x(x+1)}{2}\right] \tag{3.6}$$

$$= x + 1 + \frac{1}{3}\left[(x^2+x)(2x+1)\right] + \frac{3}{2}\left(x^2+x\right) \tag{3.7}$$

$$= x + 1 + \left(\frac{2x+1}{3} + \frac{3}{2}\right)(x^2+x) \tag{3.8}$$

$$= \frac{2}{3}x^3 + \frac{5}{2}x^2 + \frac{17}{6}x + 1 \tag{3.9}$$

Simplifying through several steps, and reinstating $\frac{n-1}{2} = \frac{n}{2} - 1$ since n is even for x:

$$= \frac{2}{3}\left(\frac{n}{2} - 1\right)^3 + \frac{5}{2}\left(\frac{n}{2} - 1\right)^2 + \frac{17}{6}\left(\frac{n}{2} - 1\right) + 1 \tag{3.10}$$

$$= \frac{1}{12}\left(n^3 + \frac{3}{2}n^2 - n\right) \tag{3.11}$$

Odd-length n compute similarly over the even increments $2, 4, 6, \ldots, n-1$ to yield:

$$|triples| = \sum_{j=2,j+2}^{n-1} \sum_{1}^{j} i = \frac{1}{12}\left(n^3 + \frac{3}{2}n^2 - n - \frac{3}{2}\right) \tag{3.12}$$

Exhaustive enumeration of all structures, without pseudoknots, can be accomplished constructively by combining triples such that they have numerically nested footprints, serially ordered footprints, or a combination thereof. Exhaustive enumeration of all structures with pseudoknots adds to the pseudoknot-free enumeration by breaking up footprints to allow intercalations of opening and closing bases from different triples. This approach allows construction algorithms that accordingly establish an ordering of all structures. Considering the subsets of legal triples, the upper bound on all possible secondary structures, including pseudoknots, is $O(2^{n^3})$ since there are $O(n^3)$ possible triples in a length n oligonucleotide.

Support Algorithms

Algorithm **Triples** computes legal triples, such as those shown in Table 3.5. The algorithm has $O(n^3)$ running time and space usage for a length n oligonucleotide.

Algorithm Triples
Input: `Length n.`
Output: `List of individual triples (i,s,o).`

```
 \*  Standard list append function used to build up vector   *\
 \*  of triples.                                             *\
 1: triples ← [  ]
 2: i ← 0
 3: for i < n-1 do
 4:    s ← 1
 5:    for s < (n/2+1) do
 6:       o ← 1
 7:       for o < n-2 do
```

```
 8:       if i + 2*s + o < n then
 9:         triples ← append(triples,(i,s,o))
10:       o ← o+1
11:     s ← s+1
12:   i ← i+1
13: return triples
```

Algorithm **ValidISO** checks triple combinations to ensure there are no bindings occupying the same locations. It employs algorithm **Bindings**, which causes it to have O($n \log n$) running time and space usage for a length n oligonucleotide.

Algorithm ValidISO
Input: List of triples iso, length n.
Output: True or False boolean.

```
 \*  Assumes bindings returned in   *\
 \*  sorted order.                  *\
 1: if |iso| < 1 then
 2:   return True
 3: allbindings ← bindings(iso)
 4: idx ← 0
 5: while idx < |allbindings|-1 do
 6:   if allbindings[idx] ≡ allbindings[idx+1] then
 7:     return False
 8:   idx ← idx+1
 9: return True
```

Algorithm **Exhaustive Multitriples** generates all secondary structures using a brute-force approach that first generates all triples, then all combinations of triples, and finally filters out ones that are invalid. The running time and space usage is driven by whatever approach is used to compute and store the combinations, thus this is not an efficient algorithm.

Algorithm Exhaustive Multitriples
Input: Length n.

Output: Exhaustive list of all possible pseudoknot-free secondary structures.

```
 \*  Assumes function to compute combinations and standard   *\
 \*  append function to build up final answer.               *\
 1: allTriples ← triples(n)
 2: allCombinations ← combinations(allTriples)
 3: legalCombinations ← [  ]
 4: idx ← 0
 5: for idx < |allcombinations| do
 6:   if ValidISO(allCombinations[idx] then
 7:     legalCombinations ←
        append(legalCombinations,allCombinations[idx])
 8:   idx ← i+1
 9: return legalCombinations
```

Chapter 4

Implementation

Ultimately, we can do chemical synthesis. A chemist comes to us and says, "Look, I want a molecule that has the atoms arranged thus and so; make me that molecule." – Richard P. Feynman [39]

A process flow for creating deoxyribozyme computation instances that can replace the current process shown in Figure 2.1 incorporates the following steps.

1. Determine requirements: write DNADL file.
2. Determine molecule netlist: use Pyxis to search solution space for netlist that satisfies requirements specified in DNADL file.
3. Verify design: laboratory build and test.
4. Iterate Steps 1-3 until satisfying solution is verified.
5. Incorporate Step 3 findings into DNADL and Pyxis.

In this chapter we cover implementation development of the abstractions, algorithms and methods into the DNADL language, the KCA simulation and the Pyxis compiler.

4.1 DNADL: DNA Description Language

DNADL is used to specify instances of chemical reaction network systems that use DNA or RNA and are devised for synthetic function purposes. We saw in Chapter 1 where programming deoxyribozyme platforms has started from three different places: a truth table, a decision tree, and an encoding to produce a dot-matrix display. A high-level language will need to be able to equivalently generate the same set of final Boolean formulas worked out by hand using these previous methods. At present, the only consistent thread is in fact the set of Boolean formulas, and their distribution into each of the wells. Equally important is how the chemistry should work. This disjoint set of concerns is not intrinsic in the training of a programmer, nor is logic part of the training of a chemist, hence our approach is a description-based design language that can serve as the target of a true higher-level programming language and as well be decomposed into finer chemical-level details.

A descriptive specification of a chemical reaction network requires a modification to the set of concepts that we normally associate with chemical systems, and with building programmable devices and machines. Are we describing a machine acting as a chemical reaction network, or a chemical system acting as a machine? The semantics of chemical systems include properties such as the stoichiometric balance of the involved tranformation mechanisms, the concentrations of reactants and products, heat loss or gain, and elapsed time. The semantics of device or machine building include operating characteristics, resource requirements, waste products, and environmental conditions such as temperature and pressure. The semantics of programmable devices are trickier since a wide variety of outcomes may be possible using the same basic substrate, hence leading to a construction-

centric focus. Missing are the familiar and concrete notions of state and time, as neatly digital units, because the nucleic acid basis of a chemical system that is realizing a programmable device brings with it the attendant characteristics of a many-bodied system undergoing continuous change. In this regime, the states are uncountable, and reaction events occur in time ranges rather than at specific and identifiable instances.

DNADL handles these issues through operational semantics, analogous to hardware description languages devised for electronic design automation. DNADL allows definition of Boolean logic formulae, what the system components are, and how they should individually behave and together interact. A linkage of logic to components is provided to show how deductions are worked out as one or more physical reaction steps. The textual description, allows simulation or emulation testing its design at the degree dictated by the description. Since the textual description is also a specification, it serves as a requirements document for initiating the automated search for one or more sets of satisfying nucleic acid oligonucleotides operating within multiple possible environments. The textual format with a preset type keyword lexicon also yields standardization benefits and a consistent record-keeping of designs and programs.

4.1.1 Description Levels

To break up the attendant complexity of DNA chemistry and enable expanded functionality and construction scale, DNADL separates description into three levels of concerns.

- **Level 1** The device or program in purpose and functional logic terms. A Level 1 description of a program and platform to carry out propositional logic deduction states

the premisses, deducible conclusions, and evaluation order, without any appeal to how these will be effected. This basic treatment serves as a foundation for layering a more sophisticated language above it.

- **Level 2** The active physical events and behavioral aspects of the chemistry that together can instantiate the purpose and functional expectations of the preceding level. A Level 2 description covers how reaction steps can serve as premisses and conclusions, and what required actions must take place to set up these steps.

- **Level 3** The individual elements and their properties which will behave as described in the preceding level. This is the level of the molecular groups, how they are to be formed, and whatever characteristics the elements must possess in order to carry out required behaviors and interactions. For reactions carrying out propositional logic, this corresponds to the set of DNA oligonucleotides and what reporting molecules will be attached to signal final Boolean outputs.

4.1.2 Types and Identifiers

Every DNADL description line is uniquely identified and typed to provide uniform constraints on exactly how descriptions can be declared, and what bounds are present on the strings, natural numbers, or predefined sets that each is defined with. The format for all descriptions is type name, followed by identifiers of that type. Effectively in DNADL, typing acts as a data and process abstraction technique. Data and process terms may be elemental, such that decomposition into something smaller does not meaningfully relate to the overall description, or they may be a collection of elemental terms that act as a record. Record-style descriptions are akin to abstract data types found in several mainstream programming languages as they allow at-will type creation that wraps distinct concepts together reflective of their physical interconnectedness. Elemental descriptions are classified as **single**

entities, while collection descriptions are classified as **compound** entities. In contrast to mainstream programming languages, there is no association between the type declaration and how much memory is required for each identifier. Memory usage is not yet relevant because there has not been any capability demonstration of a true data store that is both readable and overwritable within a reaction network.

Types are usually not associated with hardware descriptions languages, because languages of this genre are not thought of as true programming languages. However, this view is short-sighted. Types in any language create benefits. The benefits we are in need of for a chemical reaction network are safety, abstract model building, and automation of finding and verifying checkable properties of the systems being described. All of these support determination of correctness. Similar to traditional circuits and programming languages, there are often possible redundant designs that will equally satisfy the goals of the device. This redundancy portends the need for ranking of solutions, but according to what measure? Low cost and high reliability are suitable attributes to shoot for, but we need a way to say exactly what is meant with these terms. Cost may not just imply materials and personnel time, it may also make sense to consider a complexity measure and prove a bounded range of time for system operation. Reliability suggests ease of replication by other researchers outside a laboratory devising a new and better DNA system. And while replicating results across labs is not currently a goal of the university funding environment, moving DNA nanotechnology out of research labs and into manufacturing arenas will demand reliability measures. Since we can't say what is important, and can see that ranking schemes will vary across labs, instead we aim to incorporate the ability to work these issues out when it becomes a real priority to do so.

The following table and subsections show the DNADL base types. Programmed device or machine functionality is described in Layer 1 and is customized to the logic goals of the

intended application. Levels 2 and 3 cover data and process abstractions for the physical steps the reaction network must take. The type PHYSICALMAP is outside the layers and shows the mapping between device steps outlined in Layer 1, to data, processes, and attributes outlined in Layers 2 and 3. The purpose of this type is to show a clear linkage between device construction and operational steps, to events within the chemical reaction network that is realizing the device.

Layer	*Classification*	*Type*
1	single	**ADDRESS**
1	single	**PROGRAM**
1	compound	**PREMISS**
1	compound	**CONCLUSION**
2	compound	**ENTRY**
2	compound	**TRANSITION**
2	compound	**EXECUTIONMECHANISM**
2	compound	**SIGNAL**
2	compound	**EVENTSTREAM**
3	single	**LENGTH**
3	single	**FLUOROPHORE**
3	single	**QUENCHER**
3	single	**SEQUENCE**
3	single	**DOMAIN**
3	single	**ISO**
3	single	**CONCENTRATION**
3	compound	**STRAND**
-	compound	**PHYSICALMAP**

Table 4.1: DNADL Base Types

Layer 1 Base Types

Definition ADDRESS. An identifier a of type **ADDRESS** gives the grid location for a chemical reaction network within a well plate $a = (row, col) \in N^2$ where $(0,0)$ corresponds to the lower left well. Reaction networks constructed in single vessels and not using a well plate are declared with the address type, but are not given a value.

```
ADDRESS

/* a1 at row 10, col 11 on well plate, potMAIN self-contained */

a1 = (10, 11);

potMAIN;
```

Definition PROGRAM. An identifier *prog* of type **PROGRAM** denotes a complete program expressed in propositional logic where each formula is composed of literals and propositional connectives AND ($\wedge$), OR ($\vee$), NOT ($\neg$), or IMPLICATION ($\rightarrow$), formatted with each line numbered.

```
PROGRAM program4layer

1:  (E2 ∧ SCS2) → ACT2;

2:  (E3 ∧ SCS3) → ACT3;
```

Definition PREMISS. An identifier p of type **PREMISS** denotes the assignment of a formula f to a vessel or well plate located at ADDRESS a, $p = (a, f)$, where f is a logically valid formula composed of literals and propositional connectives AND ($\wedge$), OR ($\vee$), NOT ($\neg$), or IMPLICATION ($\rightarrow$).

```
PREMISS

p13 = (a1, I14);

p14 = (a1, I21 ∧ I62);

p110 = (a1, I62 ∧ I23 ∧ ¬ I21);
```

Definition CONCLUSION. An identifier c of type **CONCLUSION** denotes the logical deduction f of a vessel or well plate located at ADDRESS a, $c = (a, f)$, where f is a logically valid formula composed of literals and propositional connectives AND ($\wedge$), OR ($\vee$), NOT ($\neg$), or IMPLICATION ($\rightarrow$).

```
CONCLUSION

conclusion2 = (a2, I22);

conclusion8 = (a8, I61 ∧ I12 ∧ ¬ I82);
```

Layer 2 Base Types

Definition ENTRY. An identifier e of type **ENTRY** denotes the complete action of introducing a STRAND s at CONCENTRATION c into a well at ADDRESS a, $e = (a, s, c)$.

```
ENTRY

enZYME1INH1 = (potMAIN,strDNAZYME1INH1,100);

enZYME2INH2 = (potMAIN,strDNAZYME2INH2,100);

enZYME3INH3 = (potMAIN,strDNAZYME3INH3,100);
```

Definition TRANSITION. An identifier t of type **TRANSITION** denotes a transformation happening in a vessel or well at ADDRESS a. Transformations may be one STRAND s_a taking on a new structural form s_b through an alternative hybridization pattern (*fold*), STRAND s_a and STRAND s_b becoming a complex s_c through hybridization (*bind*), complex s_c dissociating into strands s_a, s_b (*unbind*), strands s_a, s_b becoming a longer strand s_c through backbone linkage (*ligate*), a complex and a single oligonucleotide undergoing strand exchange (*exchange*), or a complex or single oligonucleotide s_a splitting along a backbone linkage into parts (*cleave*). Transitions are described through use of the fold, bind, unbind, ligate, exchange or cleave operators $t = (a, s_a, s_b, s_c, s_d, op)$, where inclusion of STRAND entities s_a, s_b, s_c, or s_d depends on choice of operator op.

- ***fold*** is an unary operation on a STRAND s_a to produce s_b with the same sequence and length, but with a different structure, $s_b = fold(s_a)$.

- ***bind*** is a binary operation on two STRAND entities s_a, s_b to produce a new complexed STRAND entity s_c, $s_c = bind(s_a, s_b)$.

- ***unbind*** is a unary operation on one STRAND s_c to produce two STRAND entities s_a and s_b, $s_a, s_b = unbind(s_c)$.

- ***ligate*** is a binary operation on two STRAND entities s_a and s_b to produce a new ligated STRAND s_c, $s_c = ligate(s_a, s_b)$.

- ***exchange*** is a binary operation on two STRAND entities s_a and s_b to swap single oligonucleotides and form s_c and s_d. The operator implicitly expects that at least s_a

or s_b be a complex of at least two oligonucleotides and that at least one of s_c or s_d be a complex of at least two oligonucleotides, $s_c, s_d = exchange(s_a, s_b)$.

- ***cleave*** is a unary operation on one STRAND s_a to create two STRAND entities s_b, s_c, $s_b, s_c = cleave(s_a)$.

```
TRANSITION

tzymeinh1 = (potPREPZYME,strDNAZYME1,strINH1,strDNAZYME1INH1,bind);

tzyme2scs2split = (potMAIN,strDNAZYME2SCS2.stage2,strZYME2WASTE2,
strACT2FOLDED,cleave);

treleasezyme1 = (potMAIN,strDNAZYME1INH1,strACT2UNFOLDED,
strDNAZYME1UNFOLDED,strACT2INH1,exchange);
```

Definition EXECUTIONMECHANISM. An identifier *em* of type **EXECUTIONMECHANISM** denotes an ordered sequence of TRANSITION entities $t_0, t_1, \ldots$. Any reordering of transitions, insertions of new transitions, or deletions of transitions from the ordered transition sequence is an off-nominal mechanism that may not correctly carry out the required physical changes instantiating some device operational step.

```
EXECUTIONMECHANISM

layer2releaseactivator = [tzyme2scs2stage1, tzyme2scs2stage2,
tzyme2scs2split, tzyme2recovery];

layer2releasegate = [treformact2,treleasesyme1,tactive1];

layer1signal = [tzyme1substrate1];
```

Definition SIGNAL. An identifier z of type **SIGNAL** denotes the observation of threshold fluorescence color resulting from a free in solution FLUOROPHORE $flphr$ at ADDRESS a, $z = (a, color)$. The value of *color* must be either red, green, pink or purple, corresponding to a subset of FRET capabilities.

```
SIGNAL

visualgreen = (potMAIN, green);
```

Definition EVENTSTREAM. An identifier *exec* of type **EVENTSTREAM** denotes the ordered stream of events occuring at ADDRESS a. Events may be of types ENTRY, SIGNAL, or EXECUTIONMECHANISM and their particular ordering stipulates exactly how the reaction network correctly functions. In absence of a system clock, with physical events occurring in tightly precise intervals, the ordered stream describes the chemical pipeline of directed physical activity. The stream is formatted in the style of a program, with each event numbered in order. Events that occur together, with no expectation of a local ordering, are grouped together on a single line and distinguished by surrounding $<$ and $>$ characters.

```
EVENTSTREAM

execCASCADE

1:  <enZYME1INH1,enZYME2INH2,enZYME2INH3>;

2:  <enSCS2FOLDED,enSCS3FOLDED,enSCS4FOLDED>;

3:  enDNAZYME4;

4:  layer4releaseactivator;
```

Layer 3 Base Types

Definition LENGTH. An identifier l of type **LENGTH** denotes oligonucleotide length. The value must be nonzero and within N.

```
LENGTH

/* SCS and INH strand lengths in nt */

lengthSCS2 = 47;

lengthSCS3 = 46;

lengthSCS4 = 46;

lengthINH = 23;
```

Definition FLUOROPHORE. An identifier f of type **FLUOROPHORE** indicates selection of a fluorescent signaling molecule used as part of the FRET reporting scheme. The value must be either FAM or TAMRA.

```
FLUOROPHORE

FAM;
```

Definition QUENCHER. An identifier q of type **QUENCHER** indicates selection of a fluorescent absorbing molecule used as part of the FRET reporting scheme. The value must be either BH2 or TAMRA.

```
QUENCHER

BH2;
```

Definition SEQUENCE. An identifier *seq* of type **SEQUENCE** is the primary sequence string of a DNA oligonucleotide $seq = q_0 q_1 \ldots q_{n-1}$, $q_i \in \Sigma, \Sigma = \{A, C, G, T\}, n > 2$. For an RNA oligonucleotide the alphabet changes to $\Sigma = \{A, C, G, U\}$. SEQUENCE values may be described using DOMAIN identifiers, or as mixture between strings drawn from Σ and DOMAIN identifiers. SEQUENCE values may also be described as reverse complements of other SEQUENCE values through use of the revcomp operator. However a value of type SEQUENCE is described, it is always formatted using the biological $5'$ to $3'$ convention.

- ***revcomp*** is a unary operation on a sequence or domain to produce a new reverse complemented sequence or domain of the same length, where complementation follows Watson-Crick $A : T$ and $C : G$, $s_b = revcomp(s_a), s_a = q_0 q_1 \ldots q_{n-1}, s_b = q_{n-1} q_n \ldots q_o, q_i / q_{i'} [A/T, T/A, C/G, G/C]$.

Concatenation of two or more sequences seq_a, seq_b, ... uses a short dash $-$ between sequences.

```
SEQUENCE

seqSCS2 = CGCCCTAATCTTAGGTCGAAAACTAAGATACATACTAGGGCGTGATG;

seqINH1 = ATGTATCTTAGTTTTCGACCGGC;

seqCOMPLEX = seqDNAZYME2-seqSCS2;
```

Definition DOMAIN. An identifier *dom* of type **DOMAIN** is a subsequence string of any sequence *seq*. Domains may be described as reverse complements of other domains through use of the revcomp operator. Concatenation of two or more DOMAIN subsequence strings dom_a, dom_b uses a short dash $-$ between them.

```
DOMAIN

domENZ = TCCGAGCCGGTCGAAA;
```

Definition ISO. An identifier *struct* of type **ISO** is the secondary structure of an oligonucleotide in ISO format. ISO describes secondary structure as a list of numeric triples (i,s,o), $i,s,o \in N$ where each triple defines a distinct binding region for a sequence string $q_0q_1 \ldots q_{n-1}$. $iso = [(i,s,o)_0,(i,s,o)_1,\ldots,(i,s,o)_{m-1}]$ is a unique representation of secondary structure such that for each binding region:

- i defines the zero-based indexing location relative to the q_0 $5'$ end,
- s defines the length of binding,
- o defines the opening enclosed by s, equal to the number of bases, paired or unpaired, which are intermediate between the last opening base and first closing base of the binding stem.

```
ISO

structSUBSTRATE = [];

structSCS2 = [(0,7,28),(7,7,8)];

structSCS3 = [(0,7,28),(7,6,10)];
```

Definition CONCENTRATION. An identifier *conc* of type **CONCENTRATION** is the nanomolar concentration of a strand, $c \in R, c > 0$.

```
CONCENTRATION

concSUBSTRATE = 100.;
```

Definition STRAND. An identifier s of type **STRAND** is a single oligonucleotide o or complex of n oligonucleotides $o_0, o_1, \ldots, o_{n-1}$ characterized by its SEQUENCE s_{SEQ}, ISO s_{STRC}, and LENGTH s_L, $s = (s_{SEQ}, s_{STRC}, s_L)$. In cases where s_{SEQ} has already been completely described, the length information is redundant. In cases where a compiler or some other tool will be used to find a satisfying s_{SEQ}, the inclusion of length information is required since the ISO structure encoding does not implicitly provide length information. Signaling molecules are prepended and appended to s_{SEQ} if the behavior of s includes reporting.

- s_{SEQ} is a SEQUENCE entity for oligonucleotide s or a concatenation of SEQUENCE entities $s_0, s_1 \ldots s_{n-1}$, $s_{SEQ} = s_0 - s_1 - \cdots - s_{n-1}$, where each s_i is the SEQUENCE entity for oligonucleotides o_i ordered $5'$ to $3'$ as part of a complex.
- s_{STRC} is the ISO entity, $s_{STRC} = [(i,s,o)_0, \ldots]$ for one or more oligonucleotides with primary sequence s_{SEQ} and length s_L. Note that ISO handles single oligonucleotides, or multiple oligonucleotides bound as a complex, hence there is no compositing scheme of individual structure strings required.
- s_L is the length of the single oligonucleotide o, or the total length of n oligonucleotides o_0, o_1, …, o_{n-1} lengths, $s_L = l_0 + l_1, + \cdots + l_{n-1}$ if s is a complex of n oligonucleotides.

```
STRAND

strACT2INH1 = (seqACT2-seqINH1,[(13,20,3)],lengthACT+lengthINH);

strACT3INH2 = (seqACT3-seqINH2,[(13,20,3)],lengthACT+lengthINH);

strACT4INH3 = (seqACT4-seqINH3,[(11,22,3)],lengthACT+lengthINH);

strSUBSTRATE1 = (FAM-seqSUBSTRATE1-TAM,[],lengthSUBSTRATE1);
```

Definition PHYSICALMAP. An identifier *map* of type **PHYSICALMAP** describes a matching between the union of Level 1 sets P of PREMISS entities and C of CONCLUSION entities, to a subset of the union of Level 2 sets E of ENTRY entities and EM of EXECUTIONMECHANISM entities, and Level 3 set S of STRAND entities, $\{x \leftrightarrow y | x \in P \cup C, y \in Y, Y \subseteq E \cup EM \cup S\}$. If the Level 1 propositional logic description uses PROGRAM *prog* in alternative, then the set of *prog* numbered lines replaces $P \cup C$.

```
mapCASCADE

1 ↔ layer2releaseactivator; /* (E2 ∧ SCS2) → ACT 2 */

2 ↔ layer3releaseactivator; /* (E3 ∧ SCS3) → ACT 3 */

3 ↔ layer4releaseactivator; /* (E4 ∧ SCS4) → ACT 4 */
```

4.1.3 Four Layer Cascade Example

A recent example of a four layer cascade [18] uses a new enzyme-based logic gate design that combines deoxyribozymes and strand-displacement. A single active deoxyribozyme, termed a DNAzyme, initiates a domino effect of releasing and activating subsequent DNAzymes through use of additional single oligonucleotides with particular secondary structure. The additional oligonucleotides are termed structured chimeric substrate (SCS) molecules by virtue of their ability to conditionally switch between two shapes. Operation of the four layer cascade depends on attaining specific structural forms, binding between the SCS molecules and active DNAzymes, and strand-exchange between inhibited DNAzymes and intermediately produced stem-loop oligonucleotides aptly termed Activators. Inhibited DNAzymes are created a priori of the cascade operation through binding of the DNAzymes to complementary single-stranded oligonucleotides termed Inhibitors.

The development time for debugging the cascade required over 12 months. An advantage of using a textual description is to make this effort programmable to the fullest extent possible, and therefore reduce development time by at least 50%. The cascade steps are uniform, and clearly amenable to a programming approach. The complete description using DNADL is shown in Appendix C.1 corresponding to the illustrations shown in Appendix C.2[1].

4.1.4 Deoxyribozyme Logic Gates

For the original Stefanovic and Stojanovic deoxyribozyme-based logic gates, the desired behaviors are well understood and amenable to subtyping. The YES and AND gates signal in response to the specific presence of positive input, whereas the NOT, AND-NOT, and AND-AND-NOT require both presence of required positive input, and absence of the negated input. Since the "yes" function is essentially a recognition, the main application of this gate is in a detector device. Implicitly, more than one *recognizer* (yes) gate within the same vessel or well plate location is a logical *or* formula.

Recognizer Logic

Signal generating case: when input is present transitions as 5 stages
→ from no shape to stem-loop structure:
s-stage1 = fold(s-stage0)
→ from stem-loop structure to binding with input strand:
s-stage2 = bind(s-stage1,s-input)
→ opened stem with enzyme exposed:

[1]Figures from [18] reproduced with permission of Matthew Lakin, 2014.

s-stage3 = fold(s-stage2)

→ from activated gate to binding with substrate strand:

s-stage4 = bind(s-stage3,s-substrate)

→ substrate cut in half:

s-substrateF, s-substrateQ = cleave(s-substrate)

Transition subtypes:

TCASE1RECOG0-1 = (a, s-stage0, s-stage1, fold);

TCASE1RECOG1-2 = (a, s-stage1, s-inputa, s-stage2, bind);

TCASE1RECOG2-3 = (a, s-stage2, s-stage3, fold);

TCASE1RECOG3-4 = (a, s-stage3, s-substrate, s-stage4, bind);

TCASE1RECOG4-SIGNAL = (a, s-substrate, s-substrateF, s-substrateQ, cleave);

Not Logic

Signal generating case: when input is absent transitions as 3 stages

→ from no shape to stem-loop structure:

s-stage1 = fold(s-stage0)

→ from activated gate to binding with substrate strand:

s-stage2 = bind(s-stage1,s-substrate)

→ substrate cut in half:

s-substrateF, s-substrateQ = cleave(s-substrate)

Transition subtypes:

TCASE1NEGATE0-1 = (a, s-stage0, s-stage1, fold);

tCase1negate1-2 = (a, s-stage1, s-substrate, s-stage2, bind);

tCase1negate2-signal = (a, s-substrate, s-substrateF, s-substrateQ, cleave);

And Logic

Signal generating cases: +inputa+inputb, +inputb+inputa transitions as 6 stages

→ from no shape to two stem-loops:

s-stage1 = fold(s-stage0)

→ from stem-loops to binding with one input strand:

s-stage2 = bind(s-stage1,s-inputa)

→ from stem-loops with one binding to both bindings:

s-stage3 = bind(s-stage2,s-inputb)

→ from input bindings to exposed enzyme region:

s-stage4 = fold(s-stage3)

→ from activated enzyme to binding with substrate strand:

s-stage5 = bind(s-stage4,s-substrate)

→ substrate cut in half:

s-substrateF, s-substrateQ = cleave(substrate)

Transition subtypes:

tCase1and0-1 = (a, s-stage0, s-stage1, fold);

tCase1and1-2 = (a, s-stage1, s-inputa, s-stage2, bind);

tCase1and2-3 = (a, s-stage2, s-inputb, s-stage3, bind);

tCase1and3-4 = (a, s-stage3, s-stage4, fold);

tCase1and4-5 = (a, s-stage4, s-substrate, s-stage5, bind);

tCase1and5-signal = (a, s-substrate, s-substrateF, s-substrateQ, cleave);

And-Not Logic

Signal generating case: +input transitions as 5 stages

→ from no shape to two stem-loops:

s-stage1 = fold(s-stage0)

→ from stem-loops to binding with input strand:

s-stage2 = bind(s-stage1,s-input)

→ from input binding to exposed enzyme region:

s-stage3 = fold(s-stage2)

→ from activated enzyme to binding with substrate strand:

s-stage4 = bind(s-stage3,s-substrate)

→ substrate cut in half:

s-substrateF, s-substrateQ = cleave(substrate)

Transition subtypes:

tCase1andnot0-1 = (a, s-stage0, s-stage1, fold);

tCase1andnot1-2 = (a, s-stage1, s-input, s-stage2, bind);

TCASE1ANDNOT2-3 = (a, s-stage2, s-stage3, fold);

TCASE1ANDNOT3-4 = (a, s-stage3, s-substrate, s-stage4, bind);

TCASE1ANDNOT4-SIGNAL = (a, s-substrate, s-substrateF, s-substrateQ, cleave);

And-And-Not Logic

Signal generating cases: +inputa+inputb, +inputb+inputa transitions as 6 stages

→ from no shape to three stem-loops:

s-stage1 = fold(s-stage0)

→ from stem-loops to binding with one input strand:

s-stage2 = bind(s-stage1, s-input)

→ from stem-loops with one binding to both input bindings:

s-stage3 = bind(s-stage2,s-input)

→ from input binding to exposed enzyme region:

s-stage4 = fold(s-stage3)

→ from activated enzyme to binding with substrate strand:

s-stage5 = bind(s-stage4,s-substrate)

→ substrate cut in half:

s-substrateF, s-substrateQ = cleave(substrate)

Transition subtypes:

TCASE1ANDANDNOT0-1 = (a, s-stage0, s-stage1, fold);

TCASE1ANDANDNOT1-2 = (a, s-stage1, s-input, s-stage2, bind);

TCASE1ANDANDNOT2-3 = (a, s-stage2, s-input, s-stage3, bind);

TCASE1ANDANDNOT3-4 = (a, s-stage3, s-stage4, fold);

TCASE1ANDANDNOT4-5 = (a, s-stage4, s-substrate, s-stage5, bind);

TCASE1ANDANDNOT5-SIGNAL = (a, s-substrate, s-substrateF, s-substrateQ, cleave);

4.1.5 MAYAII revisited

The MAYAII application [73] played 76 tic-tac-toe games on the same board, where both marks and responses, and the board itself were all realized as 9 chemical reaction networks in 9 separate wells in a standard 384 well-plate. Another way to view how this was done is to consider the board as a constructed platform and the games as 76 different propositional logic programs that were executed on the platform. Through writing out exactly the requirements for every aspect of the programs and the machine they executed on, we are able to identify what the constraints are on the entire set of DNA oligonucleotides. This enables automatic search for a satisfying set within a compiler program. Efficiency gains in the overall process may cut down on development time, and allow scaling to larger propositional logic realizations.

Each game used a subset of premisses named as inputs and a conclusions set named as outputs, and can be treated as an individual program, as shown below for Game 18, Quarter D within the game tree. The remaining premisses were named gates and were identical for all games. In the example, conclusions are denoted in blue text and follow from the premisses. Premisses are divided into two groups indicated using black text for gates and red text for inputs. Deduced conclusions were physically persistent, and did not always result in an observable signal. The example DNADL file in Appendix C.3 shows the complete description for all 76 games and the platform. The size reflects the full complexity of what was accomplished over 36 months of development time [2] solely using nucleic acid chemistry.

[2] Joanne Macdonald, personal communication, 2008.

MAYA II Quarter D, Game 18 (M2QDG18) Propositional Logic Execution

Well 1	**Well 2**
I11 $\vee$ I12 $\vee$ I13 $\vee$ I14	I21 $\vee$ I22 $\vee$ I23 $\vee$ I24 $\vee$ I61 $\vee$ I91
I21 $\wedge$ I62	I62 $\wedge$ I13
I33 $\wedge$ I44	I93 $\wedge$ I34
I82 $\wedge$ I73	I11 $\wedge$ I32 $\wedge \neg$ I22
I71 $\wedge$ I42	I11 $\wedge$ I42 $\wedge \neg$ I22
I62 $\wedge$ I73 $\wedge \neg$ I21	I11 $\wedge$ I62 $\wedge \neg$ I22
I62 $\wedge$ I83 $\wedge \neg$ I21	I11 $\wedge$ I72 $\wedge \neg$ I22
I62 $\wedge$ I23 $\wedge \neg$ I21	I11 $\wedge$ I92 $\wedge \neg$ I22
I42 $\wedge$ I73 $\wedge \neg$ I71	I41 $\wedge$ I12 $\wedge \neg$ I22
I42 $\wedge$ I33 $\wedge \neg$ I71	I41 $\wedge$ I32 $\wedge \neg$ I22
I42 $\wedge$ I23 $\wedge \neg$ I71	I41 $\wedge$ I62 $\wedge \neg$ I22
	I41 $\wedge$ I72 $\wedge \neg$ I22
	I41 $\wedge$ I92 $\wedge \neg$ I22
I31	I31
$\Rightarrow \square$	$\Rightarrow \square$
I62	I62
$\Rightarrow \square$	$\Rightarrow \square$
I13	I13
$\Rightarrow$ I13	$\Rightarrow$ I13 $\wedge$ I62
I74	I74
$\Rightarrow$ I13	$\Rightarrow$ I13 $\wedge$ I62

Well 3	**Well 4**
I31 $\vee$ I32 $\vee$ I33 $\vee$ I34	I41 $\vee$ I42 $\vee$ I43 $\vee$ I44 $\vee$ I21 $\vee$ I31
I11 $\wedge$ I22	I22 $\wedge$ I73
I61 $\wedge$ I82	I33 $\wedge$ I14
I42 $\wedge$ I13	I81 $\wedge$ I92 $\wedge \neg$ I42
I93 $\wedge$ I24	I81 $\wedge$ I72 $\wedge \neg$ I42
I22 $\wedge$ I63 $\wedge \neg$ I11	I81 $\wedge$ I32 $\wedge \neg$ I42
I22 $\wedge$ I93 $\wedge \neg$ I11	I81 $\wedge$ I22 $\wedge \neg$ I42
I22 $\wedge$ I13 $\wedge \neg$ I11	I81 $\wedge$ I12 $\wedge \neg$ I42
I82 $\wedge$ I63 $\wedge \neg$ I61	I71 $\wedge$ I92 $\wedge \neg$ I42
I82 $\wedge$ I43 $\wedge \neg$ I61	I71 $\wedge$ I82 $\wedge \neg$ I42
I82 $\wedge$ I13 $\wedge \neg$ I61	I71 $\wedge$ I32 $\wedge \neg$ I42
	I71 $\wedge$ I22 $\wedge \neg$ I42
	I71 $\wedge$ I12 $\wedge \neg$ I42
I31	I31
$\Rightarrow$ I31	$\Rightarrow$ I31
I62	I62
$\Rightarrow$ I31	$\Rightarrow$ I31
I13	I13
$\Rightarrow$ I31	$\Rightarrow$ I31
I74	I74
$\Rightarrow$ I31	$\Rightarrow$ I31

Well 5	**Well 6**
`True`	I61 $\vee$ I62 $\vee$ I63 $\vee$ I64 $\vee$ I71 $\vee$ I81
	I73 $\wedge$ I94
	I82 $\wedge$ I33
	I21 $\wedge$ I92 $\wedge \neg$ I62
	I21 $\wedge$ I82 $\wedge \neg$ I62
	I21 $\wedge$ I72 $\wedge \neg$ I62
	I21 $\wedge$ I32 $\wedge \neg$ I62
	I21 $\wedge$ I12 $\wedge \neg$ I62
	I31 $\wedge$ I92 $\wedge \neg$ I62
	I31 $\wedge$ I82 $\wedge \neg$ I62
	I31 $\wedge$ I72 $\wedge \neg$ I62
	I31 $\wedge$ I22 $\wedge \neg$ I62
	I31 $\wedge$ I12 $\wedge \neg$ I62
I31	I31
$\Rightarrow$ `True`	$\Rightarrow \square$
I62	I62
$\Rightarrow$ `True`	$\Rightarrow$ I62
I13	I13
$\Rightarrow$ `True`	$\Rightarrow$ I62
I74	I74
$\Rightarrow$ `True`	$\Rightarrow$ I62

Well 7

$I71 \vee I72 \vee I73 \vee I74$

$I41 \wedge I22$

$I62 \wedge I93$

$I13 \wedge I84$

$I91 \wedge I82$

$I22 \wedge I93 \wedge \neg\, I41$

$I22 \wedge I63 \wedge \neg\, I41$

$I22 \wedge I43 \wedge \neg\, I41$

$I82 \wedge I93 \wedge \neg\, I91$

$I82 \wedge I43 \wedge \neg\, I91$

$I82 \wedge I13 \wedge \neg\, I91$

$I31$

$\Rightarrow \square$

$I62$

$\Rightarrow \square$

$I13$

$\Rightarrow \square$

$I74$

$\Rightarrow I74$

Well 8

$I81 \vee I82 \vee I83 \vee I84 \vee I11 \vee I41$

$I13 \wedge I74$

$I42 \wedge I93$

$I91 \wedge I72 \wedge \neg\, I82$

$I91 \wedge I62 \wedge \neg\, I82$

$I91 \wedge I42 \wedge \neg\, I82$

$I91 \wedge I32 \wedge \neg\, I82$

$I91 \wedge I12 \wedge \neg\, I82$

$I61 \wedge I92 \wedge \neg\, I82$

$I61 \wedge I72 \wedge \neg\, I82$

$I61 \wedge I42 \wedge \neg\, I82$

$I61 \wedge I32 \wedge \neg\, I82$

$I61 \wedge I12 \wedge \neg\, I82$

$I31$

$\Rightarrow \square$

$I62$

$\Rightarrow \square$

$I13$

$\Rightarrow \square$

$I74$

$\Rightarrow I13 \wedge I74$

Well 9

I91 $\vee$ I92 $\vee$ I93 $\vee$ I94

I73 $\wedge$ I64

I22 $\wedge$ I33

I31 $\wedge$ I62

I81 $\wedge$ I42

I62 $\wedge$ I83 $\wedge \neg$ I31

I62 $\wedge$ I73 $\wedge \neg$ I31

I62 $\wedge$ I33 $\wedge \neg$ I31

I42 $\wedge$ I83 $\wedge \neg$ I81

I42 $\wedge$ I33 $\wedge \neg$ I81

I42 $\wedge$ I23 $\wedge \neg$ I81

I31

$\Rightarrow \square$

I62

$\Rightarrow$ I31 $\wedge$ I62

I13

$\Rightarrow$ I31 $\wedge$ I62

I74

$\Rightarrow$ I31 $\wedge$ I62

4.2 Examining Cross-Talk Using The Kinetic Cellular Automaton (KCA) Simulation

Synthetically designed DNA chemical reaction networks do not execute perfectly. Degraded performance, where results differ from expectation, can occur naturally and normally. The ease of precise helix formation by way of sequence selection is what makes DNA an engineerable material, yet the basic promiscuity of DNA polymers to form hydrogen bonds wherever there are complementary subsequences also leads to off-nominal, accidental reactions resulting in unwanted helices. Unwanted helix formation is informally termed the "cross-talk" problem. Cross-talk covers any bindings that are not part of the design plan. Such bindings may outcompete designed bindings, leading to diminished overall output signalling, or output signal generation at the wrong time. Effectively, cross-talk represents system noise in a chemical reaction network.

An example of how cross-talk can arise is found in the yes-gate reaction sequence shown in Table A.2.1.1 (Appendix A). The expected reaction order is: (1) the logic gate strand $G_{\overline{a}}$ binds to the input I_a and becomes activated, (2) the logic gate strand then binds to the substrate S, followed by (3) cleavage of the substrate into two short strands where one strand P_f has the attached fluorophore and the the other strand P_q has the attached quencher. Sequence assignments for the 8.17.1 substrate and 8.17.1 Left Yes gate strands are shown in Tables A.1 and A.2. As designed, the gate strand has a base assignment in the stem region which allows it to self-bind and form the stem, and also to bind to the substrate after the gate has been activated by way of gate-input binding. Since the same subsequence is used in two different places, there is the possibility of a race condition if both gate and substrate strands are introduced simultaneously as shown in Figure 4.1. There is additionally the possibility that two gate strands will hybridize together, rather than each folding separately into the expected stem-loop form.

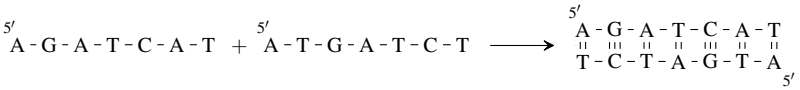


Figure 4.1: Subsequence *AGATCAT*, present on the gate strand starting at base 3, and on the substrate strand starting at base 10, is the reverse complement of subsequence *ATGATCT*, present on the gate strand starting at base 25. These regions admit a binding and can hybridize with opposing directions as shown on the right.

Each of these off-nominal reactions is possible cross-talk and to the degree that they occur in solution, a reduced percentage of required reactants and products are available at each step of the reaction pathway. Cross-talk may result in a false logical *yes* when some number of gate and substrate strands hybridize before the gate strands form their stems. These "leaky" gates are simply playing out a game of chance: unless there is zero possibility of unwanted bindings, eventually some number of strands with complementary subsequences will find each other in solution and bind even if they were never intended to do so. Similarly, some number of gate strands may bind together rather than self-fold, a different form of cross-talk that yields a reduced number of correctly formed gates able to support the overall reaction pathway and leading to a weaker output signal. Drexler in his Ph.D. thesis comments on misreactions similarly and notes that "In diffusive synthesis, achieving 95% yield in each of a long series of steps is typically considered excellent. At the end of a 100-step process, however, the net product would be about 0.6%." Thus the overall technology limitation is not construction, but rather the difficulty of avoiding mistaken additions [32].

Estimating Potential Cross-Talk Scale

Since more than one strand set S may serve to execute a programmed chemical reaction network, choosing a satisfying set that minimizes cross-talk is part of the compile cycle. To this end, we can cast mitigating cross-talk within a network using optimization

terminology. We consider only unimolecular reactions where a strand folds with particular secondary structure, or bimolecular where two strands hybridize to form a complex. This assumption is consistent with preceding approaches for stochastic chemical kinetic simulation where trimolecular and reversible reactions occur primarily as a series of unimolecular or bimolecular events [50]. Let S be a putative satisfying strand set for some programmed reaction network. At each step where a new species $x_i \in S$ is introduced, there are n pairings between strand x_i and each of the n existing strand species to consider for potential duplex formation plus one considering two x_i species binding together. Assuming that these potential duplex products–hybridizations–will use at least three contiguous complementary bases from each strand, there are $l_i - l_j + 1$ ways to hybridize x_j to x_i within the extent of x_i, and $(l_j - 3) * 2$ ways to hybridize x_j to x_i where x_j overhangs on either side, leading to $l_i + l_j - 5$ overall possible duplex reactions where we are assuming $l_j \leq l_i$. These orientation variations exist for all $\binom{n}{2}$ co-located species x_i, x_j, and for every x_i species alone where in the second case the sequence strings are identical but the resulting possible molecules are different in their secondary structures. For the purpose of quantifying cross-talk, *every* possible new molecule production is considered since we characterize a molecule using the sequence of the underlying nucleic acid bases, its secondary structure, and total base-count. Considering the expected value of potential hybridizations, and the unlimited size solution set S since we always have a possibility of a better solution by adding more species, cross-talk minimization is then the search for set S_k such that:

$$\min_{k \in S} \binom{n}{2}(l_i + l_j - 5) * \tfrac{1}{r_{ij}} + n(l_i + l_i - 5) * \tfrac{1}{r_i} :$$
$$|S_k| = n,$$
$$\forall \text{ strands } x_i, x_j \in S_k, \text{ with lengths of } l_i, l_j,$$
$$\text{and hybridization probabilities } \tfrac{1}{r_{ij}} \text{ (between strands), } \tfrac{1}{r_i} \text{ (intra-strand).}$$

Not only is it infeasible to examine all solution sets to find a minimal one, the cross-talk problem is also subtle because the classification of a reaction as off-nominal is not always straightforward. An off-nominal reaction might technically be an unwanted one, but if such a reaction occurs as an intermediate within a larger reaction event chain that eventually yields sufficient desired final species concentration production, then we might be inclined to say that an alternative acceptable reaction pathway exists. This observation gets at the heart of the cross-talk problem: we cannot know given current state-of-the-art experimental techniques what is actually happening, even for a trivially small system. No modality yet exists to precisely capture the full interplay of one or more DNA species in solution. This restricts all modeling and simulation approaches and makes validation difficult. Regardless of this constraint, attempting to understand a chemical reaction system remains a critical task. It is pursued here for the single purpose of quantifying unwanted species production for any arbitrary DNA strand set considered as a possible solution to a programming problem.

Approaches To Characterizing Cross-Talk

The cross-talk problem can be approached in two ways. The first approach is to avoid unwanted bindings by ensuring sequence assignments contain no reverse complementary subsequences that are not deliberately planned. This approach is the well-studied DNA codeword problem, and is solved in this work with Algorithm **Non-Intersecting Sequence Set**. Other codeword methods [8, 45] endeavor to choose sequences that possess minimal cross-talk between them, and may include thermodynamic property table look-ups to further predict binding formation. The second approach is worked out in the laboratory, through arranging reactions within a mechanism such that possible unwanted reactions are not particularly competitive. A further variation is a base by base modification iterated until results are acceptable [73], but this quickly becomes unsustainable in terms of time

and materials costs.

Our kinetics-based approach is to simulate the chemical reaction network and consider diffusive transport and hybridization reactions between all pairs of oligonucleotides concurrently present for determination of likely species production. This treatment assumes all possible hybridizations as not equally probable, consistent with physical reality, and in contrast to codeword methods which often make the opposite assumption. Furthermore, combinatorially generating all possible hybridizations and then subjecting them one at a time to thermodynamic modeling is computationally impractical because the number of combinations grows steeply even for small sets of initial reactants. Although a compiler program can be directed to look at all possible reactions, the subsequent output data *itself* is also too large and creates a different type of noise: design noise. It becomes impractical to separate useful findings from the entire data mass. Other researchers have previously described this situation as a "combinatorial explosion." Therefore, we simulate potential system designs as a way to infer the most probable combinations without paying the prohibitive modeling cost.

4.2.1 DNA Chemistry

DNA chemical reaction networks are reaction-diffusion systems. With the use of enzymes, there is a range of reaction rates that generally can be classified as either fast or slow. Use of enzymes entails a multi-step mechanism including enzyme-substrate binding, substrate cleavage, and enzyme recovery such that it is available to repeat the cycle with the next encountered substrate molecule. Use of strand-displacement entails a different multi-step mechanism wherein one oligonucleotide displaces another in an existing duplex. KCA covers a subset of the following DNA properties and interactive behaviors for measuring cross-talk, and provides a suitable foundation which can be developed further into a full-

fledged simulation of any arbitrary DNA chemical reaction network.

Physical Chemistry of Hybridization Reactions

A number of factors influence the kinetics of DNA hybridization, dictating how duplex formation nucleates and proceeds base-by-base in a zippering fashion to form a double helix between two single-stranded oligonucleotides, or a partial helix between any two regions that have unbound bases available for pairing. Duplex formation is not a continuous one-way event, and may involve intermediates that influence the on-rate reaction kinetics [83, 97]. Duplex stability, the tendency of a helical region to stay bound, is a function of denaturation off-rates which also may involve visiting intermediate forms. Additional kinetic rate dependencies are stem-length, loop size, sequence composition, and presence of a buffer salt such as NaCl [83]. Hybridization reaction rates slow in the presence of secondary structure [44], an important consideration since chemical reaction networks designed for non-trivial function all rely on precise formation of structure.

Physical Chemistry of Diffusion

Molecules in liquid are so close together that collisions are the billion to one dominant effect suffered by a single molecule moving in solution [14]. Yet collisions are not equivalent to reactions, and these collisions are part of the overall transport mechanism that moves molecules from higher to lower concentrations within local areas. Collision-induced motion creates diffusion as a spontaneous net flux process that proceeds as a result of following negative system free energy (ΔG). This free energy decrease is a natural consequence of systems moving towards equilibrium in absence of introduction of external energy sources. As a counter example, membrane-transport biochemistry in a cell must employ metabolic energy to do work in the form of moving a solute against a concentration

gradient [141].

Diffusion can be regarded as a random walk with z^2 spatial displacement for some position vector $\vec{z}$. It is expressed as a partial differential equation linking concentration n change in time t to concentration change in space z [98]:

$$\frac{\partial n}{\partial t} = D\frac{\partial^2 n}{\partial z^2} \tag{4.1}$$

The diffusion coefficient D has been measured experimentally for DNA and found to have dependence on length and topology [99], however these measurements are for homogeneous populations of DNA at a minimum of 6k base-pairs. This scale is not consistent with systems we aim to capture with KCA, and requires a molecular dynamics approach.

4.2.2 Simulation and Modeling Approaches

Techniques for kinetics-based capture of DNA interactions are classified as continuous and deterministic, discrete and stochastic, or continuous and stochastic [56]. Each reflects a degree of knowledge of the underlying physical chemistry factors, and a belief as to how they should be mirrored in an artificial version of real system activity.

Deterministic models assume each species type in the system is known *a priori*, and that it is quantitatively understood how the reactions they can undergo will proceed. These models also assume that each species can be adequately represented as continuous concentration variables in one or more rate equations, and that it is sufficient to ignore stochasticity of the physical system [40]. When species, rates, and all reactions are known, and the assumptions are acceptable, comprehensive simulation codes can evolve overall system behavior in time [60, 91].

Stochastic models vary in what is tracked in evolving the system over an artificial timeline using the same assumption of species types, and their possible reactions. System assessment is computed over the simulation time course using probabilistic decision making of which reactions will occur, and how diffusion will affect participants. An early stochastic model is that from McQuarrie who put forth what is now termed the *Chemical Master Equation* in 1967. This model defines a system as a set of N species $S_0,\ldots,S_{N-1}$ participating in M reactions $R_0,\ldots,R_{M-1}$, and tallies system state using a vector $X(t) = (X_0(t),X_1(t),\ldots,X_N(t))$ where each $X_i(t)$ counts the discrete number of species molecules S_i at time t [79]. A Bayesian probability function $P(X(t)|X(t-dt))$ advances the system at each time step by computing the likelihood of the system being found in some state $X(t)$ given only the known system state vector X_0 at initiating time t_0. The conditional probability makes use of an expected value of discrete species counts termed a propensity function. In practice, the method is impractical because the state space is too large.

Gillespie found a work-around to the state space problem with development of the *Stochastic Simulation Algorithm* in 1976 [48]. Gillespie's model instead computes realizations of $X(t)$ by sampling the M possible reactions using the propensity function and a Poisson distributed waiting time until a chosen one will occur [56]. Gillespie additionally contributed a version of the *Chemical Langevin Equation* [49] that computes N different probability distributions for each of the N species [56] in place of computing the probability distribution of all possible system states. As a continuous and stochastic approach, this model actively considers diffusive movement undertaken by system constituents through evolution of a set of stochastic differential equations intended to match the average and covariance of each species concentration to that computed by the Chemical Master Equation at any time point t [56, 80].

Within the camp of discrete and stochastic, inspiration for the KCA comes from [41] and [46]. Gerhardt and Schuster reported good simulation results of the Belousov-Zhabotinsky

reaction using a cellular automata rule they named the "hodge-podge machine" [46]. Belousov-Zhabotinsky reactions are non-linear oscillators that produce recurrent concentration maxima for two different products. These maxima trade off and persist in a non-equilibrium cycling state for long time periods. Their visually compelling spiral patterns mimic biochemical reactions, where local interaction effects across spatial and temporal scales confer gene regulation and organism adaptability in variable environments [41]. Weimar additionally used cellular automata to simulate enzymatic reaction networks with excellent reproduction of the standard Michaelis-Menten rate equations typically used to model enzymes [133]. The most significant use of a discrete simulation approach for biochemical systems has been in the area of random boolean networks pioneered by Kauffman [65].

4.2.3 KCA Implementation

A stochastic method with rule-based updating yields a straightforward simulation of chemical reaction networks. The typical reaction rate equation approaches fail in this context because there are too many possible reactions to anticipate, many of which are not part of any planned reaction sequence and for which we don't actually have a way to determine their rates. Additionally, rate equations are only approximate formalisms representing the most likely outcomes of their underlying stochastic systems, and are only accurate at a large scale [50, 56]. The milieu of intended and unintended reactions resemble a branching process describable as an n-ary tree where edges are labelled with the probability of proceeding from node to node and each node describes the state of the system. This conception is the same as the Chemical Master Equation with the key difference that the reaction set M is not written down ahead of time. Instead, we get to the branching process tree through dynamic discovery starting with an initial species population, and using

a stochastic update rule to either execute a reaction between co-located strands found to have reverse-complemented subsequences, or allow diffusive movement. Possible reactions are exhaustively identified with the chance of selection based on reactant discrete counts and kinetic, rather than thermodynamic, factors.

KCA is a 2.3K line C code. It accepts an input lattice dimension, and a vector of oligonucleotide species presented as sequence and nanomolar concentration pairs. The single grid dimension n is used to create an n x n lattice world of initially empty cells. Dynamic cell update follows a von Neumann neighborhood such that each cell is affected by its four neighbors as shown in Figure 4.2. As needed, movement between cells and neighbor look-up wraps around at the boundaries over the course of the simulation. Starting real-valued nanoscale concentrations for each input species are converted to discrete molecule counts, and then evenly distributed into the lattice cells via random selection to complete simulation initialization.

Each automaton cell acts as an independent reaction vessel such that system evolution is the sum effect of parallel update over all cells with non-zero discrete species counts. We employ the same abstract triple of (sequence, secondary structure, length), termed a strand in DNADL, to serve as a unique species. This means that two oligonucleotides with the same sequence and length but differing secondary structures are considered as two unique species. KCA maintains a global store of all system-wide species. Initially, these are just the input species, but upon simulation execution as reactions are fired and new species are created, the global store is updated to include them. KCA additionally manages a global store of all possible hybridization reactions by pair-wise examination of all species using the **FindAllHybrids** algorithm. As shown in Section 3.1.1., this algorithm determines all possible hybridization opportunities including complete or partial double helices that are at least three base-pairs in length. KCA combines opportunity information along with the

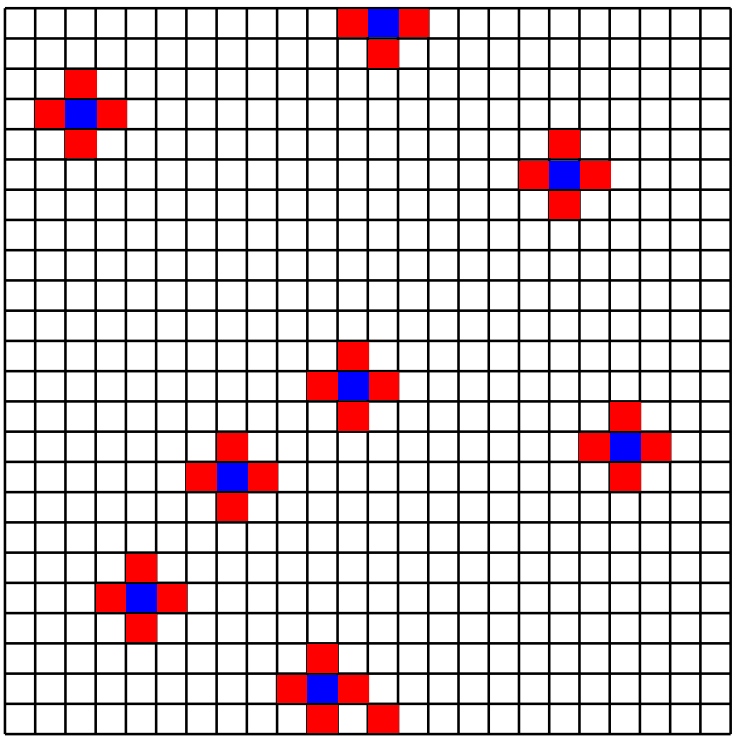


Figure 4.2: 24x24 KCA lattice. Each of the blue cells update based on the status of their directly adjacent red neighbor cells.

current structure status of each reactant to filter out cases where existing strand structure precludes a further hybridization because one or more of the required bases are not available. We recall that **FindAllHybrids** is a sequence-only algorithm, hence this extra step is necessary in the context of the simulation where species are more than just their sequences.

When a cell is selected for update, KCA consults the global reaction store along with the local population of species, and builds a possible local reaction set. All hybridization reactions are treated as bimolecular and thus there must be a count of at least one for each of the reactants. This decision is consistent with the observation that biological-based reactions are either monomolecular or biomolecular because collisions involving three or more molecules are unlikely [50, 141]. We allow the possibility of identical species

hybridizing (A + A $\rightarrow$ AA), or two different species hybridizing (A + B $\rightarrow$ AB), but do not include single species transformations such as self-folding. Monomolecular reactions are considered secondary effects in the hunt for new species production that may constitute cross-talk, and as well they are separately addressed through thermodynamic modeling in other parts of the Pyxis compiler program. In addition to the local possible reaction set, KCA treats diffusion as another possible event that may be randomly selected for execution as the cell update. Reassignment of species from one cell to a neighbor occurs in high-to-low fashion where the chosen neighbor results in the best overall local average count for that species. In cases where more than one neighbor has the same locally minimum count compared to the update cell, a random selection is made to break the tie.

All cells are examined and updated in a randomly reshuffled order with each automaton iteration to prevent location bias in diffusive movement and reaction firing. Similar to the Gillespie Stochastic Simulation Algorithm, a propensity to fire is computed for each reaction factoring in the discrete molecule count for each of the reactants, the number of bound bases, length of the shortest reactant strand, and sequence complexity of the local reactant strand. Diffusion propensity is computed based on the minimum discrete molecule count within the local area. The set of reaction propensities, and the diffusion propensity, are normalized together. A [0,1] uniform random number probabilisitically chooses which reaction fires, or if a diffusive move is made instead.

4.3 Pyxis

A chemical reaction network has structure, requires energy supplied as chemical species, and performs work by transforming reactants into products. These characteristics are consistent with the meaning of a machine. Within the network, at least one transformative trajectory is not incidental and produces output. A key distinction, which may in time be

eliminated, is that current verified reaction networks working in laboratories are single use only. They cannot be powered up and down or renewed in any way. They are constructed, and execute directed function a single time. Function may include computation [89, 96], diagnostic sensing and drug delivery [53, 75], data storage and playback [22, 51], or building devices for specific applications [30, 54]. In all these scenarios it is imperative to optimize set-up precision to better the opportunity for overall success.

A common need to achieve these goals and allow for scaling to greater functionality is to get away from special purpose or one-off design and manufacturing techniques. While other groups have gravitated towards graphical CAD-style drawing tools [31], the goal of Pyxis is ultimately to head in the opposite direction and focus on building an abstraction stack that can support synthesis testing and compiler function. Pyxis is currently a command line interaction-based 15K line Python application that originated as both a modeling framework and ensemble code to combine the results of multiple models.

4.3.1 Compiling DNA Systems

Traditional compilers transform programs written in high-level languages into machine code operable on specific architectures, hence we use the term compile loosely and make two observations. First, the analogies between silicon and molecular computing apply in some areas, but not all. Present-day molecular computing is akin to computer hardware engineering, and although the effort to arrange logic execution within a chemical network is computation, it is much closer to combinational circuit design. Second, the concept of compiling fits all forms of DNA chemical reaction network engineering, including those areas that do not specifically align as a computation, because the nature of a chemical reaction network remains consistent between application areas. Whether the goal is a motor, sensor, transporter, storage device, crystalline deposition template, or algorithm

executor, the goal must be written down and ultimately converted to DNA sequences and step-wise laboratory procedures. The utility of a compiler is to systematize design and build process steps, allow benchmarking, and provide a motivation for creating true high-level languages at a later date.

For consistency and building up an abstraction stack, compiler input requires a uniform method of encoding design specifications. The description language DNADL was created with this in mind. It functions analogously to hardware description languages, and therefore is not high-level. Instead, it is anticipated that a true high-level language will later be developed with DNADL taking on an universal instruction set role within Pyxis.

4.3.2 Pyxis Features

System design and architecture requires coupled search through sequence and structure spaces. Acting as a synthesizer, Pyxis exposes a full suite of sequence and structure evaluation methods to allow for by-hand testing and exploration. Acting as a compiler, Pyxis will conduct a search in a generation-evaluation loop using a subset of the sequence and structure methods, until application-specific results indicate threshold clearance of overall system functionality. For deoxyribozyme-based computing systems this entails assigning gates to formulas, designing sequences for gates and inputs, and evaluating gate/input/substrate interactions as a whole system.

Pyxis features include the following. Features have been grouped into six areas including 1) sequence and structure generation, 2) sequence analysis, 3) structure analysis, 4) whole system interaction analysis, 5) database interaction, and 6) utilities.

Sequence and Structure Generation

1. Exhaustive generation of all oligonucleotide sequence n-mers for any length n, or random sample generation from all possible n-mers for any length n.

2. Exhaustive generation of all secondary structures for an oligonucleotide of any length n, or random sample generation from all possible secondary structures for any length n oligonucleotide.

3. Generation of minimally reactive sequences using the **Non-Intersecting Sequence Set** algorithm.

4. Generation of all possible $n-k+1$ k-mer subsequences for a length n oligonucleotide sequence string.

5. Method to generate all deoxyribozyme gates from the original gate set (Table 1.1) for user-provided inputs.

6. Methods to generate all single input deoxyribozyme gates from the original gate set in high-throughput fashion (Table 1.1, page 8) using templates for the fixed domains and three different options for the loop recognition regions. The first option reads in 15-mers from a user supplied comma separated value (CSV) format file, the second option parses GenBank [2, 10] genome .gb files, and the third option parses FASTA input files. With options two and three, all possible gates are built from slicing parsed input into all possible 15-mers via a sliding window. Since gates are usually evaluated with inputs, the reverse complement of each 15-mer is inserted into the appropriate place within the gate subsequence string, and the original 15-mer is saved as an input. A variety of gate naming methods support user-directed names, or generation of unique names.

Sequence Analysis Features

1. Oligonucleotide sequence analysis using BioPython [1] library routines including GC-content, information content (as a measure of sequence complexity) [43, 68], melting temperature, and molecular weight.

2. Sequence edit distance computation between two sequences using the Hamming distance algorithm.

3. Sequence edit distance computation between two sequences using the Levenshtein distance algorithm [71]. This distance measure supersedes Hamming distance and is much closer in spirit to the conformational search played out by two oligonucleotides in a potential binding, therefore it is a better way to determine whether two strands will remain agnostic in solution together, or attempt to bind. A small study comparing the two distance measures for a selection of oligonucleotide pairs was undertaken to confirm this finding. DNA codeword approaches often only incorporate Hamming distance as a metric, and while this was appropriate for the original signal transmission problems it was invented for, it has less applicability for biological-based signals. Any length oligonucleotides are handled; for two sequences with lengths n and m respectively, the algorithm has $\mathrm{O}(nm)$ time and space usage.

4. A cost function to compute the range of oligonucleotide materials cost with and without purification. Cost factors including length and purification are based on supplier provided data.

Structure Analysis Features

1. A novel function, SelfFoldQuantification, to determine the tendency of a single oligonucleotide to self-fold. The algorithm was originally designed as a quick check for 15-mer input oligonucleotides and runs in $O(n^2)$ time and space for a length n sequence. The basis for the algorithm is the observation that where a sliding window comparison between a sequence and its reverse complement yields indices with identical values, all possible hybridization pairs within the sequence itself can be written down. This is a unique approach, and it is not the same thing as the n total pair bonds between a length n sequence string and its reverse complement. For example, the short sequence GTTGCA yields a self-fold pair count of 4, located at indices (0,4), (1,5), (2,5), and (3,4).

2. Two conversion functions to convert secondary structure dot-parenthesis strings to ISO strings, and ISO strings to dot-parenthesis strings.

3. A function to identify the largest binding region (longest stem) within a secondary structure.

4. Functions to report on the binding status of an oligonucleotide secondary structure including binding partner data, and lists of all bound and unbound bases.

5. Functions to identify and report on all secondary structure shapes such as upper and lower bulges, internal loops, and isolated stem-loops.

6. A function to identify and report on all secondary structure multibranches, including nested multibranches.

7. A function to identify hierarchical parent-child relationships between secondary structure sub-features.

8. A function to identify and report on all secondary structure pseudoknots.

9. A function to compute structure edit distance between two oligonucleotide secondary structures using a novel metric algorithm.

10. A function to compute structure base-weighted edit distance between two oligonucleotide secondary structures; algorithm performs a weighted base-by-base comparison, using user provided indication of which areas are important to match.

11. A function to compute secondary structure utility scores for the 8.17.1 Left Yes deoxyribozyme gate from the original gate catalog (Table 1.1) for the gate alone, and gate with input, modeling scenarios. The score is computed from a custom secondary structure evaluation rule set based on deoxyribozyme gate engineering laboratory experience. The complete evaluation scoring tables for all gates in the original catalog are shown in Appendix D, Deoxyribozyme Gate Evaluation Rules.

Whole System Interaction Analysis Features

1. A function to generate all possible Boolean expressions that can be built from the original deoxyribozyme logic connectives shown in Table 1.1, for any input literal count. Additional functions return subsets such as all positive literal expressions, all negative literal expressions, all *and-not* expressions, etc. A second variant function set generates random samples of the exhaustive sets.

2. Custom interface to the NUPACK Version 2.1 thermodynamic model, allowing up to four oligonucleotides to be tested together. NUPACK comprises eleven separate but related executables listed below. The web-based version was useful in verifying that no errors were introduced in interfacing to the direct source code within Pyxis; however, to support high-throughput analysis of potentially millions of oligonucleotide

combinations as part of large-scale studies, the web interface was not realistic. Additionally, an error in the web version was discovered as part of a special study done to understand the impact of setting a flag for inclusion of dangling-end energy contributions.

(a) pfunc, calculation of the partition function

(b) pairs, calculation of base-pairing observables

(c) mfe, determination of the minimum free energy (MFE) structure

(d) subopt, determination of all secondary structures within a specified free energy gap of the MFE

(e) count, determination of the total number of structures found in the ensemble

(f) energy, calculation of free energy and secondary structure for a particular sequence

(g) prob, calculation of the equilibrium probability of a particular secondary structure

(h) defect, calculation of a measure "ensemble defect" defined as the average number of incorrectly paired nucleotides evaluated over the ensemble

(i) complexes, calculation of the partition function of all complexes up to a specified size

(j) concentrations, calculation of complex concentrations

(k) distributions, calculation of complex population distribution

3. Incorporation of Python ctypes library code to speed up execution of NUPACK modeling. NUPACK is a C program that is input/output bound through use of small, temporarily used input files. Input type handling and a direct interface into the main

NUPACK modeling functions bypasses the out-of-the-box program organization resulting in run-time reduction of at least 60%.

4. A combinatorial strategy to handle all unique strand orderings for NUPACK testing. The underlying dynamic programming algorithm incorporated within NUPACK orders oligonucleotides based on user input, yet the documentation notes that non-circularly related orderings will yield different answers. For example, in a two strand test order A-B yields equivalent results to order B-A, but in a three strand test orders A-B-C and A-C-B may yield different results. Pyxis determines all unique orderings, and automatically tests each one. NUPACK results are collected to find the true MFE structure, average reported MFE, etc., for all tested unique orderings.

5. Two test matrix submission methods for submitting NUPACK model executions. The first reads inputs directly from a CSV file containing user directed combinations such as a gate tested alone, a gate and its designated input, a gate and a substrate, or a gate, input, and substrate together. The second method allows for several variations of exhaustive testing by automatically generating the following four sets of length n subsequences:

   (a) A random sample from a set of all possible n-mers, up to all n elements, with handling to prevent duplicate n-mers.

   (b) The set of all possible n-mers.

   (c) A single specific n-mer.

   (d) A range of n-mers, starting with a specific one, where remaining n-mers follow by virtue of a custom encoding within a Kyoto Cabinet database.

6. Integrated handling of NUPACK subopt executable output. Supoptimal modeling returns tens to thousands of all predicted secondary structures and their free energies within an energy gap above that of the minimum free energy (MFE) secondary

structure. Each reported free energy is converted into a probability of occurrence and is combined with a secondary structure evaluation to compute an overall expectation value. Structure evaluation is computed using either the custom rules shown in Appendix D, or a structure edit difference result between a reported secondary structure and an ideal template form using one or both of the structure edit distance functions.

Database Interaction

1. An interface to an early version of an oligonucleotide library holding gate and input sequences is available such that arbitrary combinations of stored gates and inputs can be tested together. The library is set up as an PostgreSQL [3] database.

2. A general PostgreSQL database interface is incorporated. Methods have been written for new table creation, row insert or update, multiple extraction variations, as well as CSV format input and output.

3. Exhaustive or random generation of oligonucleotide n-mers may be stored and retrieved from a DBM-based database, Kyoto Cabinet [57]. Kyoto Cabinet is written in C++ and has a Python interface; the code allows for customization of how the underlying name-value data records are stored. Currently the B+-tree option with O($log n$) operation cost is in use, with additional internal optimization to facilitate sequential access useful for exhaustive combinatorial test matrix execution.

Utilities

1. Standard command line argument parsing using the Python argparse module for a UNIX-like command line user experience.

2. An array of output reporting into CSV files is available. The CSV format was initially chosen to provide ease of use when moving reports into an Excel spreadsheet, and over time has proven to be a highly practical format overall in the handling of copious amounts of data.

3. A custom set of methods rewritten in Python such as the UNIX cut function for CSV files, and various functional language list processing techniques.

4. Methods to compute basic statistical measures such as average, standard deviation, minimum and maximum.

5. Output plot creation using the R Project for Statistical Computing [4] library. R is a free software environment and has grown in popularity, due to its comprehensive feature set, to the point of supporting its own refereed journal. Scatter plots, sunflower scatter plots (a technique borrowed from astrophysics studies to handle $>$1M points), bar charts, and 3D surface plots may be produced.

Chapter 5

Conclusion

I would like to describe a field, in which little has been done, but in which an enormous amount can be done in principle. – Richard P. Feynman [39]

The central thesis question has been answered constructively. A summary of contributions to the fields of DNA nanotechnology and genomics are the following conceptions, original ideas, and development. Artifacts include novel abstractions, algorithms, methods, a language, a simulation, and a DNA compiler program, as documented in this thesis. It is hoped that these efforts will further the fields of nanotechnology and genomics such that Feynman gets his wish.

There are many directions that this research project can develop further, pending a successful outcome in the highly competitive research and development grant arena. These directions include follow-on development of DNADL, KCA, and Pyxis, to ultimately bring all three together into a single package published for distribution and mainstream use. Continued laboratory validation and interaction is the single best way to achieve an eventual UNM DNA nanotechnology software package. Efforts in this direction, over the course of

project development, include (1) initial reconstruction of the MAYA II experiments [34], (2) multiple detailed evaluations of deoxyribozyme gate designs for detection of six different flaviviruses (Murray Valley, Koutango, Japanese Encephalitis, St. Louis, Usutu, and West Nile), (3) creation and maintenance of a virus detection platform development intranet, the *McogVirusDetection Web*, (4) training and one-on-one assistance for Columbia University laboratory personnel experiencing intranet usage and interaction difficulties, (5) creation of multiple proposals for an Oligonucleotide Library, also documented in the McogVirusDetection Web, and (6) internal Pyxis versions that have contributed to successful laboratory outcomes at the Columbia University and UNM laboratories [18, 92].

5.1 Contributions of Thesis

DNA chemistry has been used in the construction of a variety of devices and machines, many of which are recapitulations of what was achieved in silicon decades ago. For sensing and computing applications, the current technological capability is at the level of propositional logic and rudimentary circuits. For building nanoscale templates or transport modules, DNA origami and branched lattice techniques are able to build nearly any two dimensional shape and should shortly achieve any three dimensional shape. These accomplishments have largely been won by getting very good at hacking the bare metal, or in this case, precise designs to guide the chemistry to specific ends, not just specific yields. The goal of this thesis has been to build on the pioneering work accomplished by others in the field, and enable further achievements in scale with commensurate decrease in development time and materials costs.

1. **Abstract numeric representation of nucleic acid sequences**. Binary representation of sequences is not new, however, encoding of base-pairing inference as a

bit-level operation is. There are obvious extensions to this theme which enable high-throughput sequence motif search, one of the most frequently required low-level tasks in mainstream genomics, particularly in identifying gene coding regions and function for newly sequenced organisms.

2. **Abstract numeric representation of nucleic acid secondary structure.** This contribution is the first structure representation that is completely numeric. As a regular language it is superior to competing approaches [38, 47, 59] in measures of simplicity, efficiency, and analytical ability. We show a linear time algorithm to convert dot-parenthesis string representation [59] into ISO that makes ISO compatible with thermodynamic modeling codes.

3. **Abstract representation of reactions.** Reactions are abstracted in the style of functional programming where reactants and products are arguments and function application is chemical transformation. This idea is deeper than what has been explored in this thesis. Cheminformatics, and the enumeration of chemical reactions are old and venerable research areas. As a contrast to the various graph formalisms for reactions, a functional programming treatment marries the underlying concept of recursion to chemistry which may yield new insights.

4. **Sequence algorithms.** Both algorithms are enabled by the numeric representation of sequences and are first-ever algorithms that solve these enumeration and search problems through exploitation of bit-level operations and bitstrings for sequence strings.

    (a) **Find All Hybrids.** Dirks [26] reports $O(n^4)$ time and $O(n^3)$ space resource usage to consider formation of the complete ensemble of secondary structures for a length n oligonucleotide. Shapiro [114] reports $O(n^3)$ time. This algorithm uses $O(n^3)$ time and space resource usage.

(b) **Non-Intersecting Sequence Set**. The algorithm differs from competitors in two distinct ways. First, there is no dependence on thermodynamic lookup tables [8, 45, 63], yet results show confirmation of minimal hybridization when separately tested with the NUPACK thermodynamic modeling code [139]. Second, all variations of possible hybridization are tested, which supersedes methods that only use Hamming distance such as [45, 63]. Because this approach is solely based on sequence strings, it can be adapted to any other search problem with the same $\Theta(4^n)$ space.

5. **Structure algorithms.** The structure algorithms are unique. The RNA as Graphs approach [38] dispenses with stem-lengths and unpaired regions, thus cannot be used to track basic bound/unbound state in a secondary structure. Giegerich [47] introduces a term "shreps" as a descriptor of shape, but fails to give resource analysis of grammar-based secondary structure inference. To the best of our knowledge, there are no representation methodologies or analytical tools able to accomplish all aspects of ISO and the related algorithms below. Resource usage analysis as a function of the number of ISO triples maps to usage in the length of an oligonucleotide as shown below since a linear time step occurs prior to filtering to convert dot-parenthesis strings into ISO. Other basic representations, such as just a list of paired bases, will also only pay linear time in conversion to ISO since each pair only must be examined once.

   (a) **Dot-Parenthesis to ISO Conversion.** Enables direct parsing of thermodynamic modeling output which can then serve as input into one or more of the shape filters. Algorithm is linear time.

   (b) **ISO to Dot-Parenthesis Conversion.** The reverse conversion algorithm is added for completeness. Algorithm is linear time.

(c) **Stem-Loop Inference.** This basic algorithm identifies all stem-loops. A possible genomics application is high-throughput screening of microRNAs which have been implicated as biomarkers for certain cancers [78]. Algorithm is linear time.

(d) **Hairpin Filter.** Hairpins are often given the same meaning, a stem with a loop of unpaired bases, as our classification of stem-loop in this work. The distinction given here is important for synthetic DNA constructions that may use this shape without intervening bases. Algorithm is linear time.

(e) **Bulge Filter.** Linear time algorithm to infer bulges.

(f) **Internal Loop Filter.** Linear time algorithm to infer internal loops.

(g) **Parent-Child Determination.** Hierarchical relationship inference between stems that handles any degree of nesting in quadratic time.

(h) **R-Way Multibranch Inference.** Filter for multibranches that supports any degree of nesting in quadratic time.

(i) **Pseudoknot Filter.** Linear time algorithm to infer pseudoknots. We know of no other representation capable of handling pseudoknots as a simple check.

(j) **Bindings, Unbound, Partners.** Three linear time algorithms to characterize bound and unbound regions in an oligonucleotide.

(k) **Multitriples Exhaustive.** A straightforward cubic time algorithm that writes out all legal individual stems for an oligonucleotide. It is intuited, but not yet proven, that this approach will lead to simpler expressions for structure enumeration. As a construction algorithm it naturally gives an ordering to all individual possible stem regions.

6. **The DNA description language (DNADL)**. A design language for architecting molecular computing devices constructed from DNA. The language formalizes the

low-level attributes and behaviors required to achieve programmed function. DNADL instances act as system requirements documents and can both serve as a target of higher-level languages and as a parsable file for compiler program input.

7. **The Kinetic Cellular Automaton (KCA) kinetics-based simulation.** A discrete stochastic DNA simulation that determines likely DNA species which can form from initial species at nanoscale concentrations. A Gillespie-style propensity function comprising kinetic factors such as length, number of bound bases and concentration of reactants is employed to make probabilistic decisions at each time step to either fire a reaction, or make a diffusive move, for each cell within a cellular automaton.

8. **The Pyxis synthesizer and compiler program.** A fully featured design and analysis program for DNA systems comprising a number of specialized sequence and structure algorithms, reporting tools, database interaction capability, and a custom high-throughput port of the NUPACK thermodynamic modeling program. The program admits addition of modules to expand further physico-chemical modeling and simulation studies of DNA properties and interactions.

Appendix A

Deoxyribozyme Gate Catalog

A.1 Gate Catalog

A.1.1 Gate Schematics

8.17 LEFT YES

5’ |— stem (10 nt) —|— loop (15 nt) —|— stem (10 nt) —|— core (20 nt) —| 3’, gate

5’ |— (15 nt) —| 3’, activating input

5’ |— (17 nt) —| 3’, substrate

8.17 RIGHT YES

5’ |— core (20 nt) —|— stem (10 nt) —|— loop (15 nt) —|— stem (10 nt) —| 3’, gate

5’ |— (15 nt) —| 3’, activating input

5’ |— (17 nt) —| 3’, substrate

E6 LEFT YES

5’ |— stem (7 nt) —|— loop (15 nt) —|— stem (7 nt) —|— core (29 nt) —| 3’, gate

5’ |— (15 nt) —| 3’, activating input

5’ |— (15 nt) —| 3’, substrate

E6 RIGHT YES

5’ |— core (28 nt) —|— stem (8 nt) —|— loop (15 nt) —|— stem (8 nt) —| 3’, gate

5’ |— (15 nt) —| 3’, activating input

5’ |— (17 nt) —| 3’, substrate

E6 NOT

5’ |— (12 nt) —|— stem (5 nt) —|— loop (15 nt) —|— stem (5 nt) —|— (15 nt) —| 3’, gate

5’ |— (15 nt) —| 3’, inhibitory input

5’ |— (17 nt) —| 3’, substrate

8.17 AND

5’ |— stem (10 nt) —|— loop (15 nt) —|— stem (10 nt) —|— core (10 nt) —|
|— stem (10 nt) —|— loop (15 nt) —|— stem (10 nt) —| 3’, gate

5’ |— (15 nt) —| 3’, activating input a

5’ |— (15 nt) —| 3’, activating input b

5’ |— (17 nt) —| 3’, substrate

E6 AND

5’ |— stem (8 nt) —|— loop (15 nt) —|— stem (8 nt) —|— core (20 nt) —|
|— stem (8 nt) —|— loop (15 nt) —|— stem (8 nt) —| 3’, gate

5’ |— (15 nt) —| 3’, activating input a

5’ |— (15 nt) —| 3’, activating input b

5’ |— (15 nt) —| 3’, substrate

E6 AND-NOT

5’ |— stem (7 nt) —|— loop (15 nt) —|— stem (7 nt) —|— (7 nt) —|
|— stem (5 nt) —|— loop (15 nt) —|— stem (5 nt) —|— (15 nt) —| 3’, gate

5’ |— (15 nt) —| 3’, activating input

5’ |— (15 nt) —| 3’, inhibitory input

5’ |— (15 nt) —| 3’, substrate

E6 AND-AND-NOT

5’ |— stem (8 nt) —|— loop (15 nt) —|— stem (8 nt) —|— (4 nt) —|
|— stem (5 nt) —|— loop (15 nt) —|— stem (5 nt) —|— (7 nt) —|
|— stem (8 nt) —|— loop (15 nt) —|— stem (8 nt) —| 3’, gate

5’ |— (15 nt) —| 3’, activating input a

5’ |— (15 nt) —| 3’, activating input b

5’ |— (15 nt) —| 3’, inhibitory input

5’ |— (15 nt) —| 3’, substrate

A.1.2 Gate Sequence Specifications

Substrate Sequences																	
	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
8.17.1	T	A	G	T	A	A	C	T	rA	G	A	G	A	T	C	A	T
E6	T	C	A	C	T	A	T	rA	G	G	A	A	G	A	G		

Table A.1: Sequences listed 5' to 3' for substrates that recognize the 8.17.1 and E6 enzymes.

8.17.1 Left Yes Gate Sequence														
0 G	1 G	2 A	3 A	4 G	5 A	6 T	7 C	8 A	9 T	10 a	11 a	12 a	13 a	14 a
15 a	16 a	17 a	18 a	19 a	20 a	21 a	22 a	23 a	24 a	25 A	26 T	27 G	28 A	29 T
30 C	31 T	32 T	33 C	34 C	35 G	36 A	37 G	38 C	39 C	40 G	41 G	42 T	43 C	44 G
45 A	46 A	47 A	48 G	49 T	50 T	51 A	52 C	53 T	54 A					

Table A.2: Sequence listed 5’ to 3’ for the 8.17.1 Left Yes Deoxyribozyme Logic Gate.

8.17.1 Right Yes Gate Sequence														
0 A	1 T	2 G	3 A	4 T	5 C	6 T	7 T	8 C	9 C	10 G	11 A	12 G	13 C	14 C
15 G	16 G	17 T	18 C	19 G	20 A	21 A	22 A	23 G	24 T	25 T	26 A	27 C	28 T	29 A
30 a	31 a	32 a	33 a	34 a	35 a	36 a	37 a	38 a	39 a	40 a	41 a	42 a	43 a	44 a
45 T	46 A	47 G	48 T	49 A	50 A	51 C	52 T	53 T	54 T					

Table A.3: Sequence listed 5' to 3' for the 8.17.1 Right Yes Deoxyribozyme Logic Gate.

E6 Left Yes Gate Sequence														
0 T	1 G	2 A	3 A	4 G	5 A	6 G	7 a	8 a	9 a	10 a	11 a	12 a	13 a	14 a
15 a	16 a	17 a	18 a	19 a	20 a	21 a	22 C	23 T	24 C	25 T	26 T	27 C	28 A	29 G
30 C	31 G	32 A	33 T	34 G	35 G	36 C	37 G	38 A	39 A	40 G	41 C	42 C	43 C	44 A
45 C	46 C	47 C	48 A	49 T	50 G	51 T	52 T	53 A	54 G	55 T	56 G	57 A		

Table A.4: Sequence listed 5’ to 3’ for the E6 Left Yes Deoxyribozyme Logic Gate.

E6 Right Yes Gate Sequence														
0 C	1 T	2 C	3 T	4 T	5 C	6 A	7 G	8 C	9 G	10 A	11 T	12 G	13 G	14 C
15 G	16 A	17 A	18 G	19 C	20 C	21 C	22 A	23 C	24 C	25 C	26 A	27 T	28 G	29 T
30 T	31 A	32 G	33 T	34 G	35 A	36 a	37 a	38 a	39 a	40 a	41 a	42 a	43 a	44 a
45 a	46 a	47 a	48 a	49 a	50 a	51 T	52 C	53 A	54 C	55 T	56 A	57 A	58 C	

Table A.5: Sequence listed 5' to 3' for the E6 Right Yes Deoxyribozyme Logic Gate.

E6 Not Gate Sequence														
0 C	1 T	2 C	3 T	4 T	5 C	6 A	7 G	8 C	9 G	10 A	11 T	12 G	13 A	14 C
15 T	16 G	17 a	18 a	19 a	20 a	21 a	22 a	23 a	24 a	25 a	26 a	27 a	28 a	29 a
30 a	31 a	32 C	33 A	34 G	35 T	36 C	37 C	38 A	39 C	40 C	41 C	42 A	43 T	44 G
45 T	46 T	47 A	48 G	49 T	50 G	51 A								

Table A.6: Sequence listed 5' to 3' for the E6 Not Deoxyribozyme Logic Gate.

8.17.1 And Gate Sequence														
0 G	1 G	2 A	3 A	4 G	5 A	6 T	7 C	8 A	9 T	10 a	11 a	12 a	13 a	14 a
15 a	16 a	17 a	18 a	19 a	20 a	21 a	22 a	23 a	24 a	25 A	26 T	27 G	28 A	29 T
30 C	31 T	32 T	33 C	34 C	35 G	36 A	37 G	38 C	39 C	40 G	41 G	42 T	43 C	44 G
45 A	46 A	47 A	48 G	49 T	50 T	51 A	52 C	53 T	54 A	55 b	56 b	57 b	58 b	59 b
60 b	61 b	62 b	63 b	64 b	65 b	66 b	67 b	68 b	69 b	70 T	71 A	72 G	73 T	74 A
75 A	76 C	77 T	78 T	79 T										

Table A.7: Sequence listed 5’ to 3’ for the 8.17.1 And Deoxyribozyme Logic Gate.

E6 And Gate Sequence														
0 C	1 T	2 G	3 A	4 A	5 G	6 A	7 G	8 a	9 a	10 a	11 a	12 a	13 a	14 a
15 a	16 a	17 a	18 a	19 a	20 a	21 a	22 a	23 C	24 T	25 C	26 T	27 T	28 C	29 A
30 G	31 C	32 G	33 A	34 T	35 G	36 G	37 C	38 G	39 A	40 A	41 G	42 C	43 C	44 C
45 A	46 C	47 C	48 C	49 A	50 T	51 G	52 T	53 T	54 A	55 G	56 T	57 G	58 A	59 b
60 b	61 b	62 b	63 b	64 b	65 b	66 b	67 b	68 b	69 b	70 b	71 b	72 b	73 b	74 T
75 C	76 A	77 C	78 T	79 A	80 A	81 C								

Table A.8: Sequence listed 5’ to 3’ for the E6 And Deoxyribozyme Logic Gate.

E6 Left AndNot Gate Sequence														
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
T	G	A	A	G	A	G	a	a	a	a	a	a	a	a
15	16	17	18	19	20	21	22	23	24	25	26	27	28	29
a	a	a	a	a	a	a	C	T	C	T	T	C	A	G
30	31	32	33	34	35	36	37	38	39	40	41	42	43	44
C	G	A	T	G	A	C	T	G	b	b	b	b	b	b
45	46	47	48	49	50	51	52	53	54	55	56	57	58	59
b	b	b	b	b	b	b	b	b	C	A	G	T	C	C
60	61	62	63	64	65	66	67	68	69	70	71	72	73	74
A	C	C	C	A	T	G	T	T	A	G	T	G	A	

Table A.9: Sequence listed 5’ to 3’ for the E6 Left AndNot Deoxyribozyme Logic Gate.

E6 Right AndNot Gate Sequence														
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
C	T	C	T	T	C	A	G	C	G	A	T	G	A	C
15	16	17	18	19	20	21	22	23	24	25	26	27	28	29
T	G	b	b	b	b	b	b	b	b	b	b	b	b	b
30	31	32	33	34	35	36	37	38	39	40	41	42	43	44
b	b	C	A	G	T	C	C	A	C	C	C	A	T	G
45	46	47	48	49	50	51	52	53	54	55	56	57	58	59
T	T	A	G	T	G	A	a	a	a	a	a	a	a	a
60	61	62	63	64	65	66	67	68	69	70	71	72	73	74
a	a	a	a	a	a	a	T	C	A	C	T	A	A	C

Table A.10: Sequence listed 5’ to 3’ for the E6 Right AndNot Deoxyribozyme Logic Gate.

E6 AndAndNot Gate Sequence														
0 C	1 T	2 G	3 A	4 A	5 G	6 A	7 G	8 a	9 a	10 a	11 a	12 a	13 a	14 a
15 a	16 a	17 a	18 a	19 a	20 a	21 a	22 a	23 C	24 T	25 C	26 T	27 T	28 C	29 A
30 G	31 C	32 G	33 A	34 T	35 G	36 A	37 C	38 T	39 G	40 c	41 c	42 c	43 c	44 c
45 c	46 c	47 c	48 c	49 c	50 c	51 c	52 c	53 c	54 c	55 C	56 A	57 G	58 T	59 C
60 C	61 A	62 C	63 C	64 C	65 A	66 T	67 G	68 T	69 T	70 A	71 G	72 T	73 G	74 A
75 b	76 b	77 b	78 b	79 b	80 b	81 b	82 b	83 b	84 b	85 b	86 b	87 b	88 b	89 b
90 T	91 C	92 A	93 C	94 T	95 A	96 A	97 C							

Table A.11: Sequence listed 5’ to 3’ for the E6 AndAndNot Deoxyribozyme Logic Gate.

A.1.3 Gate Structure Specification

Gate Identifier	**Input Count**	**Length**	**Secondary Structure**
8.17 Left Yes	1	55	[(0,10,15),(32,2,5)]
8.17 Right Yes	1	55	[(8,3,3),(20,10,5)]
E6 Left Yes	1	58	[(0,7,15),(34,3,3),(43,3,8)]
E6 Right Yes	1	59	[(12,3,3),(28,8,15)]
E6 Not	1	52	[(12,5,15),(37,3,8)]
8.17 And	2	80	[(0,10,15),(38,2,0),(45,10,15)]
E6 And	2	82	[(0,8,15),(35,3,3),(51,8,15)]
E6 Left AndNot	2	74	[(0,7,15),(34,5,15)]
E6 Right AndNot	2	75	[(12,5,15),(44,8,15)]
E6 AndAndNot	3	98	[(0,8,15),(35,5,15),(67,8,15)]

Table A.12: Expected structure for all deoxyribozyme logic gates in absence of input or substrate.

A.1.4 Gate Stem-Loop Specification

Stem-Loop Loop Identifier	**Start Index**	**Stop Index**	**Stem Length**	**Loop Opening**	**Extent**
8.17 Left Yes Activating A	0	34	10	15	35
8.17 Right Yes Activating A	20	54	10	15	35
E6 Left Yes Activating A	0	28	7	15	29
E6 Right Yes Activating A	28	58	8	15	31
E6 Not Inhibitory A	12	36	5	15	25
8.17 And Activating A	0	34	10	15	35
8.17 And Activating B	45	79	10	15	35
E6 And Activating A	0	30	8	15	31
E6 And Activating B	51	81	8	15	31
E6 Left AndNot Activating A	0	28	7	15	29
E6 Left AndNot Inhibitory B	34	58	5	15	25
E6 Right AndNot Activating A	44	74	8	15	31
E6 Left AndNot Inhibitory B	34	58	5	15	25
E6 Right AndNot Activating A	44	74	8	15	31
E6 Right AndNot Inhibitory B	12	36	5	15	25
E6 AndAndNot Activating A	0	30	8	15	31
E6 AndAndNot Activating B	67	97	8	15	31
E6 AndAndNot Inhibitory C	35	59	5	15	25

Table A.13: Expected stem-loop formation for all deoxyribozyme logic gates in absence of input or substrate.

A.1.5 Gate-Input Binding Structure Specification

Gate Identifier	Inputs	Length
8.17 Left Yes	act a	71
8.17 Right Yes	act a	71
E6 Left Yes	act a	74
E6 Right Yes	act a	75
E6 Not	inh a	68
8.17 And	act a, act b	112
E6 And	act a, act b	114
E6 Left AndNot	act a, inh b	106
E6 Right AndNot	act a, inh b	108
E6 AndAndNot	act a, act b, inh c	146

Table A.14: Gates, inputs and total complex lengths for all deoxyribozyme logic gates bound to programmed input.

Gate Identifier	Ordering	Secondary Structure
8.17 Left Yes	(5')g+(3')ia	[(10,15,31)]
8.17 Right Yes	(5')g+(3')ia	[(30,15,11)]
E6 Left Yes	(5')g+(3')ia	[(7,15,37)]
E6 Right Yes	(5')g+(3')ia	[(36,15,9)]
E6 Not	(5')g+(3')ia	[(17,15,21)]
8.17 And	(5')g+(3')ia+(3')ib	[(10,15,56),(55,15,27)]
E6 And	(5')g+(3')ia+(3')ib	[(8,15,60),(59,15,25)]
E6 Left AndNot	(5')g+(3')ia+(3')ib	[(7,15,53),(39,15,37)]
E6 Right AndNot	(5')g+(3')ib+(3')ia	[(17,15,44),(52,15,25)]
E6 AndAndNot	(5')g+(3')ia+(3')ic+(3')ib	[(8,15,76),(40,15,60),(75,15,41)]

Table A.15: Expected structure, per given strand ordering, for all deoxyribozyme logic gates bound to programmed input.

A.1.6 Gate-Input Binding Stem-Loop Specification

Stem-Loop Loop Identifier	**Start Index**	**Stop Index**	**Stem Length**	**Loop Opening**	**Extent**
8.17 Left Yes Activating A	10	70	15	31	61
8.17 Right Yes Activating A	30	70	15	11	41
E6 Left Yes Activating A	7	73	15	37	67
E6 Right Yes Activating A	36	74	15	9	39
E6 Not Inhibitory A	17	67	15	21	51
8.17 And Activating A	10	95	15	56	86
8.17 And Activating B	55	111	15	27	57
E6 And Activating A	8	97	15	60	90
E6 And Activating B	59	113	15	25	55
E6 Left AndNot Activating A	7	89	15	53	83
E6 Left AndNot Inhibitory B	39	105	15	37	67
E6 Right AndNot Activating A	52	106	15	25	55
E6 Right AndNot Inhibitory B	17	90	15	44	74
E6 AndAndNot Activating A	8	113	15	76	106
E6 AndAndNot Activating B	75	145	15	41	71
E6 AndAndNot Inhibitory C	40	129	15	60	90

Table A.16: Expected stem-loop formation for all deoxyribozyme logic gates bound to programmed input.

A.1.7 Gate-Substrate Structure Specification

Gate Identifier	Substrate	Ordering	Length	Secondary Structure
8.17 Left Yes	8.17.1	(5')g+(3')s	73	[(25,8,23),(47,8,10)]
8.17 Right Yes	8.17.1	(5')g+(3')s	73	[(0,8,48),(22,8,35)]
E6 Left Yes	E6	(5')g+(3')s	74	[(22,6,31),(52,6,10)]
E6 Right Yes	E6	(5')g+(3')s	75	[(0,6,54),(30,6,33)]
E6 Not	E6	(5')g+(3')s	68	[(0,6,47),(46,6,10)]
8.17 And	8.17.1	(5')g+(3')s	96	[(25,8,48),(47,8,35)]
E6 And	E6	(5')g+(3')s	98	[(23,6,54),(53,6,33)]
E6 Left AndNot	E6	(5')g+(3')s	90	[(22,6,47),(68,6,10)]
E6 Right AndNot	E6	(5')g+(3')s	92	[(0,6,70),(46,6,33)]
E6 AndAndNot	E6	(5')g+(3')s	114	[(23,6,70),(69,6,33)]

Table A.17: Expected gate-substrate binding for all deoxyribozyme logic gates.

Stem-Loop Loop Identifier	**Start Index**	**Stop Index**	**Stem Length**	**Loop Opening**	**Extent**
8.17 Left Yes Docking Arm A	25	63	8	23	39
8.17 Left Yes Docking Arm B	47	72	8	10	26
8.17 Right Yes Docking Arm A	0	63	8	48	64
8.17 Right Yes Docking Arm B	22	72	8	35	51
E6 Left Yes Docking Arm A	22	64	6	31	43
E6 Left Yes Docking Arm B	52	73	6	10	22
E6 Right Yes Docking Arm A	0	65	6	54	66
E6 Right Yes Docking Arm B	30	74	6	33	45
E6 Not Docking Arm A	0	58	6	47	59
E6 Not Docking Arm B	46	67	6	10	22
E6 Not Docking Arm A	0	58	6	47	59
E6 Not Docking Arm B	46	67	6	10	22
8.17 And Docking Arm A	25	88	8	48	64
8.17 And Docking Arm B	47	97	8	35	51
E6 And Docking Arm A	23	88	6	54	66
E6 And Docking Arm B	53	97	6	33	45
E6 Left AndNot Docking Arm A	22	80	6	47	59
E6 Left AndNot Docking Arm B	68	89	6	10	22
E6 Right AndNot Docking Arm A	0	81	6	70	82
E6 Right AndNot Docking Arm B	46	90	6	33	45
E6 AndAndNot Docking Arm A	23	104	6	70	82
E6 AndAndNot Docking Arm B	69	113	6	33	45

Table A.18: Expected gate-substrate binding stem-loop specifications for all deoxyribozyme logic gates.

A.2 Reactions

Using standard chemical reaction notation where reactant species listed on the left side of an arrow are converted into product species listed on the right side, the relevant reactions for the basic set of deoxyribozyme logic gates are shown. There are several reversible reactions; in physical terms this corresponds to hybridization proceeding in the left-to-right

direction, and dissociation proceeding in the right-to-left direction. Inputs I are indexed as a, b, and c. Since their sequence reverse complements are part of the gate sequences, gates have been similarly named with the addition of an overbar to indicate the reverse complementarity status of the input sequence(s) they are designed to recognize. For the signalling reactions, substrate S is cleaved to form products P_f (fluorophore) and P_q (quencher).

A.2.1 Reactions for gates with only positive inputs.

When input is added to a solution containing gates and substrates, three reactions commence: a) gates and inputs reversibly bind together, b) gate-input complexes reversibly bind to substrates, and c) substrates are cleaved into products. We assume the binding k_{on} and k_{off} rates in the first two steps are the same. Since the last step recovers the gate-input complex, it acts as a catalyst and is able to either bind to and cleave other substrates, or fall apart. Accumulation of P_f allows fluorescence build-up for the *on*-state signal. In the case of the AND gate, if only one of the required inputs is introduced, the opposing stem loop remains intact and therefore half of the substrate binding region remains sequestered. This prevents correct gate-input-substrate complex formation, and no signal is produced.

Gate	Reactions
YES	(1) $G_{\overline{a}} + I_a \underset{k_{off}}{\overset{k_{on}}{\rightleftharpoons}} G_{\overline{a}}I_a$ (2) $G_{\overline{a}}I_a + S \underset{k_{off}}{\overset{k_{on}}{\rightleftharpoons}} G_{\overline{a}}I_aS$ (3) $G_{\overline{a}}I_aS \xrightarrow{k_{cat}} G_{\overline{a}}I_a + P_f + P_q$

Table A.2.1.1: Reaction sequence for the YES gate that requires a single positive input.

Gate	Reactions
AND	(1) $G_{\overline{ab}} + I_a + I_b \underset{k_{off}}{\overset{k_{on}}{\rightleftharpoons}} G_{\overline{ab}} I_a I_b$ (2) $G_{\overline{ab}} I_a I_b + S \underset{k_{off}}{\overset{k_{on}}{\rightleftharpoons}} G_{\overline{ab}} I_a I_b S$ (3) $G_{\overline{ab}} I_a I_b S \xrightarrow{k_{cat}} G_{\overline{ab}} I_a I_b + P_f + P_q$

Table A.2.1.2: Reaction sequence for the AND gate that requires two positive inputs.

A.2.2 Reactions for a gate with a single negative input.

The reaction sequences for the NOT gate are shown both with and without input introduction. The action of this gate is based on interaction with the deoxyribozyme core region. Lack of input introduction leads to gate activation, while presence of input leads to gate inhibition.

Gate	Reactions
NOT	(1) $G_{\overline{c}} + I_c \underset{k_{off}}{\overset{k_{on}}{\rightleftharpoons}} G_{\overline{c}} I_c$ (2) $G_{\overline{c}} I_c + S \underset{k_{off}}{\overset{k_{on}}{\rightleftharpoons}} G_{\overline{c}} I_c S$

Table A.2.2.1: Reaction sequence for the NOT gate when input *is* introduced. With input, the gate core region is distorted such that the gate-input-substrate complex is rendered inactive. No signal occurs, thus negating the input.

Gate	Reactions
NOT	(1) $G_{\overline{c}} S \xrightarrow{k_{cat}} G_{\overline{c}} + P_f + P_q$

Table A.2.2.2: Reaction sequence for the NOT gate when input *is not* introduced. In this case the gate is immediately active and signaling occurs.

A.2.3 Reactions for gates with positive inputs and a single negative input.

These gates combine aspects of the simpler architectures to provide signals only when the correct positive inputs have been introduced, and no inhibitory input is present. In the case of the ANDANDNOT gate, similar to the AND gate, if only one of the required positive inputs is introduced, again correct gate-input-substrate binding fails to occur and no signal is produced.

Gate	Reactions
ANDNOT	(1) $G_{\overline{ac}} + I_a + I_c \underset{k_{off}}{\overset{k_{on}}{\rightleftharpoons}} G_{\overline{ac}} I_a I_c$ (2) $G_{\overline{ac}} I_a I_c + S \underset{k_{off}}{\overset{k_{on}}{\rightleftharpoons}} G_{\overline{ac}} I_a I_c S$

Table A.2.3.1: Reaction sequence for the ANDNOT gate when both the single positive input and single negative input are introduced. Since the negative input I_c acts as an inhibitor, the core is deformed and the gate fails to be activated.

Gate	Reactions
ANDNOT	(1) $G_{\overline{ac}} + I_a \underset{k_{off}}{\overset{k_{on}}{\rightleftharpoons}} G_{\overline{ac}} I_a$ (2) $G_{\overline{ac}} I_a + S \underset{k_{off}}{\overset{k_{on}}{\rightleftharpoons}} G_{\overline{ac}} I_a S$ (3) $G_{\overline{ac}} I_a S \xrightarrow{k_{cat}} G_{\overline{ac}} I_a + P_f + P_q$

Table A.2.3.2: Reaction sequence for ANDNOT gate when only the positive input is introduced. Since the inhibitory input I_c is not present, the gate is activated and a signal is produced.

Gate	Reactions
ANDANDNOT	(1) $G_{\overline{abc}} + I_a + I_b + I_c \underset{k_{off}}{\overset{k_{on}}{\rightleftharpoons}} G_{\overline{abc}} I_a I_b I_c$ (2) $G_{\overline{abc}} I_a I_b I_c + S \underset{k_{off}}{\overset{k_{on}}{\rightleftharpoons}} G_{\overline{abc}} I_a I_b I_c S$

Table A.2.3.3: Reaction sequence for ANDANDNOT gate when both positive inputs and the negative input are introduced. Since the inhibitory input I_c is present, no signal is produced.

Gate	Reactions
ANDANDNOT	(1) $G_{\overline{abc}} + I_a + I_b \underset{k_{off}}{\overset{k_{on}}{\rightleftharpoons}} G_{\overline{abc}} I_a I_b$ (2) $G_{\overline{abc}} I_a I_b + S \underset{k_{off}}{\overset{k_{on}}{\rightleftharpoons}} G_{\overline{abc}} I_a I_b S$ (3) $G_{\overline{abc}} I_a I_b S \xrightarrow{k_{cat}} G_{\overline{abc}} I_a I_b + P_f + P_q$

Table A.2.3.4: Reaction sequence for ANDANDNOT gate when both the positive inputs are introduced, but not the negative (inhibitory) one, leading to signal production.

A.3 Logic Examples

A.3.1 Adders

Each adder uses a single well, and is able to support simultaneous signaling for both the red and green channels. For the half-adder, the logic does not strictly require two-color signaling, but the full-adder does when both the sum and carry bit are on in the case of $01_2 + 01_2 = 11_2$.

Formula Half-Adder; sum bit signaled in red, carry bit signaled in green.

A.3.1.2 Well 1.

RED (TAMRA-BH_2 FRET):

$(i_1 \wedge \neg i_2) \vee (i_2 \wedge \neg i_1)$

GREEN (FLUORESCEIN-BH_1 FRET):

$(i_1 \wedge i_2)$

Formula Full-Adder; sum bit signaled in red, carry bit signaled in green.

A.3.1.2 Well 1.

RED (TAMRA-BH_2 FRET):

$(i_1 \wedge \neg i_2 \wedge \neg i_3) \vee (i_2 \wedge \neg i_1 \wedge \neg i_3) \vee (i_3 \wedge \neg i_1 \wedge \neg i_2) \vee (i_1 \wedge i_2 \wedge i_3)$

GREEN (FLUORESCEIN-BH_1 FRET):

$(i_1 \wedge i_2) \vee (i_2 \wedge i_3) \wedge (i_1 \wedge i_3)$

A.3.2 MAYA2

There are two indices used for naming each literal. The first index denotes the tic-tac-toe square number, where the upper left of the board is square 1 and the lower right is square 9.

The second index denotes the play number made by the human, since up to four plays are possible in some games this index ranged from 1 through 4. Each well has two formulas, where the first encodes human moves signaled with green fluorescence and the second encodes automaton response moves signaled with red fluorescence. Only a single color is signaled because a square can be claimed at most once in game play. Since the two formulas per well are coexistent in solution, low-level programming requires screening formulas against each other to ensure intended logic is not compromised by unwanted interactions. Well 5 is always claimed by the automaton as the initial move of all games, therefore it is a formula consisting of a single literal signaled in red.

Formula A.3.2.1 Well 1.

GREEN (FLUORESCEIN-BH_1 FRET):

$i_{11} \vee i_{12} \vee i_{13} \vee i_{14}$

RED (TAMRA-BH_2 FRET):

$(i_{21} \wedge i_{62}) \vee (i_{33} \wedge i_{44}) \vee (i_{73} \wedge i_{82}) \vee (i_{42} \wedge i_{71}) \vee (i_{62} \wedge i_{73} \wedge \neg i_{21}) \vee (i_{62} \wedge i_{83} \wedge \neg i_{21}) \vee$
$(i_{23} \wedge i_{62} \wedge \neg i_{21}) \vee (i_{42} \wedge i_{73} \wedge \neg i_{71}) \vee (i_{33} \wedge i_{42} \wedge \neg i_{71}) \vee (i_{23} \wedge i_{42} \wedge \neg i_{71})$

Formula A.3.2.2 Well 2.

GREEN (FLUORESCEIN-BH_1 FRET):

$i_{21} \vee i_{22} \vee i_{23} \vee i_{24}$

RED (TAMRA-BH_2 FRET):

$i_{61} \vee i_{91} \vee (i_{13} \wedge i_{62}) \vee (i_{34} \wedge i_{93}) \vee (i_{11} \wedge i_{32} \wedge \neg i_{22}) \vee (i_{11} \wedge i_{42} \wedge \neg i_{22}) \vee (i_{11} \wedge i_{62} \wedge \neg i_{22}) \vee$
$(i_{11} \wedge i_{72} \wedge \neg i_{22}) \vee (i_{11} \wedge i_{92} \wedge \neg i_{22}) \vee (i_{41} \wedge i_{12} \wedge \neg i_{22}) \vee (i_{41} \wedge i_{32} \wedge \neg i_{22}) \vee (i_{41} \wedge i_{62} \wedge \neg i_{22}) \vee$
$(i_{41} \wedge i_{72} \wedge \neg i_{22}) \vee (i_{41} \wedge i_{92} \wedge \neg i_{22})$

Formula A.3.2.3 Well 3.

GREEN (FLUORESCEIN-BH_1 FRET):

$i_{31} \vee i_{32} \vee i_{33} \vee i_{34}$

RED (TAMRA-BH_2 FRET):

$(i_{11} \wedge i_{22}) \vee (i_{61} \wedge i_{82}) \vee (i_{13} \wedge i_{42}) \vee (i_{24} \wedge i_{93}) \vee (i_{22} \wedge i_{63} \wedge \neg i_{11}) \vee (i_{22} \wedge i_{93} \wedge \neg i_{11}) \vee$
$(i_{22} \wedge i_{13} \wedge \neg i_{11}) \vee (i_{82} \wedge i_{63} \wedge \neg i_{61}) \vee (i_{82} \wedge i_{43} \wedge \neg i_{61}) \vee (i_{82} \wedge i_{13} \wedge \neg i_{61})$

Formula A.3.2.4 Well 4.

GREEN (FLUORESCEIN-BH_1 FRET):

$i_{41} \vee i_{42} \vee i_{43} \vee i_{44}$

RED (TAMRA-BH_2 FRET):

$i_{21} \vee i_{31} \vee (i_{22} \wedge i_{73}) \vee (i_{14} \wedge i_{33}) \vee (i_{81} \wedge i_{22} \wedge \neg i_{42}) \vee (i_{81} \wedge i_{92} \wedge \neg i_{42}) \vee (i_{81} \wedge i_{72} \wedge \neg i_{42}) \vee$
$(i_{81} \wedge i_{32} \wedge \neg i_{42}) \vee (i_{81} \wedge i_{12} \wedge \neg i_{42}) \vee (i_{71} \wedge i_{92} \wedge \neg i_{42}) \vee (i_{71} \wedge i_{82} \wedge \neg i_{42}) \vee (i_{71} \wedge i_{32} \wedge \neg i_{42}) \vee$
$(i_{71} \wedge i_{22} \wedge \neg i_{42}) \vee (i_{71} \wedge i_{12} \wedge \neg i_{42})$

Formula A.3.2.5 Well 5.

RED (TAMRA-BH_2 FRET):

i_{50}

Formula A.3.2.6 Well 6.

GREEN (FLUORESCEIN-BH_1 FRET):

$i_{61} \vee i_{62} \vee i_{63} \vee i_{64}$

RED (TAMRA-BH_2 FRET):

$i_{71} \vee i_{81} \vee (i_{73} \wedge i_{94}) \vee (i_{33} \wedge i_{82}) \vee (i_{21} \wedge i_{12} \wedge \neg i_{62}) \vee (i_{21} \wedge i_{32} \wedge \neg i_{62}) \vee (i_{21} \wedge i_{72} \wedge \neg i_{62}) \vee$
$(i_{21} \wedge i_{82} \wedge \neg i_{62}) \vee (i_{21} \wedge i_{92} \wedge \neg i_{62}) \vee (i_{31} \wedge i_{12} \wedge \neg i_{62}) \vee (i_{31} \wedge i_{22} \wedge \neg i_{62}) \vee (i_{31} \wedge i_{72} \wedge \neg i_{62}) \vee$
$(i_{31} \wedge i_{82} \wedge \neg i_{62}) \vee (i_{31} \wedge i_{92} \wedge \neg i_{62})$

Formula A.3.2.7 Well 7.

GREEN (FLUORESCEIN-BH_1 FRET):

$i_{71} \vee i_{72} \vee i_{73} \vee i_{74}$

RED (TAMRA-BH_2 FRET):

$(i_{22} \wedge i_{41}) \vee (i_{62} \wedge i_{93}) \vee (i_{13} \wedge i_{84}) \vee (i_{82} \wedge i_{91}) \vee (i_{22} \wedge i_{63} \wedge \neg i_{41}) \vee (i_{22} \wedge i_{93} \wedge \neg i_{41}) \vee$
$(i_{22} \wedge i_{43} \wedge \neg i_{41}) \vee (i_{82} \wedge i_{93} \wedge \neg i_{91}) \vee (i_{82} \wedge i_{43} \wedge \neg i_{91}) \vee (i_{82} \wedge i_{13} \wedge \neg i_{91})$

Formula A.3.2.8 Well 8.

GREEN (FLUORESCEIN-BH_1 FRET):

$i_{81} \vee i_{82} \vee i_{83} \vee i_{84}$

RED (TAMRA-BH_2 FRET):

$i_{11} \vee i_{41} \vee (i_{13} \wedge i_{74}) \vee (i_{42} \wedge i_{93}) \vee (i_{91} \wedge i_{72} \wedge \neg i_{82}) \vee (i_{91} \wedge i_{62} \wedge \neg i_{82}) \vee (i_{91} \wedge i_{42} \wedge \neg i_{82}) \vee$
$(i_{91} \wedge i_{32} \wedge \neg i_{82}) \vee (i_{91} \wedge i_{12} \wedge \neg i_{82}) \vee (i_{61} \wedge i_{92} \wedge \neg i_{82}) \vee (i_{61} \wedge i_{72} \wedge \neg i_{82}) \vee (i_{61} \wedge i_{42} \wedge \neg i_{82}) \vee$
$(i_{61} \wedge i_{32} \wedge \neg i_{82}) \vee (i_{61} \wedge i_{12} \wedge \neg i_{82})$

Formula A.3.2.9 Well 9.

GREEN (FLUORESCEIN-BH_1 FRET):

$i_{91} \vee i_{92} \vee i_{93} \vee i_{94}$

RED (TAMRA-BH_2 FRET):

$(i_{64} \wedge i_{73}) \vee (i_{22} \wedge i_{33}) \vee (i_{31} \wedge i_{62}) \vee (i_{42} \wedge i_{81}) \vee (i_{62} \wedge i_{73} \wedge \neg i_{31}) \vee (i_{62} \wedge i_{83} \wedge \neg i_{31}) \vee$
$(i_{62} \wedge i_{33} \wedge \neg i_{31}) \vee (i_{42} \wedge i_{83} \wedge \neg i_{81}) \vee (i_{42} \wedge i_{33} \wedge \neg i_{81}) \vee (i_{42} \wedge i_{23} \wedge \neg i_{81})$

A.3.3 Sensor Platform

For the H1N1 example, the following formulas are required. If we wanted further sensing capability for additional pathogens we would first need to select a short human-readable identifier, and then or together all single literals clauses where the display called for using the same wells for the dot-matrix display. We add square brackets to the formulas to distinguish the individual wells, and address each well in 2-D grid format where the bottom

left well is located at (0,0), and the top right well is located at (23,15), using 0-based numbering for the row and column positions of a 384-well plate respectively.

Formula A.3.3.1 H1N1

$[i_{h1n1}]_{3,6}$, $[i_{h1n1}]_{3,5}$, $[i_{h1n1}]_{3,4}$, $[i_{h1n1}]_{3,3}$, $[i_{h1n1}]_{3,2}$, $[i_{h1n1}]_{4,4}$, $[i_{h1n1}]_{5,4}$, $[i_{h1n1}]_{6,6}$
$[i_{h1n1}]_{6,5}$, $[i_{h1n1}]_{6,4}$, $[i_{h1n1}]_{6,3}$, $[i_{h1n1}]_{6,2}$, $[i_{h1n1}]_{8,6}$, $[i_{h1n1}]_{8,5}$, $[i_{h1n1}]_{8,4}$, $[i_{h1n1}]_{8,3}$
$[i_{h1n1}]_{8,2}$, $[i_{h1n1}]_{10,6}$, $[i_{h1n1}]_{10,5}$, $[i_{h1n1}]_{10,4}$, $[i_{h1n1}]_{10,3}$, $[i_{h1n1}]_{10,2}$, $[i_{h1n1}]_{11,5}$, $[i_{h1n1}]_{12,4}$
$[i_{h1n1}]_{13,3}$, $[i_{h1n1}]_{14,6}$, $[i_{h1n1}]_{14,5}$, $[i_{h1n1}]_{14,4}$, $[i_{h1n1}]_{14,3}$, $[i_{h1n1}]_{14,2}$, $[i_{h1n1}]_{16,6}$, $[i_{h1n1}]_{16,5}$
$[i_{h1n1}]_{16,4}$, $[i_{h1n1}]_{16,3}$, $[i_{h1n1}]_{16,2}$

Appendix B

Structure Shape and Binding Inference Report

B.1 Inference report for multibranch structure.

The combined output from shape inference and binding characterization algorithms for the 4-level multiply nested multibranch-stemloop-hairpin structure shown in Figure 3.5 is the following.

```
structure is  [(1,5,214),
               (7,4,49),
               (12,4,7),
               (28,4,7),
               (44,4,7),
               (65,5,101),
               (71,4,0),
```

```
                (80,12,46),
                (93,3,14),
                (114,3,14),
                (151,4,11),
                (177,4,34),
                (182,4,0),
                (191,4,0),
                (200,4,6)]

length is  225

bindings are  [1, 2, 3, 4, 5, 7, 8, 9, 10, 12, 13, 14, 15, 23, 24,
25, 26, 28, 29, 30, 31, 39, 40, 41, 42, 44, 45, 46, 47, 55, 56, 57,
58, 60, 61, 62, 63, 65, 66, 67, 68, 69, 71, 72, 73, 74, 75, 76, 77,
78, 80, 81, 82, 83, 84, 85, 86, 87, 88, 89, 90, 91, 93, 94, 95,
110, 111, 112, 114, 115, 116, 131, 132, 133, 138, 139, 140, 141,
142, 143, 144, 145, 146, 147, 148, 149, 151, 152, 153, 154, 166,
167, 168, 169, 171, 172, 173, 174, 175, 177, 178, 179, 180, 182,
183, 184, 185, 186, 187, 188, 189, 191, 192, 193, 194, 195, 196,
197, 198, 200, 201, 202, 203, 210, 211, 212, 213, 215, 216, 217,
218, 220, 221, 222, 223, 224]

partners are  [(1, 224), (2, 223), (3, 222), (4, 221), (5, 220),
(7, 63), (8, 62), (9, 61), (10, 60), (12, 26), (13, 25), (14, 24),
(15, 23), (28, 42), (29, 41), (30, 40), (31, 39), (44, 58),
(45, 57), (46, 56), (47, 55), (65, 175), (66, 174), (67, 173),
(68, 172), (69, 171), (71, 78), (72, 77), (73, 76), (74, 75),
```

```
(80, 149), (81, 148), (82, 147), (83, 146), (84, 145), (85, 144),
(86, 143), (87, 142), (88, 141), (89, 140), (90, 139), (91, 138),
(93, 112), (94, 111), (95, 110), (114, 133), (115, 132),
(116, 131), (151, 169), (152, 168), (153, 167), (154, 166),
(177, 218), (178, 217), (179, 216), (180, 215), (182, 189),
(183, 188), (184, 187), (185, 186), (191, 198), (192, 197),
(193, 196), (194, 195), (200, 213), (201, 212), (202, 211),
(203, 210)]

base state is  [(0, 'U'), (1, 224), (2, 223), (3, 222), (4, 221),
(5, 220), (6, 'U'), (7, 63), (8, 62), (9, 61), (10, 60), (11, 'U'),
(12, 26), (13, 25), (14, 24), (15, 23), (16, 'U'), (17, 'U'),
(18, 'U'), (19, 'U'), (20, 'U'), (21, 'U'), (22, 'U'), (23, 15),
(24, 14), (25, 13), (26, 12), (27, 'U'), (28, 42), (29, 41),
(30, 40), (31, 39), (32, 'U'), (33, 'U'), (34, 'U'), (35, 'U'),
(36, 'U'), (37, 'U'), (38, 'U'), (39, 31), (40, 30), (41, 29),
(42, 28), (43, 'U'), (44, 58), (45, 57), (46, 56), (47, 55),
(48, 'U'), (49, 'U'), (50, 'U'), (51, 'U'), (52, 'U'), (53, 'U'),
(54, 'U'), (55, 47), (56, 46), (57, 45), (58, 44), (59, 'U'),
(60, 10), (61, 9), (62, 8), (63, 7), (64, 'U'), (65, 175),
(66, 174), (67, 173), (68, 172), (69, 171), (70, 'U'), (71, 78),
(72, 77), (73, 76), (74, 75), (75, 74), (76, 73), (77, 72),
(78, 71), (79, 'U'), (80, 149), (81, 148), (82, 147), (83, 146),
(84, 145), (85, 144), (86, 143), (87, 142), (88, 141), (89, 140),
(90, 139), (91, 138), (92, 'U'), (93, 112), (94, 111), (95, 110),
(96, 'U'), (97, 'U'), (98, 'U'), (99, 'U'), (100, 'U'), (101, 'U'),
(102, 'U'), (103, 'U'), (104, 'U'), (105, 'U'), (106, 'U'),
```

```
(107, 'U'), (108, 'U'), (109, 'U'), (110, 95), (111, 94),
(112, 93), (113, 'U'), (114, 133), (115, 132), (116, 131),
(117, 'U'), (118, 'U'), (119, 'U'), (120, 'U'), (121, 'U'),
(122, 'U'), (123, 'U'), (124, 'U'), (125, 'U'), (126, 'U'),
(127, 'U'), (128, 'U'), (129, 'U'), (130, 'U'), (131, 116),
(132, 115), (133, 114), (134, 'U'), (135, 'U'), (136, 'U'),
(137, 'U'), (138, 91), (139, 90), (140, 89), (141, 88), (142, 87),
(143, 86), (144, 85), (145, 84), (146, 83), (147, 82), (148, 81),
(149, 80), (150, 'U'), (151, 169), (152, 168), (153, 167),
(154, 166), (155, 'U'), (156, 'U'), (157, 'U'), (158, 'U'),
(159, 'U'), (160, 'U'), (161, 'U'), (162, 'U'), (163, 'U'),
(164, 'U'), (165, 'U'), (166, 154), (167, 153), (168, 152),
(169, 151), (170, 'U'), (171, 69), (172, 68), (173, 67), (174, 66),
(175, 65), (176, 'U'), (177, 218), (178, 217), (179, 216),
(180, 215), (181, 'U'), (182, 189), (183, 188), (184, 187),
(185, 186), (186, 185), (187, 184), (188, 183), (189, 182),
(190, 'U'), (191, 198), (192, 197), (193, 196), (194, 195),
(195, 194), (196, 193), (197, 192), (198, 191), (199, 'U'),
(200, 213), (201, 212), (202, 211), (203, 210), (204, 'U'),
(205, 'U'), (206, 'U'), (207, 'U'), (208, 'U'), (209, 'U'),
(210, 203), (211, 202), (212, 201), (213, 200), (214, 'U'),
(215, 180), (216, 179), (217, 178), (218, 177), (219, 'U'),
(220, 5), (221, 4), (222, 3), (223, 2), (224, 1)]

unbound is  [0, 6, 11, 16, 17, 18, 19, 20, 21, 22, 27, 32, 33, 34,
35, 36, 37, 38, 43, 48, 49, 50, 51, 52, 53, 54, 59, 64, 70, 79, 92,
96, 97, 98, 99, 100, 101, 102, 103, 104, 105, 106, 107, 108, 109,
```

```
113, 117, 118, 119, 120, 121, 122, 123, 124, 125, 126, 127, 128,
129, 130, 134, 135, 136, 137, 150, 155, 156, 157, 158, 159, 160,
161, 162, 163, 164, 165, 170, 176, 181, 190, 199, 204, 205, 206,
207, 208, 209, 214, 219]

hairpin triples are  [[(71, 4, 0)], [(182, 4, 0)], [(191, 4, 0)]]
hairpin stem lengths are  [[4], [4], [4]]
hairpin stem addresses are  [[71], [182], [191]]

stemloop triples are
[[(12, 4, 7)], [(28, 4, 7)], [(44, 4, 7)], [(93, 3, 14)],
[(114, 3, 14)], [(151, 4, 11)], [(200, 4, 6)]]
stemloop stem lengths are
[[4], [4], [4], [3], [3], [4], [4]]
stemloop stem addresses are
[[12], [28], [44], [93], [114], [151], [200]]
stemloop loop counts are
[[7], [7], [7], [14], [14], [11], [6]]

bulge triples are  []
bulge stem lengths are  []
bulge stem addresses are  []
bulge loop counts are  []

internalloop triples are  []
internalloop stem lengths are  []
internalloop stem addresses are  []
```

```
internalloop loop counts are  []

multibranch triples are
[[(1, 5, 214), (7, 4, 49), (65, 5, 101), (177, 4, 34)],
[(7, 4, 49), (12, 4, 7), (28, 4, 7), (44, 4, 7)],
[(65, 5, 101), (71, 4, 0), (80, 12, 46), (151, 4, 11)],
[(80, 12, 46), (93, 3, 14), (114, 3, 14)],
[(177, 4, 34), (182, 4, 0), (191, 4, 0), (200, 4, 6)]]
multibranch stem lengths are
[[5, 4, 5, 4], [4, 4, 4, 4], [5, 4, 12, 4], [12, 3, 3],
[4, 4, 4, 4]]
multibranch stem addresses are
[[1, 7, 65, 177], [7, 12, 28, 44], [65, 71, 80, 151],
[80, 93, 114], [177, 182, 191, 200]]
multibranch loop counts are
[4, 4, 4, 6, 4]

parent-child relationships are
reported from parent perspective (my child triples are these)
[[1, 5, 11], [2, 3, 4], [], [], [], [6, 7, 10], [], [8, 9], [], [],
[], [12, 13, 14], [], [], []]
report from the child perspective (my parent is this)
[-1, 0, 1, 1, 1, 0, 5, 5, 7, 7, 5, 0, 11, 11, 11]
```

Appendix C

Four Layer Cascade DNADL File

C.1 Four Layer Cascade DDL File

```
/* 4 Layer Cascade */

LEVEL 1

ADDRESS

potMAIN;
potPREPZYME;
potPREPSCS;

PROGRAM

program_4layer
1:  (potMAIN, (E2 ^ SCS2) -> ACT2);
2:  (potMAIN, (E3 ^ SCS3) -> ACT3);
3:  (potMAIN, (E4 ^ SCS4) -> ACT4);
4:  (potMAIN, (E1INH ^ ACT2) -> E1);
5:  (potMAIN, (E2INH ^ ACT3) -> E2);
```

```
6:  (potMAIN, (E3INH ^ ACT4) -> E3);
7:  (potMAIN, SCS2);
9:  (potMAIN, SCS4);
10: (potMAIN, E1INH);
11: (potMAIN, E2INH);
12: (potMAIN, E3INH);
13: (potMAIN, E4);
14: (potMAIN, ACT4);   /* by 3, 9, 13,  modus ponens */
15: (potMAIN, E3);     /* by 6, 12, 14, modus ponens */
16: (potMAIN, ACT3);   /* by 2, 8, 15,  modus ponens */
17: (potMAIN, E2);     /* by 5, 11, 16, modus ponens */
18: (potMAIN, ACT2);   /* by 1, 7, 17,  modus ponens */
19: (potMAIN, E1);     /* by 4, 10, 18, modus ponens */

LEVEL 2

ENTRY

enDNAZYME1 = (potPREPZYME,strDNAZYME1UNFOLDED,100);
enDNAZYME2 = (potPREPZYME,strDNAZYME2UNFOLDED,100);
enDNAZYME3 = (potPREPZYME,strDNAZYME3UNFOLDED,100);

enINH1 = (potPREPZYME,strINH1,125);
enINH2 = (potPREPZYME,strINH2,125);
enINH3 = (potPREPZYME,strINH3,125);

enSCS2 = (potPREPSCS,strSCS2UNFOLDED,100);
enSCS3 = (potPREPSCS,strSCS3UNFOLDED,100);
enSCS4 = (potPREPSCS,strSCS4UNFOLDED,100);

enZYME1INH1 = (potMAIN,strDNAZYME1INH1,100);
enZYME2INH2 = (potMAIN,strDNAZYME2INH2,100);
enZYME3INH3 = (potMAIN,strDNAZYME3INH3,100);

enSCS2FOLDED = (potMAIN,strSCS2FOLDED,100);
enSCS3FOLDED = (potMAIN,strSCS3FOLDED,100);
enSCS4FOLDED = (potMAIN,strSCS4FOLDED,100);

enDNAZYME4 = (potMAIN,strDNAZYME4,100);
enSUBSTRATE1 = (potMAIN,strSUBSTRATE1,250);
```

```
SIGNAL

visualgreen = (potMAIN, green);

TRANSITION

tzymeinh1 = (potPREPZYME,strDNAZYME1,strINH1,strDNAZYME1INH1,bind);
tzymeinh2 = (potPREPZYME,strDNAZYME2,strINH2,strDNAZYME2INH2,bind);
tzymeinh3 = (potPREPZYME,strDNAZYME3,strINH3,strDNAZYME3INH3,bind);

tscs2 = (potPREPSCS,strSCS2UNFOLDED,strSCS2FOLDED,fold);
tscs3 = (potPREPSCS,strSCS3UNFOLDED,strSCS3FOLDED,fold);
tscs4 = (potPREPSCS,strSCS4UNFOLDED,strSCS4FOLDED,fold);

tzyme2scs2stage1 =
(potMAIN,strDNAZYME2,strSCS2FOLDED,strDNAZYME2SCS2.stage1,bind);
tzyme2scs2stage2 =
(potMAIN,strDNAZYME2SCS2.stage1,strDNAZYME2SCS2.stage2,bind);
tzyme2scs2split  =
(potMAIN,strDNAZYME2SCS2.stage2,strZYME2WASTE2,strACT2FOLDED,cleave);
tzyme2recovery   =
(potMAIN,strZYME2WASTE2,strDNAZYME2,strWASTE2,unbind);

tzyme3scs3stage1 =
(potMAIN,strDNAZYME3,strSCS3FOLDED,strDNAZYME3SCS3.stage1,bind);
tzyme3scs3stage2 =
(potMAIN,strDNAZYME3SCS3.stage1,strDNAZYME3SCS3.stage2,bind);
tzyme3scs3split  =
(potMAIN,strDNAZYME3SCS3.stage2,strZYME3WASTE3,strACT3FOLDED,cleave);
tzyme3recovery   =
(potMAIN,strZYME3WASTE3,strDNAZYME3,strWASTE3,unbind);

tzyme4scs4stage1 =
(potMAIN,strDNAZYME4,strSCS4FOLDED,strDNAZYME4SCS4.stage1,bind);
tzyme4scs4stage2 =
(potMAIN,strDNAZYME4SCS4.stage1,strDNAZYME4SCS4.stage2,bind);
tzyme4scs4split  =
(potMAIN,strDNAZYME4SCS4.stage2,strZYME4WASTE4,strACT4FOLDED,cleave);
tzyme4recovery   =
(potMAIN,strZYME4WASTE4,strDNAZYME4,strWASTE4,unbind);
```

```
treformact2      =
(potMAIN,strACT2FOLDED,strACT2UNFOLDED,fold);
treleasezyme1    =
(potMAIN,strDNAZYME1INH1,strACT2UNFOLDED,strDNAZYME1UNFOLDED,
strACT2INH1,exchange);

treformact3      = (potMAIN,strACT3FOLDED,strACT3UNFOLDED,fold);
treleasezyme2    =
(potMAIN,strDNAZYME2INH2,strACT3UNFOLDED,strDNAZYME2UNFOLDED,
strACT3INH2,exchange);

treformact4      = (potMAIN,strACT4FOLDED,strACT4UNFOLDED,fold);
treleasezyme3    =
(potMAIN,strDNAZYME3INH3,strACT4UNFOLDED,strDNAZYME3UNFOLDED,
strACT4INH3,exchange);

tactive1         = (potMAIN,strDNAZYME1UNFOLDED,strDNAZYME1,fold);
tactive2         = (potMAIN,strDNAZYME2UNFOLDED,strDNAZYME2,fold);
tactive3         = (potMAIN,strDNAZYME3UNFOLDED,strDNAZYME3,fold);

tzyme1substrate1 = (potMAIN,strDNAZYME1,strSUBSTRATE1,
strDNAZYME1SUBSTRATE1,bind);

EXECUTIONMECHANISM

layer2releaseactivator  = [tzyme2scs2stage1, tzyme2scs2stage2,
tzyme2scs2split, tzyme2recovery];
layer3releaseactivator  = [tzyme3scs3stage1, tzyme3scs3stage2,
tzyme3scs3split, tzyme3recovery];
layer4releaseactivator  = [tzyme4scs4stage1, tzyme4scs4stage2,
tzyme4scs4split, tzyme4recovery];

layer2releasegate = [treformact2,treleasesyme1,tactive1];
layer3releasegate = [treformact3,treleasezyme2,tactive2];
layer4releasegate = [treformact4,treleasezyme3,tactive3];

layer1signal = [tzyme1substrate1];

EVENTSTREAM

annealSCS
```

```
1: <enSCS2,enSCS3,enSCS4>;
2: <tscs2,tscs3,tscs4>;

annealDNAZYMES
1: <enDNAZYME1,enDNAZYME2,enDNAZYME3>;
2: <enINH1,enINH2,enINH3>;
3: <tzymeinh1,tzymeinh2,tzymeinh3>;

execCASCADE
1: <enZYME1INH1,enZYME2INH2,enZYME2INH3>;
2: <enSCS2FOLDED,enSCS3FOLDED,enSCS4FOLDED>;
3: enDNAZYME4;
4: layer4releaseactivator;
5: layer4activategate;
6: layer3releaseactivator;
7: layer3activategate;
8: layer2releaseactivator;
9: layer2activategate;
10: enSUBSTRATE1;
11: layer1signal;
12: visualgreen;

LEVEL 3

FLUOROPHORE

FAM;

QUENCHER

TAMRA;

LENGTH

lengthSCS2 = 47;
lengthSCS3 = 46;
lengthSCS4 = 46;
lengthINH = 23;
lengthDNAZYME = 31;
lengthACT = 36;
lengthWASTE2 = 11;
```

```
lengthWASTE3 = 10;
lengthWASTE4 = 10;
lengthSUBSTRATE1 = 20;

ISO

structSUBSTRATE = [];

structSCS2 = [(0,7,28),(7,7,8)];
structSCS3 = [(0,7,28),(7,6,10)];
structSCS4 = [(0,7,28),(7,6,10)];

structDNAZYME1ACTIVE = [(10,3,5)];
structDNAZYME2ACTIVE = [(10,3,5)];
structDNAZYME3ACTIVE = [(10,3,5)];

SEQUENCE

seqSUBSTRATE1 = TCTTAGTTAGGATAGTTCAT;

seqSCS2 = CGCCCTAATCTTAGGTCGAAAACTAAGATACATACTAGGGCGTGATG;
seqSCS3 = GCCGCTAATACATGGTCGAAAGTATGTATCCCCTGTAGCGGCATGT;
seqSCS4 = CGCGCTATTCCCCGGTCGAAACAGGGGAACTTCTGTAGCGCGACAG;

seqINH1 = ATGTATCTTAGTTTTCGACCGGC;
seqINH2 = GGGGATACATACTTTCGACCGGC;
seqINH3 = GAAGTTCCCCTGTTTCGACCGGC;

seqDNAZYME1 = GAACTATCTCCGAGCCGGTCGAAAACTAAGA;
seqDNAZYME2 = ATCACGCCTCCGAGCCGGTCGAAAGTATGTA;
seqDNAZYME3 = ACATGCCGTCCGAGCCGGTCGAAACAGGGGA;
seqDNAZYME4 = CTGTCGCGTCCGAGCCGGTCGAAACAGAAGT;

seqACT2 = CGCCCTAATCTTAGGTCGAAAACTAAGATACATACT;
seqACT3 = GCCGCTAATACATGGTCGAAAGTATGTATCCCCTGT;
seqACT4 = CGCGCTATTCCCCGGTCGAAACAGGGGAACTTCTGT;

seqWASTE2       = AGGGCGTGATG;
seqWASTE3       = AGCGGCATGT;
seqWASTE4       = AGCGCGACAG;
```

```
seqZYME2 = ATCACGCCTCCGAGCCGGTCGAAAGTATGTA;
seqZYME3 = ACATGCCGTCCGAGCCGGTCGAAACAGGGGA;
seqZYME4 = CTGTCGCGTCCGAGCCGGTCGAAACAGAAGT;

STRAND

strDNAZYME1 = (seqDNAZYME1,[(10,3,5)],lengthDNAZYME);
strDNAZYME2 = (seqDNAZYME2,[(10,3,5)],lengthDNAZYME);
strDNAZYME3 = (seqDNAZYME3,[(10,3,5)],lengthDNAZYME);
strDNAZYME4 = (seqDNAZYME4,[(10,3,5)],lengthDNAZYME);

strDNAZYME1UNFOLDED = (seqDNAZYME1,[],lengthDNAZYME);
strDNAZYME2UNFOLDED = (seqDNAZYME2,[],lengthDNAZYME);
strDNAZYME3UNFOLDED = (seqDNAZYME3,[],lengthDNAZYME);

strINH1 = (seqINH1,[],lengthINH);
strINH2 = (seqINH2,[],lengthINH);
strINH3 = (seqINH3,[],lengthINH);

strSCS2UNFOLDED = (seqSCS2,[],lengthSCS2);
strSCS3UNFOLDED = (seqSCS3,[],lengthSCS3);
strSCS4UNFOLDED = (seqSCS4,[],lengthSCS4);

strSCS2FOLDED = (seqSCS2,structSCS2,lengthSCS2);
strSCS3FOLDED = (seqSCS3,structSCS3,lengthSCS3);
strSCS4FOLDED = (seqSCS4,structSCS4,lengthSCS4);

strDNAZYME1INH1 = (seqDNAZYME1-seqINH1,
[(13,18,5)],lengthDNAZYME1+lengthINH1);
strDNAZYME2INH2 = (seqDNAZYME2-seqINH2,
[(13,18,5)],lengthDNAZYME2+lengthINH2);
strDNAZYME3INH3 = (seqDNAZYME3-seqINH3,
[(13,18,5)],lengthDNAZYME3+lengthINH3);

strDNAZYME2SCS2.stage1 = (seqDNAZYME2-seqSCS2,
[(0,8,61)],lengthDNAZYME+lengthSCS2);
strDNAZYME2SCS2.stage2 = (seqDNAZYME2-seqSCS2,
[(0,8,61),(23,7,30),(38,7,8)],lengthDNAZYME+lengthSCS2);
strDNAZYME2SCS2.stage3 = (seqDNAZYME2-seqACT2-seqWASTE2,
[(0,8,61),(23,7,30),(38,7,8)],lengthDNAZYME+lengthACT);
strZYME2WASTE2  = (seqZYME2-seqWASTE2,
```

```
[(0,8,25)],lengthDNAZYME+lengthWASTE2);

strDNAZYME3SCS3.stage1 = (seqDNAZYME3-seqSCS3,
[(0,8,61)],lengthDNAZYME+lengthSCS3);
strDNAZYME3SCS3.stage2 = (seqDNAZYME3-seqSCS3,
[(0,8,61),(23,7,30),(38,7,8)],lengthDNAZYME+lengthSCS3);
strDNAZYME3SCS3.stage3 = (seqDNAZYME3-seqACT3-seqWASTE3,
[(0,8,61),(23,7,30),(38,7,8)],lengthDNAZYME+lengthACT);
strZYME3WASTE3  = (seqZYME3-seqWASTE3,
[(0,8,25)],lengthDNAZYME+lengthWASTE3);

strDNAZYME4SCS4.stage1 = (seqDNAZYME4-seqSCS4,
[(0,8,61)],lengthDNAZYME+lengthSCS4);
strDNAZYME4SCS4.stage2 = (seqDNAZYME4-seqSCS4,
[(0,8,61),(23,7,30),(38,6,10)],lengthDNAZYME+lengthSCS4);
strDNAZYME4SCS4.stage3 = (seqDNAZYME4-seqACT-seqWASTE4,
[(0,8,61),(23,7,30),(38,6,10)],lengthDNAZYME+lengthACT);
strZYME4WASTE4  = (seqZYME4-seqWASTE4,
[(0,8,25)],lengthDNAZYME+lengthWASTE4);

strACT2FOLDED = (seqACT2,[(7,7,8)],lengthACT);
strACT3FOLDED = (seqACT3,[(7,7,8)],lengthACT);
strACT4FOLDED = (seqACT4,[(7,6,10)],lengthACT);

strACT2UNFOLDED = (seqACT2,[],lengthACT);
strACT3UNFOLDED = (seqACT3,[],lengthACT);
strACT4UNFOLDED = (seqACT4,[],lengthACT);

strACT2INH1 = (seqACT2-seqINH1,[(13,20,3)],lengthACT+lengthINH);
strACT3INH2 = (seqACT3-seqINH2,[(13,20,3)],lengthACT+lengthINH);
strACT4INH3 = (seqACT4-seqINH3,[(11,22,3)],lengthACT+lengthINH);

strSUBSTRATE1 = (FAM-seqSUBSTRATE1-TAM,[],lengthSUBSTRATE1);

strDNAZYME1SUBSTRATE1 = (seqDNAZYME1-seqSUBSTRATE1,
[(0,8,33),(23,8,0)],lengthDNAZYME+lengthSUBSTRATE1);

PHYSICALMAP

mapCASCADE
with program_4layer
```

```
1  <-> layer2releaseactivator;       /* (E2 ^ SCS2) -> ACT2     */
2  <-> layer3releaseactivator;       /* (E3 ^ SCS3) -> ACT3     */
3  <-> layer4releaseactivator;       /* (E4 ^ SCS4) -> ACT4     */
4  <-> layer2releasegate;            /* (E1INH ^ ACT2) -> E1    */
5  <-> layer3releasegate;            /* (E2INH ^ ACT3) -> E2    */
6  <-> layer4releasegate;            /* (E3INH ^ ACT4) -> E3    */
7  <-> enSCS2;                       /* SCS2                    */
8  <-> enSCS3;                       /* SCS3                    */
9  <-> enSCS3;                       /* SCS4                    */
10 <-> enZYME1INH1;                  /* E1INH                   */
11 <-> enZYME2INH2;                  /* E2INH                   */
12 <-> enZYME3INH3;                  /* E3INH                   */
13 <-> enDNAZYME4;                   /* E4                      */
14 <-> strACT4FOLDED;                /* ACT4                    */
15 <-> strDNAZYME3;                  /* E3                      */
16 <-> strACT3FOLDED;                /* ACT3                    */
17 <-> strDNAZYME2;                  /* E2                      */
18 <-> strACT2FOLDED;                /* ACT2                    */
19 <-> strDNAZYME1;                  /* E1                      */
```

C.2 Four Layer Cascade Diagrams

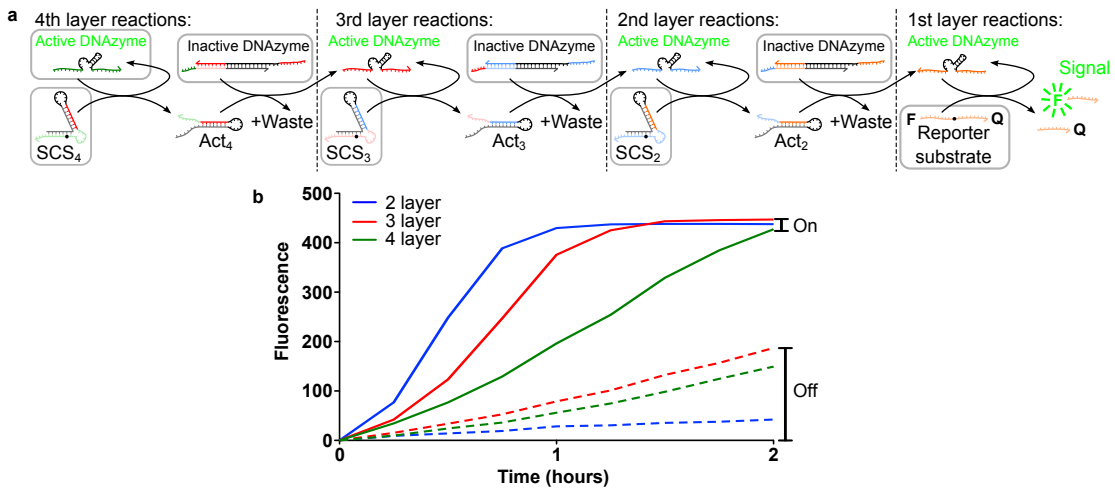


Figure C.1: The four layers of the cascade were linked using the Structured Chimeric Molecule (SCS) to enable layer-to-layer signal propagation. Each layer contained an inactivated deoxyribozyme-based gate designed to respond to an output product oligonucleotide from the preceding layer. The final layer used FRET signaling.

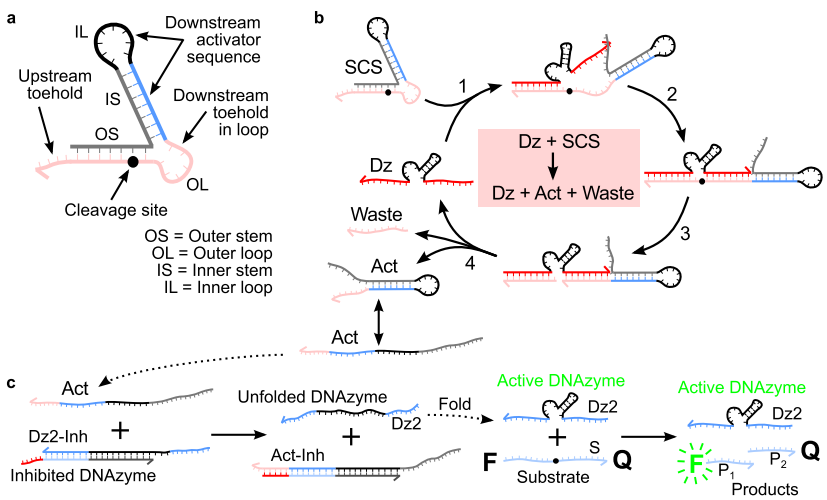


Figure C.2: The multi-step mechanism of the SCS and the deoxyribozyme gate encompassed structural conformation changes and critical binding events.

C.3 Maya II DDL File

```
/* MAYA 2 Level 1 Description */

/* indexing scheme prefix+address+number                    */

/*          first formula is     prefix = p, i=1, j=0:  p10   */
/*         second formula is     prefix = p, i=1, j=1:  p11   */
/*  ...                                                      */
/*           last formula is     prefix = p, i=9, j=13: p913  */

LEVEL 1

ADDRESS

a1 = (ROW 10, COL 11); a2 = (ROW 10, COL 12); a3 = (ROW 10, COL 13);
a4 = (ROW 10, COL 14); a6 = (ROW 10, COL 16); a7 = (ROW 10, COL 17);
a8 = (ROW 10, COL 18); a9 = (ROW 10, COL 19);

PREMISS

with a1formulas
p10 = (a1, I11);
p11 = (a1, I12);
p12 = (a1, I13);
p13 = (a1, I14);
p14 = (a1, I21 ^ I62);
p15 = (a1, I33 ^ I44);
p16 = (a1, I73 ^ I82);
p17 = (a1, I42 ^ I71);
p18 = (a1, I62 ^ I73 ^ neg I21);
p19 = (a1, I62 ^ I83 ^ neg I21);
p110 = (a1, I62 ^ I23 ^ neg I21);
p111 = (a1, I42 ^ I73 ^ neg I71);
p112 = (a1, I42 ^ I33 ^ neg I71);
p113 = (a1, I42 ^ I23 ^ neg I71);
```

```
with a2formulas
p20 = (a2, I21);
p21 = (a2, I22);
p22 = (a2, I23);
p23 = (a2, I24);
p24 = (a2, I61);
p25 = (a2, I91);
p26 = (a2, I13 ^ I62);
p27 = (a2, I93 ^ I34);
p28 = (a2, I11 ^ I32 ^ neg I22);
p29 = (a2, I11 ^ I42 ^ neg I22);
p210 = (a2, I11 ^ I62 ^ neg I22);
p211 = (a2, I11 ^ I72 ^ neg I22);
p212 = (a2, I11 ^ I92 ^ neg I22);
p213 = (a2, I41 ^ I12 ^ neg I22);
p214 = (a2, I41 ^ I32 ^ neg I22);
p215 = (a2, I41 ^ I62 ^ neg I22);
p216 = (a2, I41 ^ I72 ^ neg I22);
p217 = (a2, I41 ^ I92 ^ neg I22);


with a3formulas
p30 = (a3, I31);
p31 = (a3, I32);
p32 = (a3, I33);
p33 = (a3, I34);
p34 = (a3, I11 ^ I22);
p35 = (a3, I61 ^ I82);
p36 = (a3, I42 ^ I13);
p37 = (a3, I93 ^ I24);
p38 = (a3, I22 ^ I63 ^ neg I11);
p39 = (a3, I22 ^ I93 ^ neg I11);
p310 = (a3, I22 ^ I13 ^ neg I11);
p311 = (a3, I82 ^ I63 ^ neg I61);
p312 = (a3, I82 ^ I43 ^ neg I61);
p313 = (a3, I82 ^ I13 ^ neg I61);


with a4formulas
p40 = (a4, I41);
p41 = (a4, I42);
```

```
p42 = (a4, I43);
p43 = (a4, I44);
p44 = (a4, I21);
p45 = (a4, I31);
p46 = (a4, I22 ^ I73);
p47 = (a4, I33 ^ I14);
p48 = (a4, I81 ^ I22 ^ neg I42);
p49 = (a4, I81 ^ I92 ^ neg I42);
p410 = (a4, I81 ^ I72 ^ neg I42);
p411 = (a4, I81 ^ I32 ^ neg I42);
p412 = (a4, I81 ^ I12 ^ neg I42);
p413 = (a4, I71 ^ I92 ^ neg I42);
p414 = (a4, I71 ^ I82 ^ neg I42);
p415 = (a4, I71 ^ I32 ^ neg I42);
p416 = (a4, I71 ^ I22 ^ neg I42);
p417 = (a4, I71 ^ I12 ^ neg I42);


with a6formulas
p60 = (a6, I61);
p61 = (a6, I62);
p62 = (a6, I63);
p63 = (a6, I64);
p64 = (a6, I71);
p65 = (a6, I81);
p66 = (a6, I73 ^ I94);
p67 = (a6, I82 ^ I33);
p68 = (a6, I21 ^ I12 ^ neg I62);
p69 = (a6, I21 ^ I32 ^ neg I62);
p610 = (a6, I21 ^ I72 ^ neg I62);
p611 = (a6, I21 ^ I82 ^ neg I62);
p612 = (a6, I21 ^ I92 ^ neg I62);
p613 = (a6, I31 ^ I12 ^ neg I62);
p614 = (a6, I31 ^ I22 ^ neg I62);
p615 = (a6, I31 ^ I72 ^ neg I62);
p616 = (a6, I31 ^ I82 ^ neg I62);
p617 = (a6, I31 ^ I92 ^ neg I62);


with a7formulas
p70 = (a7, I71);
```

```
p71 = (a7, I72);
p72 = (a7, I73);
p73 = (a7, I74);
p74 = (a7, I41 ^ I22);
p75 = (a7, I62 ^ I93);
p76 = (a7, I13 ^ I84);
p77 = (a7, I91 ^ I82);
p78 = (a7, I22 ^ I63 ^ neg I41);
p79 = (a7, I22 ^ I93 ^ neg I41);
p710 = (a7, I22 ^ I43 ^ neg I41);
p711 = (a7, I82 ^ I93 ^ neg I91);
p712 = (a7, I82 ^ I43 ^ neg I91);
p713 = (a7, I82 ^ I13 ^ neg I91);


with a8formulas
p80 = (a8, I81);
p81 = (a8, I82);
p82 = (a8, I83);
p83 = (a8, I84);
p84 = (a8, I11);
p85 = (a8, I41);
p86 = (a8, I13 ^ I74);
p87 = (a8, I42 ^ I93);
p88 = (a8, I91 ^ I72 ^ neg I82);
p89 = (a8, I91 ^ I62 ^ neg I82);
p810 = (a8, I91 ^ I42 ^ neg I82);
p811 = (a8, I91 ^ I32 ^ neg I82);
p812 = (a8, I91 ^ I12 ^ neg I82);
p813 = (a8, I61 ^ I92 ^ neg I82);
p814 = (a8, I61 ^ I72 ^ neg I82);
p815 = (a8, I61 ^ I42 ^ neg I82);
p816 = (a8, I61 ^ I32 ^ neg I82);
p817 = (a8, I61 ^ I12 ^ neg I82);


with a9formulas
p90 = (a9, I91);
p91 = (a9, I92);
p92 = (a9, I93);
p93 = (a9, I94);
```

```
p94 = (a9, I73 ^ I64);
p95 = (a9, I22 ^ I33);
p96 = (a9, I31 ^ I62);
p97 = (a9, I81 ^ I42);
p98 = (a9, I62 ^ I73 ^ neg I31);
p99 = (a9, I62 ^ I83 ^ neg I31);
p910 = (a9, I62 ^ I33 ^ neg I31);
p911 = (a9, I42 ^ I83 ^ neg I81);
p912 = (a9, I42 ^ I33 ^ neg I81);
p913 = (a9, I42 ^ I23 ^ neg I81);


with board
a1formulas;
a2formulas;
a3formulas;
a4formulas;
a6formulas;
a7formulas;
a8formulas;
a9formulas;


with i1
with gameC1, gameC2, gameC3, gameC4, gameC5,
gameC11, gameC12, gameC13, gameC18, gameC19
input1 = (a1, I11);
input2 = (a2, I11);
input3 = (a3, I11);
input4 = (a4, I11);
input6 = (a6, I11);
input7 = (a7, I11);
input8 = (a8, I11);
input9 = (a9, I11);


with i1
with gameD1, gameD2, gameD3, gameD4, gameD5,
gameD11, gameD12, gameD13, gameD14
input1 = (a1, I21);
input2 = (a2, I21);
```

```
input3 = (a3, I21);
input4 = (a4, I21);
input6 = (a6, I21);
input7 = (a7, I21);
input8 = (a8, I21);
input9 = (a9, I21);


with i1
with gameD6, gameD7, gameD8, gameD9, gameD10,
gameD15, gameD16, gameD17, gameD18, gameD19
input1 = (a1, I31);
input2 = (a2, I31);
input3 = (a3, I31);
input4 = (a4, I31);
input6 = (a6, I31);
input7 = (a7, I31);
input8 = (a8, I31);
input9 = (a9, I31);


with i1
with gameC6, gameC7, gameC8, gameC9, gameC10,
gameC14, gameC15, gameC16, gameC17
input1 = (a1, I41);
input2 = (a2, I41);
input3 = (a3, I41);
input4 = (a4, I41);
input6 = (a6, I41);
input7 = (a7, I41);
input8 = (a8, I41);
input9 = (a9, I41);


with i1
with gameA1, gameA2, gameA3, gameA4, gameA5,
gameA11, gameA12, gameA13, gameA14
input1 = (a1, I61);
input2 = (a2, I61);
input3 = (a3, I61);
input4 = (a4, I61);
```

```
input6 = (a6, I61);
input7 = (a7, I61);
input8 = (a8, I61);
input9 = (a9, I61);


with i1
with gameB6, gameB7, gameB8, gameB9, gameB10,
gameB16, gameB17, gameB18, gameB19
input1 = (a1, I71);
input2 = (a2, I71);
input3 = (a3, I71);
input4 = (a4, I71);
input6 = (a6, I71);
input7 = (a7, I71);
input8 = (a8, I71);
input9 = (a9, I71);


with i1
with gameB1, gameB2, gameB3, gameB4, gameB5,
gameB11, gameB12, gameB13, gameB14, gameB15
input1 = (a1, I81);
input2 = (a2, I81);
input3 = (a3, I81);
input4 = (a4, I81);
input6 = (a6, I81);
input7 = (a7, I81);
input8 = (a8, I81);
input9 = (a9, I81);


with i1
with gameA6, gameA7, gameA8, gameA9, gameA10,
gameA15, gameA16, gameA17, gameA18, gameA19
input1 = (a1, I91);
input2 = (a2, I91);
input3 = (a3, I91);
input4 = (a4, I91);
input6 = (a6, I91);
input7 = (a7, I91);
```

```
input8 = (a8, I91);
input9 = (a9, I91);


with i2
with gameA1, gameA6, gameB1, gameB6, gameC6,
gameD1, gameD6
input1 = (a1, I12);
input2 = (a2, I12);
input3 = (a3, I12);
input4 = (a4, I12);
input6 = (a6, I12);
input7 = (a7, I12);
input8 = (a8, I12);
input9 = (a9, I12);


with i2
with gameB2, gameB7, gameC11, gameC12, gameC13, gameC14,
gameC15, gameC16, gameC17, gameC18, gameC19, gameD7
input1 = (a1, I22);
input2 = (a2, I22);
input3 = (a3, I22);
input4 = (a4, I22);
input6 = (a6, I22);
input7 = (a7, I22);
input8 = (a8, I22);
input9 = (a9, I22);


with i2
with gameA2, gameA7, gameB3, gameB8, gameC1,
gameC7, gameD2
input1 = (a1, I32);
input2 = (a2, I32);
input3 = (a3, I32);
input4 = (a4, I32);
input6 = (a6, I32);
input7 = (a7, I32);
input8 = (a8, I32);
input9 = (a9, I32);
```

```
with i2
with gameA3, gameA8, gameB11, gameB12, gameB13, gameB14,
gameB15, gameB16, gameB17, gameB18, gameB19, gameC2
input1 = (a1, I42);
input2 = (a2, I42);
input3 = (a3, I42);
input4 = (a4, I42);
input6 = (a6, I42);
input7 = (a7, I42);
input8 = (a8, I42);
input9 = (a9, I42);


with i2
with gameA9, gameC3, gameC8, gameD11, gameD12, gameD13,
gameD14, gameD15, gameD16, gameD17, gameD18, gameD19
input1 = (a1, I62);
input2 = (a2, I62);
input3 = (a3, I62);
input4 = (a4, I62);
input6 = (a6, I62);
input7 = (a7, I62);
input8 = (a8, I62);
input9 = (a9, I62);


with i2
with gameA4, gameA10, gameB4, gameC4, gameC9,
gameD3, gameD8
input1 = (a1, I72);
input2 = (a2, I72);
input3 = (a3, I72);
input4 = (a4, I72);
input6 = (a6, I72);
input7 = (a7, I72);
input8 = (a8, I72);
input9 = (a9, I72);
```

```
with i2
with gameA11, gameA12, gameA13, gameA14, gameA15, gameA16,
gameA17, gameA18, gameA19, gameB9, gameD4, gameD9
input1 = (a1, I82);
input1 = (a2, I82);
input1 = (a3, I82);
input1 = (a4, I82);
input1 = (a6, I82);
input1 = (a7, I82);
input1 = (a8, I82);
input1 = (a9, I82);


with i2
with gameA5, gameB5, gameB10, gameC5, gameC10,
gameD5, gameD10
input1 = (a1, I92);
input1 = (a2, I92);
input1 = (a3, I92);
input1 = (a4, I92);
input1 = (a6, I92);
input1 = (a7, I92);
input1 = (a8, I92);
input1 = (a9, I92);


with i3
with gameA11, gameA15, gameB14, gameC14, gameD18, gameD19
input1 = (a1, I13);
input2 = (a2, I13);
input3 = (a3, I13);
input4 = (a4, I13);
input6 = (a6, I13);
input7 = (a7, I13);
input8 = (a8, I13);
input9 = (a9, I13);


with i3
with gameB11, gameB15, gameD15
input1 = (a1, I23);
```

```
input2 = (a2, I23);
input3 = (a3, I23);
input4 = (a4, I23);
input6 = (a6, I23);
input7 = (a7, I23);
input8 = (a8, I23);
input9 = (a9, I23);


with i3
with gameA18, gameA19, gameB12, gameB16, gameC15, gameD11
input1 = (a1, I33);
input2 = (a2, I33);
input3 = (a3, I33);
input4 = (a4, I33);
input6 = (a6, I33);
input7 = (a7, I33);
input8 = (a8, I33);
input9 = (a9, I33);


with i3
with gameA12, gameA16, gameC11
input1 = (a1, I43);
input2 = (a2, I43);
input3 = (a3, I43);
input4 = (a4, I43);
input6 = (a6, I43);
input7 = (a7, I43);
input8 = (a8, I43);
input9 = (a9, I43);


with i3
with gameA17, gameC12, gameC16
input1 = (a1, I63);
input2 = (a2, I63);
input3 = (a3, I63);
input4 = (a4, I63);
input6 = (a6, I63);
input7 = (a7, I63);
```

```
input8 = (a8, I63);
input9 = (a9, I63);


with i3
with gameA13, gameB13, gameC18, gameC19, gameD12, gameD16
input1 = (a1, I73);
input2 = (a2, I73);
input3 = (a3, I73);
input4 = (a4, I73);
input6 = (a6, I73);
input7 = (a7, I73);
input8 = (a8, I73);
input9 = (a9, I73);


with i3
with gameB17, gameD13, gameD17
input1 = (a1, I83);
input2 = (a2, I83);
input3 = (a3, I83);
input4 = (a4, I83);
input6 = (a6, I83);
input7 = (a7, I83);
input8 = (a8, I83);
input9 = (a9, I83);


with i3
with gameA14, gameB18, gameB19, gameC13, gameC17, gameD14
input1 = (a1, I93);
input2 = (a2, I93);
input3 = (a3, I93);
input4 = (a4, I93);
input6 = (a6, I93);
input7 = (a7, I93);
input8 = (a8, I93);
input9 = (a9, I93);


with i4
```

```
with gameA18
input1 = (a1, I14);
input2 = (a2, I14);
input3 = (a3, I14);
input4 = (a4, I14);
input6 = (a6, I14);
input7 = (a7, I14);
input8 = (a8, I14);
input9 = (a9, I14);


with i4
with gameB18
input1 = (a1, I24);
input2 = (a2, I24);
input3 = (a3, I24);
input4 = (a4, I24);
input6 = (a6, I24);
input7 = (a7, I24);
input8 = (a8, I24);
input9 = (a9, I24);


with i4
with gameB19
input1 = (a1, I34);
input2 = (a2, I34);
input3 = (a3, I34);
input4 = (a4, I34);
input6 = (a6, I34);
input7 = (a7, I34);
input8 = (a8, I34);
input9 = (a9, I34);


with i4
with gameA19
input1 = (a1, I44);
input2 = (a2, I44);
input3 = (a3, I44);
input4 = (a4, I44);
```

```
input6 = (a6, I44);
input7 = (a7, I44);
input8 = (a8, I44);
input9 = (a9, I44);


with i4
with gameC18
input1 = (a1, I64);
input2 = (a2, I64);
input3 = (a3, I64);
input4 = (a4, I64);
input6 = (a6, I64);
input7 = (a7, I64);
input8 = (a8, I64);
input9 = (a9, I64);


with i4
with gameD18
input1 = (a1, I74);
input2 = (a2, I74);
input3 = (a3, I74);
input4 = (a4, I74);
input6 = (a6, I74);
input7 = (a7, I74);
input8 = (a8, I74);
input9 = (a9, I74);


with i4
with gameD19
input1 = (a1, I84);
input2 = (a2, I84);
input3 = (a3, I84);
input4 = (a4, I84);
input6 = (a6, I84);
input7 = (a7, I84);
input8 = (a8, I84);
input9 = (a9, I84);
```

```
with i4
with gameC19
input1 = (a1, I94);
input2 = (a2, I94);
input3 = (a3, I94);
input4 = (a4, I94);
input6 = (a6, I94);
input7 = (a7, I94);
input8 = (a8, I94);
input9 = (a9, I94);


CONCLUSION

with o1
with gameC1, gameC2, gameC3, gameC4, gameC5, gameC11,
gameC12, gameC13, gameC18, gameC19
conclusion1 = (a1, I11);
conclusion8 = (a8, I11);


with o1
with gameD1, gameD2, gameD3, gameD4, gameD5, gameD11,
gameD12, gameD13, gameD14
conclusion2 = (a2, I21);
conclusion4 = (a4, I21);


with o1
with gameD6, gameD7, gameD8, gameD9, gameD10, gameD15,
gameD16, gameD17, gameD18, gameD19
conclusion3 = (a3, I31);
conclusion4 = (a4, I31);


with o1
with gameC6, gameC7, gameC8, gameC9, gameC10, gameC14,
gameC15, gameC16, gameC17
conclusion4 = (a4, I41);
conclusion8 = (a8, I41);
```

```
with o1
with gameA1, gameA2, gameA3, gameA4, gameA5, gameA11,
gameA12, gameA13, gameA14, gameA15
conclusion6 = (a6, I61);
conclusion2 = (a2, I61);


with o1
with gameB6, gameB7, gameB8, gameB9, gameB10, gameB15,
gameB16, gameB17, gameB18, gameB19
conclusion7 = (a7, I71);
conclusion6 = (a6, I71);


with o1
with gameB1, gameB2, gameB3, gameB4, gameB5, gameB11,
gameB12, gameB13, gameB14
conclusion8 = (a8, I81);
conclusion6 = (a6, I81);


with o1
with gameA6, gameA7, gameA8, gameA9, gameA10, gameA16,
gameA17, gameA18, gameA19
conclusion9 = (a9, I91);
conclusion2 = (a2, I91);


with o2
with gameA1, gameA6, gameB1, gameB6, gameC6, gameD1, gameD6
conclusion1 = (a1, I12);


with o2
with gameD1
conclusion6 = (a6, I21 ^ I12 ^ ~I62);


with o2
```

```
with gameD6
conclusion6 = (a6, I31 ^ I12 ^ ~I62);


with o2
with gameC6
conclusion2 = (a2, I41 ^ I12 ^ ~I22);


with o2
with gameA1, gameA6
conclusion8 = (a8, I61 ^ I12 ^ ~I82);


with o2
with gameB6
conclusion4 = (a4, I71 ^ I12 ^ ~I42);


with o2
with gameB1
conclusion4 = (a4, I81 ^ I12 ^ ~I42);


with o2
with gameB2, gameB7, gameC11, gameC12, gameC13, gameC14,
gameC15, gameC16, gameC17, gameC18, gameC19, gameD7
conclusion2 = (a2, I22);


with o2
with gameC11, gameC12, gameC13, gameC18, gameC19
conclusion3 = (a3, I11 ^ I22);


with o2
with gameD7
conclusion6 = (a6, I31 ^ I22 ^ ~I62);


with o2
```

```
with gameC14, gameC15, gameC16, gameC17
conclusion7 = (a7, I41 ^ I22);


with o2
with gameB2
conclusion4 = (a4, I81 ^ I22 ^ ~I42);


with o2
with gameB7
conclusion4 = (a4, I71 ^ I22 ^ ~I42);


with o2
with gameA2, gameA7, gameB3, gameB8, gameC1, gameC7, gameD2
conclusion3 = (a3, I32);


with o2
with gameC1
conclusion2 = (a2, I11 ^ I32 ^ ~I22);


with o2
with gameD2
conclusion6 = (a6, I21 ^ I32 ^ ~I62);


with o2
with gameC7
conclusion2 = (a2, I41 ^ I32 ^ ~I22);


with o2
with gameA2
conclusion8 = (a8, I61 ^ I32 ^ ~I82);


with o2
with gameB8
```

```
conclusion4 = (a4, I71 ^ I32 ^ ~I42);


with o2
with gameB3
conclusion4 = (a4, I81 ^ I32 ^ ~I42);


with o2
with gameA3, gameA8, gameB11, gameB12, gameB13, gameB14,
gameB15, gameB16, gameB17, gameB18, gameB19, gameC2
conclusion4 = (a4, I42);


with o2
with gameC2
conclusion2 = (a2, I11 ^ I42 ^ ~I22);


with o2
with gameA3
conclusion8 = (a8, I61 ^ I42 ^ ~I82);


with o2
with gameB15, gameB16, gameB17, gameB18, gameB19
conclusion1 = (a1, I71 ^ I42);


with o2
with gameB11, gameB12, gameB13, gameB14
conclusion9 = (a9, I81 ^ I42);


with o2
with gameA9, gameC3, gameC8, gameD11, gameD12, gameD13,
gameD14, gameD15, gameD16, gameD17, gameD18, gameD19
conclusion6 = (a6, I62);


with o2
```

```
with gameC3
conclusion2 = (a2, I11 ^ I62 ^ ~I22);


with o2
with gameD11, gameD12, gameD13, gameD14
conclusion1 = (a1, I21 ^ I62);


with o2
with gameD15, gameD16, gameD17, gameD18, gameD19
conclusion9 = (a9, I31 ^ I62);


with o2
with gameC8
conclusion2 = (a2, I41 ^ I62 ^ ~I22);


with o2
with gameA7
conclusion8 = (a8, I91 ^ I32 ^ ~I82);


with o2
with gameA8
conclusion8 = (a8, I91 ^ I42 ^ ~I82);


with o2
with gameA9
conclusion8 = (a8, I91 ^ I62 ^ ~I82);


with o2
with gameA4, gameA10, gameB4, gameC4, gameC9, gameD3, gameD8
conclusion7 = (a7, I72);


with o2
with gameC4
```

```
conclusion2 = (a2, I11 ^ I72 ^ ~I22);


with o2
with gameD3
conclusion6 = (a6, I21 ^ I72 ^ ~I62);


with o2
with gameD8
conclusion6 = (a6, I31 ^ I72 ^ ~I62);


with o2
with gameC9
conclusion2 = (a2, I41 ^ I72 ^ ~I22);


with o2
with gameA4
conclusion8 = (a8, I61 ^ I72 ^ ~I82);


with o2
with gameB4
conclusion4 = (a4, I81 ^ I72 ^ ~I42);


with o2
with gameB5
conclusion4 = (a4, I81 ^ I92 ^ ~I42);


with o2
with gameA10
conclusion8 = (a8, I91 ^ I72 ^ ~I82);


with o2
with gameA11, gameA12, gameA13, gameA14, gameA15, gameA16,
gameA17, gameA18, gameA19, gameB9, gameD4, gameD9
```

```
conclusion2 = (a2, I82);


with o2
with gameD4
conclusion6 = (a6, I21 ^ I82 ^ ~I62);


with o2
with gameD9
conclusion6 = (a6, I31 ^ I82 ^ ~I62);


with o2
with gameA11, gameA12, gameA13, gameA14
conclusion3 = (a3, I61 ^ I82);


with o2
with gameB9
conclusion4 = (a4, I71 ^ I82 ^ ~I42);


with o2
with gameB10
conclusion4 = (a4, I71 ^ I92 ^ ~I42);


with o2
with gameA15, gameA16, gameA17, gameA18, gameA19
conclusion7 = (a7, I91 ^ I82);


with o2
with gameB5, gameB10, gameC5, gameC10, gameD5, gameD10
conclusion9 = (a9, I92);


with o2
with gameC5
conclusion2 = (a2, I11 ^ I92 ^ ~I22);
```

```
with o2
with gameD5
conclusion6 = (a6, I21 ^ I92 ^ ~I22);


with o2
with gameD10
conclusion6 = (a6, I31 ^ I92 ^ ~I22);


with o2
with gameC10
conclusion2 = (a2, I41 ^ I92 ^ ~I22);


with o3
with gameA11, gameA15, gameB14, gameC14, gameD18, gameD19
conclusion1 = (a1, I13);


with o3
with gameC14
conclusion3 = (a3, I22 ^ I13 ^ ~I11);


with o3
with gameB14
conclusion3 = (a3, I42 ^ I13);


with o3
with gameA11
conclusion7 = (a7, I82 ^ I13 ^ ~I91);


with o3
with gameD18, gameD19
conclusion2 = (a2, I62 ^ I13);
```

```
with o3
with gameA15
conclusion3 = (a3, I82 ^ I13 ^ ~I61);


with o3
with gameB11, gameB15, gameD15
conclusion2 = (a2, I23);


with o3
with gameD15
conclusion1 = (a1, I62 ^ I23 ^ ~I21);


with o3
with gameB11
conclusion1 = (a1, I42 ^ I23 ^ ~I71);


with o3
with gameB15
conclusion9 = (a9, I42 ^ I23 ^ ~I81);


with o3
with gameA18, gameA19, gameB12, gameB16, gameC15, gameD11
conclusion3 = (a3, I33);


with o3
with gameC15
conclusion9 = (a9, I22 ^ I33);


with o3
with gameB12
conclusion1 = (a1, I42 ^ I33 ^ ~I71);
```

```
with o3
with gameB16
conclusion9 = (a9, I42 ^ I33 ^ ~I91);


with o3
with gameD11
conclusion9 = (a9, I62 ^ I33 ^ ~I31);


with o3
with gameA18, gameA19
conclusion6 = (a6, I82 ^ I33);


with o3
with gameA12, gameA16, gameC11
conclusion4 = (a4, I43);


with o3
with gameC11
conclusion7 = (a7, I22 ^ I43 ^ ~I41);


with o3
with gameA16
conclusion3 = (a3, I82 ^ I43 ^ ~I61);


with o3
with gameA12
conclusion7 = (a7, I82 ^ I43 ^ ~I91);


with o3
with gameA17, gameC12, gameC16
conclusion6 = (a6, I63);


with o3
```

```
with gameC12
conclusion7 = (a7, I22 ^ I63 ^ ~I41);


with o3
with gameC16
conclusion3 = (a3, I22 ^ I63 ^ ~I11);


with o3
with gameA17
conclusion3 = (a3, I82 ^ I63 ^ ~I61);


with o3
with gameA13, gameB13, gameC18, gameC19, gameD12, gameD16
conclusion7 = (a7, I73);


with o3
with gameC18, gameC19
conclusion4 = (a4, I22 ^ I73);


with o3
with gameB13
conclusion1 = (a1, I42 ^ I73 ^ ~I71);


with o3
with gameD12
conclusion9 = (a9, I62 ^ I73 ^ ~I31);


with o3
with gameD16
conclusion1 = (a1, I62 ^ I73 ^ ~I21);


with o3
with gameA13
```

```
conclusion1 = (a1, I82 ^ I73);


with o3
with gameB17, gameD13, gameD17
conclusion3 = (a3, I83);


with o3
with gameD17
conclusion1 = (a1, I62 ^ I83 ^ ~I21);


with o3
with gameD13
conclusion9 = (a9, I62 ^ I83 ^ ~I31);


with o3
with gameB17
conclusion9 = (a9, I42 ^ I83 ^ ~I81);


with o3
with gameA14, gameB18, gameB19, gameC13, gameC17, gameD14
conclusion9 = (a9, I93);


with o3
with gameC13
conclusion7 = (a7, I22 ^ I93 ^ ~I41);


with o3
with gameC17
conclusion3 = (a3, I22 ^ I93 ^ ~I11);


with o3
with gameB18, gameB19
conclusion8 = (a8, I42 ^ I93);
```

```
with o3
with gameD14
conclusion7 = (a7, I62 ^ I93);


with o3
with gameA14
conclusion7 = (a7, I82 ^ I93 ^ ~I91);


with o4
with gameA18
conclusion1 = (a1, I14);
conclusion4 = (a4, I33 ^ I14);


with o4
with gameB18
conclusion2 = (a2, I24);
conclusion3 = (a3, I93 ^ I24);


with o4
with gameB19
conclusion3 = (a3, I34);
conclusion2 = (a2, I93 ^ I34);


with o4
with gameA19
conclusion4 = (a4, I44);
conclusion1 = (a1, I33 ^ I44);


with o4
with gameC18
conclusion6 = (a6, I64);
conclusion9 = (a9, I73 ^ I64);
```

```
with o4
with gameD18
conclusion7 = (a7, I74);
conclusion8 = (a8, I13 ^ I74);


with o4
with gameD19
conclusion8 = (a8, I84);
conclusion7 = (a7, I13 ^ I84);


with o4
with gameC19
conclusion9 = (a9, I94);
conclusion6 = (a6, I73 ^ I94);


/* M2 Level 2 Description */

LEVEL 2

ENTRY

/* gameC1,  gameC2,  gameC3,  gameC4,  gameC5, */
/* gameC11, gameC12, gameC13, gameC18, gameC19 */
en1I11 = (a1, str.lI11, 100);
en2I11 = (a2, str.lI11, 100);
en3I11 = (a3, str.lI11, 100);
en4I11 = (a4, str.lI11, 100);
en6I11 = (a6, str.lI11, 100);
en7I11 = (a7, str.lI11, 100);
en8I11 = (a8, str.lI11, 100);
en9I11 = (a9, str.lI11, 100);

/*  gameD1,  gameD2,  gameD3,  gameD4, gameD5,  */
/*  gameD11, gameD12, gameD13, gameD14          */
en1I21 = (a1, str.lI21, 100);
en2I21 = (a2, str.lI21, 100);
en3I21 = (a3, str.lI21, 100);
```

```
en4I21 = (a4, str.lI21, 100);
en6I21 = (a6, str.lI21, 100);
en7I21 = (a7, str.lI21, 100);
en8I21 = (a8, str.lI21, 100);
en9I21 = (a9, str.lI21, 100);

/* gameD6,  gameD7,  gameD8,  gameD9,  gameD10, */
/* gameD15, gameD16, gameD17, gameD18, gameD19  */
en1I31 = (a1, str.lI31, 100);
en2I31 = (a2, str.lI31, 100);
en3I31 = (a3, str.lI31, 100);
en4I31 = (a4, str.lI31, 100);
en6I31 = (a6, str.lI31, 100);
en7I31 = (a7, str.lI31, 100);
en8I31 = (a8, str.lI31, 100);
en9I31 = (a9, str.lI31, 100);

/* gameC6,  gameC7,  gameC8,  gameC9, gameC10,  */
/* gameC14, gameC15, gameC16, gameC17           */
en1I41 = (a1, str.lI41, 100);
en2I41 = (a2, str.lI41, 100);
en3I41 = (a3, str.lI41, 100);
en4I41 = (a4, str.lI41, 100);
en6I41 = (a6, str.lI41, 100);
en7I41 = (a7, str.lI41, 100);
en8I41 = (a8, str.lI41, 100);
en9I41 = (a9, str.lI41, 100);

/* gameA1,  gameA2,  gameA3,  gameA4, gameA5,   */
/* gameA11, gameA12, gameA13, gameA14           */
en1I61 = (a1, str.lI61, 100);
en2I61 = (a2, str.lI61, 100);
en3I61 = (a3, str.lI61, 100);
en4I61 = (a4, str.lI61, 100);
en6I61 = (a6, str.lI61, 100);
en7I61 = (a7, str.lI61, 100);
en8I61 = (a8, str.lI61, 100);
en9I61 = (a9, str.lI61, 100);

/* gameB6,  gameB7,  gameB8,  gameB9, gameB10,  */
/* gameB16, gameB17, gameB18, gameB19           */
```

```
en1I71 = (a1, str.lI71, 100);
en2I71 = (a2, str.lI71, 100);
en3I71 = (a3, str.lI71, 100);
en4I71 = (a4, str.lI71, 100);
en6I71 = (a6, str.lI71, 100);
en7I71 = (a7, str.lI71, 100);
en8I71 = (a8, str.lI71, 100);
en9I71 = (a9, str.lI71, 100);

/* gameB1,  gameB2,  gameB3,  gameB4,  gameB5,  */
/* gameB11, gameB12, gameB13, gameB14, gameB15  */
en1I81 = (a1, str.lI81, 100);
en2I81 = (a2, str.lI81, 100);
en3I81 = (a3, str.lI81, 100);
en4I81 = (a4, str.lI81, 100);
en6I81 = (a6, str.lI81, 100);
en7I81 = (a7, str.lI81, 100);
en8I81 = (a8, str.lI81, 100);
en9I81 = (a9, str.lI81, 100);

/* gameA6,  gameA7,  gameA8,  gameA9,  gameA10, */
/* gameA15, gameA16, gameA17, gameA18, gameA19  */
en1I91 = (a1, str.lI91, 100);
en2I91 = (a2, str.lI91, 100);
en3I91 = (a3, str.lI91, 100);
en4I91 = (a4, str.lI91, 100);
en6I91 = (a6, str.lI91, 100);
en7I91 = (a7, str.lI91, 100);
en8I91 = (a8, str.lI91, 100);
en9I91 = (a9, str.lI91, 100);

/* gameA1, gameA6, gameB1, gameB6, gameC6,     */
/* gameD1, gameD1, gameD6                      */
en1I12 = (a1, str.lI12, 100);
en2I12 = (a2, str.lI12, 100);
en3I12 = (a3, str.lI12, 100);
en4I12 = (a4, str.lI12, 100);
en6I12 = (a6, str.lI12, 100);
en7I12 = (a7, str.lI12, 100);
en8I12 = (a8, str.lI12, 100);
en9I12 = (a9, str.lI12, 100);
```

```
/* gameB2,  gameB7,  gameC11, gameC12, gameC13, gameC14,  */
/* gameC15, gameC16, gameC17, gameC18, gameC19, gameD7    */
en1I22 = (a1, str.lI22, 100);
en2I22 = (a2, str.lI22, 100);
en3I22 = (a3, str.lI22, 100);
en4I22 = (a4, str.lI22, 100);
en6I22 = (a6, str.lI22, 100);
en7I22 = (a7, str.lI22, 100);
en8I22 = (a8, str.lI22, 100);
en9I22 = (a9, str.lI22, 100);

/* gameA2, gameA7, gameB3, gameB8, gameC1, gameC7, gameD2 */
en1I32 = (a1, str.lI32, 100);
en2I32 = (a2, str.lI32, 100);
en3I32 = (a3, str.lI32, 100);
en4I32 = (a4, str.lI32, 100);
en6I32 = (a6, str.lI32, 100);
en7I32 = (a7, str.lI32, 100);
en8I32 = (a8, str.lI32, 100);
en9I32 = (a9, str.lI32, 100);

/* gameA3,  gameA8,  gameB11, gameB12, gameB13, gameB14,  */
/* gameB15, gameB16, gameB17, gameB18, gameB19, gameC2    */
en1I42 = (a1, str.lI42, 100);
en2I42 = (a2, str.lI42, 100);
en3I42 = (a3, str.lI42, 100);
en4I42 = (a4, str.lI42, 100);
en6I42 = (a6, str.lI42, 100);
en7I42 = (a7, str.lI42, 100);
en8I42 = (a8, str.lI42, 100);
en9I42 = (a9, str.lI42, 100);

/* gameA9,  gameC3,  gameC8,  gameD11, gameD12, gameD13,  */
/* gameD14, gameD15, gameD16, gameD17, gameD18, gameD19   */
en1I62 = (a1, str.lI62, 100);
en2I62 = (a2, str.lI62, 100);
en3I62 = (a3, str.lI62, 100);
en4I62 = (a4, str.lI62, 100);
en6I62 = (a6, str.lI62, 100);
en7I62 = (a7, str.lI62, 100);
```

```
en8I62 = (a8, str.lI62, 100);
en9I62 = (a9, str.lI62, 100);

/* gameA4, gameA10, gameB4, gameC4, gameC9, gameD3, gameD8 */
en1I72 = (a1, str.lI72, 100);
en2I72 = (a2, str.lI72, 100);
en3I72 = (a3, str.lI72, 100);
en4I72 = (a4, str.lI72, 100);
en6I72 = (a6, str.lI72, 100);
en7I72 = (a7, str.lI72, 100);
en8I72 = (a8, str.lI72, 100);
en9I72 = (a9, str.lI72, 100);

/* gameA11, gameA12, gameA13, gameA14, gameA15, gameA16,   */
/* gameA17, gameA18, gameA19, gameB9,  gameD4,  gameD9     */
en1I82 = (a1, str.lI82, 100);
en2I82 = (a2, str.lI82, 100);
en3I82 = (a3, str.lI82, 100);
en4I82 = (a4, str.lI82, 100);
en6I82 = (a6, str.lI82, 100);
en7I82 = (a7, str.lI82, 100);
en8I82 = (a8, str.lI82, 100);
en9I82 = (a9, str.lI82, 100);

/* gameA5, gameB5, gameC5, gameC10, gameD5, gameD10 */
en1I92 = (a1, str.lI92, 100);
en2I92 = (a2, str.lI92, 100);
en3I92 = (a3, str.lI92, 100);
en4I92 = (a4, str.lI92, 100);
en6I92 = (a6, str.lI92, 100);
en7I92 = (a7, str.lI92, 100);
en8I92 = (a8, str.lI92, 100);
en9I92 = (a9, str.lI92, 100);

/* gameA11, gameA15, gameB14, gameC14, gameD18, gameD19 */
en1I13 = (a1, str.lI13, 100);
en2I13 = (a2, str.lI13, 100);
en3I13 = (a3, str.lI13, 100);
en4I13 = (a4, str.lI13, 100);
en6I13 = (a6, str.lI13, 100);
en7I13 = (a7, str.lI13, 100);
```

```
en8I13 = (a8, str.lI13, 100);
en9I13 = (a9, str.lI13, 100);

/* gameB11, gameB15, gameD15 */
en1I23 = (a1, str.lI23, 100);
en2I23 = (a2, str.lI23, 100);
en3I23 = (a3, str.lI23, 100);
en4I23 = (a4, str.lI23, 100);
en6I23 = (a6, str.lI23, 100);
en7I23 = (a7, str.lI23, 100);
en8I23 = (a8, str.lI23, 100);
en9I23 = (a9, str.lI23, 100);

/* gameA18, gameA19, gameB12, gameB16, gameC15, gameD11 */
en1I33 = (a1, str.lI33, 100);
en2I33 = (a2, str.lI33, 100);
en3I33 = (a3, str.lI33, 100);
en4I33 = (a4, str.lI33, 100);
en6I33 = (a6, str.lI33, 100);
en7I33 = (a7, str.lI33, 100);
en8I33 = (a8, str.lI33, 100);
en9I33 = (a9, str.lI33, 100);

/* gameA12, gameA16, gameC11 */
en1I43 = (a1, str.lI43, 100);
en2I43 = (a2, str.lI43, 100);
en3I43 = (a3, str.lI43, 100);
en4I43 = (a4, str.lI43, 100);
en6I43 = (a6, str.lI43, 100);
en7I43 = (a7, str.lI43, 100);
en8I43 = (a8, str.lI43, 100);
en9I43 = (a9, str.lI43, 100);

/* gameA17, gameC12, gameC16 */
en1I63 = (a1, str.lI63, 100);
en2I63 = (a2, str.lI63, 100);
en3I63 = (a3, str.lI63, 100);
en4I63 = (a4, str.lI63, 100);
en6I63 = (a6, str.lI63, 100);
en7I63 = (a7, str.lI63, 100);
en8I63 = (a8, str.lI63, 100);
```

```
en9I63 = (a9, str.lI63, 100);

/* gameA13, gameB13, gameC18, gameC19, gameD12, gameD16 */
en1I73 = (a1, str.lI73, 100);
en2I73 = (a2, str.lI73, 100);
en3I73 = (a3, str.lI73, 100);
en4I73 = (a4, str.lI73, 100);
en6I73 = (a6, str.lI73, 100);
en7I73 = (a7, str.lI73, 100);
en8I73 = (a8, str.lI73, 100);
en9I73 = (a9, str.lI73, 100);

/* gameB17, gameD13, gameD17 */
en1I83 = (a1, str.lI83, 100);
en2I83 = (a2, str.lI83, 100);
en3I83 = (a3, str.lI83, 100);
en4I83 = (a4, str.lI83, 100);
en6I83 = (a6, str.lI83, 100);
en7I83 = (a7, str.lI83, 100);
en8I83 = (a8, str.lI83, 100);
en9I83 = (a9, str.lI83, 100);

/* gameA14, gameB18, gameB19, gameC13, gameC17, gameD14 */
en1I93 = (a1, str.lI93, 100);
en2I93 = (a2, str.lI93, 100);
en3I93 = (a3, str.lI93, 100);
en4I93 = (a4, str.lI93, 100);
en6I93 = (a6, str.lI93, 100);
en7I93 = (a7, str.lI93, 100);
en8I93 = (a8, str.lI93, 100);
en9I93 = (a9, str.lI93, 100);

/* gameA18 */
en1I14 = (a1, str.lI14, 100);
en2I14 = (a2, str.lI14, 100);
en3I14 = (a3, str.lI14, 100);
en4I14 = (a4, str.lI14, 100);
en6I14 = (a6, str.lI14, 100);
en7I14 = (a7, str.lI14, 100);
en8I14 = (a8, str.lI14, 100);
en9I14 = (a9, str.lI14, 100);
```

```
/* gameB18 */
en1I24 = (a1, str.lI24, 100);
en2I24 = (a2, str.lI24, 100);
en3I24 = (a3, str.lI24, 100);
en4I24 = (a4, str.lI24, 100);
en6I24 = (a6, str.lI24, 100);
en7I24 = (a7, str.lI24, 100);
en8I24 = (a8, str.lI24, 100);
en9I24 = (a9, str.lI24, 100);

/* gameB19 */
en1I34 = (a1, str.lI34, 100);
en2I34 = (a2, str.lI34, 100);
en3I34 = (a3, str.lI34, 100);
en4I34 = (a4, str.lI34, 100);
en6I34 = (a6, str.lI34, 100);
en7I34 = (a7, str.lI34, 100);
en8I34 = (a8, str.lI34, 100);
en9I34 = (a9, str.lI34, 100);

/* gameA19 */
en1I44 = (a1, str.lI44, 100);
en2I44 = (a2, str.lI44, 100);
en3I44 = (a3, str.lI44, 100);
en4I44 = (a4, str.lI44, 100);
en6I44 = (a6, str.lI44, 100);
en7I44 = (a7, str.lI44, 100);
en8I44 = (a8, str.lI44, 100);
en9I44 = (a9, str.lI44, 100);

/* gameC18 */
en1I64 = (a1, str.lI64, 100);
en2I64 = (a2, str.lI64, 100);
en3I64 = (a3, str.lI64, 100);
en4I64 = (a4, str.lI64, 100);
en6I64 = (a6, str.lI64, 100);
en7I64 = (a7, str.lI64, 100);
en8I64 = (a8, str.lI64, 100);
en9I64 = (a9, str.lI64, 100);
```

```
/* gameD18 */
en1I74 = (a1, str.lI74, 100);
en2I74 = (a2, str.lI74, 100);
en3I74 = (a3, str.lI74, 100);
en4I74 = (a4, str.lI74, 100);
en6I74 = (a6, str.lI74, 100);
en7I74 = (a7, str.lI74, 100);
en8I74 = (a8, str.lI74, 100);
en9I74 = (a9, str.lI74, 100);

/* gameD19 */
en1I84 = (a1, str.lI84, 100);
en2I84 = (a2, str.lI84, 100);
en3I84 = (a3, str.lI84, 100);
en4I84 = (a4, str.lI84, 100);
en6I84 = (a6, str.lI84, 100);
en7I84 = (a7, str.lI84, 100);
en8I84 = (a8, str.lI84, 100);
en9I84 = (a9, str.lI84, 100);

/* gameC19 */
en1I94 = (a1, str.lI94, 100);
en2I94 = (a2, str.lI94, 100);
en3I94 = (a3, str.lI94, 100);
en4I94 = (a4, str.lI94, 100);
en6I94 = (a6, str.lI94, 100);
en7I94 = (a7, str.lI94, 100);
en8I94 = (a8, str.lI94, 100);
en9I94 = (a9, str.lI94, 100);

SIGNAL

/* FLUORESCEIN produces green */
/* TAMRA produces red */

visual1green = (a1, green);
visual1red = (a1, red);
visual2green = (a2, green);
visual2red = (a2, red);
visual3green = (a3, green);
visual3red = (a3, red);
```

```
visual4green = (a4, green);
visual4red = (a4, red);
visual6green = (a6, green);
visual6red = (a6, red);
visual7green = (a7, green);
visual7red = (a7, red);
visual8green = (a8, green);
visual8red = (a8, red);
visual9green = (a9, green);
visual9red = (a9, red);

TRANSITION

with tCase1recog0-1

/* gameA1,  gameA2,  gameA3,  gameA4,  gameA5,  gameA6,  gameA7,  */
/* gameA8,  gameA9,  gameA10, gameA11, gameA12, gameA13, gameA14, */
/* gameA15, gameA16, gameA17, gameA18, gameA19,                   */
/* gameB1,  gameB2,  gameB3,  gameB4,  gameB5,  gameB6,  gameB7,  */
/* gameB8,  gameB9,  gameB10, gameB11, gameB12, gameB13, gameB14, */
/* gameB15, gameB16, gameB17, gameB18, gameB19,                   */
/* gameC1,  gameC2,  gameC3,  gameC4,  gameC5,  gameC6,  gameC7,  */
/* gameC8,  gameC9,  gameC10, gameC11, gameC12, gameC13, gameC14, */
/* gameC15, gameC16, gameC17, gameC18, gameC19,                   */
/* gameD1,  gameD2,  gameD3,  gameD4,  gameD5,  gameD6,  gameD7,  */
/* gameD8,  gameD9,  gameD10, gameD11, gameD12, gameD13, gameD14, */
/* gameD15, gameD16, gameD17, gameD18, gameD19                    */

t1I11 = (a1, str.rI11.stage0, str.rI11.stage1,fold);
t1I12 = (a1, str.rI12.stage0, str.rI12.stage1,fold);
t1I13 = (a1, str.rI13.stage0, str.rI13.stage1,fold);
t1I14 = (a1, str.rI14.stage0, str.rI14.stage1,fold);

t2I21 = (a2, str.rI21.stage0, str.rI21.stage1,fold);
t2I22 = (a2, str.rI22.stage0, str.rI22.stage1,fold);
t2I23 = (a2, str.rI23.stage0, str.rI23.stage1,fold);
t2I24 = (a2, str.rI24.stage0, str.rI24.stage1,fold);
t2I61 = (a2, str.rI61.stage0, str.rI61.stage1,fold);
t2I91 = (a2, str.rI91.stage0, str.rI91.stage1,fold);

t3I31 = (a3, str.rI31.stage0, str.rI31.stage1,fold);
```

```
t3I32 = (a3, str.rI32.stage0, str.rI32.stage1,fold);
t3I33 = (a3, str.rI33.stage0, str.rI33.stage1,fold);
t3I34 = (a3, str.rI34.stage0, str.rI34.stage1,fold);

t4I31 = (a4, str.rI31.stage0, str.rI31.stage1,fold);
t4I41 = (a4, str.rI41.stage0, str.rI41.stage1,fold);
t4I42 = (a4, str.rI42.stage0, str.rI42.stage1,fold);
t4I43 = (a4, str.rI43.stage0, str.rI43.stage1,fold);
t4I44 = (a4, str.rI44.stage0, str.rI44.stage1,fold);
t4I21 = (a4, str.rI21.stage0, str.rI21.stage1,fold);

t6I61 = (a6, str.rI61.stage0, str.rI61.stage1,fold);
t6I62 = (a6, str.rI62.stage0, str.rI62.stage1,fold);
t6I63 = (a6, str.rI63.stage0, str.rI64.stage1,fold);
t6I64 = (a6, str.rI64.stage0, str.rI64.stage1,fold);
t6I71 = (a6, str.rI71.stage0, str.rI71.stage1,fold);
t6I81 = (a6, str.rI81.stage0, str.rI81.stage1,fold);

t7I71 = (a7, str.rI71.stage0, str.rI71.stage1,fold);
t7I72 = (a7, str.rI72.stage0, str.rI72.stage1,fold);
t7I73 = (a7, str.rI73.stage0, str.rI73.stage1,fold);
t7I74 = (a7, str.rI74.stage0, str.rI74.stage1,fold);

t8I81 = (a8, str.rI81.stage0, str.rI81.stage1,fold);
t8I82 = (a8, str.rI82.stage0, str.rI82.stage1,fold);
t8I83 = (a8, str.rI83.stage0, str.rI83.stage1,fold);
t8I84 = (a8, str.rI84.stage0, str.rI84.stage1,fold);
t8I11 = (a8, str.rI11.stage0, str.rI11.stage1,fold);
t8I41 = (a8, str.rI41.stage0, str.rI41.stage1,fold);

t9I91 = (a9, str.rI91.stage0, str.rI91.stage1,fold);
t9I92 = (a9, str.rI92.stage0, str.rI92.stage1,fold);
t9I93 = (a9, str.rI93.stage0, str.rI93.stage1,fold);
t9I94 = (a9, str.rI94.stage0, str.rI94.stage1,fold);


with tCase1recog1-2

/* gameC1,  gameC2,  gameC3, gameC4, gameC5, gameC11, gameC12,    */
/* gameC13, gameC18, gameC19                                      */
t1I11 = (a1, str.rI11.stage1, slI11, str.rI11.stage2, bind);
```

```
t8I11 = (a8, str.rI11.stage1, slI11, str.rI11.stage2, bind);

/* gameD1,  gameD2,  gameD3, gameD4, gameD5, gameD11, gameD12,    */
/* gameD13, gameD14                                               */
t2I21 = (a2, str.rI21.stage1, slI21, str.rI21.stage2, bind);
t4I21 = (a4, str.rI21.stage1, slI21, str.rI21.stage2, bind);

/* gameD6,  gameD7,  gameD8, gameD9, gameD10, gameD15, gameD16,   */
/* gameD17, gameD18, gameD19                                      */
t3I31 = (a3, str.rI31.stage1, slI31, str.rI31.stage2, bind);
t4I31 = (a4, str.rI31.stage1, slI31, str.rI31.stage2, bind);

/* gameC6,  gameC7,  gameC8, gameC9, gameC10, gameC14, gameC15,   */
/* gameC16, gameC17                                               */
t4I41 = (a4, str.rI41.stage1, slI41, str.rI41.stage2, bind);
t8I41 = (a8, str.rI41.stage1, slI41, str.rI41.stage2, bind);

/* gameA1,  gameA2,  gameA3, gameA4, gameA5, gameA11, gameA12,    */
/* gameA13, gameA14, gameA15                                      */
t2I61 = (a2, str.rI61.stage1, slI61, str.rI61.stage2, bind);
t6I61 = (a6, str.rI61.stage1, slI61, str.rI61.stage2, bind);

/* gameB6,  gameB7,  gameB8, gameB9, gameB10, gameB15, gameB16,   */
/* gameB17, gameB18, gameB19                                      */
t6I71 = (a6, str.rI71.stage1, slI71, str.rI71.stage2, bind);
t7I71 = (a7, str.rI71.stage1, slI71, str.rI71.stage2, bind);

/* gameB1,  gameB2,  gameB3, gameB4, gameB5, gameB11, gameB12,    */
/* gameB13, gameB14                                               */
t6I81 = (a6, str.rI81.stage1, slI81, str.rI81.stage2, bind);
t8I81 = (a8, str.rI81.stage1, slI81, str.rI81.stage2, bind);

/* gameA6,  gameA7,  gameA8, gameA9, gameA10, gameA16, gameA17,   */
/* gameA18, gameA19                                               */
t2I91 = (a2, str.rI91.stage1, slI91, str.rI91.stage2, bind);
t9I91 = (a9, str.rI91.stage1, slI91, str.rI91.stage2, bind);

/* gameA1,  gameA6,  gameB1,  gameB6,  gameC6,  gameD1,  gameD6   */
t1I12 = (a1, str.rI12.stage1, slI12, str.rI12.stage2, bind);

/* gameB2,  gameB7,  gameC11, gameC12, gameC13, gameC14, gameC15, */
```

```
/* gameC16, gameC17, gameC18, gameC19, gameD7                    */
t2I22 = (a2, str.rI22.stage1, slI22, str.rI22.stage2, bind);

/* gameA2,  gameA7,  gameB3,  gameB8,  gameC1,  gameC7,  gameD2  */
t3I32 = (a2, str.rI32.stage1, slI32, str.rI32.stage2, bind);

/* gameA3,  gameA8,  gameB11, gameB12, gameB13, gameB14, gameB15, */
/* gameB16, gameB17, gameB18, gameB19, gameC2                    */
t4I42 = (a4, str.rI42.stage1, slI42, str.rI42.stage2, bind);

/* gameA9,  gameC3,  gameC8,  gameD11, gameD12, gameD13, gameD14, */
/* gameD15, gameD16, gameD17, gameD18, gameD19                   */
t6I62 = (a6, str.rI62.stage1, slI62, str.rI62.stage2, bind);

/* gameA4,  gameA10, gameB4,  gameC4,  gameC9,  gameD3,  gameD8  */
t7I72 = (a7, str.rI72.stage1, slI72, str.rI72.stage2, bind);

/* gameA11, gameA12, gameA13, gameA14, gameA15, gameA16, gameA17, */
/* gameA18, gameA19, gameB9,  gameD4,  gameD9                    */
t8I82 = (a8, str.rI82.stage1, slI82, str.rI82.stage2, bind);

/* gameA5,  gameB5,  gameB10, gameC5,  gameC10, gameD5,  gameD10 */
t9I92 = (a9, str.rI92.stage1, slI92, str.rI92.stage2, bind);

/* gameA11, gameA15, gameB14, gameC14, gameD18, gameD19          */
t1I13 = (a1, str.rI13.stage1, slI13, str.rI13.stage2, bind);

/* gameB11, gameB15, gameD15                                     */
t2I23 = (a2, str.rI23.stage1, slI23, str.rI23.stage2, bind);

/* gameA18, gameA19, gameB12, gameB16, gameC15, gameD11          */
t3I33 = (a3, str.rI33.stage1, slI33, str.rI33.stage2, bind);

/* gameA12, gameA16, gameC11                                     */
t4I43 = (a4, str.rI43.stage1, slI43, str.rI43.stage2, bind);

/* gameA17, gameC12, gameC16                                     */
t6I63 = (a6, str.rI63.stage1, slI63, str.rI63.stage2, bind);

/* gameA13, gameB13, gameC18, gameC19, gameD12, gameD16          */
t7I73 = (a7, str.rI73.stage1, slI73, str.rI73.stage2, bind);
```

```
/* gameB17, gameD13, gameD17                                          */
t8I83 = (a8, str.rI83.stage1, slI83, str.rI83.stage2, bind);

/* gameA14, gameB18, gameB19, gameC13, gameC17, gameD14               */
t9I93 = (a9, str.rI93.stage1, slI93, str.rI93.stage2, bind);

/* gameA18                                                            */
t1I14 = (a1, str.rI14.stage1, slI14, str.rI14.stage2, bind);

/* gameB18                                                            */
t2I24 = (a2, str.rI24.stage1, slI24, str.rI24.stage2, bind);

/* gameB19                                                            */
t3I34 = (a3, str.rI34.stage1, slI34, str.rI34.stage2, bind);

/* gameA19                                                            */
t4I44 = (a4, str.rI44.stage1, slI44, str.rI44.stage2, bind);

/* gameC18                                                            */
t6I64 = (a6, str.rI64.stage1, slI64, str.rI64.stage2, bind);

/* gameD18                                                            */
t7I74 = (a7, str.rI74.stage1, slI74, str.rI74.stage2, bind);

/* gameD19                                                            */
t8I84 = (a8, str.rI84.stage1, slI84, str.rI84.stage2, bind);

/* gameC19                                                            */
t9I94 = (a9, str.rI94.stage1, slI94, str.rI94.stage2, bind);


with tCase1recog2-3

/* gameC1,  gameC2,  gameC3, gameC4, gameC5, gameC11, gameC12,        */
/* gameC13, gameC18, gameC19                                          */
t1I11 = (a1, str.rI11.stage2, str.rI11.stage3,fold);
t8I11 = (a8, str.rI11.stage2, str.rI11.stage3,fold);

/* gameD1,  gameD2,  gameD3, gameD4, gameD5, gameD11, gameD12,        */
/* gameD13, gameD14                                                   */
```

```
t2I21 = (a2, str.rI21.stage2, str.rI21.stage3,fold);
t4I21 = (a4, str.rI21.stage2, str.rI21.stage3,fold);

/* gameD6,  gameD7,  gameD8, gameD9, gameD10, gameD15, gameD16,   */
/* gameD17, gameD18, gameD19                                      */
t3I31 = (a3, str.rI31.stage2, str.rI31.stage3,fold);
t4I31 = (a4, str.rI31.stage2, str.rI31.stage3,fold);

/* gameC6,  gameC7,  gameC8, gameC9, gameC10, gameC14, gameC15,   */
/* gameC16, gameC17                                               */
t4I41 = (a4, str.rI41.stage2, str.rI41.stage3,fold);
t8I41 = (a8, str.rI41.stage2, str.rI41.stage3,fold);

/* gameA1,  gameA2,  gameA3, gameA4, gameA5,  gameA11, gameA12,   */
/* gameA13, gameA14, gameA15                                      */
t2I61 = (a2, str.rI61.stage2, str.rI61.stage3,fold);
t6I61 = (a6, str.rI61.stage2, str.rI61.stage3,fold);

/* gameB6,  gameB7,  gameB8, gameB9, gameB10, gameB15, gameB16,   */
/* gameB17, gameB18, gameB19                                      */
t6I71 = (a6, str.rI71.stage2, str.rI71.stage3,fold);
t7I71 = (a7, str.rI71.stage2, str.rI71.stage3,fold);

/* gameB1,  gameB2,  gameB3, gameB4, gameB5,  gameB11, gameB12,   */
/* gameB13, gameB14                                               */
t6I81 = (a6, str.rI81.stage2, str.rI81.stage3,fold);
t8I81 = (a8, str.rI81.stage2, str.rI81.stage3,fold);

/* gameA6,  gameA7,  gameA8, gameA9, gameA10, gameA16, gameA17,   */
/* gameA18, gameA19                                               */
t2I91 = (a2, str.rI91.stage2, str.rI91.stage3,fold);
t9I91 = (a9, str.rI91.stage2, str.rI91.stage3,fold);

/* gameA1,  gameA6,  gameB1,  gameB6,  gameC6,  gameD1,  gameD6   */
t1I12 = (a1, str.rI12.stage2, str.rI12.stage3,fold);

/* gameB2,  gameB7,  gameC11, gameC12, gameC13, gameC14, gameC15, */
/* gameC16, gameC17, gameC18, gameC19, gameD7                     */
t2I22 = (a2, str.rI22.stage2, str.rI22.stage3,fold);

/* gameA2,  gameA7,  gameB3,  gameB8,  gameC1,  gameC7,  gameD2   */
```

```
t3I32 = (a3, str.rI32.stage2, str.rI32.stage3,fold);

/* gameA3,  gameA8,  gameB11, gameB12, gameB13, gameB14, gameB15, */
/* gameB16, gameB17, gameB18, gameB19, gameC2                     */
t4I42 = (a4, str.rI42.stage2, str.rI42.stage3,fold);

/* gameA9,  gameC3,  gameC8,  gameD11, gameD12, gameD13, gameD14, */
/* gameD15, gameD16, gameD17, gameD18, gameD19                    */
t6I62 = (a6, str.rI62.stage2, str.rI62.stage3,fold);

/* gameA4,  gameA10, gameB4,  gameC4,  gameC9,  gameD3,  gameD8   */
t7I72 = (a7, str.rI72.stage2, str.rI72.stage3,fold);

/* gameA11, gameA12, gameA13, gameA14, gameA15, gameA16, gameA17, */
/* gameA18, gameA19, gameB9,  gameD4,  gameD9                     */
t8I82 = (a8, str.rI82.stage2, str.rI82.stage3,fold);

/* gameA5,  gameB5,  gameB10, gameC5,  gameC10, gameD5,  gameD10  */
t9I92 = (a9, str.rI92.stage2, str.rI92.stage3,fold);

/* gameA11, gameA15, gameB14, gameC14, gameD18, gameD19           */
t1I13 = (a1, str.rI13.stage2, str.rI13.stage3,fold);

/* gameB11, gameB15, gameD15                                      */
t2I23 = (a2, str.rI23.stage2, str.rI23.stage3,fold);

/* gameA18, gameA19, gameB12, gameB16, gameC15, gameD11           */
t3I33 = (a3, str.rI33.stage2, str.rI33.stage3,fold);

/* gameA12, gameA16, gameC11                                      */
t4I43 = (a4, str.rI43.stage2, str.rI43.stage3,fold);

/* gameA17, gameC12, gameC16                                      */
t6I63 = (a6, str.rI63.stage2, str.rI63.stage3,fold);

/* gameA13, gameB13, gameC18, gameC19, gameD12, gameD16           */
t7I73 = (a7, str.rI73.stage2, str.rI73.stage3,fold);

/* gameB17, gameD13, gameD17                                      */
t8I83 = (a8, str.rI83.stage2, str.rI83.stage3,fold);
```

```
/* gameA14, gameB18, gameB19, gameC13, gameC17, gameD14          */
t9I93 = (a9, str.rI93.stage2, str.rI93.stage3,fold);

/* gameA18                                                       */
t1I14 = (a1, str.rI14.stage2, str.rI14.stage3,fold);

/* gameB18                                                       */
t2I24 = (a2, str.rI24.stage2, str.rI24.stage3,fold);

/* gameB19                                                       */
t3I34 = (a3, str.rI34.stage2, str.rI34.stage3,fold);

/* gameA19                                                       */
t4I44 = (a4, str.rI44.stage2, str.rI44.stage3,fold);

/* gameC18                                                       */
t6I64 = (a6, str.rI64.stage2, str.rI64.stage3,fold);

/* gameD18                                                       */
t7I74 = (a7, str.rI74.stage2, str.rI74.stage3,fold);

/* gameD19                                                       */
t8I84 = (a8, str.rI84.stage2, str.rI84.stage3,fold);

/* gameC19                                                       */
t9I94 = (a9, str.rI94.stage2, str.rI94.stage3,fold);


with tCase1recog3-4

/* gameC1,  gameC2,  gameC3, gameC4, gameC5, gameC11, gameC12,   */
/* gameC13, gameC18, gameC19                                     */
t1I11 = (a1, str.rI11.stage3, substrateFAM, str.rI11.stage4, bind);
t8I11 = (a8, str.rI11.stage3, substrateTAMRA, str.rI11.stage4, bind);

/* gameD1,  gameD2, gameD3, gameD4, gameD5, gameD11, gameD12,    */
/* gameD13, gameD14                                              */
t2I21 = (a2, str.rI21.stage3, substrateFAM, str.rI21.stage4, bind);
t4I21 = (a4, str.rI21.stage3, substrateTAMRA, str.rI21.stage4, bind);

/* gameD6,  gameD7,  gameD8, gameD9, gameD10, gameD15, gameD16,  */
```

```
/* gameD17, gameD18, gameD19                                          */
t3I31 = (a3, str.rI31.stage3, substrateFAM, str.rI31.stage4, bind);
t4I31 = (a4, str.rI31.stage3, substrateTAMRA, str.rI31.stage4, bind);

/* gameC6,  gameC7, gameC8, gameC9, gameC10, gameC14, gameC15,    */
/* gameC16, gameC17                                                   */
t4I41 = (a4, str.rI41.stage3, substrateFAM, str.rI41.stage4, bind);
t8I41 = (a8, str.rI41.stage3, substrateTAMRA, str.rI41.stage4, bind);

/* gameA1,  gameA2,  gameA3, gameA4, gameA5, gameA11, gameA12,    */
/* gameA13, gameA14, gameA15                                          */
t2I61 = (a2, str.rI61.stage3, substrateTAMRA, str.rI61.stage4, bind);
t6I61 = (a6, str.rI61.stage3, substrateFAM, str.rI61.stage4, bind);

/* gameB6,  gameB7,  gameB8, gameB9, gameB10, gameB15, gameB16,   */
/* gameB17, gameB18, gameB19                                          */
t6I71 = (a6, str.rI71.stage3, substrateTAMRA, str.rI71.stage4, bind);
t7I71 = (a7, str.rI71.stage3, substrateFAM, str.rI71.stage4, bind);

/* gameB1,  gameB2, gameB3, gameB4, gameB5, gameB11, gameB12,     */
/* gameB13, gameB14                                                   */
t6I81 = (a6, str.rI81.stage3, substrateTAMRA, str.rI81.stage4, bind);
t8I81 = (a8, str.rI81.stage3, substrateFAM, str.rI81.stage4, bind);

/* gameA6,  gameA7, gameA8, gameA9, gameA10, gameA16, gameA17,    */
/* gameA18, gameA19                                                   */
t2I91 = (a2, str.rI91.stage3, substrateTAMRA, str.rI91.stage4, bind);
t9I91 = (a9, str.rI91.stage3, substrateFAM, str.rI91.stage4, bind);

/* gameA1,  gameA6, gameB1, gameB6, gameC6,  gameD1,  gameD6      */
t1I12 = (a1, str.rI12.stage3, substrateFAM, str.rI12.stage4, bind);

/* gameB2,  gameB7,  gameC11, gameC12, gameC13, gameC14, gameC15, */
/* gameC16, gameC17, gameC18, gameC19, gameD7                     */
t2I22 = (a2, str.rI22.stage3, substrateFAM, str.rI22.stage4, bind);

/* gameA2,  gameA7,  gameB3,  gameB8,  gameC1,  gameC7,  gameD2   */
t3I32 = (a3, str.rI32.stage3, substrateFAM, str.rI32.stage4, bind);

/* gameA3,  gameA8,  gameB11, gameB12, gameB13, gameB14, gameB15, */
/* gameB16, gameB17, gameB18, gameB19, gameC2                     */
```

```
t4I42 = (a4, str.rI42.stage3, substrateFAM, str.rI42.stage4, bind);

/* gameA9,  gameC3,  gameC8,  gameD11, gameD12, gameD13, gameD14, */
/* gameD15, gameD16, gameD17, gameD18, gameD19                    */
t6I62 = (a6, str.rI62.stage3, substrateFAM, str.rI62.stage4, bind);

/* gameA4,  gameA10, gameB4,  gameC4,  gameC9,  gameD3,  gameD8   */
t7I72 = (a7, str.rI72.stage3, substrateFAM, str.rI72.stage4, bind);

/* gameA11, gameA12, gameA13, gameA14, gameA15, gameA16, gameA17, */
/* gameA18, gameA19, gameB9,  gameD4,  gameD9                     */
t8I82 = (a8, str.rI82.stage3, substrateFAM, str.rI82.stage4, bind);

/* gameA5,  gameB5,  gameB10, gameC5,  gameC10, gameD5,  gameD10  */
t9I92 = (a9, str.rI92.stage3, substrateFAM, str.rI92.stage4, bind);

/* gameA11, gameA15, gameB14, gameC14, gameD18, gameD19           */
t1I13 = (a1, str.rI13.stage3, substrateFAM, str.rI13.stage4, bind);

/* gameB11, gameB15, gameD15                                      */
t2I23 = (a2, str.rI23.stage3, substrateFAM, str.rI23.stage4, bind);

/* gameA18, gameA19, gameB12, gameB16, gameC15, gameD11           */
t3I33 = (a3, str.rI33.stage3, substrateFAM, str.rI33.stage4, bind);

/* gameA12, gameA16, gameC11                                      */
t4I43 = (a4, str.rI43.stage3, substrateFAM, str.rI43.stage4, bind);

/* gameA17, gameC12, gameC16                                      */
t6I63 = (a6, str.rI63.stage3, substrateFAM, str.rI63.stage4, bind);

/* gameA13, gameB13, gameC18, gameC19, gameD12, gameD16           */
t7I73 = (a7, str.rI73.stage3, substrateFAM, str.rI73.stage4, bind);

/* gameB17, gameD13, gameD17                                      */
t8I83 = (a8, str.rI83.stage3, substrateFAM, str.rI83.stage4, bind);

/* gameA14, gameB18, gameB19, gameC13, gameC17, gameD14           */
t9I93 = (a9, str.rI93.stage3, substrateFAM, str.rI93.stage4, bind);

/* gameA18                                                        */
```

```
t1I14 = (a1, str.rI14.stage3, substrateFAM, str.rI14.stage4, bind);

/* gameB18                                                          */
t2I24 = (a2, str.rI24.stage3, substrateFAM, str.rI24.stage4, bind);

/* gameB19                                                          */
t3I34 = (a3, str.rI34.stage3, substrateFAM, str.rI34.stage4, bind);

/* gameA19                                                          */
t4I44 = (a4, str.rI44.stage3, substrateFAM, str.rI44.stage4, bind);

/* gameC18                                                          */
t6I64 = (a6, str.rI64.stage3, substrateFAM, str.rI64.stage4, bind);

/* gameD18                                                          */
t7I74 = (a7, str.rI74.stage3, substrateFAM, str.rI74.stage4, bind);

/* gameD19                                                          */
t8I84 = (a8, str.rI84.stage3, substrateFAM, str.rI84.stage4, bind);

/* gameC19                                                          */
t9I94 = (a9, str.rI94.stage3, substrateFAM, str.rI94.stage4, bind);


with tCase1and0-1

/* gameA1,  gameA2,  gameA3,  gameA4,  gameA5,  gameA6,  gameA7,  */
/* gameA8,  gameA9,  gameA10, gameA11, gameA12, gameA13, gameA14, */
/* gameA15, gameA16, gameA17, gameA18, gameA19,                   */
/* gameB1,  gameB2,  gameB3,  gameB4,  gameB5,  gameB6,  gameB7,  */
/* gameB8,  gameB9,  gameB10, gameB11, gameB12, gameB13, gameB14, */
/* gameB15, gameB16, gameB17, gameB18, gameB19,                   */
/* gameC1,  gameC2,  gameC3,  gameC4,  gameC5,  gameC6,  gameC7,  */
/* gameC8,  gameC9,  gameC10, gameC11, gameC12, gameC13, gameC14, */
/* gameC15, gameC16, gameC17, gameC18, gameC19,                   */
/* gameD1,  gameD2,  gameD3,  gameD4,  gameD5,  gameD6,  gameD7,  */
/* gameD8,  gameD9,  gameD10, gameD11, gameD12, gameD13, gameD14, */
/* gameD15, gameD16, gameD17, gameD18, gameD19                    */

t1I21I62 = (a1, str.aI21I62.stage0, str.aI21I62.stage1,fold);
t1I71I42 = (a1, str.aI71I42.stage0, str.aI71I42.stage1,fold);
```

```
t1I82I73 = (a1, str.aI82I73.stage0, str.aI82I73.stage1,fold);
t1I33I44 = (a1, str.aI33I44.stage0, str.aI33I44.stage1,fold);

t2I62I13 = (a2, str.aI62I13.stage0, str.aI62I13.stage1,fold);
t2I93I34 = (a2, str.aI93I34.stage0, str.aI93I34.stage1,fold);

t3I11I22 = (a3, str.aI11I22.stage0, str.aI11I22.stage1,fold);
t3I61I82 = (a3, str.aI61I82.stage0, str.aI61I82.stage1,fold);
t3I42I13 = (a3, str.aI42I13.stage0, str.aI42I13.stage1,fold);
t3I93I24 = (a3, str.aI93I24.stage0, str.aI93I24.stage1,fold);

t4I22I73 = (a4, str.aI22I73.stage0, str.aI22I73.stage1,fold);
t4I33I14 = (a4, str.aI33I14.stage0, str.aI33I14.stage1,fold);

t6I82I33 = (a6, str.aI82I33.stage0, str.aI82I33.stage1,fold);
t6I73I94 = (a6, str.aI73I94.stage0, str.aI73I94.stage1,fold);

t7I41I22 = (a7, str.aI41I22.stage0, str.aI41I22.stage1,fold);
t7I91I82 = (a7, str.aI91I82.stage0, str.aI91I82.stage1,fold);
t7I62I93 = (a7, str.aI62I93.stage0, str.aI62I93.stage1,fold);
t7I13I84 = (a7, str.aI13I84.stage0, str.aI13I84.stage1,fold);

t8I42I93 = (a8, str.aI42I93.stage0, str.aI42I93.stage1,fold);
t8I13I74 = (a8, str.aI13I74.stage0, str.aI13I74.stage1,fold);

t9I31I62 = (a9, str.aI31I62.stage0, str.aI31I62.stage1,fold);
t9I81I42 = (a9, str.aI81I42.stage0, str.aI81I42.stage1,fold);
t9I22I33 = (a9, str.aI22I33.stage0, str.aI22I33.stage1,fold);
t9I73I64 = (a9, str.aI73I64.stage0, str.aI73I64.stage1,fold);


with tCase1and1-2

/* gameC11, gameC12, gameC13, gameC18, gameC19                    */
t3I11I22 = (a3, str.aI11I22.stage1, slI11,
            str.aI11I22.stage2, bind);

/* gameC14, gameC15, gameC16, gameC17                             */
t7I41I22 = (a7, str.aI41I22.stage1, slI41,
            str.aI41I22.stage2, bind);
```

```
/* gameB15, gameB16, gameB17, gameB18, gameB19                    */
t1I71I42 = (a1, str.aI71I42.stage1, slI71,
            str.aI71I42.stage2, bind);

/* gameB11, gameB12, gameB13, gameB14                             */
t9I81I42 = (a9, str.aI81I42.stage1, slI81,
            str.aI81I42.stage2, bind);

/* gameD15, gameD16, gameD17, gameD18, gameD19                    */
t9I31I62 = (a9, str.aI31I62.stage1, slI31,
            str.aI31I62.stage2, bind);

/* gameA11, gameA12, gameA13, gameA14                             */
t3I61I82 = (a3, str.aI61I82.stage1, slI61,
            str.aI61I82.stage2, bind);

/* gameD11, gameD12, gameD13, gameD14                             */
t1I21I62 = (a1, str.aI21I62.stage1, slI21,
            str.aI21I62.stage2, bind);

/* gameA15, gameA16, gameA17, gameA18, gameA19                    */
t7I91I82 = (a7, str.aI91I82.stage1, slI91,
            str.aI91I82.stage2, bind);

/* gameB18, gameB19                                               */
t8I42I93 = (a8, str.aI42I93.stage1, slI42,
            str.aI42I93.stage2, bind);

/* gameD14                                                        */
t7I62I93 = (a7, str.aI62I93.stage1, slI62,
            str.aI62I93.stage2, bind);

/* gameB14                                                        */
t3I42I13 = (a3, str.aI42I13.stage1, slI42,
            str.aI42I13.stage2, bind);

/* gameA18                                                        */
t4I33I14 = (a4, str.aI33I14.stage1, slI33,
            str.aI33I14.stage2, bind);

/* gameB18                                                        */
```

```
t3I93I24 = (a3, str.aI93I24.stage1, slI93,
            str.aI93I24.stage2, bind);

/* gameB19                                                    */
t2I93I34 = (a2, str.aI93I34.stage1, slI93,
            str.aI93I34.stage2, bind);

/* gameA19                                                    */
t1I33I44 = (a1, str.aI33I44.stage1, slI33,
            str.aI33I44.stage2, bind);

/* gameC18                                                    */
t9I73I64 = (a9, str.aI73I64.stage1, slI73,
            str.aI73I64.stage2, bind);

/* gameD18                                                    */
t8I13I74 = (a8, str.aI13I74.stage1, slI13,
            str.aI13I74.stage2, bind);

/* gameD19                                                    */
t7I13I84 = (a7, str.aI13I84.stage1, slI13,
            str.aI13I84.stage2, bind);

/* gameC19                                                    */
t6I73I94 = (a6, str.aI73I94.stage1, slI73,
            str.aI73I94.stage2, bind);

/* gameD18, gameD19                                           */
t2I62I13 = (a2, str.aI62I13.stage1, slI62,
            str.aI62I13.stage2, bind);

/* gameC15                                                    */
t9I22I33 = (a9, str.aI22I33.stage1, slI22,
            str.aI22I33.stage2, bind);

/* gameA18, gameA19                                           */
t6I82I33 = (a6, str.aI82I33.stage1, slI82,
            str.aI82I33.stage2, bind);

/* gameC18, gameC19                                           */
t4I22I73 = (a4, str.aI22I73.stage1, slI22,
```

```
            str.aI22I73.stage2, bind);

/* gameA13                                                          */
t1I82I73 = (a1, str.aI82I73.stage1, slI82,
            str.aI82I73.stage2, bind);


with tCase1and2-3

/* gameC11, gameC12, gameC13, gameC18, gameC19                      */
t3I11I22 = (a3, str.aI11I22.stage2, slI22,
            str.aI11I22.stage3, bind);

/* gameC14, gameC15, gameC16, gameC17                               */
t7I41I22 = (a7, str.aI41I22.stage2, slI22,
            str.aI41I22.stage3, bind);

/* gameB15, gameB16, gameB17, gameB18, gameB19                      */
t1I71I42 = (a1, str.aI71I42.stage2, slI42,
            str.aI71I42.stage3, bind);

/* gameB11, gameB12, gameB13, gameB14                               */
t9I81I42 = (a9, str.aI81I42.stage2, slI42,
            str.aI81I42.stage3, bind);

/* gameD15, gameD16, gameD17, gameD18, gameD19                      */
t9I31I62 = (a9, str.aI31I62.stage2, slI62,
            str.aI31I62.stage3, bind);

/* gameA11, gameA12, gameA13, gameA14                               */
t3I61I82 = (a3, str.aI61I82.stage2, slI82,
            str.aI61I82.stage3, bind);

/* gameD11, gameD12, gameD13, gameD14                               */
t1I21I62 = (a1, str.aI21I62.stage2, slI62,
            str.aI21I62.stage3, bind);

/* gameA15, gameA16, gameA17, gameA18, gameA19                      */
t7I91I82 = (a7, str.aI91I82.stage2, slI82,
            str.aI91I82.stage3, bind);
```

```
/* gameB18, gameB19                                                    */
t8I42I93 = (a8, str.aI42I93.stage2, slI93,
            str.aI42I93.stage3, bind);

/* gameD14                                                             */
t7I62I93 = (a7, str.aI62I93.stage2, slI93,
            str.aI62I93.stage3, bind);

/* gameB14                                                             */
t3I42I13 = (a3, str.aI42I13.stage2, slI13,
            str.aI42I13.stage3, bind);

/* gameA18                                                             */
t4I33I14 = (a4, str.aI33I14.stage2, slI14,
            str.aI33I14.stage3, bind);

/* gameB18                                                             */
t3I93I24 = (a3, str.aI93I24.stage2, slI24,
            str.aI93I24.stage3, bind);

/* gameB19                                                             */
t2I93I34 = (a2, str.aI93I34.stage2, slI34,
            str.aI93I34.stage3, bind);

/* gameA19                                                             */
t1I33I44 = (a1, str.aI33I44.stage2, slI44,
            str.aI33I44.stage3, bind);

/* gameC18                                                             */
t9I73I64 = (a9, str.aI73I64.stage2, slI64,
            str.aI73I64.stage3, bind);

/* gameD18                                                             */
t8I13I74 = (a8, str.aI13I74.stage2, slI74,
            str.aI13I74.stage3, bind);

/* gameD19                                                             */
t7I13I84 = (a7, str.aI13I84.stage2, slI84,
            str.aI13I84.stage3, bind);

/* gameC19                                                             */
```

```
t6I73I94 = (a6, str.aI73I94.stage2, slI94,
            str.aI73I94.stage3, bind);

/* gameD18, gameD19                                                    */
t2I62I13 = (a2, str.aI62I13.stage2, slI13,
            str.aI62I13.stage3, bind);

/* gameC15                                                             */
t9I22I33 = (a9, str.aI22I33.stage2, slI33,
            str.aI22I33.stage3, bind);

/* gameA18, gameA19                                                    */
t6I82I33 = (a6, str.aI82I33.stage2, slI33,
            str.aI82I33.stage3, bind);

/* gameC18, gameC19                                                    */
t4I22I73 = (a4, str.aI22I73.stage2, slI73,
            str.aI22I73.stage3, bind);

/* gameA13                                                             */
t1I82I73 = (a1, str.aI82I73.stage2, slI73,
            str.aI82I73.stage3, bind);


with tCase1and3-4

/* gameC11, gameC12, gameC13, gameC18, gameC19                         */
t3I11I22 = (a3, str.aI11I22.stage3, str.aI11I22.stage4,fold);

/* gameC14, gameC15, gameC16, gameC17                                  */
t7I41I22 = (a7, str.aI41I22.stage3, str.aI41I22.stage4,fold);

/* gameB15, gameB16, gameB17, gameB18, gameB19                         */
t1I71I42 = (a1, str.aI71I42.stage3, str.aI71I42.stage4,fold);

/* gameB11, gameB12, gameB13, gameB14                                  */
t9I81I42 = (a9, str.aI81I42.stage3, str.aI81I42.stage4,fold);

/* gameD15, gameD16, gameD17, gameD18, gameD19                         */
t9I31I62 = (a9, str.aI31I62.stage3, str.aI31I62.stage4,fold);
```

```
/* gameA11, gameA12, gameA13, gameA14                                  */
t3I61I82 = (a3, str.aI61I82.stage3, str.aI61I82.stage4,fold);

/* gameD11, gameD12, gameD13, gameD14                                  */
t1I21I62 = (a1, str.aI21I62.stage3, str.aI21I62.stage4,fold);

/* gameA15, gameA16, gameA17, gameA18, gameA19                         */
t7I91I82 = (a7, str.aI91I82.stage3, str.aI91I82.stage4,fold);

/* gameB18, gameB19                                                    */
t8I42I93 = (a8, str.aI42I93.stage3, str.aI42I93.stage4,fold);

/* gameD14                                                             */
t7I62I93 = (a7, str.aI62I93.stage3, str.aI62I93.stage4,fold);

/* gameB14                                                             */
t3I42I13 = (a3, str.aI42I13.stage3, str.aI42I13.stage4,fold);

/* gameA18                                                             */
t4I33I14 = (a4, str.aI33I14.stage3, str.aI33I14.stage4,fold);

/* gameB18                                                             */
t3I93I24 = (a3, str.aI93I24.stage3, str.aI93I24.stage4,fold);

/* gameB19                                                             */
t2I93I34 = (a2, str.aI93I34.stage3, str.aI93I34.stage4,fold);

/* gameA19                                                             */
t1I33I44 = (a1, str.aI33I44.stage3, str.aI33I44.stage4,fold);

/* gameC18                                                             */
t9I73I64 = (a9, str.aI73I64.stage3, str.aI73I64.stage4,fold);

/* gameD18                                                             */
t8I13I74 = (a8, str.aI13I74.stage3, str.aI13I74.stage4,fold);

/* gameD19                                                             */
t7I13I84 = (a7, str.aI13I84.stage3, str.aI13I84.stage4,fold);

/* gameC19                                                             */
t6I73I94 = (a6, str.aI73I94.stage3, str.aI73I94.stage4,fold);
```

```
/* gameD18, gameD19                                                    */
t2I62I13 = (a2, str.aI62I13.stage3, str.aI62I13.stage4,fold);

/* gameC15                                                             */
t9I22I33 = (a9, str.aI22I33.stage3, str.aI22I33.stage4,fold);

/* gameA18, gameA19                                                    */
t6I82I33 = (a6, str.aI82I33.stage3, str.aI82I33.stage4,fold);

/* gameC18, gameC19                                                    */
t4I22I73 = (a4, str.aI22I73.stage3, str.aI22I73.stage4,fold);

/* gameA13                                                             */
t1I82I73 = (a1, str.aI82I73.stage3, str.aI82I73.stage4,fold);


with tCase1and4-5

/* gameC11, gameC12, gameC13, gameC18, gameC19                         */
t3I11I22 = (a3, str.aI11I22.stage4, substrateTAMRA,
            str.aI11I22.stage5, bind);

/* gameC14, gameC15, gameC16, gameC17                                  */
t7I41I22 = (a7, str.aI41I22.stage4, substrateTAMRA,
            str.aI41I22.stage5, bind);

/* gameB15, gameB16, gameB17, gameB18, gameB19                         */
t1I71I42 = (a1, str.aI71I42.stage4, substrateTAMRA,
            str.aI71I42.stage5, bind);

/* gameB11, gameB12, gameB13, gameB14                                  */
t9I81I42 = (a9, str.aI81I42.stage4, substrateTAMRA,
            str.aI81I42.stage5, bind);

/* gameD15, gameD16, gameD17, gameD18, gameD19                         */
t9I31I62 = (a9, str.aI31I62.stage4, substrateTAMRA,
            str.aI31I62.stage5, bind);

/* gameA11, gameA12, gameA13, gameA14                                  */
t3I61I82 = (a3, str.aI61I82.stage4, substrateTAMRA,
```

```
            str.aI61I82.stage5, bind);

/* gameD11, gameD12, gameD13, gameD14                                  */
t1I21I62 = (a1, str.aI21I62.stage4, substrateTAMRA,
            str.aI21I62.stage5, bind);

/* gameA15, gameA16, gameA17, gameA18, gameA19                         */
t7I91I82 = (a7, str.aI91I82.stage4, substrateTAMRA,
            str.aI91I82.stage5, bind);

/* gameB18, gameB19                                                   */
t8I42I93 = (a8, str.aI42I93.stage4, substrateTAMRA,
            str.aI42I93.stage5, bind);

/* gameD14                                                            */
t7I62I93 = (a7, str.aI62I93.stage4, substrateTAMRA,
            str.aI62I93.stage5, bind);

/* gameB14                                                            */
t3I42I13 = (a3, str.aI42I13.stage4, substrateTAMRA,
            str.aI42I13.stage5, bind);

/* gameA18                                                            */
t4I33I14 = (a4, str.aI33I14.stage4, substrateTAMRA,
            str.aI33I14.stage5, bind);

/* gameB18                                                            */
t3I93I24 = (a3, str.aI93I24.stage4, substrateTAMRA,
            str.aI93I24.stage5, bind);

/* gameB19                                                            */
t2I93I34 = (a2, str.aI93I34.stage4, substrateTAMRA,
            str.aI93I34.stage5, bind);

/* gameA19                                                            */
t1I33I44 = (a1, str.aI33I44.stage4, substrateTAMRA,
            str.aI33I44.stage5, bind);

/* gameC18                                                            */
t9I73I64 = (a9, str.aI73I64.stage4, substrateTAMRA,
            str.aI73I64.stage5, bind);
```

```
/* gameD18                                                          */
t8I13I74 = (a8, str.aI13I74.stage4, substrateTAMRA,
            str.aI13I74.stage5, bind);

/* gameD19                                                          */
t7I13I84 = (a7, str.aI13I84.stage4, substrateTAMRA,
            str.aI13I84.stage5, bind);

/* gameC19                                                          */
t6I73I94 = (a6, str.aI73I94.stage4, substrateTAMRA,
            str.aI73I94.stage5, bind);

/* gameD18, gameD19                                                 */
t2I62I13 = (a2, str.aI62I13.stage4, substrateTAMRA,
            str.aI62I13.stage5, bind);

/* gameC15                                                          */
t9I22I33 = (a9, str.aI22I33.stage4, substrateTAMRA,
            str.aI22I33.stage5, bind);

/* gameA18, gameA19                                                 */
t6I82I33 = (a6, str.aI82I33.stage4, substrateTAMRA,
            str.aI82I33.stage5, bind);

/* gameC18, gameC19                                                 */
t4I22I73 = (a4, str.aI22I73.stage4, substrateTAMRA,
            str.aI22I73.stage5, bind);

/* gameA13                                                          */
t1I82I73 = (a1, str.aI82I73.stage4, substrateTAMRA,
            str.aI82I73.stage5, bind);


with tCase1andandnot0-1

/* gameA1,  gameA2,  gameA3,  gameA4,  gameA5,  gameA6,  gameA7,  */
/* gameA8,  gameA9,  gameA10, gameA11, gameA12, gameA13, gameA14, */
/* gameA15, gameA16, gameA17, gameA18, gameA19,                   */
/* gameB1,  gameB2,  gameB3,  gameB4,  gameB5,  gameB6,  gameB7,  */
/* gameB8,  gameB9,  gameB10, gameB11, gameB12, gameB13, gameB14, */
```

```
/* gameB15, gameB16, gameB17, gameB18, gameB19,                    */
/* gameC1,  gameC2,  gameC3,  gameC4,  gameC5,  gameC6,  gameC7,  */
/* gameC8,  gameC9,  gameC10, gameC11, gameC12, gameC13, gameC14, */
/* gameC15, gameC16, gameC17, gameC18, gameC19,                    */
/* gameD1,  gameD2,  gameD3,  gameD4,  gameD5,  gameD6,  gameD7,  */
/* gameD8,  gameD9,  gameD10, gameD11, gameD12, gameD13, gameD14, */
/* gameD15, gameD16, gameD17, gameD18, gameD19                     */

t1I42I73I71 = (a1, str.aanI42I73I71.stage0,
               str.aanI42I73I71.stage1,fold);
t1I42I33I71 = (a1, str.aanI42I33I71.stage0,
               str.aanI42I33I71.stage1,fold);
t1I42I23I71 = (a1, str.aanI42I23I71.stage0,
               str.aanI42I23I71.stage1,fold);
t1I62I73I21 = (a1, str.aanI62I73I21.stage0,
               str.aanI62I73I21.stage1,fold);
t1I62I83I21 = (a1, str.aanI62I83I21.stage0,
               str.aanI62I83I21.stage1,fold);
t1I62I23I21 = (a1, str.aanI62I23I21.stage0,
               str.aanI62I23I21.stage1,fold);

t2I11I32I22 = (a2, str.aanI11I32I22.stage0,
               str.aanI11I32I22.stage1,fold);
t2I11I42I22 = (a2, str.aanI11I42I22.stage0,
               str.aanI11I42I22.stage1,fold);
t2I11I62I22 = (a2, str.aanI11I62I22.stage0,
               str.aanI11I62I22.stage1,fold);
t2I11I72I22 = (a2, str.aanI11I72I22.stage0,
               str.aanI11I72I22.stage1,fold);
t2I11I92I22 = (a2, str.aanI11I92I22.stage0,
               str.aanI11I92I22.stage1,fold);
t2I41I12I22 = (a2, str.aanI4I12I22.stage0,
               str.aanI41I12I22.stage1,fold);
t2I41I32I22 = (a2, str.aanI41I32I22.stage0,
               str.aanI41I32I22.stage1,fold);
t2I41I62I22 = (a2, str.aanI41I62I22.stage0,
               str.aanI41I62I22.stage1,fold);
t2I41I72I22 = (a2, str.aanI41I72I22.stage0,
               str.aanI41I72I22.stage1,fold);
t2I41I92I22 = (a2, str.aanI41I92I22.stage0,
               str.aanI41I92I22.stage1,fold);
```

```
t3I22I13I11 = (a3, str.aanI22I13I11.stage0,
               str.aanI22I13I11.stage1,fold);
t3I22I63I11 = (a3, str.aanI22I63I11.stage0,
               str.aanI22I63I11.stage1,fold);
t3I22I93I11 = (a3, str.aanI22I93I11.stage0,
               str.aanI22I93I11.stage1,fold);
t3I82I31I61 = (a3, str.aanI82I13I61.stage0,
               str.aanI82I13I61.stage1,fold);
t3I81I43I61 = (a3, str.aanI82I43I61.stage0,
               str.aanI82I43I61.stage1,fold);
t3I82I63I61 = (a3, str.aanI82I63I61.stage0,
               str.aanI82I63I61.stage1,fold);

t4I71I12I42 = (a4, str.aanI71I12I42.stage0,
               str.aanI71I12I42.stage1,fold);
t4I71I22I42 = (a4, str.aanI71I22I42.stage0,
               str.aanI71I22I42.stage1,fold);
t4I71I32I42 = (a4, str.aanI71I32I42.stage0,
               str.aanI71I32I42.stage1,fold);
t4I71I82I42 = (a4, str.aanI71I82I42.stage0,
               str.aanI71I82I42.stage1,fold);
t4I71I92I42 = (a4, str.aanI71I92I42.stage0,
               str.aanI71I92I42.stage1,fold);
t4I81I12I42 = (a4, str.aanI81I12I42.stage0,
               str.aanI81I12I42.stage1,fold);
t4I81I22I42 = (a4, str.aanI81I22I42.stage0,
               str.aanI81I22I42.stage1,fold);
t4I81I32I42 = (a4, str.aanI81I32I42.stage0,
               str.anI81I32I42.stage1,fold);
t4I81I72I42 = (a4, str.aanI81I72I42.stage0,
               str.anI81I72I42.stage1,fold);
t4I81I92I42 = (a4, str.aanI81I92I42.stage0,
               str.sanI81I92I42.stage1,fold);

t6I21I12I62 = (a6, str.aanI21I12I62.stage0,
               str.aanI21I12I62.stage1,fold);
t6I21I32I62 = (a6, str.aanI21I32I62.stage0,
               str.aanI21I32I62.stage1,fold);
t6I21I72I62 = (a6, str.aanI21I72I62.stage0,
               str.aanI21I72I62.stage1,fold);
```

```
t6I21I82I62 = (a6, str.aanI21I82I62.stage0,
               str.aanI21I82I62.stage1,fold);
t6I21I92I62 = (a6, str.aanI21I92I62.stage0,
               str.aanI21I92I62.stage1,fold);
t6I31I12I62 = (a6, str.aanI31I12I62.stage0,
               str.aanI31I12I62.stage1,fold);
t6I31I22I62 = (a6, str.aanI31I22I62.stage0,
               str.aanI31I22I62.stage1,fold);
t6I31I72I62 = (a6, str.aanI31I72I62.stage0,
               str.aanI31I72I62.stage1,fold);
t6I31I82I62 = (a6, str.aanI31I82I62.stage0,
               str.aanI31I82I62.stage1,fold);
t6I31I92I62 = (a6, str.aanI31I92I62.stage0,
               str.aanI31I92I62.stage1,fold);

t7I22I43I41 = (a7, str.aanI22I43I41.stage0,
               str.aanI22I43I41.stage1,fold);
t7I22I63I41 = (a7, str.aanI22I63I41.stage0,
               str.aanI22I63I41.stage1,fold);
t7I22I93I41 = (a7, str.aanI22I93I41.stage0,
               str.aanI22I93I41.stage1,fold);
t7I82I13I91 = (a7, str.aanI82I13I91.stage0,
               str.aanI82I13I91.stage1,fold);
t7I82I43I91 = (a7, str.aanI82I43I91.stage0,
               str.aanI82I43I91.stage1,fold);
t7I82I93I91 = (a7, str.aanI82I93I91.stage0,
               str.aanI82I93I91.stage1,fold);

t8I61I12I82 = (a8, str.aanI61I12I82.stage0,
               str.aanI61I12I82.stage1,fold);
t8I61I32I82 = (a8, str.aanI61I32I82.stage0,
               str.aanI61I32I82.stage1,fold);
t8I61I42I82 = (a8, str.aanI61I42I82.stage0,
               str.aanI61I42I82.stage1,fold);
t8I61I72I82 = (a8, str.aanI61I72I82.stage0,
               str.aanI61I72I82.stage1,fold);
t8I61I92I82 = (a8, str.aanI61I92I82.stage0,
               str.aanI61I92I82.stage1,fold);
t8I91I12I82 = (a8, str.aanI91I12I82.stage0,
               str.aanI91I12I82.stage1,fold);
t8I91I32I82 = (a8, str.aanI91I32I82.stage0,
```

```
               str.aanI91I32I82.stage1,fold);
t8I91I42I82 = (a8, str.aanI91I42I82.stage0,
               str.aanI91I42I82.stage1,fold);
t8I91I62I82 = (a8, str.aanI91I62I82.stage0,
               str.aanI91I62I82.stage1,fold);
t8I91I72I82 = (a8, str.aanI91I72I82.stage0,
               str.aanI91I72I82.stage1,fold);

t9I42I23I81 = (a9, str.aanI42I23I81.stage0,
               str.aanI42I23I81.stage1,fold);
t9I42I33I81 = (a9, str.aanI42I33I81.stage0,
               str.aanI42I33I81.stage1,fold);
t9I42I83I81 = (a9, str.aanI42I83I81.stage0,
               str.aanI42I83I81.stage1,fold);
t9I62I33I31 = (a9, str.aanI62I33I31.stage0,
               str.aanI62I33I31.stage1,fold);
t9I62I73I31 = (a9, str.aanI62I73I31.stage0,
               str.aanI62I73I31.stage1,fold);
t9I62I83I31 = (a9, str.aanI62I83I31.stage0,
               str.aanI62I83I31.stage1,fold);


with tCase1andandnot1-2

/* gameD1                                                          */
t6I21I12I62 = (a6, str.aanI21I12I62.stage1, slI21,
               str.aanI21I12I62.stage2, bind);

/* gameD6                                                          */
t6I31I12I62 = (a6, str.aanI31I12I62.stage1, slI31,
               str.aanI31I12I62.stage2, bind);

/* gameC6                                                          */
t2I41I12I22 = (a2, str.aanI41I12I22.stage1, slI41,
               str.aanI41I12I22.stage2, bind);

/* gameA1, gameA6                                                  */
t8I61I12I82 = (a8, str.aanI61I12I82.stage1, slI61,
               str.aanI61I12I82.stage2, bind);

/* gameB6                                                          */
```

```
t4I71I12I42 = (a4, str.aanI71I12I42.stage1, slI71,
               str.aanI71I12I42.stage2, bind);

/* gameB1                                                    */
t4I81I12I42 = (a4, str.aanI81I12I42.stage1, slI81,
               str.aanI81I12I42.stage2, bind);

/* gameD7                                                    */
t6I31I22I62 = (a6, str.aanI31I22I62.stage1, slI31,
               str.aanI31I22I62.stage2, bind);

/* gameB2                                                    */
t4I81I22I42 = (a4, str.aanI81I22I42.stage1, slI81,
               str.aanI81I22I42.stage2, bind);

/* gameB7                                                    */
t4I71I22I42 = (a4, str.aanI71I22I42.stage1, slI71,
               str.aanI71I22I42.stage2, bind);

/* gameC1                                                    */
t2I11I32I22 = (a2, str.aanI11I32I22.stage1, slI11,
               str.aanI11I32I22.stage2, bind);

/*gameD2                                                     */
t6I21I32I62 = (a6, str.aanI21I32I62.stage1, slI21,
               str.aanI21I32I62.stage2, bind);

/* gameC7                                                    */
t2I41I32I22 = (a2, str.aanI41I32I22.stage1, slI41,
               str.aanI41I32I22.stage2, bind);

/* gameA2                                                    */
t8I61I32I82 = (a8, str.aanI61I32I82.stage1, slI61,
               str.aanI61I32I82.stage2, bind);

/* gameB8                                                    */
t4I71I32I42 = (a4, str.aanI71I32I42.stage1, slI71,
               str.aanI71I32I42.stage2, bind);

/* gameB3                                                    */
t4I81I32I42 = (a4, str.aanI81I32I42.stage1, slI81,
```

```
              str.aanI81I32I42.stage2, bind);

/* gameC2                                                          */
t2I11I42I22 = (a2, str.aanI11I42I22.stage1, slI11,
              str.aanI11I42I22.stage2, bind);

/* gameA3                                                          */
t8I61I42I82 = (a8, str.aanI61I42I82.stage1, slI61,
              str.aanI61I42I82.stage2, bind);

/* gameC3                                                          */
t2I11I62I22 = (a2, str.aanI11I62I22.stage1, slI11,
              str.aanI11I62I22.stage2, bind);

/* gameC8                                                          */
t2I41I62I22 = (a2, str.aanI41I62I22.stage1, slI41,
              str.aanI41I62I22.stage2, bind);

/* gameA7                                                          */
t8I91I32I82 = (a8, str.aanI91I32I82.stage1, slI91,
              str.aanI91I32I82.stage2, bind);

/* gameA8                                                          */
t8I91I42I82 = (a8, str.aanI91I42I82.stage1, slI91,
              str.aanI91I42I82.stage2, bind);

/* gameA9                                                          */
t8I91I62I82 = (a8, str.aanI91I62I82.stage1, slI91,
              str.aanI91I62I82.stage2, bind);

/* gameC4                                                          */
t2I11I72I22 = (a2, str.aanI11I72I22.stage1, slI11,
              str.aanI11I72I22.stage2, bind);

/* gameD3                                                          */
t6I21I72I62 = (a6, str.aanI21I72I62.stage1, slI21,
              str.aanI21I72I62.stage2, bind);

/* gameD8                                                          */
t6I31I72I62 = (a6, str.aanI31I72I62.stage1, slI31,
              str.aanI31I72I62.stage2, bind);
```

```
/* gameC9                                                    */
t2I42I72I22 = (a2, str.aanI41I72I22.stage1, slI41,
               str.aanI41I72I22.stage2, bind);

/* gameA4                                                    */
t8I61I72I82 = (a8, str.aanI61I72I82.stage1, slI61,
               str.aanI61I72I82.stage2, bind);

/* gameA5                                                    */
t8I61I92I82 = (a8, str.aanI61I92I82.stage1, slI61,
               str.aanI61I92I82.stage2, bind);

/* gameB4                                                    */
t4I81I72I42 = (a4, str.aanI81I72I42.stage1, slI81,
               str.aanI81I72I42.stage2, bind);

/* gameB5                                                    */
t4I81I92I42 = (a4, str.aanI81I92I42.stage1, slI81,
               str.aanI81I92I42.stage2, bind);

/* gameA10                                                   */
t8I91I72I82 = (a8, str.aanI91I72I82.stage1, slI91,
               str.aanI91I72I82.stage2, bind);

/* gameD4                                                    */
t6I21I82I62 = (a6, str.aanI21I82I62.stage1, slI21,
               str.aanI21I82I62.stage2, bind);

/* gameD9                                                    */
t6I31I82I62 = (a6, str.aanI31I82I62.stage1, slI31,
               str.aanI31I82I62.stage2, bind);

/* gameB9                                                    */
t4I71I82I42 = (a4, str.aanI71I82I42.stage1, slI71,
               str.aanI71I82I42.stage2, bind);

/* gameB10                                                   */
t4I71I92I42 = (a4, str.aanI71I92I42.stage1, slI71,
               str.aanI71I92I42.stage2, bind);
```

```
/* gameC5                                                          */
t2I11I92I22 = (a2, str.aanI11I92I22.stage1, slI11,
               str.aanI11I92I22.stage2, bind);

/* gameD5                                                          */
t6I21I92I22 = (a6, str.aanI21I92I22.stage1, slI21,
               str.aanI21I92I22.stage2, bind);

/* gameD10                                                         */
t6I31I92I22 = (a6, str.aanI31I92I22.stage1, slI31,
               str.aanI31I92I22.stage2, bind);

/* gameC10                                                         */
t2I41I92I22 = (a2, str.aanI41I92I22.stage1, slI41,
               str.aanI41I92I22.stage2, bind);

/* gameC14                                                         */
t3I22I13I11 = (a3, str.aanI22I13I11.stage1, slI22,
               str.aanI22I13I11.stage2, bind);

/* gameA11                                                         */
t7I82I13I91 = (a7, str.aanI82I13I91.stage1, slI82,
               str.aanI82I13I91.stage2, bind);

/* gameA15                                                         */
t3I82I13I61 = (a3, str.aanI82I13I61.stage1, slI82,
               str.aanI82I13I61.stage2, bind);

/* gameD15                                                         */
t1I62I23I21 = (a1, str.aanI62I23I21.stage1, slI62,
               str.aanI62I23I21.stage2, bind);

/* gameB11                                                         */
t1I42I23I71 = (a1, str.aanI42I23I71.stage1, slI42,
               str.aanI42I23I71.stage2, bind);

/* gameB15                                                         */
t9I42I23I81 = (a9, str.aanI42I23I81.stage1, slI42,
               str.aanI42I23I81.stage2, bind);

/* gameB12                                                         */
```

```
t1I42I33I71 = (a1, str.aanI42I33I71.stage1, slI42,
               str.aanI42I33I71.stage2, bind);

/* gameB16                                                    */
t9I42I33I91 = (a9, str.aanI42I33I91.stage1, slI42,
               str.aanI42I33I91.stage2, bind);

/* gameD11                                                    */
t9I61I33I31 = (a9, str.aanI62I33I31.stage1, slI62,
               str.aanI62I33I31.stage2, bind);

/* gameC11                                                    */
t7I22I43I41 = (a7, str.aanI22I43I41.stage1, slI22,
               str.aanI22I43I41.stage2, bind);

/* gameA16                                                    */
t3I82I43I61 = (a3, str.aanI82I43I61.stage1, slI82,
               str.aanI82I43I61.stage2, bind);

/* gameA12                                                    */
t7I81I43I91 = (a7, str.aanI82I43I91.stage1, slI82,
               str.aanI82I43I91.stage2, bind);

/* gameC12                                                    */
t7I22I63I41 = (a7, str.aanI22I63I41.stage1, slI22,
               str.aanI22I63I41.stage2, bind);

/* gameC16                                                    */
t3I22I63I11 = (a3, str.aanI22I63I11.stage1, slI22,
               str.aanI22I63I11.stage2, bind);

/* gameA17                                                    */
t3I81I63I61 = (a3, str.aanI82I63I61.stage1, slI82,
               str.aanI82I63I61.stage2, bind);

/* gameB13                                                    */
t1I41I73I71 = (a1, str.aanI42I73I71.stage1, slI42,
               str.aanI42I73I71.stage2, bind);

/* gameD12                                                    */
t9I62I73I31 = (a9, str.aanI62I73I31.stage1, slI62,
```

```
              str.aanI62I73I31.stage2, bind);

/* gameD16                                                          */
t1I62I73I21 = (a1, str.aanI62I73I21.stage1, slI62,
              str.aanI62I73I21.stage2, bind);

/* gameD17                                                          */
t1I62I83I21 = (a1, str.aanI62I83I21.stage1, slI62,
              str.aanI62I83I21.stage2, bind);

/* gameD13                                                          */
t9I62I83I31 = (a9, str.aanI62I83I31.stage1, slI62,
              str.aanI62I83I31.stage2, bind);

/* gameB17                                                          */
t9I42I83I81 = (a9, str.aanI42I83I81.stage1, slI42,
              str.aanI42I83I81.stage2, bind);

/* gameC13                                                          */
t7I22I93I41 = (a7, str.aanI22I93I41.stage1, slI22,
              str.aanI22I93I41.stage2, bind);

/* gameC17                                                          */
t3I22I93I11 = (a3, str.aanI22I93I11.stage1, slI22,
              str.aanI22I93I11.stage2, bind);

/* gameA14                                                          */
t7I82I93I91 = (a7, str.aanI82I93I91.stage1, slI82,
              str.aanI82I93I91.stage2, bind);


with tCase1andandnot2-3

/* gameD1                                                           */
t6I21I12I62 = (a6, str.aanI21I12I62.stage2, slI12,
              str.aanI21I12I62.stage3, bind);

/* gameD6                                                           */
t6I31I12I62 = (a6, str.aanI31I12I62.stage2, slI12,
              str.aanI31I12I62.stage3, bind);
```

```
/* gameC6                                                          */
t2I41I12I22 = (a2, str.aanI41I12I22.stage2, slI12,
               str.aanI41I12I22.stage3, bind);

/* gameA1, gameA6                                                  */
t8I61I12I82 = (a8, str.aanI61I12I82.stage2, slI12,
               str.aanI61I12I82.stage3, bind);

/* gameB6                                                          */
t4I71I12I42 = (a4, str.aanI71I12I42.stage2, slI12,
               str.aanI71I12I42.stage3, bind);

/* gameB1                                                          */
t4I81I12I42 = (a4, str.aanI81I12I42.stage2, slI12,
               str.aanI81I12I42.stage3, bind);

/* gameD7                                                          */
t6I31I22I62 = (a6, str.aanI31I22I62.stage2, slI22,
               str.aanI31I22I62.stage3, bind);

/* gameB2                                                          */
t4I81I22I42 = (a4, str.aanI81I22I42.stage2, slI22,
               str.aanI81I22I42.stage3, bind);

/* gameB7                                                          */
t4I71I22I42 = (a4, str.aanI71I22I42.stage2, slI22,
               str.aanI71I22I42.stage3, bind);

/* gameC1                                                          */
t2I11I32I22 = (a2, str.aanI11I32I22.stage2, slI32,
               str.aanI11I32I22.stage3, bind);

/* gameD2                                                          */
t6I21I32I62 = (a6, str.aanI21I32I62.stage2, slI32,
               str.aanI21I32I62.stage3, bind);

/* gameC7                                                          */
t2I41I32I22 = (a2, str.aanI41I32I22.stage2, slI32,
               str.aanI41I32I22.stage3, bind);

/* gameA2                                                          */
```

```
t8I61I32I82 = (a8, str.aanI61I32I82.stage2, slI32,
               str.aanI61I32I82.stage3, bind);

/* gameB8                                                    */
t4I71I32I42 = (a4, str.aanI71I32I42.stage2, slI32,
               str.aanI71I32I42.stage3, bind);

/* gameB3                                                    */
t4I81I32I42 = (a4, str.aanI81I32I42.stage2, slI32,
               str.aanI81I32I42.stage3, bind);

/* gameC2                                                    */
t2I11I42I22 = (a2, str.aanI11I42I22.stage2, slI42,
               str.aanI11I42I22.stage3, bind);

/* gameA3                                                    */
t8I61I42I82 = (a8, str.aanI61I42I82.stage2, slI42,
               str.aanI61I42I82.stage3, bind);

/* gameC3                                                    */
t2I11I62I22 = (a2, str.aanI11I62I22.stage2, slI62,
               str.aanI11I62I22.stage3, bind);

/* gameC8                                                    */
t2I41I62I22 = (a2, str.aanI41I62I22.stage2, slI62,
               str.aanI41I62I22.stage3, bind);

/* gameA7                                                    */
t8I91I32I82 = (a8, str.aanI91I32I82.stage2, slI32,
               str.aanI91I32I82.stage3, bind);

/* gameA8                                                    */
t8I91I42I82 = (a8, str.aanI91I42I82.stage2, slI42,
               str.aanI91I42I82.stage3, bind);

/* gameA9                                                    */
t8I91I62I82 = (a8, str.aanI91I62I82.stage2, slI62,
               str.aanI91I62I82.stage3, bind);

/* gameC4                                                    */
t2I11I72I22 = (a2, str.aanI11I72I22.stage2, slI72,
```

```
              str.aanI11I72I22.stage3, bind);

/* gameD3                                                          */
t6I21I72I62 = (a6, str.aanI21I72I62.stage2, slI72,
              str.aanI21I72I62.stage3, bind);

/* gameD8                                                          */
t6I31I72I62 = (a6, str.aanI31I72I62.stage2, slI72,
              str.aanI31I72I62.stage3, bind);

/* gameC9                                                          */
t2I41I72I22 = (a2, str.aanI41I72I22.stage2, slI72,
              str.aanI41I72I22.stage3, bind);

/* gameA4                                                          */
t8I61I72I82 = (a8, str.aanI61I72I82.stage2, slI72,
              str.aanI61I72I82.stage3, bind);

/* gameA5                                                          */
t8I61I92I82 = (a8, str.aanI61I92I82.stage2, slI92,
              str.aanI61I92I82.stage3, bind);

/* gameB4                                                          */
t4I81I72I42 = (a4, str.aanI81I72I42.stage2, slI72,
              str.aanI81I72I42.stage3, bind);

/* gameB5                                                          */
t4I81I92I42 = (a4, str.aanI81I92I42.stage2, slI92,
              str.aanI81I92I42.stage3, bind);

/* gameA10                                                         */
t8I91I72I82 = (a8, str.aanI91I72I82.stage2, slI72,
              str.aanI91I72I82.stage3, bind);

/* gameD4                                                          */
t6I21I82I62 = (a6, str.aanI21I82I62.stage2, slI82,
              str.aanI21I82I62.stage3, bind);

/* gameD9                                                          */
t6I31I82I62 = (a6, str.aanI31I82I62.stage2, slI82,
              str.aanI31I82I62.stage3, bind);
```

```
/* gameB9                                                          */
t4I71I82I42 = (a4, str.aanI71I82I42.stage2, slI82,
               str.aanI71I82I42.stage3, bind);

/* gameB10                                                         */
t4I71I92I42 = (a4, str.aanI71I92I42.stage2, slI92,
               str.aanI71I92I42.stage3, bind);

/* gameC5                                                          */
t2I11I92I22 = (a2, str.aanI11I92I22.stage2, slI92,
               str.aanI11I92I22.stage3, bind);

/* gameD5                                                          */
t6I21I92I22 = (a6, str.aanI21I92I22.stage2, slI92,
               str.aanI21I92I22.stage3, bind);

/* gameD10                                                         */
t6I31I92I22 = (a6, str.aanI31I92I22.stage2, slI92,
               str.aanI31I92I22.stage3, bind);

/* gameC10                                                         */
t2I41I92I22 = (a2, str.aanI41I92I22.stage2, slI92,
               str.aanI41I92I22.stage3, bind);

/* gameC14                                                         */
t3I22I13I11 = (a3, str.aanI22I13I11.stage2, slI13,
               str.aanI22I13I11.stage3, bind);

/* gameA11                                                         */
t7I82I13I91 = (a7, str.aanI82I13I91.stage2, slI13,
               str.aanI82I13I91.stage3, bind);

/* gameA15                                                         */
t3I82I13I61 = (a3, str.aanI82I13I61.stage2, slI13,
               str.aanI82I13I61.stage3, bind);

/* gameD15                                                         */
t1I62I23I21 = (a1, str.aanI62I23I21.stage2, slI23,
               str.aanI62I23I21.stage3, bind);
```

```
/* gameB11                                                    */
t1I42I23I71 = (a1, str.aanI42I23I71.stage2, slI23,
               str.aanI42I23I71.stage3, bind);

/* gameB15                                                    */
t9I42I23I81 = (a9, str.aanI42I23I81.stage2, slI23,
               str.aanI42I23I81.stage3, bind);

/* gameB12                                                    */
t1I42I33I71 = (a1, str.aanI42I33I71.stage2, slI33,
               str.aanI42I33I71.stage3, bind);

/* gameB16                                                    */
t9I42I33I91 = (a9, str.aanI42I33I91.stage2, slI33,
               str.aanI42I33I91.stage3, bind);

/* gameD11                                                    */
t9I62I33I31 = (a9, str.aanI62I33I31.stage2, slI33,
               str.aanI62I33I31.stage3, bind);

/* gameC11                                                    */
t7I22I43I41 = (a7, str.aanI22I43I41.stage2, slI43,
               str.aanI22I43I41.stage3, bind);

/* gameA16                                                    */
t3I82I43I61 = (a3, str.aanI82I43I61.stage2, slI43,
               str.aanI82I43I61.stage3, bind);

/* gameA12                                                    */
t7I82I43I91 = (a7, str.aanI82I43I91.stage2, slI43,
               str.aanI82I43I91.stage3, bind);

/* gameC12                                                    */
t7I22I63I41 = (a7, str.aanI22I63I41.stage2, slI63,
               str.aanI22I63I41.stage3, bind);

/* gameC16                                                    */
t3I22I63I11 = (a3, str.aanI22I63I11.stage2, slI63,
               str.aanI22I63I11.stage3, bind);

/* gameA17                                                    */
```

```
t3I82I63I61 = (a3, str.aanI82I63I61.stage2, slI63,
               str.aanI82I63I61.stage3, bind);

/* gameB13                                                    */
t1I42I73I71 = (a1, str.aanI42I73I71.stage2, slI73,
               str.aanI42I73I71.stage3, bind);

/* gameD12                                                    */
t9I62I73I31 = (a9, str.aanI62I73I31.stage2, slI73,
               str.aanI62I73I31.stage3, bind);

/* gameD16                                                    */
t1I62I73I21 = (a1, str.aanI62I73I21.stage2, slI73,
               str.aanI62I73I21.stage3, bind);

/* gameD17                                                    */
t1I62I83I21 = (a1, str.aanI62I83I21.stage2, slI83,
               str.aanI62I83I21.stage3, bind);

/* gameD13                                                    */
t9I62I83I31 = (a9, str.aanI62I83I31.stage2, slI83,
               str.aanI62I83I31.stage3, bind);

/* gameB17                                                    */
t9I42I83I81 = (a9, str.aanI42I83I81.stage2, slI83,
               str.aanI42I83I81.stage3, bind);

/* gameC13                                                    */
t7I22I93I41 = (a7, str.aanI22I93I41.stage2, slI93,
               str.aanI22I93I41.stage3, bind);

/* gameC17                                                    */
t3I22I93I11 = (a3, str.aanI22I93I11.stage2, slI93,
               str.aanI22I93I11.stage3, bind);

/* gameA14                                                    */
t7I82I93I91 = (a7, str.aanI82I93I91.stage2, slI93,
               str.aanI82I93I91.stage3, bind);


with tCase1andandnot3-4
```

```
/* gameD1                                                          */
t6I21I12I62 = (a6, str.aanI21I12I62.stage3,
               str.aanI21I12I62.stage4,fold);

/* gameD6                                                          */
t6I31I12I62 = (a6, str.aanI31I12I62.stage3,
               str.aanI31I12I62.stage4,fold);

/* gameC6                                                          */
t2I41I12I22 = (a2, str.aanI41I12I22.stage3,
               str.aanI41I12I22.stage4,fold);

/* gameA1, gameA6                                                  */
t8I61I12I82 = (a8, str.aanI61I12I82.stage3,
               str.aanI61I12I82.stage4,fold);

/* gameB6                                                          */
t4I71I12I42 = (a4, str.aanI71I12I42.stage3,
               str.aanI71I12I42.stage4,fold);

/* gameB1                                                          */
t4I81I12I42 = (a4, str.aanI81I12I42.stage3,
               str.aanI81I12I42.stage4,fold);

/* gameD7                                                          */
t6I31I22I62 = (a6, str.aanI31I22I62.stage3,
               str.aanI31I22I62.stage4,fold);

/* gameB2                                                          */
t4I81I22I42 = (a4, str.aanI81I22I42.stage3,
               str.aanI81I22I42.stage4,fold);

/* gameB7                                                          */
t4I71I22I42 = (a4, str.aanI71I22I42.stage3,
               str.aanI71I22I42.stage4,fold);

/* gameC1                                                          */
t2I11I32I22 = (a2, str.aanI11I32I22.stage3,
               str.aanI11I32I22.stage4,fold);
```

```
/* gameD2                                                          */
t6I21I32I62 = (a6, str.aanI21I32I62.stage3,
               str.aanI21I32I62.stage4,fold);

/* gameC7                                                          */
t2I41I32I22 = (a2, str.aanI41I32I22.stage3,
               str.aanI41I32I22.stage4,fold);

/* gameA2                                                          */
t8I61I32I82 = (a8, str.aanI61I32I82.stage3,
               str.aanI61I32I82.stage4,fold);

/* gameB8                                                          */
t4I71I32I42 = (a4, str.aanI71I32I42.stage3,
               str.aanI71I32I42.stage4,fold);

/* gameB3                                                          */
t4I81I32I42 = (a4, str.aanI81I32I42.stage3,
               str.aanI81I32I42.stage4,fold);

/* gameC2                                                          */
t2I11I42I22 = (a2, str.aanI11I42I22.stage3,
               str.aanI11I42I22.stage4,fold);

/* gameA3                                                          */
t8I61I42I82 = (a8, str.aanI61I42I82.stage3,
               str.aanI61I42I82.stage4,fold);

/* gameC3                                                          */
t2I11I62I22 = (a2, str.aanI11I62I22.stage3,
               str.aanI11I62I22.stage4,fold);

/* gameC8                                                          */
t2I41I62I22 = (a2, str.aanI41I62I22.stage3,
               str.aanI41I62I22.stage4,fold);

/* gameA7                                                          */
t8I91I32I82 = (a8, str.aanI91I32I82.stage3,
               str.aanI91I32I82.stage4,fold);

/* gameA8                                                          */
```

```
t8I91I42I82 = (a8, str.aanI91I42I82.stage3,
               str.aanI91I42I82.stage4,fold);

/* gameA9                                                    */
t8I91I62I82 = (a8, str.aanI91I62I82.stage3,
               str.aanI91I62I82.stage4,fold);

/* gameC4                                                    */
t2I11I72I22 = (a2, str.aanI11I72I22.stage3,
               str.aanI11I72I22.stage4,fold);

/* gameD3                                                    */
t6I21I72I62 = (a6, str.aanI21I72I62.stage3,
               str.aanI21I72I62.stage4,fold);

/* gameD8                                                    */
t6I31I72I62 = (a6, str.aanI31I72I62.stage3,
               str.aanI31I72I62.stage4,fold);

/* gameC9                                                    */
t2I41I72I22 = (a2, str.aanI41I72I22.stage3,
               str.aanI41I72I22.stage4,fold);

/* gameA4                                                    */
t8I61I72I82 = (a8, str.aanI61I72I82.stage3,
               str.aanI61I72I82.stage4,fold);

/* gameA5                                                    */
t8I61I92I82 = (a8, str.aanI61I92I82.stage3,
               str.aanI61I92I82.stage4,fold);

/* gameB4                                                    */
t4I81I72I42 = (a4, str.aanI81I72I42.stage3,
               str.aanI81I72I42.stage4,fold);

/* gameB5                                                    */
t4I81I92I42 = (a4, str.aanI81I92I42.stage3,
               str.aanI81I92I42.stage4,fold);

/* gameA10                                                   */
t8I91I72I82 = (a8, str.aanI91I72I82.stage3,
```

```
              str.aanI91I72I82.stage4,fold);

/* gameD4                                                          */
t6I21I82I62 = (a6, str.aanI21I82I62.stage3,
              str.aanI21I82I62.stage4,fold);

/* gameD9                                                          */
t6I31I82I62 = (a6, str.aanI31I82I62.stage3,
              str.aanI31I82I62.stage4,fold);

/* gameB9                                                          */
t4I71I82I42 = (a4, str.aanI71I82I42.stage3,
              str.aanI71I82I42.stage4,fold);

/* gameB10                                                         */
t4I71I92I42 = (a4, str.aanI71I92I42.stage3,
              str.aanI71I92I42.stage4,fold);

/* gameC5                                                          */
t2I11I92I22 = (a2, str.aanI11I92I22.stage3,
              str.aanI11I92I22.stage4,fold);

/* gameD5                                                          */
t6I21I92I22 = (a6, str.aanI21I92I22.stage3,
              str.aanI21I92I22.stage4,fold);

/* gameD10                                                         */
t6I31I92I22 = (a6, str.aanI31I92I22.stage3,
              str.aanI31I92I22.stage4,fold);

/* gameC10                                                         */
t2I41I92I22 = (a2, str.aanI41I92I22.stage3,
              str.aanI41I92I22.stage4,fold);

/* gameC14                                                         */
t3I22I13I11 = (a3, str.aanI22I13I11.stage3,
              str.aanI22I13I11.stage4,fold);

/* gameA11                                                         */
t7I82I13I91 = (a7, str.aanI82I13I91.stage3,
              str.aanI82I13I91.stage4,fold);
```

```
/* gameA15                                                    */
t3I82I13I61 = (a3, str.aanI82I13I61.stage3,
               str.aanI82I13I61.stage4,fold);

/* gameD15                                                    */
t1I62I23I21 = (a1, str.aanI62I23I21.stage3,
               str.aanI62I23I21.stage4,fold);

/* gameB11                                                    */
t1I42I23I71 = (a1, str.aanI42I23I71.stage3,
               str.aanI42I23I71.stage4,fold);

/* gameB15                                                    */
t9I42I23I81 = (a9, str.aanI42I23I81.stage3,
               str.aanI42I23I81.stage4,fold);

/* gameB12                                                    */
t1I42I33I71 = (a1, str.aanI42I33I71.stage3,
               str.aanI42I33I71.stage4,fold);

/* gameB16                                                    */
t9I42I33I91 = (a9, str.aanI42I33I91.stage3,
               str.aanI42I33I91.stage4,fold);

/* gameD11                                                    */
t9I62I33I31 = (a9, str.aanI62I33I31.stage3,
               str.aanI62I33I31.stage4,fold);

/* gameC11                                                    */
t7I22I43I41 = (a7, str.aanI22I43I41.stage3,
               str.aanI22I43I41.stage4,fold);

/* gameA16                                                    */
t3I82I43I61 = (a3, str.aanI82I43I61.stage3,
               str.aanI82I43I61.stage4,fold);

/* gameA12                                                    */
t7I82I43I91 = (a7, str.aanI82I43I91.stage3,
               str.aanI82I43I91.stage4,fold);
```

```
/* gameC12                                                    */
t7I22I63I41 = (a7, str.aanI22I63I41.stage3,
               str.aanI22I63I41.stage4,fold);

/* gameC16                                                    */
t3I22I63I11 = (a3, str.aanI22I63I11.stage3,
               str.aanI22I63I11.stage4,fold);

/* gameA17                                                    */
t3I82I63I61 = (a3, str.aanI82I63I61.stage3,
               str.aanI82I63I61.stage4,fold);

/* gameB13                                                    */
t1I42I73I71 = (a1, str.aanI42I73I71.stage3,
               str.aanI42I73I71.stage4,fold);

/* gameD12                                                    */
t9I62I73I31 = (a9, str.aanI62I73I31.stage3,
               str.aanI62I73I31.stage4,fold);

/* gameD16                                                    */
t1I62I73I21 = (a1, str.aanI62I73I21.stage3,
               str.aanI62I73I21.stage4,fold);

/* gameD17                                                    */
t1I62I83I21 = (a1, str.aanI62I83I21.stage3,
               str.aanI62I83I21.stage4,fold);

/* gameD13                                                    */
t9I62I83I31 = (a9, str.aanI62I83I31.stage3,
               str.aanI62I83I31.stage4,fold);

/* gameB17                                                    */
t9I42I83I81 = (a9, str.aanI42I83I81.stage3,
               str.aanI42I83I81.stage4,fold);

/* gameC13                                                    */
t7I22I93I41 = (a7, str.aanI22I93I41.stage3,
               str.aanI22I93I41.stage4,fold);

/* gameC17                                                    */
```

```
t3I22I93I11 = (a3, str.aanI22I93I11.stage3,
               str.aanI22I93I11.stage4,fold);

/* gameA14                                                    */
t7I82I93I91 = (a7, str.aanI82I93I91.stage3,
               str.aanI82I93I91.stage4,fold);


with tCase1andandnot4-5

/* gameD1                                                     */
t6I21I12I62 = (a6, str.aanI21I12I62.stage4, substrateTAMRA,
               str.aanI21I12I62.stage5, bind);

/* gameD6                                                     */
t6I31I12I62 = (a6, str.aanI31I12I62.stage4, substrateTAMRA,
               str.aanI31I12I62.stage5, bind);

/* gameC6                                                     */
t2I41I21I22 = (a2, str.aanI41I12I22.stage4, substrateTAMRA,
               str.aanI41I12I22.stage5, bind);

/* gameA1, gameA6                                             */
t8I61I12I82 = (a8, str.aanI61I12I82.stage4, substrateTAMRA,
               str.aanI61I12I82.stage5, bind);

/* gameB6                                                     */
t4I71I12I42 = (a4, str.aanI71I12I42.stage4, substrateTAMRA,
               str.aanI71I12I42.stage5, bind);

/* gameB1                                                     */
t4I81I12I42 = (a4, str.aanI81I12I42.stage4, substrateTAMRA,
               str.aanI81I12I42.stage5, bind);

/* gameD7                                                     */
t6I31I22I62 = (a6, str.aanI31I22I62.stage4, substrateTAMRA,
               str.aanI31I22I62.stage5, bind);

/* gameB2                                                     */
t4I81I22I42 = (a4, str.aanI81I22I42.stage4, substrateTAMRA,
               str.aanI81I22I42.stage5, bind);
```

```
/* gameB7                                                          */
t4I71I22I42 = (a4, str.aanI71I22I42.stage4, substrateTAMRA,
               str.aanI71I22I42.stage5, bind);

/* gameC1                                                          */
t2I11I32I22 = (a2, str.aanI11I32I22.stage4, substrateTAMRA,
               str.aanI11I32I22.stage5, bind);

/* gameD2                                                          */
t6I21I32I62 = (a6, str.aanI21I32I62.stage4, substrateTAMRA,
               str.aanI21I32I62.stage5, bind);

/* gameC7                                                          */
t2I41I32I22 = (a2, str.aanI41I32I22.stage4, substrateTAMRA,
               str.aanI41I32I22.stage5, bind);

/* gameA2                                                          */
t8I61I32I82 = (a8, str.aanI61I32I82.stage4, substrateTAMRA,
               str.aanI61I32I82.stage5, bind);

/* gameB8                                                          */
t4I71I32I42 = (a4, str.aanI71I32I42.stage4, substrateTAMRA,
               str.aanI71I32I42.stage5, bind);

/* gameB3                                                          */
t4I81I32I42 = (a4, str.aanI81I32I42.stage4, substrateTAMRA,
               str.aanI81I32I42.stage5, bind);

/* gameC2                                                          */
t2I11I42I22 = (a2, str.aanI11I42I22.stage4, substrateTAMRA,
               str.aanI11I42I22.stage5, bind);

/* gameA3                                                          */
t8I61I42I82 = (a8, str.aanI61I42I82.stage4, substrateTAMRA,
               str.aanI61I42I82.stage5, bind);

/* gameC3                                                          */
t2I11I62I22 = (a2, str.aanI11I62I22.stage4, substrateTAMRA,
               str.aanI11I62I22.stage5, bind);
```

```
/* gameC8                                                          */
t2I41I62I22 = (a2, str.aanI41I62I22.stage4, substrateTAMRA,
               str.aanI41I62I22.stage5, bind);

/* gameA7                                                          */
t8I91I32I82 = (a8, str.aanI91I32I82.stage4, substrateTAMRA,
               str.aanI91I32I82.stage5, bind);

/* gameA8                                                          */
t8I91I42I82 = (a8, str.aanI91I42I82.stage4, substrateTAMRA,
               str.aanI91I42I82.stage5, bind);

/* gameA9                                                          */
t8I91I62I82 = (a8, str.aanI91I62I82.stage4, substrateTAMRA,
               str.aanI91I62I82.stage5, bind);

/* gameC4                                                          */
t2I11I72I22 = (a2, str.aanI11I72I22.stage4, substrateTAMRA,
               str.aanI11I72I22.stage5, bind);

/* gameD3                                                          */
t6I21I72I62 = (a6, str.aanI21I72I62.stage4, substrateTAMRA,
               str.aanI21I72I62.stage5, bind);

/* gameD8                                                          */
t6I31I72I62 = (a6, str.aanI31I72I62.stage4, substrateTAMRA,
               str.aanI31I72I62.stage5, bind);

/* gameC9                                                          */
t2I41I72I22 = (a2, str.aanI41I72I22.stage4, substrateTAMRA,
               str.aanI41I72I22.stage5, bind);

/* gameA4                                                          */
t8I61I72I82 = (a8, str.aanI61I72I82.stage4, substrateTAMRA,
               str.aanI61I72I82.stage5, bind);

/* gameA5                                                          */
t8I61I92I82 = (a8, str.aanI61I92I82.stage4, substrateTAMRA,
               str.aanI61I92I82.stage5, bind);

/* gameB4                                                          */
```

```
t4I81I72I42 = (a4, str.aanI81I72I42.stage4, substrateTAMRA,
               str.aanI81I72I42.stage5, bind);

/* gameB5                                                    */
t4I81I92I42 = (a4, str.aanI81I92I42.stage4, substrateTAMRA,
               str.aanI81I92I42.stage5, bind);

/* gameA10                                                   */
t8I91I72I82 = (a8, str.aanI91I72I82.stage4, substrateTAMRA,
               str.aanI91I72I82.stage5, bind);

/* gameD4                                                    */
t6I21I82I62 = (a6, str.aanI21I82I62.stage4, substrateTAMRA,
               str.aanI21I82I62.stage5, bind);

/* gameD9                                                    */
t6I31I82I62 = (a6, str.aanI31I82I62.stage4, substrateTAMRA,
               str.aanI31I82I62.stage5, bind);

/* gameB9                                                    */
t4I71I82I42 = (a4, str.aanI71I82I42.stage4, substrateTAMRA,
               str.aanI71I82I42.stage5, bind);

/* gameB10                                                   */
t4I71I92I42 = (a4, str.aanI71I92I42.stage4, substrateTAMRA,
               str.aanI71I92I42.stage5, bind);

/* gameC5                                                    */
t2I11I92I22 = (a2, str.aanI11I92I22.stage4, substrateTAMRA,
               str.aanI11I92I22.stage5, bind);

/* gameD5                                                    */
t6I21I92I22 = (a6, str.aanI21I92I22.stage4, substrateTAMRA,
               str.aanI21I92I22.stage5, bind);

/* gameD10                                                   */
t6I31I92I22 = (a6, str.aanI31I92I22.stage4, substrateTAMRA,
               str.aanI31I92I22.stage5, bind);

/* gameC10                                                   */
t2I41I92I22 = (a2, str.aanI41I92I22.stage4, substrateTAMRA,
```

```
              str.aanI41I92I22.stage5, bind);

/* gameC14                                                          */
t3I22I13I11 = (a3, str.aanI22I13I11.stage4, substrateTAMRA,
              str.aanI22I13I11.stage5, bind);

/* gameA11                                                          */
t7I82I13I91 = (a7, str.aanI82I13I91.stage4, substrateTAMRA,
              str.aanI82I13I91.stage5, bind);

/* gameA15                                                          */
t3I82I13I61 = (a3, str.aanI82I13I61.stage4, substrateTAMRA,
              str.aanI82I13I61.stage5, bind);

/* gameD15                                                          */
t1I62I23I21 = (a1, str.aanI62I23I21.stage4, substrateTAMRA,
              str.aanI62I23I21.stage5, bind);

/* gameB11                                                          */
t1I42I23I71 = (a1, str.aanI42I23I71.stage4, substrateTAMRA,
              str.aanI42I23I71.stage5, bind);

/* gameB15                                                          */
t9I42I23I81 = (a9, str.aanI42I23I81.stage4, substrateTAMRA,
              str.aanI42I23I81.stage5, bind);

/* gameB12                                                          */
t1I42I33I71 = (a1, str.aanI42I33I71.stage4, substrateTAMRA,
              str.aanI42I33I71.stage5, bind);

/* gameB16                                                          */
t9I42I33I91 = (a9, str.aanI42I33I91.stage4, substrateTAMRA,
              str.aanI42I33I91.stage5, bind);

/* gameD11                                                          */
t9I62I33I31 = (a9, str.aanI62I33I31.stage4, substrateTAMRA,
              str.aanI62I33I31.stage5, bind);

/* gameC11                                                          */
t7I22I43I41 = (a7, str.aanI22I43I41.stage4, substrateTAMRA,
              str.aanI22I43I41.stage5, bind);
```

```
/* gameA16                                                    */
t3I82I43I61 = (a3, str.aanI82I43I61.stage4, substrateTAMRA,
               str.aanI82I43I61.stage5, bind);

/* gameA12                                                    */
t7I82I43I91 = (a7, str.aanI82I43I91.stage4, substrateTAMRA,
               str.aanI82I43I91.stage5, bind);

/* gameC12                                                    */
t7I22I63I41 = (a7, str.aanI22I63I41.stage4, substrateTAMRA,
               str.aanI22I63I41.stage5, bind);

/* gameC16                                                    */
t3I22I63I11 = (a3, str.aanI22I63I11.stage4, substrateTAMRA,
               str.aanI22I63I11.stage5, bind);

/* gameA17                                                    */
t3I82I63I61 = (a3, str.aanI82I63I61.stage4, substrateTAMRA,
               str.aanI82I63I61.stage5, bind);

/* gameB13                                                    */
t1I42I73I71 = (a1, str.aanI42I73I71.stage4, substrateTAMRA,
               str.aanI42I73I71.stage5, bind);

/* gameD12                                                    */
t9I62I73I31 = (a9, str.aanI62I73I31.stage4, substrateTAMRA,
               str.aanI62I73I31.stage5, bind);

/* gameD16                                                    */
t1I62I73I21 = (a1, str.aanI62I73I21.stage4, substrateTAMRA,
               str.aanI62I73I21.stage5, bind);

/* gameD17                                                    */
t1I62I83I21 = (a1, str.aanI62I83I21.stage4, substrateTAMRA,
               str.aanI62I83I21.stage5, bind);

/* gameD13                                                    */
t9I62I83I31 = (a9, str.aanI62I83I31.stage4, substrateTAMRA,
               str.aanI62I83I31.stage5, bind);
```

```
/* gameB17                                                          */
t9I42I83I81 = (a9, str.aanI42I83I81.stage4, substrateTAMRA,
               str.aanI42I83I81.stage5, bind);

/* gameC13                                                          */
t7I22I93I41 = (a7, str.aanI22I93I41.stage4, substrateTAMRA,
               str.aanI22I93I41.stage5, bind);

/* gameC17                                                          */
t3I22I93I11 = (a3, str.aanI22I93I11.stage4, substrateTAMRA,
               str.aanI22I93I11.stage5, bind);

/* gameA14                                                          */
t7I82I93I91 = (a7, str.aanI82I93I91.stage4, substrateTAMRA,
               str.aanI82I93I91.stage5, bind);


EXECUTIONMECHANISM

with cmCase1recog

/* gameA1,  gameA2,  gameA3,  gameA4,  gameA5,  gameA6,  gameA7,  */
/* gameA8,  gameA9,  gameA10, gameA11, gameA12, gameA13, gameA14, */
/* gameA15, gameA16, gameA17, gameA18, gameA19,                   */
/* gameB1,  gameB2,  gameB3,  gameB4,  gameB5,  gameB6,  gameB7,  */
/* gameB8,  gameB9,  gameB10, gameB11, gameB12, gameB13, gameB14, */
/* gameB15, gameB16, gameB17, gameB18, gameB19,                   */
/* gameC1,  gameC2,  gameC3,  gameC4,  gameC5,  gameC6,  gameC7,  */
/* gameC8,  gameC9,  gameC10, gameC11, gameC12, gameC13, gameC14, */
/* gameC15, gameC16, gameC17, gameC18, gameC19,                   */
/* gameD1,  gameD2,  gameD3,  gameD4,  gameD5,  gameD6,  gameD7,  */
/* gameD8,  gameD9,  gameD10, gameD11, gameD12, gameD13, gameD14, */
/* gameD15, gameD16, gameD17, gameD18, gameD19                    */

c1I11 = [tCase1recog0-1.t1I11];
c1I12 = [tCase1recog0-1.t1I12];
c1I13 = [tCase1recog0-1.t1I13];
c1I14 = [tCase1recog0-1.t1I14];

c2I21 = [tCase1recog0-1.t2I21];
c2I22 = [tCase1recog0-1.t2I22];
```

```
c2I23 = [tCase1recog0-1.t2I23];
c2I24 = [tCase1recog0-1.t2I24];
c2I61 = [tCase1recog0-1.t2I61];
c2I91 = [tCase1recog0-1.t2I91];

c3I31 = [tCase1recog0-1.t3I31];
c3I32 = [tCase1recog0-1.t3I32];
c3I33 = [tCase1recog0-1.t3I33];
c3I34 = [tCase1recog0-1.t3I34];

c4I41 = [tCase1recog0-1.t4I41];
c4I42 = [tCase1recog0-1.t4I42];
c4I43 = [tCase1recog0-1.t4I32];
c4I44 = [tCase1recog0-1.t4I44];
c4I21 = [tCase1recog0-1.t4I21];
c4I31 = [tCase1recog0-1.t4I31];

c6I61 = [tCase1recog0-1.t6I61];
c6I62 = [tCase1recog0-1.t6I62];
c6I63 = [tCase1recog0-1.t6I63];
c6I64 = [tCase1recog0-1.t6I64];
c6I71 = [tCase1recog0-1.t6I71];
c6I81 = [tCase1recog0-1.t6I81];

c7I71 = [tCase1recog0-1.t7I71];
c7I72 = [tCase1recog0-1.t7I72];
c7I73 = [tCase1recog0-1.t7I73];
c7I74 = [tCase1recog0-1.t7I74];

c8I81 = [tCase1recog0-1.t8I81];
c8I82 = [tCase1recog0-1.t8I82];
c8I83 = [tCase1recog0-1.t8I83];
c8I84 = [tCase1recog0-1.t8I84];
c8I11 = [tCase1recog0-1.t8I11];
c8I41 = [tCase1recog0-1.t8I41];

c9I91 = [tCase1recog0-1.t9I91];
c9I92 = [tCase1recog0-1.t9I92];
c9I93 = [tCase1recog0-1.t9I93];
c9I94 = [tCase1recog0-1.t9I94];
```

```
with cmCase1and

/* gameA1,  gameA2,  gameA3,  gameA4,  gameA5,  gameA6,  gameA7,  */
/* gameA8,  gameA9,  gameA10, gameA11, gameA12, gameA13, gameA14, */
/* gameA15, gameA16, gameA17, gameA18, gameA19,                   */
/* gameB1,  gameB2,  gameB3,  gameB4,  gameB5,  gameB6,  gameB7,  */
/* gameB8,  gameB9,  gameB10, gameB11, gameB12, gameB13, gameB14, */
/* gameB15, gameB16, gameB17, gameB18, gameB19,                   */
/* gameC1,  gameC2,  gameC3,  gameC4,  gameC5,  gameC6,  gameC7,  */
/* gameC8,  gameC9,  gameC10, gameC11, gameC12, gameC13, gameC14, */
/* gameC15, gameC16, gameC17, gameC18, gameC19,                   */
/* gameD1,  gameD2,  gameD3,  gameD4,  gameD5,  gameD6,  gameD7,  */
/* gameD8,  gameD9,  gameD10, gameD11, gameD12, gameD13, gameD14, */
/* gameD15, gameD16, gameD17, gameD18, gameD19                    */

c1I21I62 = [tCase1and0-1.t1I21I62];
c1I33I44 = [tCase1and0-1.t1I33I44];
c1I82I73 = [tCase1and0-1.t1I82I73];
c1I71I42 = [tCase1and0-1.t1I71I42];

c2I62I13 = [tCase1and0-1.t2I62I13];
c2I93I34 = [tCase1and0-1.t2I93I34];

c3I11I22 = [tCase1and0-1.t3I11I22];
c3I61I82 = [tCase1and0-1.t3I61I82];
c3I42I13 = [tCase1and0-1.t3I42I13];
c3I93I24 = [tCase1and0-1.t3I93I24];

c4I22I73 = [tCase1and0-1.t4I22I73];
c4I33I14 = [tCase1and0-1.t4I33I14];

c6I73I94 = [tCase1and0-1.t6I73I94];
c6I82I33 = [tCase1and0-1.t6I82I33];

c7I41I22 = [tCase1and0-1.t7I41I22];
c7I62I93 = [tCase1and0-1.t7I62I93];
c7I13I84 = [tCase1and0-1.t7I13I84];
c7I91I82 = [tCase1and0-1.t7I91I82];

c8I13I74 = [tCase1and0-1.t8I13I74];
```

```
c8I42I93 = [tCase1and0-1.t8I42I93];

c9I73I64 = [tCase1and0-1.t9I73I64];
c9I22I33 = [tCase1and0-1.t9I22I33];
c9I31I62 = [tCase1and0-1.t9I31I62];
c9I81I42 = [tCase1and0-1.t9I81I42];


with cmCase1andandnot

/* gameA1,  gameA2,  gameA3,  gameA4,  gameA5,  gameA6,  gameA7,  */
/* gameA8,  gameA9,  gameA10, gameA11, gameA12, gameA13, gameA14, */
/* gameA15, gameA16, gameA17, gameA18, gameA19,                   */
/* gameB1,  gameB2,  gameB3,  gameB4,  gameB5,  gameB6,  gameB7,  */
/* gameB8,  gameB9,  gameB10, gameB11, gameB12, gameB13, gameB14, */
/* gameB15, gameB16, gameB17, gameB18, gameB19,                   */
/* gameC1,  gameC2,  gameC3,  gameC4,  gameC5,  gameC6,  gameC7,  */
/* gameC8,  gameC9,  gameC10, gameC11, gameC12, gameC13, gameC14, */
/* gameC15, gameC16, gameC17, gameC18, gameC19,                   */
/* gameD1,  gameD2,  gameD3,  gameD4,  gameD5,  gameD6,  gameD7,  */
/* gameD8,  gameD9,  gameD10, gameD11, gameD12, gameD13, gameD14, */
/* gameD15, gameD16, gameD17, gameD18, gameD19                    */

c1I62I73I21 = [tCase1andandnot0-1.t1I62I73I21];
c1I62I83I21 = [tCase1andandnot0-1.t1I62I83I21];
c1I62I23I21 = [tCase1andandnot0-1.t1I62I23I21];
c1I42I73I71 = [tCase1andandnot0-1.t1I42I73I71];
c1I42I33I71 = [tCase1andandnot0-1.t1I42I33I71];
c1I42I23I71 = [tCase1andandnot0-1.t1I42I23I71];

c2I11I32I22 = [tCase1andandnot0-1.t2I11I32I22];
c2I11I42I22 = [tCase1andandnot0-1.t2I11I42I22];
c2I11I62I22 = [tCase1andandnot0-1.t2I11I62I22];
c2I11I72I22 = [tCase1andandnot0-1.t2I11I72I22];
c2I11I92I22 = [tCase1andandnot0-1.t2I11I92I22];
c2I41I12I22 = [tCase1andandnot0-1.t2I41I12I22];
c2I41I32I22 = [tCase1andandnot0-1.t2I41I32I22];
c2I41I62I22 = [tCase1andandnot0-1.t2I41I62I22];
c2I41I72I22 = [tCase1andandnot0-1.t2I41I72I22];
c2I41I92I22 = [tCase1andandnot0-1.t2I41I92I22];
```

```
c3I22I63I11 = [tCase1andandnot0-1.t3I22I63I11];
c3I22I93I11 = [tCase1andandnot0-1.t3I22I93I11];
c3I22I13I11 = [tCase1andandnot0-1.t3I22I13I11];
c3I82I63I61 = [tCase1andandnot0-1.t3I82I63I61];
c3I82I43I61 = [tCase1andandnot0-1.t3I82I43I61];
c3I82I13I61 = [tCase1andandnot0-1.t3I82I13I61];

c4I81I22I42 = [tCase1andandnot0-1.t4I81I22I42];
c4I81I92I42 = [tCase1andandnot0-1.t4I81I92I42];
c4I81I72I42 = [tCase1andandnot0-1.t4I81I72I42];
c4I81I32I42 = [tCase1andandnot0-1.t4I81I32I42];
c4I81I12I42 = [tCase1andandnot0-1.t4I81I12I42];
c4I71I92I42 = [tCase1andandnot0-1.t4I71I92I42];
c4I71I82I42 = [tCase1andandnot0-1.t4I71I82I42];
c4I71I32I42 = [tCase1andandnot0-1.t4I71I32I42];
c4I71I22I42 = [tCase1andandnot0-1.t4I71I22I42];
c4I71I12I42 = [tCase1andandnot0-1.t4I71I12I42];

c6I21I12I62 = [tCase1andandnot0-1.t6I21I12I62];
c6I21I32I62 = [tCase1andandnot0-1.t6I21I32I62];
c6I21I72I62 = [tCase1andandnot0-1.t6I21I72I62];
c6I21I82I62 = [tCase1andandnot0-1.t6I21I82I62];
c6I21I92I62 = [tCase1andandnot0-1.t6I21I92I62];
c6I31I12I62 = [tCase1andandnot0-1.t6I31I12I62];
c6I31I22I62 = [tCase1andandnot0-1.t6I31I22I62];
c6I31I72I62 = [tCase1andandnot0-1.t6I31I72I62];
c6I31I82I62 = [tCase1andandnot0-1.t6I31I82I62];
c6I31I92I62 = [tCase1andandnot0-1.t6I31I92I62];

c7I22I63I41 = [tCase1andandnot0-1.t7I22I63I41];
c7I22I93I41 = [tCase1andandnot0-1.t7I22I93I41];
c7I22I43I41 = [tCase1andandnot0-1.t7I22I43I41];
c7I82I93I91 = [tCase1andandnot0-1.t7I82I93I91];
c7I82I43I91 = [tCase1andandnot0-1.t7I82I43I91];
c7I82I13I91 = [tCase1andandnot0-1.t7I82I13I91];

c8I91I72I82 = [tCase1andandnot0-1.t8I91I72I82];
c8I91I62I82 = [tCase1andandnot0-1.t8I91I62I82];
c8I91I42I82 = [tCase1andandnot0-1.t8I91I42I82];
c8I91I32I82 = [tCase1andandnot0-1.t8I91I32I82];
c8I91I12I82 = [tCase1andandnot0-1.t8I91I12I82];
```

```
c8I61I92I82 = [tCase1andandnot0-1.t8I61I92I82];
c8I61I72I82 = [tCase1andandnot0-1.t8I61I72I82];
c8I61I42I82 = [tCase1andandnot0-1.t8I61I42I82];
c8I61I32I82 = [tCase1andandnot0-1.t8I61I32I82];
c8I61I12I82 = [tCase1andandnot0-1.t8I61I12I82];

c9I62I73I31 = [tCase1andandnot0-1.t9I62I73I31];
c9I62I83I31 = [tCase1andandnot0-1.t9I62I83I31];
c9I62I33I31 = [tCase1andandnot0-1.t9I62I33I31];
c9I42I83I81 = [tCase1andandnot0-1.t9I42I83I81];
c9I42I33I81 = [tCase1andandnot0-1.t9I42I33I81];
c9I42I23I81 = [tCase1andandnot0-1.t9I42I23I81];


with emCase1recog

e1I11 = [tCase1recog1-2.t1I11, tCase1recog2-3.t1I11,
         tCase1recog3-4.t1I11];
e1I12 = [tCase1recog1-2.t1I12, tCase1recog2-3.t1I12,
         tCase1recog3-4.t1I12];
e1I13 = [tCase1recog1-2.t1I13, tCase1recog2-3.t1I13,
         tCase1recog3-4.t1I13];
e1I14 = [tCase1recog1-2.t1I14, tCase1recog2-3.t1I14,
         tCase1recog3-4.t1I14];

e2I21 = [tCase1recog1-2.t2I21, tCase1recog2-3.t2I21,
         tCase1recog3-4.t2I21];
e2I22 = [tCase1recog1-2.t2I22, tCase1recog2-3.t2I22,
         tCase1recog3-4.t2I22];
e2I23 = [tCase1recog1-2.t2I23, tCase1recog2-3.t2I23,
         tCase1recog3-4.t2I23];
e2I24 = [tCase1recog1-2.t2I24, tCase1recog2-3.t2I24,
         tCase1recog3-4.t2I24];
e2I61 = [tCase1recog1-2.t2I61, tCase1recog2-3.t2I61,
         tCase1recog3-4.t2I61];
e2I91 = [tCase1recog1-2.t2I91, tCase1recog2-3.t2I91,
         tCase1recog3-4.t2I91];

e3I31 = [tCase1recog1-2.t3I31, tCase1recog2-3.t3I31,
         tCase1recog3-4.t3I31];
e3I32 = [tCase1recog1-2.t3I32, tCase1recog2-3.t3I32,
```

```
         tCase1recog3-4.t3I32];
e3I33 = [tCase1recog1-2.t3I33, tCase1recog2-3.t3I33,
         tCase1recog3-4.t3I33];
e3I34 = [tCase1recog1-2.t3I34, tCase1recog2-3.t3I34,
         tCase1recog3-4.t3I34];

e4I41 = [tCase1recog1-2.t4I41, tCase1recog2-3.t4I41,
         tCase1recog3-4.t4I41];
e4I42 = [tCase1recog1-2.t4I42, tCase1recog2-3.t4I42,
         tCase1recog3-4.t4I42];
e4I43 = [tCase1recog1-2.t4I43, tCase1recog2-3.t4I43,
         tCase1recog3-4.t4I43];
e4I44 = [tCase1recog1-2.t4I44, tCase1recog2-3.t4I44,
         tCase1recog3-4.t4I44];
e4I21 = [tCase1recog1-2.t4I21, tCase1recog2-3.t4I21,
         tCase1recog3-4.t4I21];
e4I31 = [tCase1recog1-2.t4I31, tCase1recog2-3.t4I31,
         tCase1recog3-4.t4I31];

e6I61 = [tCase1recog1-2.t6I61, tCase1recog2-3.t6I61,
         tCase1recog3-4.t6I61];
e6I62 = [tCase1recog1-2.t6I62, tCase1recog2-3.t6I62,
         tCase1recog3-4.t6I62];
e6I63 = [tCase1recog1-2.t6I63, tCase1recog2-3.t6I63,
         tCase1recog3-4.t6I63];
e6I64 = [tCase1recog1-2.t6I64, tCase1recog2-3.t6I64,
         tCase1recog3-4.t6I64];
e6I71 = [tCase1recog1-2.t6I71, tCase1recog2-3.t6I71,
         tCase1recog3-4.t6I71];
e6I81 = [tCase1recog1-2.t6I81, tCase1recog2-3.t6I81,
         tCase1recog3-4.t6I81];

e7I71 = [tCase1recog1-2.t7I71, tCase1recog2-3.t7I71,
         tCase1recog3-4.t7I71];
e7I72 = [tCase1recog1-2.t7I72, tCase1recog2-3.t7I72,
         tCase1recog3-4.t7I72];
e7I73 = [tCase1recog1-2.t7I73, tCase1recog2-3.t7I73,
         tCase1recog3-4.t7I73];
e7I74 = [tCase1recog1-2.t7I74, tCase1recog2-3.t7I74,
         tCase1recog3-4.t7I74];
```

```
e8I81 = [tCase1recog1-2.t8I81, tCase1recog2-3.t8I81,
         tCase1recog3-4.t8I81];
e8I82 = [tCase1recog1-2.t8I82, tCase1recog2-3.t8I82,
         tCase1recog3-4.t8I82];
e8I83 = [tCase1recog1-2.t8I83, tCase1recog2-3.t8I83,
         tCase1recog3-4.t8I83];
e8I84 = [tCase1recog1-2.t8I84, tCase1recog2-3.t8I84,
         tCase1recog3-4.t8I84];
e8I11 = [tCase1recog1-2.t8I11, tCase1recog2-3.t8I11,
         tCase1recog3-4.t8I11];
e8I41 = [tCase1recog1-2.t8I41, tCase1recog2-3.t8I41,
         tCase1recog3-4.t8I41];

e9I91 = [tCase1recog1-2.t9I91, tCase1recog2-3.t9I91,
         tCase1recog3-4.t9I91];
e9I92 = [tCase1recog1-2.t9I92, tCase1recog2-3.t9I92,
         tCase1recog3-4.t9I92];
e9I93 = [tCase1recog1-2.t9I93, tCase1recog2-3.t9I93,
         tCase1recog3-4.t9I93];
e9I94 = [tCase1recog1-2.t9I94, tCase1recog2-3.t9I94,
         tCase1recog3-4.t9I94];



with emCase1and

e1I21I62 = [tCase1and1-2.t1I21I62, tCase1and2-3.t1I21I62,
            tCase1and3-4.t1I21I62, tCase1and4-5.t1I21I62];
e1I33I44 = [tCase1and1-2.t1I33I44, tCase1and2-3.t1I33I44,
            tCase1and3-4.t1I33I44, tCase1and4-5.t1I33I44];
e1I82I73 = [tCase1and1-2.t1I82I73, tCase1and2-3.t1I82I73,
            tCase1and3-4.t1I82I73, tCase1and4-5.t1I82I73];
e1I71I42 = [tCase1and1-2.t1I71I42, tCase1and2-3.t1I71I42,
            tCase1and3-4.t1I71I42, tCase1and4-5.t1I71I42];

e2I62I13 = [tCase1and1-2.t2I62I13, tCase1and2-3.t2I62I13,
            tCase1and3-4.t2I62I13, tCase1and4-5.t2I62I13];
e2I93I34 = [tCase1and1-2.t2I93I34, tCase1and2-3.t2I93I34,
            tCase1and3-4.t2I93I34, tCase1and4-5.t2I93I34];

e3I11I22 = [tCase1and1-2.t3I11I22, tCase1and2-3.t3I11I22,
```

```
            tCase1and3-4.t3I11I22, tCase1and4-5.t3I11I22];
e3I61I82 = [tCase1and1-2.t3I61I82, tCase1and2-3.t3I61I82,
            tCase1and3-4.t3I61I82, tCase1and4-5.t3I61I82];
e3I42I13 = [tCase1and1-2.t3I42I13, tCase1and2-3.t3I42I13,
            tCase1and3-4.t3I42I13, tCase1and4-5.t3I42I13];
e3I93I24 = [tCase1and1-2.t3I93I24, tCase1and2-3.t3I93I24,
            tCase1and3-4.t3I93I24, tCase1and4-5.t3I93I24];

e4I22I73 = [tCase1and1-2.t4I22I73, tCase1and2-3.t4I22I73,
            tCase1and3-4.t4I22I73, tCase1and4-5.t4I22I73];
e4I33I14 = [tCase1and1-2.t4I33I14, tCase1and2-3.t4I33I14,
            tCase1and3-4.t4I33I14, tCase1and4-5.t4I33I14];

e6I73I94 = [tCase1and1-2.t6I73I94, tCase1and2-3.t6I73I94,
            tCase1and3-4.t6I73I94, tCase1and4-5.t6I73I94];
e6I82I33 = [tCase1and1-2.t6I82I33, tCase1and2-3.t6I82I33,
            tCase1and3-4.t6I82I33, tCase1and4-5.t6I82I33];

e7I41I22 = [tCase1and1-2.t7I41I22, tCase1and2-3.t7I41I22,
            tCase1and3-4.t7I41I22, tCase1and4-5.t7I41I22];
e7I62I93 = [tCase1and1-2.t7I62I93, tCase1and2-3.t7I62I93,
            tCase1and3-4.t7I62I93, tCase1and4-5.t7I62I93];
e7I13I84 = [tCase1and1-2.t7I13I84, tCase1and2-3.t7I13I84,
            tCase1and3-4.t7I13I84, tCase1and4-5.t7I13I84];
e7I91I82 = [tCase1and1-2.t7I91I82, tCase1and2-3.t7I91I82,
            tCase1and3-4.t7I91I82, tCase1and4-5.t7I91I82];

e8I13I74 = [tCase1and1-2.t8I13I74, tCase1and2-3.t8I13I74,
            tCase1and3-4.t8I13I74, tCase1and4-5.t8I13I74];
e8I42I93 = [tCase1and1-2.t8I42I93, tCase1and2-3.t8I42I93,
            tCase1and3-4.t8I42I93, tCase1and4-5.t8I42I93];

e9I73I64 = [tCase1and1-2.t9I73I64, tCase1and2-3.t9I73I64,
            tCase1and3-4.t9I73I64, tCase1and4-5.t9I73I64];
e9I22I33 = [tCase1and1-2.t9I22I33, tCase1and2-3.t9I22I33,
            tCase1and3-4.t9I22I33, tCase1and4-5.t9I22I33];
e9I31I62 = [tCase1and1-2.t9I31I62, tCase1and2-3.t9I31I62,
            tCase1and3-4.t9I31I62, tCase1and4-5.t9I31I62];
e9I81I42 = [tCase1and1-2.t9I81I42, tCase1and2-3.t9I81I42,
            tCase1and3-4.t9I81I42, tCase1and4-5.t9I81I42];
```

```
with emCase1andandnot

e1I62I73I21 = [tCase1andandnot1-2.t1I62I73I21,
               tCase1andandnot2-3.t1I62I73I21,
               tCase1andandnot3-4.t1I62I73I21,
               tCase1andandnot4-5.t1I62I73I21];
e1I62I83I21 = [tCase1andandnot1-2.t1I62I83I21,
               tCase1andandnot2-3.t1I62I83I21,
               tCase1andandnot3-4.t1I62I83I21,
               tCase1andandnot4-5.t1I62I83I21];
e1I62I23I21 = [tCase1andandnot1-2.t1I62I23I21,
               tCase1andandnot2-3.t1I62I23I21,
               tCase1andandnot3-4.t1I62I23I21,
               tCase1andandnot4-5.t1I62I23I21];
e1I42I73I71 = [tCase1andandnot1-2.t1I42I73I71,
               tCase1andandnot2-3.t1I42I73I71,
               tCase1andandnot3-4.t1I42I73I71,
               tCase1andandnot4-5.t1I42I73I71];
e1I42I33I71 = [tCase1andandnot1-2.t1I42I33I71,
               tCase1andandnot2-3.t1I42I33I71,
               tCase1andandnot3-4.t1I42I33I71,
               tCase1andandnot4-5.t1I42I33I71];
e1I42I23I71 = [tCase1andandnot1-2.t1I42I23I71,
               tCase1andandnot2-3.t1I42I23I71,
               tCase1andandnot3-4.t1I42I23I71,
               tCase1andandnot4-5.t1I42I23I71];

e2I11I32I22 = [tCase1andandnot1-2.t2I11I32I22,
               tCase1andandnot2-3.t2I11I32I22,
               tCase1andandnot3-4.t2I11I32I22,
               tCase1andandnot4-5.t2I11I32I22];
e2I11I42I22 = [tCase1andandnot1-2.t2I11I42I22,
               tCase1andandnot2-3.t2I11I42I22,
               tCase1andandnot3-4.t2I11I42I22,
               tCase1andandnot4-5.t2I11I42I22];
e2I11I62I22 = [tCase1andandnot1-2.t2I11I62I22,
               tCase1andandnot2-3.t2I11I62I22,
               tCase1andandnot3-4.t2I11I62I22,
               tCase1andandnot4-5.t2I11I62I22];
```

```
e2I11I72I22 = [tCase1andandnot1-2.t2I11I72I22,
               tCase1andandnot2-3.t2I11I72I22,
               tCase1andandnot3-4.t2I11I72I22,
               tCase1andandnot4-5.t2I11I72I22];
e2I11I92I22 = [tCase1andandnot1-2.t2I11I92I22,
               tCase1andandnot2-3.t2I11I92I22,
               tCase1andandnot3-4.t2I11I92I22,
               tCase1andandnot4-5.t2I11I92I22];
e2I41I12I22 = [tCase1andandnot1-2.t2I41I12I22,
               tCase1andandnot2-3.t2I41I12I22,
               tCase1andandnot3-4.t2I41I12I22,
               tCase1andandnot4-5.t2I41I12I22];
e2I41I32I22 = [tCase1andandnot1-2.t2I41I32I22,
               tCase1andandnot2-3.t2I41I32I22,
               tCase1andandnot3-4.t2I41I32I22,
               tCase1andandnot4-5.t2I41I32I22];
e2I41I62I22 = [tCase1andandnot1-2.t2I41I62I22,
               tCase1andandnot2-3.t2I41I62I22,
               tCase1andandnot3-4.t2I41I62I22,
               tCase1andandnot4-5.t2I41I62I22];
e2I41I72I22 = [tCase1andandnot1-2.t2I41I72I22,
               tCase1andandnot2-3.t2I41I72I22,
               tCase1andandnot3-4.t2I41I72I22,
               tCase1andandnot4-5.t2I41I72I22];
e2I41I92I22 = [tCase1andandnot1-2.t2I41I92I22,
               tCase1andandnot2-3.t2I41I92I22,
               tCase1andandnot3-4.t2I41I92I22,
               tCase1andandnot4-5.t2I41I92I22];

e3I22I63I11 = [tCase1andandnot1-2.t3I22I63I11,
               tCase1andandnot2-3.t3I22I63I11,
               tCase1andandnot3-4.t3I22I63I11,
               tCase1andandnot4-5.t3I22I63I11];
e3I22I93I11 = [tCase1andandnot1-2.t3I22I93I11,
               tCase1andandnot2-3.t3I22I93I11,
               tCase1andandnot3-4.t3I22I93I11,
               tCase1andandnot4-5.t3I22I93I11];
e3I22I13I11 = [tCase1andandnot1-2.t3I22I13I11,
               tCase1andandnot2-3.t3I22I13I11,
               tCase1andandnot3-4.t3I22I13I11,
               tCase1andandnot4-5.t3I22I13I11];
```

```
e3I82I63I61 = [tCase1andandnot1-2.t3I82I63I61,
               tCase1andandnot2-3.t3I82I63I61,
               tCase1andandnot3-4.t3I82I63I61,
               tCase1andandnot4-5.t3I82I63I61];
e3I82I43I61 = [tCase1andandnot1-2.t3I82I43I61,
               tCase1andandnot2-3.t3I82I43I61,
               tCase1andandnot3-4.t3I82I43I61,
               tCase1andandnot4-5.t3I82I43I61];
e3I82I13I61 = [tCase1andandnot1-2.t3I82I13I61,
               tCase1andandnot2-3.t3I82I13I61,
               tCase1andandnot3-4.t3I82I13I61,
               tCase1andandnot4-5.t3I82I13I61];

e4I81I22I42 = [tCase1andandnot1-2.t4I81I22I42,
               tCase1andandnot2-3.t4I81I22I42,
               tCase1andandnot3-4.t4I81I22I42,
               tCase1andandnot4-5.t4I81I22I42];
e4I81I92I42 = [tCase1andandnot1-2.t4I81I92I42,
               tCase1andandnot2-3.t4I81I92I42,
               tCase1andandnot3-4.t4I81I92I42,
               tCase1andandnot4-5.t4I81I92I42];
e4I81I72I42 = [tCase1andandnot1-2.t4I81I72I42,
               tCase1andandnot2-3.t4I81I72I42,
               tCase1andandnot3-4.t4I81I72I42,
               tCase1andandnot4-5.t4I81I72I42];
e4I81I32I42 = [tCase1andandnot1-2.t4I81I32I42,
               tCase1andandnot2-3.t4I81I32I42,
               tCase1andandnot3-4.t4I81I32I42,
               tCase1andandnot4-5.t4I81I32I42];
e4I81I12I42 = [tCase1andandnot1-2.t4I81I12I42,
               tCase1andandnot2-3.t4I81I12I42,
               tCase1andandnot3-4.t4I81I12I42,
               tCase1andandnot4-5.t4I81I12I42];
e4I71I92I42 = [tCase1andandnot1-2.t4I71I92I42,
               tCase1andandnot2-3.t4I71I92I42,
               tCase1andandnot3-4.t4I71I19I42,
               tCase1andandnot4-5.t4I71I92I42];
e4I71I82I42 = [tCase1andandnot1-2.t4I71I82I42,
               tCase1andandnot2-3.t4I71I82I42,
               tCase1andandnot3-4.t4I71I82I42,
               tCase1andandnot4-5.t4I71I82I42];
```

```
e4I71I32I42 = [tCase1andandnot1-2.t4I71I32I42,
               tCase1andandnot2-3.t4I71I32I42,
               tCase1andandnot3-4.t4I71I32I42,
               tCase1andandnot4-5.t4I71I32I42];
e4I71I22I42 = [tCase1andandnot1-2.t4I71I22I42,
               tCase1andandnot2-3.t4I71I22I42,
               tCase1andandnot3-4.t4I71I22I42,
               tCase1andandnot4-5.t4I71I22I42];
e4I71I12I42 = [tCase1andandnot1-2.t4I71I12I42,
               tCase1andandnot2-3.t4I71I12I42,
               tCase1andandnot3-4.t4I71I12I42,
               tCase1andandnot4-5.t4I71I12I42];

e6I21I12I62 = [tCase1andandnot1-2.t6I21I12I62,
               tCase1andandnot2-3.t6I21I12I62,
               tCase1andandnot3-4.t6I21I12I62,
               tCase1andandnot4-5.t6I21I12I62];
e6I21I32I62 = [tCase1andandnot1-2.t6I21I32I62,
               tCase1andandnot2-3.t6I21I32I62,
               tCase1andandnot3-4.t6I21I32I62,
               tCase1andandnot4-5.t6I21I32I62];
e6I21I72I62 = [tCase1andandnot1-2.t6I21I72I62,
               tCase1andandnot2-3.t6I21I72I62,
               tCase1andandnot3-4.t6I21I72I62,
               tCase1andandnot4-5.t6I21I72I62];
e6I21I82I62 = [tCase1andandnot1-2.t6I21I82I62,
               tCase1andandnot2-3.t6I21I82I62,
               tCase1andandnot3-4.t6I21I82I62,
               tCase1andandnot4-5.t6I21I82I62];
e6I21I92I62 = [tCase1andandnot1-2.t6I21I92I62,
               tCase1andandnot2-3.t6I21I92I62,
               tCase1andandnot3-4.t6I21I92I62,
               tCase1andandnot4-5.t6I21I92I62];
e6I31I12I62 = [tCase1andandnot1-2.t6I31I12I62,
               tCase1andandnot2-3.t6I31I12I62,
               tCase1andandnot3-4.t6I31I12I62,
               tCase1andandnot4-5.t6I31I12I62];
e6I31I22I62 = [tCase1andandnot1-2.t6I31I22I62,
               tCase1andandnot2-3.t6I31I22I62,
               tCase1andandnot3-4.t6I31I22I62,
               tCase1andandnot4-5.t6I31I22I62];
```

```
e6I31I72I62 = [tCase1andandnot1-2.t6I31I72I62,
               tCase1andandnot2-3.t6I31I72I62,
               tCase1andandnot3-4.t6I31I72I62,
               tCase1andandnot4-5.t6I31I72I62];
e6I31I82I62 = [tCase1andandnot1-2.t6I31I82I62,
               tCase1andandnot2-3.t6I31I82I62,
               tCase1andandnot3-4.t6I31I82I62,
               tCase1andandnot4-5.t6I31I82I62];
e6I31I92I62 = [tCase1andandnot1-2.t6I31I92I62,
               tCase1andandnot2-3.t6I31I92I62,
               tCase1andandnot3-4.t6I31I92I62,
               tCase1andandnot4-5.t6I31I92I62];

e7I22I63I41 = [tCase1andandnot1-2.t7I22I63I41,
               tCase1andandnot2-3.t7I22I63I41,
               tCase1andandnot3-4.t7I22I63I41,
               tCase1andandnot4-5.t7I22I63I41];
e7I22I93I41 = [tCase1andandnot1-2.t7I22I93I41,
               tCase1andandnot2-3.t7I22I93I41,
               tCase1andandnot3-4.t7I22I93I41,
               tCase1andandnot4-5.t7I22I93I41];
e7I22I43I41 = [tCase1andandnot1-2.t7I22I43I41,
               tCase1andandnot2-3.t7I22I43I41,
               tCase1andandnot3-4.t7I22I43I41,
               tCase1andandnot4-5.t7I22I43I41];
e7I82I93I91 = [tCase1andandnot1-2.t7I82I93I91,
               tCase1andandnot2-3.t7I82I93I91,
               tCase1andandnot3-4.t7I82I93I91,
               tCase1andandnot4-5.t7I82I93I91];
e7I82I43I91 = [tCase1andandnot1-2.t7I82I43I91,
               tCase1andandnot2-3.t7I82I43I91,
               tCase1andandnot3-4.t7I82I43I91,
               tCase1andandnot4-5.t7I82I43I91];
e7I82I13I91 = [tCase1andandnot1-2.t7I82I13I91,
               tCase1andandnot2-3.t7I82I13I91,
               tCase1andandnot3-4.t7I82I13I91,
               tCase1andandnot4-5.t7I82I13I91];

e8I91I72I82 = [tCase1andandnot1-2.t8I91I72I82,
               tCase1andandnot2-3.t8I91I72I82,
               tCase1andandnot3-4.t8I91I72I82,
```

```
                tCase1andandnot4-5.t8I91I72I82];
e8I91I62I82 = [tCase1andandnot1-2.t8I91I62I82,
                tCase1andandnot2-3.t8I91I62I82,
                tCase1andandnot3-4.t8I91I62I82,
                tCase1andandnot4-5.t8I91I62I82];
e8I91I42I82 = [tCase1andandnot1-2.t8I91I42I82,
                tCase1andandnot2-3.t8I91I42I82,
                tCase1andandnot3-4.t8I91I42I82,
                tCase1andandnot4-5.t8I91I42I82];
e8I91I32I82 = [tCase1andandnot1-2.t8I91I32I82,
                tCase1andandnot2-3.t8I91I32I82,
                tCase1andandnot3-4.t8I91I32I82,
                tCase1andandnot4-5.t8I91I32I82];
e8I91I12I82 = [tCase1andandnot1-2.t8I91I12I82,
                tCase1andandnot2-3.t8I91I12I82,
                tCase1andandnot3-4.t8I91I12I82,
                tCase1andandnot4-5.t8I91I12I82];
e8I61I92I82 = [tCase1andandnot1-2.t8I61I92I82,
                tCase1andandnot2-3.t8I61I92I82,
                tCase1andandnot3-4.t8I61I92I82,
                tCase1andandnot4-5.t8I61I92I82];
e8I61I72I82 = [tCase1andandnot1-2.t8I61I72I82,
                tCase1andandnot2-3.t8I61I72I82,
                tCase1andandnot3-4.t8I61I72I82,
                tCase1andandnot4-5.t8I61I72I82];
e8I61I42I82 = [tCase1andandnot1-2.t8I61I42I82,
                tCase1andandnot2-3.t8I61I42I82,
                tCase1andandnot3-4.t8I61I42I82,
                tCase1andandnot4-5.t8I61I42I82];
e8I61I32I82 = [tCase1andandnot1-2.t8I61I32I82,
                tCase1andandnot2-3.t8I61I32I82,
                tCase1andandnot3-4.t8I61I32I82,
                tCase1andandnot4-5.t8I61I32I82];
e8I61I12I82 = [tCase1andandnot1-2.t8I61I12I82,
                tCase1andandnot2-3.t8I61I12I82,
                tCase1andandnot3-4.t8I61I12I82,
                tCase1andandnot4-5.t8I61I12I82];

e9I62I73I31 = [tCase1andandnot1-2.t9I62I73I31,
                tCase1andandnot2-3.t9I62I73I31,
                tCase1andandnot3-4.t9I62I73I31,
```

```
                tCase1andandnot4-5.t9I62I73I31];
e9I62I83I31 = [tCase1andandnot1-2.t9I62I83I31,
                tCase1andandnot2-3.t9I62I83I31,
                tCase1andandnot3-4.t9I62I83I31,
                tCase1andandnot4-5.t9I62I83I31];
e9I62I33I31 = [tCase1andandnot1-2.t9I62I33I31,
                tCase1andandnot2-3.t9I62I33I31,
                tCase1andandnot3-4.t9I62I33I31,
                tCase1andandnot4-5.t9I62I33I31];
e9I42I83I81 = [tCase1andandnot1-2.t9I42I83I81,
                tCase1andandnot2-3.t9I42I83I81,
                tCase1andandnot3-4.t9I42I83I81,
                tCase1andandnot4-5.t9I42I83I81];
e9I42I33I81 = [tCase1andandnot1-2.t9I42I33I81,
                tCase1andandnot2-3.t9I42I33I81,
                tCase1andandnot3-4.t9I42I33I81,
                tCase1andandnot4-5.t9I42I33I81];
e9I42I23I81 = [tCase1andandnot1-2.t9I42I23I81,
                tCase1andandnot2-3.t9I42I23I81,
                tCase1andandnot3-4.t9I42I23I81,
                tCase1andandnot4-5.t9I42I23I81];



EVENTSTREAM

/* Platform creation */
ttt = [<e0>,<cmCase1recog,cmCase1and,cmCase1andandnot>];

/* Program for game A1 */
programA1 =
[ <en1I61,en2I61,en3I61,en4I61,en6I61,en7I61,en8I61,en9I61>,
  <emCase1recog.e6I61,emCase1recog.e2I61>, <visual6green,visual2red>,
  <en1I12,en2I12,en3I12,en4I12,en6I12,en7I12,en8I12,en9I12>,
  <emCase1recog.e1I12,emCase1andandnot.e8I61I12I82>,
  <visual1green,visual8red>
];

/* Program for game A2 */
programA2 =
[ <en1I61,en2I61,en3I61,en4I61,en6I61,en7I61,en8I61,en9I61>,
```

```
  <emCase1recog.e6I61,emCase1recog.e2I61>, <visual6green,visual2red>,
  <en1I32,en2I32,en3I32,en4I32,en6I32,en7I32,en8I32,en9I32>,
  <emCase1recog.e3I32,emCase1andandnot.e8I61I32I82>,
  <visual3green,visual8red>
];

/* Program for game A3 */
programA3 =
[ <en1I61,en2I61,en3I61,en4I61,en6I61,en7I61,en8I61,en9I61>,
  <emCase1recog.e6I61,emCase1recog.e2I61>, <visual6green,visual2red>,
  <en1I42,en2I42,en3I42,en4I42,en6I42,en7I42,en8I42,en9I42>,
  <emCase1recog.e4I42,emCase1andandnot.e8I61I42I82>,
  <visual4green,visual8red>
];

/* Program for game A4 */
programA4 =
[ <en1I61,en2I61,en3I61,en4I61,en6I61,en7I61,en8I61,en9I61>,
  <emCase1recog.e6I61,emCase1recog.e2I61>, <visual6green,visual2red>,
  <en1I72,en2I72,en3I72,en4I72,en6I72,en7I72,en8I72,en9I72>,
  <emCase1recog.e7I72,emCase1andandnot.e8I61I72I82>,
  <visual7green,visual8red>
];

/* Program for game A5 */
programA5 =
[ <en1I61,en2I61,en3I61,en4I61,en6I61,en7I61,en8I61,en9I61>,
  <emCase1recog.e6I61,emCase1recog.e2I61>, <visual6green,visual2red>,
  <en1I92,en2I92,en3I92,en4I92,en6I92,en7I92,en8I92,en9I92>,
  <emCase1recog.e9I92,emCase1andandnot.e8I61I92I82>,
  <visual9green,visual8red>
];

/* Program for game A6 */
programA6 =
[ <en1I91,en2I91,en3I91,en4I91,en6I91,en7I91,en8I91,en9I91>,
  <emCase1recog.e9I91,emCase1recog.e2I91>, <visual9green,visual2red>,
  <en1I12,en2I12,en3I12,en4I12,en6I12,en7I12,en8I12,en9I12>,
  <emCase1recog.e1I12,emCase1andandnot.e8I61I12I82>,
  <visual1green,visual8green>
];
```

```
/* Program for game A7 */
programA7 =
[ <en1I91,en2I91,en3I91,en4I91,en6I91,en7I91,en8I91,en9I91>,
  <emCase1recog.e9I91,emCase1recog.e2I91>, <visual9green,visual2red>,
  <en1I32,en2I32,en3I32,en4I32,en6I32,en7I32,en8I32,en9I32>,
  <emCase1recog.e3I32,emCase1andandnot.e8I91I32I82>,
  <visual3green,visual8red>
];

/* Program for game A8 */
programA8 =
[ <en1I91,en2I91,en3I91,en4I91,en6I91,en7I91,en8I91,en9I91>,
  <emCase1recog.e9I91,emCase1recog.e2I91>, <visual9green,visual2red>,
  <en1I42,en2I42,en3I42,en4I42,en6I42,en7I42,en8I42,en9I42>,
  <emCase1recog.e4I42,emCase1andandnot.e8I91I42I82>,
  <visual4green,visual8red>
];

/* Program for game A9 */
programA9 =
[ <en1I91,en2I91,en3I91,en4I91,en6I91,en7I91,en8I91,en9I91>,
  <emCase1recog.e9I91,emCase1recog.e2I91>, <visual9green,visual2red>,
  <en1I62,en2I62,en3I62,en4I62,en6I62,en7I62,en8I62,en9I62>,
  <emCase1recog.e6I62,emCase1andandnot.e8I91I62I82>,
  <visual6green,visual8red>
];

/* Program for game A10 */
programA10 =
[ <en1I91,en2I91,en3I91,en4I91,en6I91,en7I91,en8I91,en9I91>,
  <emCase1recog.e9I91,emCase1recog.e2I91>, <visual9green,visual2red>,
  <en1I72,en2I72,en3I72,en4I72,en6I72,en7I72,en8I72,en9I72>,
  <emCase1recog.e7I72,emCase1andandnot.e8I91I72I82>,
  <visual7green,visual8red>
];

/* Program for game A11 */
programA11 =
[ <en1I61,en2I61,en3I61,en4I61,en6I61,en7I61,en8I61,en9I61>,
  <emCase1recog.e6I61,emCase1recog.e2I61>, <visual6green,visual2red>,
```

```
  <en1I82,en2I82,en3I82,en4I82,en6I82,en7I82,en8I82,en9I82>,
  <emCase1recog.e8I82,emCase1recog.e3I61I82>, <visual8green,visual3green>
  <en1I13,en2I13,en3I13,en4I13,en6I13,en7I13,en8I13,en9I13>,
  <emCase1recog.e1I13,emCase1andandnot.e7I82I13I91>,
  <visual1green,visual7red>
];

/* Program for game A12 */
programA12 =
[ <en1I61,en2I61,en3I61,en4I61,en6I61,en7I61,en8I61,en9I61>,
  <emCase1recog.e6I61,emCase1recog.e2I61>, <visual6green,visual2red>,
  <en1I82,en2I82,en3I82,en4I82,en6I18,en7I82,en8I82,en9I82>,
  <emCase1recog.e8I82,emCase1recog.e3I61I82>, <visual8green,visual3green>
  <en1I43,en2I43,en3I43,en4I43,en6I43,en7I43,en8I43,en9I43>,
  <emCase1recog.e4I43,emCase1andandnot.e7I82I43I91>,
  <visual4green,visual7red>
];

/* Program for game A13 */
programA13 =
[ <en1I61,en2I61,en3I61,en4I61,en6I61,en7I61,en8I61,en9I61>,
  <emCase1recog.e6I61,emCase1recog.e2I61>, <visual6green,visual2red>,
  <en1I82,en2I82,en3I82,en4I82,en6I18,en7I82,en8I82,en9I82>,
  <emCase1recog.e8I82,emCase1recog.e3I61I82>, <visual8green,visual3green>
  <en1I73,en2I73,en3I73,en4I73,en6I73,en7I73,en8I73,en9I73>,
  <emCase1recog.e7I73,emCase1.e1I82I73>, <visual7green,visual1red>
];

/* Program for game A14 */
programA14 =
[ <en1I61,en2I61,en3I61,en4I61,en6I61,en7I61,en8I61,en9I61>,
  <emCase1recog.e6I61,emCase1recog.e2I61>, <visual6green,visual2red>,
  <en1I82,en2I82,en3I82,en4I82,en6I18,en7I82,en8I82,en9I82>,
  <emCase1recog.e8I82,emCase1recog.e3I61I82>, <visual8green,visual3green>
  <en1I93,en2I93,en3I93,en4I93,en6I93,en7I93,en8I93,en9I93>,
  <emCase1recog.e9I93,emCase1andandnot.e7I82I93I91>,
  <visual9green,visual7green>
];

/* Program for game A15 */
programA15 =
```

```
[ <en1I91,en2I91,en3I91,en4I91,en6I91,en7I91,en8I91,en9I91>,
  <emCase1recog.e9I91,emCase1recog.e2I91>, <visual9green,visual2red>,
  <en1I82,en2I82,en3I82,en4I82,en6I18,en7I82,en8I82,en9I82>,
  <emCase1recog.e8I82,emCase1and.e7I91I82>, <visual8green,visual7red>,
  <en1I13,en2I13,en3I13,en4I13,en6I13,en7I13,en8I13,en9I13>,
  <emCase1recog.e1I13,emCase1andandnot.e3I82I13I61>,
  <visual1green,visual3red>
];

/* Program for game A16 */
programA16 =
[ <en1I91,en2I91,en3I91,en4I91,en6I91,en7I91,en8I91,en9I91>,
  <emCase1recog.e9I91,emCase1recog.e2I91>, <visual9green,visual2red>,
  <en1I82,en2I82,en3I82,en4I82,en6I18,en7I82,en8I82,en9I82>,
  <emCase1recog.e8I82,emCase1and.e7I91I82>, <visual8green,visual7red>,
  <en1I43,en2I43,en3I43,en4I43,en6I43,en7I43,en8I43,en9I43>,
  <emCase1recog.e4I43,emCase1andandnot.e3I82I43I61>,
  <visual4green,visual3red>
];

/* Program for game A17 */
programA17 =
[ <en1I91,en2I91,en3I91,en4I91,en6I91,en7I91,en8I91,en9I91>,
  <emCase1recog.e9I91,emCase1recog.e2I91>, <visual9green,visual2red>,
  <en1I82,en2I82,en3I82,en4I82,en6I18,en7I82,en8I82,en9I82>,
  <emCase1recog.e8I82,emCase1and.e7I91I82>, <visual8green,visual7red>,
  <en1I63,en2I63,en3I63,en4I63,en6I63,en7I63,en8I63,en9I63>,
  <emCase1recog.e6I63,emCase1andandnot.e3I82I63I61>,
  <visual6green,visual3red>
];

/* Program for game A18 */
programA18 =
[ <en1I91,en2I91,en3I91,en4I91,en6I91,en7I91,en8I91,en9I91>,
  <emCase1recog.e9I91,emCase1recog.e2I91>, <visual9green,visual2red>,
  <en1I82,en2I82,en3I82,en4I82,en6I18,en7I82,en8I82,en9I82>,
  <emCase1recog.e8I82,emCase1and.e7I91I82>, <visual8green,visual7red>,
  <en1I33,en2I33,en3I33,en4I33,en6I33,en7I33,en8I33,en9I33>,
  <emCase1recog.e3I33,emCase1and.e6I82I33>, <visual3green,visual6red>,
  <en1I14,en2I14,en3I14,en4I14,en6I14,en7I14,en8I14,en9I14>,
  <emCase1recog.e1I14,emCase1and.e4I33I14>, <visual1green,visual4red>
```

```
];

/* Program for game A19 */
[ <en1I91,en2I91,en3I91,en4I91,en6I91,en7I91,en8I91,en9I91>,
  <emCase1recog.e9I91,emCase1recog.e2I91>, <visual9green,visual2red>,
  <en1I82,en2I82,en3I82,en4I82,en6I18,en7I82,en8I82,en9I82>,
  <emCase1recog.e8I82,emCase1and.e7I91I82>, <visual8green,visual7red>,
  <en1I33,en2I33,en3I33,en4I33,en6I33,en7I33,en8I33,en9I33>,
  <emCase1recog.e3I33,emCase1and.e6I82I33>, <visual3green,visual6red>,
  <en1I44,en2I44,en3I44,en4I44,en6I44,en7I44,en8I44,en9I44>,
  <emCase1recog.e4I44,emCase1and.e1I33I44>, <visual4green,visual1red>
];

/* Program for game B1 */
programB1 =
[ <en1I81,en2I81,en3I81,en4I81,en6I81,en7I81,en8I81,en9I81>,
  <emCase1recog.e8I81,emCase1recog.e6I81>, <visual8green,visual6red>,
  <en1I12,en2I12,en3I12,en4I12,en6I12,en7I12,en8I12,en9I12>,
  <emCase1recog.e1I12,emCase1andandnot.e4I81I12I42>,
  <visual1green,visual4red>
];

/* Program for game B2 */
programB2 =
[ <en1I81,en2I81,en3I81,en4I81,en6I81,en7I81,en8I81,en9I81>,
  <emCase1recog.e8I81,emCase1recog.e6I81>, <visual8green,visual6red>,
  <en1I22,en2I22,en3I22,en4I22,en6I22,en7I22,en8I22,en9I22>,
  <emCase1recog.e2I22,emCase1andandnot.e4I81I22I42>,
  <visual2green,visual4red>
];

/* Program for game B3 */
programB3 =
[ <en1I81,en2I81,en3I81,en4I81,en6I81,en7I81,en8I81,en9I81>,
  <emCase1recog.e8I81,emCase1recog.e6I81>, <visual8green,visual6red>,
  <en1I32,en2I32,en3I32,en4I32,en6I32,en7I32,en8I32,en9I32>,
  <emCase1recog.e3I32,emCase1andandnot.e4I81I32I42>,
  <visual3green,visual4red>
];

/* Program for game B4 */
```

```
programB4 =
[ <en1I81,en2I81,en3I81,en4I81,en6I81,en7I81,en8I81,en9I81>,
  <emCase1recog.e8I81,emCase1recog.e6I81>, <visual8green,visual6red>,
  <en1I72,en2I72,en3I72,en4I72,en6I72,en7I72,en8I72,en9I72>,
  <emCase1recog.e7I72,emCase1andandnot.e4I81I72I42>,
  <visual7green,visual4red>
];

/* Program for game B5 */
programB5 =
[ <en1I81,en2I81,en3I81,en4I81,en6I81,en7I81,en8I81,en9I81>,
  <emCase1recog.e8I81,emCase1recog.e6I81>, <visual8green,visual6red>,
  <en1I92,en2I92,en3I92,en4I92,en6I92,en7I92,en8I92,en9I92>,
  <emCase1recog.e9I92,emCase1andandnot.e4I81I9242>,
  <visual9green,visual4red>
];

/* Program for game B6 */
programB6 =
[ <en1I71,en2I71,en3I71,en4I71,en6I71,en7I71,en8I71,en9I71>,
  <emCase1recog.e7I71,emCase1recog.e6I71>, <visual7green,visual6red>,
  <en1I12,en2I12,en3I12,en4I12,en6I12,en7I12,en8I12,en9I12>,
  <emCase1recog.e1I12,emCase1andandnot.e4I71I12I42>,
  <visual1green,visual4red>
];

/* Program for game B7 */
programB7 =
[ <en1I71,en2I71,en3I71,en4I71,en6I71,en7I71,en8I71,en9I71>,
  <emCase1recog.e7I71,emCase1recog.e6I71>, <visual7green,visual6red>,
  <en1I22,en2I22,en3I22,en4I22,en6I22,en7I22,en8I22,en9I22>,
  <emCase1recog.e2I22,emCase1andandnot.e4I71I22I42>,
  <visual2green,visual4red>
];

/* Program for game B8 */
programB8 =
[ <en1I71,en2I71,en3I71,en4I71,en6I71,en7I71,en8I71,en9I71>,
  <emCase1recog.e7I71,emCase1recog.e6I71>, <visual7green,visual6red>,
  <en1I32,en2I32,en3I32,en4I32,en6I32,en7I32,en8I32,en9I32>,
  <emCase1recog.e3I32,emCase1andandnot.e4I71I32I42>,
```

```
  <visual3green,visual4red>
];

/* Program for game B9 */
programB9 =
[ <en1I71,en2I71,en3I71,en4I71,en6I71,en7I71,en8I71,en9I71>,
  <emCase1recog.e7I71,emCase1recog.e6I71>, <visual7green,visual6red>,
  <en1I82,en2I82,en3I82,en4I82,en6I82,en7I82,en8I82,en9I82>,
  <emCase1recog.e2I82,emCase1andandnot.e4I71I82I42>,
  <visual2green,visual4red>
];

/* Program for game B10 */
programB10 =
[ <en1I71,en2I71,en3I71,en4I71,en6I71,en7I71,en8I71,en9I71>,
  <emCase1recog.e7I71,emCase1recog.e6I71>, <visual7green,visual6red>,
  <en1I92,en2I92,en3I92,en4I92,en6I92,en7I92,en8I92,en9I92>,
  <emCase1recog.e9I92,emCase1andandnot.e4I71I92I42>,
  <visual9green,visual4red>
];

/* Program for game B11 */
programB11 =
[ <en1I81,en2I81,en3I81,en4I81,en6I81,en7I81,en8I81,en9I81>,
  <emCase1recog.e8I81,emCase1recog.e6I81>, <visual8green,visual6red>,
  <en1I42,en2I42,en3I42,en4I42,en6I42,en7I42,en8I42,en9I42>,
  <emCase1recog.e4I42,emCase1and.e9I81I42>, <visual4green,visual9red>,
  <en1I23,en2I23,en3I23,en4I23,en6I23,en7I23,en8I23,en9I23>,
  <emCase1recog.e2I23,emCase1andandnot.e1I42I23I71>,
  <visual2green,visual1red>
];

/* Program for game B12 */
programB12 =
[ <en1I81,en2I81,en3I81,en4I81,en6I81,en7I81,en8I81,en9I81>,
  <emCase1recog.e8I81,emCase1recog.e6I81>, <visual8green,visual6red>,
  <en1I42,en2I42,en3I42,en4I42,en6I42,en7I42,en8I42,en9I42>,
  <emCase1recog.e4I42,emCase1and.e9I81I42>, <visual4green,visual9red>,
  <en1I33,en2I33,en3I33,en4I33,en6I33,en7I33,en8I33,en9I33>,
  <emCase1recog.e3I33,emCase1andandnot.e1I42I33I71>,
  <visual3green,visual1red>
```

```
];

/* Program for game B13 */
programB13 =
[ <en1I81,en2I81,en3I81,en4I81,en6I81,en7I81,en8I81,en9I81>,
  <emCase1recog.e8I81,emCase1recog.e6I81>, <visual8green,visual6red>,
  <en1I42,en2I42,en3I42,en4I42,en6I42,en7I42,en8I42,en9I42>,
  <emCase1recog.e4I42,emCase1and.e9I81I42>, <visual4green,visual9red>,
  <en1I73,en2I73,en3I73,en4I73,en6I73,en7I73,en8I73,en9I73>,
  <emCase1recog.e7I73,emCase1andandnot.e1I42I73I71>,
  <visual7green,visual1red>
];

/* Program for game B14 */
programB14 =
[ <en1I81,en2I81,en3I81,en4I81,en6I81,en7I81,en8I81,en9I81>,
  <emCase1recog.e8I81,emCase1recog.e6I81>, <visual8green,visual6red>,
  <en1I42,en2I42,en3I42,en4I42,en6I42,en7I42,en8I42,en9I42>,
  <emCase1recog.e4I42,emCase1and.e9I81I42>, <visual4green,visual9red>,
  <en1I13,en2I13,en3I13,en4I13,en6I13,en7I13,en8I13,en9I13>,
  <emCase2recog.e1I13,emCase1and.e3I42I13>, <visual1green,visual3red>
];

/* Program for game B15 */
programB15 =
[ <en1I71,en2I71,en3I71,en4I71,en6I71,en7I71,en8I71,en9I71>,
  <emCase1recog.e7I71,emCase1recog.e6I71>, <visual7green,visual6red>,
  <en1I42,en2I42,en3I42,en4I42,en6I42,en7I42,en8I42,en9I42>,
  <emCase1recog.e4I42,emCase1and.e1I7I42>, <visual4green,visual1red>,
  <en1I23,en2I23,en3I23,en4I23,en6I23,en7I23,en8I23,en9I23>,
  <emCase1recog.e2I23,emCase1andandnot.e9I42I23I81>,
  <visual2green,visual9red>
];

/* Program for game B16 */
programB16 =
[ <en1I71,en2I71,en3I71,en4I71,en6I71,en7I71,en8I71,en9I71>,
  <emCase1recog.e7I71,emCase1recog.e6I71>, <visual7green,visual6red>,
  <en1I42,en2I42,en3I42,en4I42,en6I42,en7I42,en8I42,en9I42>,
  <emCase1recog.e4I42,emCase1and.e1I7I42>, <visual4green,visual1red>,
  <en1I33,en2I33,en3I33,en4I33,en6I33,en7I33,en8I33,en9I33>,
```

```
  <emCase1recog.e3I33,emCase1andandnot.e9I42I33I91>,
  <visual3green,visual9red>
];

/* Program for game B17 */
programB17 =
[ <en1I71,en2I71,en3I71,en4I71,en6I71,en7I71,en8I71,en9I71>,
  <emCase1recog.e7I71,emCase1recog.e6I71>, <visual7green,visual6red>,
  <en1I42,en2I42,en3I42,en4I42,en6I42,en7I42,en8I42,en9I42>,
  <emCase1recog.e4I42,emCase1and.e1I7I42>, <visual4green,visual1red>,
  <en1I83,en2I83,en3I83,en4I83,en6I83,en7I83,en8I83,en9I83>,
  <emCase1recog.e8I83,emCase1andandnot.e9I42I83I81>,
  <visual8green,visual9red>
];

/* Program for game B18 */
programB18 =
[ <en1I71,en2I71,en3I71,en4I71,en6I71,en7I71,en8I71,en9I71>,
  <emCase1recog.e7I71,emCase1recog.e6I71>, <visual7green,visual6red>,
  <en1I42,en2I42,en3I42,en4I42,en6I42,en7I42,en8I42,en9I42>,
  <emCase1recog.e4I42,emCase1and.e1I7I42>, <visual4green,visual1red>,
  <en1I93,en2I93,en3I93,en4I93,en6I93,en7I93,en8I93,en9I93>,
  <emCase1recog.e9I93,emCase1and.e8I42I93>, <visual9green,visual8red>,
  <en1I24,en2I24,en3I24,en4I24,en6I24,en7I24,en8I24,en9I24>,
  <emCase1recog.e2I124,emCase2and.e3I93I24>, <visual2green,visual3green>
];

/* Program for game B19 */
programB19 =
[ <en1I71,en2I71,en3I71,en4I71,en6I71,en7I71,en8I71,en9I71>,
  <emCase1recog.e7I71,emCase1recog.e6I71>, <visual7green,visual6red>,
  <en1I42,en2I42,en3I42,en4I42,en6I42,en7I42,en8I42,en9I42>,
  <emCase1recog.e4I42,emCase1and.e1I7I42>, <visual4green,visual1red>,
  <en1I93,en2I93,en3I93,en4I93,en6I93,en7I93,en8I93,en9I93>,
  <emCase1recog.e9I93,emCase1and.e8I42I93>, <visual9green,visual8red>,
  <en1I34,en2I34,en3I34,en4I34,en6I34,en7I34,en8I34,en9I34>,
  <emCase1recog.e3I34,emCase1and.e2I93I34>, <visual3green,visual2red>
];

/* Program for game C1 */
programC1 =
```

```
[ <en1I11,en2I11,en3I11,en4I11,en6I11,en7I11,en8I11,en9I11>,
  <emCase1recog.e1I11,emCase1recog.e8I11>, <visual1green,visual8red>,
  <en1I32,en2I32,en3I32,en4I32,en6I32,en7I32,en8I32,en9I32>,
  <emCase1recog.e3I32,emCase1andandnot.e2I11I32I22>,
  <visual3green,visual2red>
];

/* Program for game C2 */
programC2 =
[ <en1I11,en2I11,en3I11,en4I11,en6I11,en7I11,en8I11,en9I11>,
  <emCase1recog.e1I11,emCase1recog.e8I11>, <visual1green,visual8red>,
  <en1I42,en2I42,en3I42,en4I42,en6I42,en7I42,en8I42,en9I42>,
  <emCase1recog.e4I42,emCase1andandnot.e2I11I42I22>,
  <visual4green,visual2red>
];

/* Program for game C3 */
programC3 =
[ <en1I11,en2I11,en3I11,en4I11,en6I11,en7I11,en8I11,en9I11>,
  <emCase1recog.e1I11,emCase1recog.e8I11>, <visual1green,visual8red>,
  <en1I62,en2I62,en3I62,en4I62,en6I62,en7I62,en8I62,en9I62>,
  <emCase1recog.e6I62,emCase1andandnot.e1I11I62I22>,
  <visual6green,visual2red>
];

/* Program for game C4 */
programC4 =
[ <en1I11,en2I11,en3I11,en4I11,en6I11,en7I11,en8I11,en9I11>,
  <emCase1recog.e1I11,emCase1recog.e8I11>, <visual1green,visual8red>,
  <en1I72,en2I72,en3I72,en4I72,en6I72,en7I72,en8I72,en9I72>,
  <emCase1recog.e7I72,emCase1andandnot.e2I11I72I22>,
  <visual7green,visual2red>
];

/* Program for game C5 */
programC5 =
[ <en1I11,en2I11,en3I11,en4I11,en6I11,en7I11,en8I11,en9I11>,
  <emCase1recog.e1I11,emCase1recog.e8I11>, <visual1green,visual8red>,
  <en1I92,en2I92,en3I92,en4I92,en6I92,en7I92,en8I92,en9I92>,
  <emCase1recog.e9I92,emCase1andandnot.e2I11I92I22>,
  <visual9green,visual2red>
```

```
];

/* Program for game C6 */
programC6 =
[ <en1I41,en2I41,en3I41,en4I41,en6I41,en7I41,en8I41,en9I41>,
  <emCase1recog.e4I41,emCase1recog.e8I41>, <visual4green,visual8red>,
  <en1I12,en2I12,en3I12,en4I12,en6I12,en7I12,en8I12,en9I12>,
  <emCase1recog.e1I12,emCase1andandnot.e2I41I12I22>,
  <visual1green,visual2red>
];

/* Program for game C7 */
programC7 =
[ <en1I41,en2I41,en3I41,en4I41,en6I41,en7I41,en8I41,en9I41>,
  <emCase1recog.e4I41,emCase1recog.e8I41>, <visual4green,visual8red>,
  <en1I32,en2I32,en3I32,en4I32,en6I32,en7I32,en8I32,en9I32>,
  <emCase1recog.e3I32,emCase1andandnot.e2I41I32I22>,
  <visual3green,visual2red>
];

/* Program for game C8 */
programC8 =
[ <en1I41,en2I41,en3I41,en4I41,en6I41,en7I41,en8I41,en9I41>,
  <emCase1recog.e4I41,emCase1recog.e8I41>, <visual4green,visual8red>,
  <en1I62,en2I62,en3I62,en4I62,en6I62,en7I62,en8I62,en9I62>,
  <emCase1recog.e6I62,emCase1andandnot.e2I41I62I22>,
  <visual6green,visual2red>
];

/* Program for game C9 */
programC9 =
[ <en1I41,en2I41,en3I41,en4I41,en6I41,en7I41,en8I41,en9I41>,
  <emCase1recog.e4I41,emCase1recog.e8I41>, <visual4green,visual8red>,
  <en1I72,en2I72,en3I72,en4I72,en6I72,en7I72,en8I72,en9I72>,
  <emCase1recog.e7I72,emCase1andandnot.e2I41I72I22>,
  <visual7green,visual2red>
];

/* Program for game C10 */
programC10 =
[ <en1I41,en2I41,en3I41,en4I41,en6I41,en7I41,en8I41,en9I41>,
```

```
  <emCase1recog.e4I41,emCase1recog.e8I41>, <visual4green,visual8red>,
  <en1I92,en2I92,en3I92,en4I92,en6I92,en7I92,en8I92,en9I92>,
  <emCase1recog.e9I92,emCase1andandnot.e2I41I92I22>,
  <visual9green,visual2red>
];

/* Program for game C11 */
programC11 =
[ <en1I11,en2I11,en3I11,en4I11,en6I11,en7I11,en8I11,en9I11>,
  <emCase1recog.e1I11,emCase1recog.e8I11>, <visual1green,visual8red>,
  <en1I22,en2I22,en3I22,en4I22,en6I22,en7I22,en8I22,en9I22>,
  <emCase1recog.e2I22,emCase1and.e3I11I22>, <visual2green,visual3red>,
  <en1I43,en2I43,en3I43,en4I43,en6I43,en7I43,en8I43,en9I43>,
  <emCase1recog.e4I43,emCase1andandnot.e7I22I43I41>,
  <visual4green,visual7red>
];

/* Program for game C12 */
programC12 =
[ <en1I11,en2I11,en3I11,en4I11,en6I11,en7I11,en8I11,en9I11>,
  <emCase1recog.e1I11,emCase1recog.e8I11>, <visual1green,visual8red>,
  <en1I22,en2I22,en3I22,en4I22,en6I22,en7I22,en8I22,en9I22>,
  <emCase1recog.e2I22,emCase1and.e3I11I22>, <visual2green,visual3red>,
  <en1I63,en2I63,en3I63,en4I63,en6I63,en7I63,en8I63,en9I63>,
  <enCase1recog.e6I63,emCase1andandnot.e7I22I63I41>,
  <visual6green,visual7red>
];

/* Program for game C13 */
programC13 =
[ <en1I11,en2I11,en3I11,en4I11,en6I11,en7I11,en8I11,en9I11>,
  <emCase1recog.e1I11,emCase1recog.e8I11>, <visual1green,visual8red>,
  <en1I22,en2I22,en3I22,en4I22,en6I22,en7I22,en8I22,en9I22>,
  <emCase1recog.e2I22,emCase1and.e3I11I22>, <visual2green,visual3red>,
  <en1I93,en2I93,en3I93,en4I93,en6I93,en7I93,en8I93,en9I93>,
  <emCase1recog.e9I93,emCase1andandnot.e7I22I93I41>,
  <visual9green,visual7red>
];

/* Program for game C14 */
programC14 =
```

```
[ <en1I41,en2I41,en3I41,en4I41,en6I41,en7I41,en8I41,en9I41>,
  <emCase1recog.e4I41,emCase1recog.e8I41>, <visual4green,visual8red>,
  <en1I22,en2I22,en3I22,en4I22,en6I22,en7I22,en8I22,en9I22>,
  <emCase1recog.e1I22,emCase1and.e7I41I22>, <visual2green,visual7red>,
  <en1I13,en2I13,en3I13,en4I13,en6I13,en7I13,en8I13,en9I13>,
  <emCase1recog.e1I13,emCase1andandnot.e3I22I13I11>,
  <visual1green,visual3red>
];

/* Program for game C15 */
programC15 =
[ <en1I41,en2I41,en3I41,en4I41,en6I41,en7I41,en8I41,en9I41>,
  <emCase1recog.e4I41,emCase1recog.e8I41>, <visual4green,visual8red>,
  <en1I22,en2I22,en3I22,en4I22,en6I22,en7I22,en8I22,en9I22>,
  <emCase1recog.e1I22,emCase1and.e7I41I22>, <visual2green,visual7red>,
  <en1I33,en2I33,en3I33,en4I33,en6I33,en7I33,en8I33,en9I33>,
  <emCase1recog.e3I33,emCase1and.e9I22I33>, <visual3green,visual9red>
];

/* Program for game C16 */
programC16 =
[ <en1I41,en2I41,en3I41,en4I41,en6I41,en7I41,en8I41,en9I41>,
  <emCase1recog.e4I41,emCase1recog.e8I41>, <visual4green,visual8red>,
  <en1I22,en2I22,en3I22,en4I22,en6I22,en7I22,en8I22,en9I22>,
  <emCase1recog.e1I22,emCase1and.e7I41I22>, <visual2green,visual7red>,
  <en1I63,en2I63,en3I63,en4I63,en6I63,en7I63,en8I63,en9I63>,
  <emCase1recog.e6I63,emCase1andandnot.e3I22I63I11>,
  <visual6green,visual3red>
];

/* Program for game C17 */
programC17 =
[ <en1I41,en2I41,en3I41,en4I41,en6I41,en7I41,en8I41,en9I41>,
  <emCase1recog.e4I41,emCase1recog.e8I41>, <visual4green,visual8red>,
  <en1I22,en2I22,en3I22,en4I22,en6I22,en7I22,en8I22,en9I22>,
  <emCase1recog.e1I22,emCase1and.e7I41I22>, <visual2green,visual7red>,
  <en1I93,en2I93,en3I93,en4I93,en6I93,en7I93,en8I93,en9I93>,
  <emCase1recog.e9I93,emCase1andandnot.e3I22I93I11>,
  <visual9green,visual3red>
];
```

```
/* Program for game C18 */
programC18 =
[ <en1I11,en2I11,en3I11,en4I11,en6I11,en7I11,en8I11,en9I11>,
  <emCase1recog.e1I11,emCase1recog.e8I11>, <visual1green,visual8red>,
  <en1I22,en2I22,en3I22,en4I22,en6I22,en7I22,en8I22,en9I22>,
  <emCase1recog.e2I22,emCase1and.e3I11I22>, <visual2green,visual3red>,
  <en1I73,en2I73,en3I73,en4I73,en6I73,en7I73,en8I73,en9I73>,
  <emCase1recog.e7I73,emCase1and.e4I22I73>, <visual7green,visual4red>,
  <en1I64,en2I64,en3I64,en4I64,en6I64,en7I64,en8I64,en9I64>,
  <emCase1recog.e6I64,emCase1and.e9I73I64>, <visual6green,visual9red>
];

/* Program for game C19 */
programC19 =
[ <en1I11,en2I11,en3I11,en4I11,en6I11,en7I11,en8I11,en9I11>,
  <emCase1recog.e1I11,emCase1recog.e8I11>, <visual1green,visual8red>,
  <en1I22,en2I22,en3I22,en4I22,en6I22,en7I22,en8I22,en9I22>,
  <emCase1recog.e2I22,emCase1and.e3I11I22>, <visual2green,visual3red>,
  <en1I73,en2I73,en3I73,en4I73,en6I73,en7I73,en8I73,en9I73>,
  <emCase1recog.e7I73,emCase1and.e4I22I73>, <visual7green,visual4red>,
  <en1I94,en2I94,en3I94,en4I94,en6I94,en7I94,en8I94,en9I94>,
  <emCase1recog.e9I94,emCase1and.e6I73I94>, <visual9green,visual6red>
];

/* Program for game D1 */
programD1 =
[ <en1I21,en2I21,en3I21,en4I21,en6I21,en7I21,en8I21,en9I21>,
  <emCase1recog.e2I21,emCase1recog.e4I21>, <visual2green,visual4red>,
  <en1I12,en2I12,en3I12,en4I12,en6I12,en7I12,en8I12,en9I12>,
  <emCase1recog.e1I12,emCase1andandnot.e6I21I12I62>,
  <visual1green,visual6red>
];

/* Program for game D2 */
programD2 =
[ <en1I21,en2I21,en3I21,en4I21,en6I21,en7I21,en8I21,en9I21>,
  <emCase1recog.e2I21,emCase1recog.e4I21>, <visual2green,visual4red>,
  <en1I32,en2I32,en3I32,en4I32,en6I32,en7I32,en8I32,en9I32>,
  <emCase1recog.e3I32,emCase1andandnot.e2I21I32I62>,
  <visual3green,visual6red>
];
```

```
/* Program for game D3 */
programD3 =
[ <en1I21,en2I21,en3I21,en4I21,en6I21,en7I21,en8I21,en9I21>,
  <emCase1recog.e2I21,emCase1recog.e4I21>, <visual2green,visual4red>,
  <en1I72,en2I72,en3I72,en4I72,en6I72,en7I72,en8I72,en9I72>,
  <emCase1recog.e7I72,emCase1andandnot.e6I21I72I62>,
  <visual7green,visual6red>
];

/* Program for game D4 */
programD4 =
[ <en1I21,en2I21,en3I21,en4I21,en6I21,en7I21,en8I21,en9I21>,
  <emCase1recog.e2I21,emCase1recog.e4I21>, <visual2green,visual4red>,
  <en1I82,en2I82,en3I82,en4I82,en6I82,en7I82,en8I82,en9I82>,
  <emCase1recog.e1I82,emCase1andandnot.e6I21I82I62>,
  <visual8green,visual6red>
];

/* Program for game D5 */
programD5 =
[ <en1I21,en2I21,en3I21,en4I21,en6I21,en7I21,en8I21,en9I21>,
  <emCase1recog.e2I21,emCase1recog.e4I21>, <visual2green,visual4red>,
  <en1I92,en2I92,en3I92,en4I92,en6I92,en7I92,en8I92,en9I92>,
  <emCase1recog.e9I92,emCase1andandnot.e6I21I92I22>,
  <visual9green,visual6red>
];

/* Program for game D6 */
programD6 =
[ <en1I31,en2I31,en3I31,en4I31,en6I31,en7I31,en8I31,en9I31>,
  <emCase1recog.e3I31,emCase1recog.e4I31>, <visual3green,visual4red>,
  <en1I12,en2I12,en3I12,en4I12,en6I12,en7I12,en8I12,en9I12>,
  <emCase1recog.e1I12,emCase1andandnot.e6I31I12I62>,
  <visual1green,visual6red>
];

/* Program for game D7 */
programD7 =
[ <en1I31,en2I31,en3I31,en4I31,en6I31,en7I31,en8I31,en9I31>,
  <emCase1recog.e3I31,emCase1recog.e4I31>, <visual3green,visual4red>,
```

```
  <en1I22,en2I22,en3I22,en4I22,en6I22,en7I22,en8I22,en9I22>,
  <emCase1recog.e2I22,emCase1andandnot.e6I31I22I62>,
  <visual2green,visual6red>
];

/* Program for game D8 */
programD8 =
[ <en1I31,en2I31,en3I31,en4I31,en6I31,en7I31,en8I31,en9I31>,
  <emCase1recog.e3I31,emCase1recog.e4I31>, <visual3green,visual4red>,
  <en1I72,en2I72,en3I72,en4I72,en6I72,en7I72,en8I72,en9I72>,
  <emCase1recog.e7I72,emCase1andandnot.e6I31I72I62>,
  <visual7green,visual6red>
];

/* Program for game D9 */
programD9 =
[ <en1I31,en2I31,en3I31,en4I31,en6I31,en7I31,en8I31,en9I31>,
  <emCase1recog.e3I31,emCase1recog.e4I31>, <visual3green,visual4red>,
  <en1I82,en2I82,en3I82,en4I82,en6I82,en7I82,en8I82,en9I82>,
  <emCase1recog.e2I82,emCase1andandnot.e6I31I82I62>,
  <visual2green,visual6red>
];

/* Program for game D10 */
programD10 =
[ <en1I31,en2I31,en3I31,en4I31,en6I31,en7I31,en8I31,en9I31>,
  <emCase1recog.e3I31,emCase1recog.e4I31>, <visual3green,visual4red>,
  <en1I92,en2I92,en3I92,en4I92,en6I92,en7I92,en8I92,en9I92>,
  <emCase1recog.e9I92,emCase1andandnot.e6I31I92I22>,
  <visual9green,visual6red>
];

/* Program for game D11 */
programD11 =
[ <en1I21,en2I21,en3I21,en4I21,en6I21,en7I21,en8I21,en9I21>,
  <emCase1recog.e2I21,emCase1recog.e4I21>, <visual2green,visual4red>,
  <en1I62,en2I62,en3I62,en4I62,en6I62,en7I62,en8I62,en9I62>,
  <emCase1recog.e6I62,emCase1and.e1I12I62>, <visual6green,visual1red>,
  <en1I33,en2I33,en3I33,en4I33,en6I33,en7I33,en8I33,en9I33>,
  <emCase1recog.e3I33,emCase1andandnot.e9I62I33I31>,
  <visual3green,visual9red>
```

```
];

/* Program for game D12 */
programD12 =
[ <en1I21,en2I21,en3I21,en4I21,en6I21,en7I21,en8I21,en9I21>,
  <emCase1recog.e2I21,emCase1recog.e4I21>, <visual2green,visual4red>,
  <en1I62,en2I62,en3I62,en4I62,en6I62,en7I62,en8I62,en9I62>,
  <emCase1recog.e6I62,emCase1and.e1I12I62>, <visual6green,visual1red>,
  <en1I73,en2I73,en3I73,en4I73,en6I73,en7I73,en8I73,en9I73>,
  <emCase1recog.e7I73,emCase1andandnot.e9I62I73I31>,
  <visual7green,visual9red>
];

/* Program for game D13 */
programD13 =
[ <en1I21,en2I21,en3I21,en4I21,en6I21,en7I21,en8I21,en9I21>,
  <emCase1recog.e2I21,emCase1recog.e4I21>, <visual2green,visual4red>,
  <en1I62,en2I62,en3I62,en4I62,en6I62,en7I62,en8I62,en9I62>,
  <emCase1recog.e6I62,emCase1and.e1I12I62>, <visual6green,visual1red>,
  <en1I83,en2I83,en3I83,en4I83,en6I83,en7I83,en8I83,en9I83>,
  <emCase1recog.e3I83,emCase1andandnot.e1I62I83I31>,
  <visual3green,visual9red>
];

/* Program for game D14 */
programD14 =
[ <en1I21,en2I21,en3I21,en4I21,en6I21,en7I21,en8I21,en9I21>,
  <emCase1recog.e2I21,emCase1recog.e4I21>, <visual2green,visual4red>,
  <en1I62,en2I62,en3I62,en4I62,en6I62,en7I62,en8I62,en9I62>,
  <emCase1recog.e6I62,emCase1and.e1I12I62>, <visual6green,visual1red>,
  <en1I93,en2I93,en3I93,en4I93,en6I93,en7I93,en8I93,en9I93>,
  <emCase1recog.e9I93,emCase1and.e7I62I93>, <visual9green,visual7red>
];

/* Program for game D15 */
programD15 =
[ <en1I31,en2I31,en3I31,en4I31,en6I31,en7I31,en8I31,en9I31>,
  <emCase1recog.e3I31,emCase1recog.e4I31>, <visual3green,visual4red>,
  <en1I62,en2I62,en3I62,en4I62,en6I62,en7I62,en8I62,en9I62>,
  <emCase1recog.e6I62,emCase1and.e9I31I62>, <visual6green,visual9red>,
  <en1I23,en2I23,en3I23,en4I23,en6I23,en7I23,en8I23,en9I23>,
```

```
  <emCase1recog.e2I23,emCase1andandnot.e1I62I23I21>
];

/* Program for game D16 */
programD16 =
[ <en1I31,en2I31,en3I31,en4I31,en6I31,en7I31,en8I31,en9I31>,
  <emCase1recog.e3I31,emCase1recog.e4I31>, <visual3green,visual4red>,
  <en1I62,en2I62,en3I62,en4I62,en6I62,en7I62,en8I62,en9I62>,
  <emCase1recog.e6I62,emCase1and.e9I31I62>, <visual6green,visual9red>,
  <en1I73,en2I73,en3I73,en4I73,en6I73,en7I73,en8I73,en9I73>,
  <emCase1recog.e7I73,emCase1andandnot.e1I62I73I21>,
  <visual7green,visual1red>
];

/* Program for game D17 */
programD17 =
[ <en1I31,en2I31,en3I31,en4I31,en6I31,en7I31,en8I31,en9I31>,
  <emCase1recog.e3I31,emCase1recog.e4I31>, <visual3green,visual4red>,
  <en1I62,en2I62,en3I62,en4I62,en6I62,en7I62,en8I62,en9I62>,
  <emCase1recog.e6I62,emCase1and.e9I31I62>, <visual6green,visual9red>,
  <en1I83,en2I83,en3I83,en4I83,en6I83,en7I83,en8I83,en9I83>,
  <emCase1recog.e8I83,emCase1andandnot.e1I62I83I21>,
  <visual8green,visual1red>
];

/* Program for game D18 */
programD18 =
[ <en1I31,en2I31,en3I31,en4I31,en6I31,en7I31,en8I31,en9I31>,
  <emCase1recog.e3I31,emCase1recog.e4I31>, <visual3green,visual4red>,
  <en1I62,en2I62,en3I62,en4I62,en6I62,en7I62,en8I62,en9I62>,
  <emCase1recog.e6I62,emCase1and.e9I31I62>, <visual6green,visual9red>,
  <en1I13,en2I13,en3I13,en4I13,en6I13,en7I13,en8I13,en9I13>,
  <emCase1recog.e1I13,emCase1and.e2I62I13>, <visual1green,visual2red>,
  <en1I74,en2I74,en3I74,en4I74,en6I74,en7I74,en8I74,en9I74>,
  <emCase1recog.e7I74,emCase1and.e8I13I74>, <visual7green,visual8red>
];

/* Program for game D19 */
programD19 =
[ <en1I31,en2I13,en3I31,en4I31,en6I31,en7I31,en8I31,en9I31>,
  <emCase1recog.e3I31,emCase1recog.e4I31>, <visual3green,visual4red>,
```

```
  <en1I62,en2I62,en3I62,en4I62,en6I62,en7I62,en8I62,en9I62>,
  <emCase1recog.e6I62,emCase1and.e9I31I62>, <visual6green,visual9red>,
  <en1I13,en2I13,en3I13,en4I13,en6I13,en7I13,en8I13,en9I13>,
  <emCase1recog.e1I13,emCase1and.e2I62I13>, <visual1green,visual2red>,
  <en1I84,en2I84,en3I84,en4I84,en6I84,en7I84,en8I84,en9I84>,
  <emCase1recog.e8I84,emCase1and.e7I13I84>, <visual8green,visual7red>,
];

/* M2 Level 3 Description */

LEVEL 3

LENGTH

lengthI = 15;

lengthE817 = 17;
lengthE6 = 15;

lengthE817LeftYes = 55;
lengthE817RightYes = 55;

lengthE6LeftYes = 58;
lengthE6RightYes = 59;

length817And = 80;
lengthE6And = 82;

lengthE6AndAndNot = 98;

SEQUENCE

/* recognizer gate templates */
seqE817LeftYes = GGAAGATCAT-ia-ATGATCTTCCGAGCCGGTCGAAAGTTACTA;
seqE817RightYes = ATGATCTTCCGAGCCGGTCGAAAGTTACTA-ia-TAGTAACTTT;
seqE6LeftYes = TGAAGAG-ia-CTCTTCAGCGATGGCGAAGCCCACCCATGTTAGTGA;
seqE6RightYes = CTCTTCAGCGATGGCGAAGCCCACCCATGTTAGTGA-ia-TCACTAAC;

/* and gate templates */
seqE817And = GGAAGATCAT-ia-ATGATCTTCCGAGCCGGTCGAAAGTTACTA-ib-TAGTAACTTT;
seqE6And = CTGAAGAG-ia-CTCTTCAGCGATGGCGAAGCCCACCCATGTTAGTGA-ib-TCACTAAC;
```

```
/* and and not gate template */
seqE6AndAndNot = CTGAAGAG-ia-CTCTTCAGCGATGACTG-ic-
                 CAGTCCACCCATGTTAGTGA-ib-TCACTAAC;

/* substrate sequences */
seqE817 = TAGTAACTAGAGATCAT;
seqE6 = TCACTATAGGAAGAG;

/* sequences serving in literal strands */

seq.lI11;
seq.lI12;
seq.lI13;
seq.lI14;
seq.lI21;
seq.lI22;
seq.lI23;
seq.lI24;
seq.lI31;
seq.lI32;
seq.lI33;
seq.lI34;
seq.lI41;
seq.lI42;
seq.lI43;
seq.lI44;
seq.lI61;
seq.lI62;
seq.lI63;
seq.lI64;
seq.lI71;
seq.lI72;
seq.lI73;
seq.lI74;
seq.lI81;
seq.lI82;
seq.lI83;
seq.lI84;
seq.lI91;
seq.lI92;
```

```
seq.lI93;
seq.lI94;

/* sequences serving in recognition function strands */

seq.rI11;
seq.rI12;
seq.rI13;
seq.rI14;
seq.rI21;
seq.rI22;
seq.rI23;
seq.rI24;
seq.rI31;
seq.rI32;
seq.rI33;
seq.rI34;
seq.rI41;
seq.rI42;
seq.rI43;
seq.rI44;
seq.rI61;
seq.rI62;
seq.rI63;
seq.rI64;
seq.rI71;
seq.rI72;
seq.rI73;
seq.rI74;
seq.rI81;
seq.rI82;
seq.rI83;
seq.rI84;
seq.rI91;
seq.rI92;
seq.rI93;
seq.rI94;

/* sequences serving in and function strands */

seq.aI21I62;
```

```
seq.aI71I42;
seq.aI82I73;
seq.aI33I44;
seq.aI62I13;
seq.aI93I34;
seq.aI11I22;
seq.aI61I82;
seq.aI42I13;
seq.aI93I24;
seq.aI22I73;
seq.aI33I14;
seq.aI82I33;
seq.aI73I94;
seq.aI41I22;
seq.aI91I82;
seq.aI62I93;
seq.aI13I84;
seq.aI42I93;
seq.aI13I74;
seq.aI31I62;
seq.aI81I42;
seq.aI22I33;
seq.aI73I64;

/* sequences serving in and-and-not function strands */

seq.aanI42I73I71;
seq.aanI42I33I71;
seq.aanI42I23I71;
seq.aanI62I73I21;
seq.aanI62I83I21;
seq.aanI62I23I21;
seq.aanI11I32I22;
seq.aanI11I42I22;
seq.aanI11I62I22;
seq.aanI11I72I22;
seq.aanI11I92I22;
seq.aanI41I12I22;
seq.aanI41I32I22;
seq.aanI41I62I22;
seq.aanI41I72I22;
```

```
seq.aanI41I92I22;
seq.aanI22I13I11;
seq.aanI22I63I11;
seq.aanI22I93I11;
seq.aanI82I13I61;
seq.aanI82I43I61;
seq.aanI82I63I61;
seq.aanI71I12I42;
seq.aanI71I22I42;
seq.aanI71I32I42;
seq.aanI71I82I42;
seq.aanI71I92I42;
seq.aanI81I12I42;
seq.aanI81I22I42;
seq.aanI81I32I42;
seq.aanI81I72I42;
seq.aanI81I92I42;
seq.aanI21I12I62;
seq.aanI21I32I62;
seq.aanI21I72I62;
seq.aanI21I82I62;
seq.aanI21I92I62;
seq.aanI31I12I62;
seq.aanI31I22I62;
seq.aanI31I72I62;
seq.aanI31I82I62;
seq.aanI31I92I62;
seq.aanI22I43I41;
seq.aanI22I63I41;
seq.aanI22I93I41;
seq.aanI82I13I91;
seq.aanI82I43I91;
seq.aanI82I93I91;
seq.aanI61I12I82;
seq.aanI61I32I82;
seq.aanI61I42I82;
seq.aanI61I72I82;
seq.aanI61I92I82;
seq.aanI91I12I82;
seq.aanI91I32I82;
seq.aanI91I42I82;
```

```
seq.aanI91I62I82;
seq.aanI91I72I82;
seq.aanI42I23I81;
seq.aanI42I33I81;
seq.aanI42I83I81;
seq.aanI62I33I31;
seq.aanI62I73I31;
seq.aanI62I83I31;

ISO

/* recognizer gate structures */

with stage1
structE817LeftYes = [(0,10,15),(35,2,5)];
structE817RightYes = [(8,3,3),(20,10,15)];
structE6LeftYes = [(0,7,15),(34,3,3),(43,3,8)];
structE6RightYes = [(12,3,3),(28,8,15)];


with stage2
structE817LeftYes = [(10,15,31)];
structE817RightYes = [(30,15,11)];
structE6LeftYes = [(7,15,37)];
structE6RightYes = [(36,15,9)];


with stage4
structE817LeftYes = [(25,8,23),(47,8,10)];
structE817RightYes = [(0,8,48),(22,8,35)];
structE6LeftYes = [(22,6,31),(52,6,10)];
structE6RightYes = [(0,6,54),(30,6,33)];


/* and gate structures */

with stage1
structE817And = [(0,10,15),(38,2,0),(45,10,15)];
structE6And = [(0,8,15),(35,3,3),(51,8,15)];
```

```
with stage2
structE817And = [(10,15,56)];
structE6And = [(8,15,60)];


with stage3
structE817And = [(55,15,27)];
structE6And = [(59,15,25)];


with stage5
structE817And = [(25,8,48),(47,8,35)];
structE6And = [(23,6,54),(53,6,33)];


/* and-and-not gate structures */

with stage1
structE6AndAndNot = [(0,8,15),(35,5,15),(67,8,15)];


with stage2
structE6AndAndNot = [(8,15,76)];


with stage3
structE6AndAndNot = [(75,15,41)];


with stage5
structE6AndAndNot = [(23,6,70),(69,6,33)];


FLUOROPHORE

FAM <-> green;
TAMRA <-> red;
JOE <-> pink;
ROX <-> purple;

QUENCHER
```

```
BH2;

STRAND

/* substrate variations */

substrateE817FAM = (FAM-seqE817-BH2,[],lengthE817);
substrateE817TAMRA = (TAMRA-seqE817-BH2,[],lengthE817);
substrateE6FAM = (FAM-seqE6-BH2,[],lengthE6);
substrateE6TAMRA = (TAMRA-seqE6-BH2,[],lengthE6);

substrateFAM = substrateE817FAM || substrateE817TAMRA;
substrateTAMRA = substrateE6FAM || substrateE817TAMRA;

/* literal strands */

str.lI11 = (seq.lI11,[],lengthI);
str.lI12 = (seq.lI12,[],lengthI);
str.lI13 = (seq.lI13,[],lengthI);
str.lI14 = (seq.lI14,[],lengthI);
str.lI21 = (seq.lI21,[],lengthI);
str.lI22 = (seq.lI22,[],lengthI);
str.lI23 = (seq.lI23,[],lengthI);
str.lI24 = (seq.lI24,[],lengthI);
str.lI31 = (seq.lI31,[],lengthI);
str.lI32 = (seq.lI32,[],lengthI);
str.lI33 = (seq.lI33,[],lengthI);
str.lI34 = (seq.lI34,[],lengthI);
str.lI41 = (seq.lI41,[],lengthI);
str.lI42 = (seq.lI42,[],lengthI);
str.lI43 = (seq.lI43,[],lengthI);
str.lI44 = (seq.lI44,[],lengthI);
str.lI61 = (seq.lI61,[],lengthI);
str.lI62 = (seq.lI62,[],lengthI);
str.lI63 = (seq.lI63,[],lengthI);
str.lI64 = (seq.lI64,[],lengthI);
str.lI71 = (seq.lI71,[],lengthI);
str.lI72 = (seq.lI72,[],lengthI);
str.lI73 = (seq.lI73,[],lengthI);
str.lI74 = (seq.lI74,[],lengthI);
```

```
str.lI81 = (seq.lI81,[],lengthI);
str.lI82 = (seq.lI82,[],lengthI);
str.lI83 = (seq.lI83,[],lengthI);
str.lI84 = (seq.lI84,[],lengthI);
str.lI91 = (seq.lI91,[],lengthI);
str.lI92 = (seq.lI92,[],lengthI);
str.lI93 = (seq.lI93,[],lengthI);
str.lI94 = (seq.lI94,[],lengthI);

/* recognizer gate strands */

build (rI11,lI11), (rI12,lI12), (rI13,lI13), (rI14,lI14),
      (rI21,lI21), (rI22,lI22), (rI23,lI23), (rI24,lI24),
      (rI31,lI31), (rI32,lI32), (rI33,lI33), (rI34,lI34),
      (rI41,lI41), (rI42,lI42), (rI43,lI43), (rI44,lI44),
      (rI61,lI61), (rI62,lI62), (rI63,lI63), (rI64,lI64),
      (rI71,lI71), (rI72,lI72), (rI73,lI73), (rI74,lI74),
      (rI81,lI81), (rI82,lI82), (rI83,lI83), (rI84,lI84),
      (rI91,lI91), (rI92,lI92), (rI93,lI93), (rI94,lI94)
 seq.*1 = seqE817LeftYes | ia = revcomp(seq.*2);
  str.*1.stage0 = (seq.*1,[],lengthE817LeftYes);
  str.*1.stage1 = (seq.*1,stage1.structE817LeftYes,lengthE817LeftYes);
  str.*1.stage2 = (seq.*1-seq.*2,stage2.structE817LeftYes,
                   lengthE817LeftYes+lengthI);
  str.*1.stage4 = (seq.*1-substrateE817FAM,stage4.structE817LeftYes,
                   lengthE817LeftYes+lengthE817);  ||
 seq.*1 = seqE817RightYes | ia = revcomp(seq.*2);
  str.*1.stage0 = (seq.*1,[],lengthE817RightYes);
  str.*1.stage1 = (seq.*1,stage1.structE817RightYes,lengthE817RightYes);
  str.*1.stage2 = (seq.*1-seq.*2,stage2.structE817RightYes,
                   lengthE817RightYes+lengthI);
  str.*1.stage4 = (seq.*1-substrateE817FAM,stage4.structE817RightYes,
                   lengthE817RightYes+lengthE817);  ||
 seq.*1 = seqE6LeftYes | ia = revcomp(seq.*2);
  str.*1.stage0 = (seq.*1,[],lengthE6LeftYes);
  str.*1.stage1 = (seq.*1,stage1.structE6LeftYes,lengthE6LeftYes);
  str.*1.stage2 = (seq.*1-seq.*2,stage2.structE6LeftYes,
                   lengthE6LeftYes+lengthI);
  str.*1.stage4 = (seq.*1-substrateE6FAM,stage4.structE6LeftYes,
                   lengthE6LeftYes+lengthE6);  ||
 seq.*1 = seqE6RightYes | ia = revcomp(seq.*2);
```

```
  str.*1.stage0 = (seq.*1,[],lengthE6RightYes);
  str.*1.stage1 = (seq.*1,stage1.structE6RightYes,lengthE6RightYes);
  str.*1.stage2 = (seq.*1-seq.*2,stage2.structE6RightYes,
                   lengthE6RightYes+lengthI);
  str.*1.stage4 = (seq.*1-substrateE6FAM,stage4.structE6RightYes,
                   lengthE6RightYes+lengthE6);

build (rI11,lI11), (rI21,lI21), (rI31,lI31), (rI41,lI41),
      (rI61,lI61), (rI71,lI71), (rI81,lI81), (rI91,lI91)
 seq.*1 = seqE817LeftYes | ia = revcomp(seq.*2);
  str.*1.stage0 = (seq.*1,[],lengthE817LeftYes);
  str.*1.stage1 = (seq.*1,stage1.structE817LeftYes,
                   lengthE817LeftYes);
  str.*1.stage2 = (seq.*1-seq.*2,stage2.structE817LeftYes,
                   lengthE817LeftYes+lengthI);
  str.*1.stage4 = (seq.*1-substrateE817TAMRA,stage4.structE817LeftYes,
                   lengthE817LeftYes+lengthE817);  ||
 seq.*1 = seqE817RightYes | ia = revcomp(seq.*2);
  str.*1.stage0 = (seq.*1,[],lengthE817RightYes);
  str.*1.stage1 = (seq.*1,stage1.structE817RightYes,lengthE817RightYes);
  str.*1.stage2 = (seq.*1-seq.*2,stage2.structE817RightYes,
                   lengthE817RightYes+lengthI);
  str.*1.stage4 = (seq.*1-substrateE817TAMRA,stage4.structE817RightYes,
                   lengthE817RightYes+lengthE817);  ||
 seq.*1 = seqE6LeftYes | ia = revcomp(seq.*2);
  str.*1.stage0 = (seq.*1,[],lengthE6LeftYes);
  str.*1.stage1 = (seq.*1,stage1.structE6LeftYes,lengthE6LeftYes);
  str.*1.stage2 = (seq.*1-seq.*2,stage2.structE6LeftYes,
                   lengthE6LeftYes+lengthI);
  str.*1.stage4 = (seq.*1-substrateE6TAMRA,stage4.structE6LeftYes,
                   lengthE6LeftYes+lengthE6);  ||
 seq.*1 = seqE6RightYes | ia = revcomp(seq.*2);
  str.*1.stage0 = (seq.*1,[],lengthE6RightYes);
  str.*1.stage1 = (seq.*1,stage1.structE6RightYes,lengthE6RightYes);
  str.*1.stage2 = (seq.*1-seq.*2,stage2.structE6RightYes,
                   lengthE6RightYes+lengthI);
  str.*1.stage4 = (seq.*1-substrateE6TAMRA,stage4.structE6RightYes,
                   lengthE6RightYes+lengthE6);

/* and gate strands */
```

```
build (aI21I62,lI21,lI62), (aI71I42,lI71,lI42), (aI82I73,lI82,lI73),
      (aI33I44,lI33,lI44), (aI62I13,lI62,lI13), (aI93I34,lI93,lI34),
      (aI11I22,lI11,lI22), (aI61I82,lI61,lI82), (aI42I13,lI42,lI13),
      (aI93I24,lI93,lI24), (aI22I73,lI22,lI73), (aI33I14,lI33,lI14),
      (aI82I33,lI82,lI33), (aI73I94,lI73,lI94), (aI41I22,lI41,lI22),
      (aI91I82,lI91,lI82), (aI62I93,lI62,lI93), (aI13I84,lI13,lI84),
      (aI42I93,lI42,lI93), (aI13I74,lI13,lI74), (aI31I62,lI13,lI62),
      (aI81I42,lI81,lI42), (aI22I33,lI22,lI33), (aI73I64,lI73,lI64)
 seq.*1 = seqE6And | ia = revcomp(seq.*2), ib = revcomp(seq.*3);
  str.*1.stage0 = (seq.*1,[],lengthE6And);
  str.*1.stage1 = (seq.*1,stage1.structE6And,lengthE6And);
  str.*1.stage2 = (seq.*1-seq.*2,stage2.structE6And,lengthE6And+lengthI);
  str.*1.stage3 = (seq.*1-seq.*3,stage3.structE6And,lengthE6And+lengthI);
  str.*1.stage5 = (seq.*1-substrateE6TAMRA,stage5.structE6And,
                   lengthE6And+lengthE6);  ||
 seq.*1 = seqE817And | ia = revcomp(seq.*2), ib = revcomp(seq.*3);
  str.*1.stage0 = (seq.*1,[],lengthE817And);
  str.*1.stage1 = (seq.*1,stage1.structE817And,lengthE817And);
  str.*1.stage2 = (seq.*1-seq.*2,stage2.structE817And,
                   lengthE817And+lengthI);
  str.*1.stage3 = (seq.*1-seq.*3,stage3.structE817And,
                   lengthE817And+lengthI);
  str.*1.stage5 = (seq.*1-substrateE817TAMRA,stage5.structE817And,
                   lengthE817And+lengthE817);

/* and-and-not gate strands */

build (aanI42I73I71,lI42,lI73,lI71), (aanI42I33I71,lI42,lI33,lI71),
      (aanI42I23I71,lI42,lI23,lI71), (aanI62I73I21,lI62,lI73,lI21),
      (aanI62I83I21,lI62,lI83,lI21), (aanI62I23I21,lI62,lI23,lI21),
      (aanI11I32I22,lI11,lI32,lI22), (aanI11I42I22,lI11,lI42,lI22),
      (aanI11I62I22,lI11,lI62,lI22), (aanI11I72I22,lI11,lI72,lI22),
      (aanI11I92I22,lI11,lI92,lI22), (aanI41I12I22,lI41,lI12,lI22),
      (aanI41I32I22,lI41,lI32,lI22), (aanI41I62I22,lI41,lI62,lI22),
      (aanI41I72I22,lI41,lI72,lI22), (aanI41I92I22,lI41,lI92,lI22),
      (aanI22I13I11,lI22,lI13,lI11), (aanI22I63I11,lI22,lI63,lI11),
      (aanI22I93I11,lI22,lI93,lI11), (aanI82I13I61,lI82,lI13,lI61),
      (aanI82I43I61,lI82,lI43,lI61), (aanI82I63I61,lI82,lI63,lI61),
      (aanI71I12I42,lI71,lI12,lI42), (aanI71I22I42,lI71,lI22,lI42),
      (aanI71I32I42,lI71,lI32,lI42), (aanI71I82I42,lI71,lI82,lI42),
      (aanI71I92I42,lI71,lI92,lI42), (aanI81I12I42,lI81,lI12,lI42),
```

```
      (aanI81I22I42,lI81,lI22,lI42), (aanI81I32I42,lI81,lI32,lI42),
      (aanI81I72I42,lI81,lI72,lI42), (aanI81I92I42,lI81,lI92,lI42),
      (aanI21I12I62,lI21,lI12,lI62), (aanI21I32I62,lI21,lI32,lI62),
      (aanI21I72I62,lI21,lI72,lI62), (aanI21I82I62,lI21,lI82,lI62),
      (aanI21I92I62,lI21,lI92,lI62), (aanI31I12I62,lI31,lI12,lI62),
      (aanI31I22I62,lI31,lI22,lI62), (aanI31I72I62,lI31,lI72,lI62),
      (aanI31I82I62,lI31,lI82,lI62), (aanI31I92I62,lI31,lI92,lI62),
      (aanI22I43I41,lI22,lI43,lI41), (aanI22I63I41,lI22,lI63,lI41),
      (aanI22I93I41,lI22,lI93,lI41), (aanI82I13I91,lI82,lI13,lI91),
      (aanI82I43I91,lI82,lI43,lI91), (aanI82I93I91,lI82,lI93,lI91),
      (aanI61I12I82,lI61,lI12,lI82), (aanI61I32I82,lI61,lI32,lI82),
      (aanI61I42I82,lI61,lI42,lI82), (aanI61I72I82,lI61,lI72,lI82),
      (aanI61I92I82,lI61,lI92,lI82), (aanI91I12I82,lI91,lI12,lI82),
      (aanI91I32I82,lI91,lI32,lI82), (aanI91I42I82,lI91,lI42,lI82),
      (aanI91I62I82,lI91,lI62,lI82), (aanI91I72I82,lI91,lI72,lI82),
      (aanI42I23I81,lI42,lI23,lI81), (aanI42I33I81,lI42,lI33,lI81),
      (aanI42I83I81,lI42,lI83,lI81), (aanI62I33I31,lI62,lI33,lI31),
      (aanI62I73I31,lI62,lI73,lI31), (aanI62I83I31,lI62,lI83,lI31)
 seq.*1 = seqE6AndAndNot | ia = revcomp(seq.*2), ib = revcomp(seq.*3),
                           ic = revcomp(seq.*4);
  str.*1.stage0 = (seq.*1,[],lengthE6AndAndNot);
  str.*1.stage1 = (seq.*1,stage1.structE6AndAndNot,lengthE6AndAndNot);
  str.*1.stage2 = (seq.*1-seq.*2,stage2.structE6AndAndNot,
                   lengthE6AndAndNot+lengthI);
  str.*1.stage3 = (seq.*1-seq.*3,stage3.structE6AndAndNot,
                   lengthE6AndAndNot+lengthI);
  str.*1.stage5 = (seq.*1-substrateE6TAMRA,stage5.structE6AndAndNot,
                   lengthE6AndAndNot+lengthE817);


PHYSICALMAP

/* premiss maps */

a1formulas.p10 <-> cmCase1recog.c1I11;
a1formulas.p11 <-> cmCase1recog.c1I12;
a1formulas.p12 <-> cmCase1recog.c1I13;
a1formulas.p13 <-> cmCase1recog.c1I14;
a1formulas.p14 <-> cmCase1and.c1I21I62;
a1formulas.p15 <-> cmCase1and.c1I33I44;
a1formulas.p16 <-> cmCase1and.c1I82I73;
```

```
a1formulas.p17 <-> cmCase1and.c1I71I42;
a1formulas.p18 <-> cmCase1andandnot.c1I62I73I21;
a1formulas.p19 <-> cmCase1andandnot.c1I62I83I21;
a1formulas.p110 <-> cmCase1andandnot.c1I62I23I21;
a1formulas.p111 <-> cmCase1andandnot.c1I42I73I71;
a1formulas.p112 <-> cmCase1andandnot.c1I42I33I71;
a1formulas.p113 <-> cmCase1andandnot.c1I42I23I71;

a2formulas.p20 <-> cmCase1recog.c2I21;
a2formulas.p21 <-> cmCase1recog.c2I22;
a2formulas.p22 <-> cmCase1recog.c2I23;
a2formulas.p23 <-> cmCase1recog.c2I24;
a2formulas.p24 <-> cmCase1recog.c2I61;
a2formulas.p25 <-> cmCase1recog.c2I91;
a2formulas.p26 <-> cmCase1and.c2I62I13;
a2formulas.p27 <-> cmCase1and.c2I93I34;
a2formulas.p28 <-> cmCase1andandnot.c2I11I32I22;
a2formulas.p29 <-> cmCase1andandnot.c2I11I42I22;
a2formulas.p210 <-> cmCase1andandnot.c2I11I62I22;
a2formulas.p211 <-> cmCase1andandnot.c2I11I72I22;
a2formulas.p212 <-> cmCase1andandnot.c2I11I92I22;
a2formulas.p213 <-> cmCase1andandnot.c2I41I12I22;
a2formulas.p214 <-> cmCase1andandnot.c2I41I32I22;
a2formulas.p215 <-> cmCase1andandnot.c2I41I62I22;
a2formulas.p216 <-> cmCase1andandnot.c2I41I72I22;
a2formulas.p217 <-> cmCase1andandnot.c2I41I92I22;

a3formulas.p30 <-> cmCase1recog.c3I31;
a3formulas.p31 <-> cmCase1recog.c3I32;
a3formulas.p32 <-> cmCase1recog.c3I33;
a3formulas.p33 <-> cmCase1recog.c3I34;
a3formulas.p34 <-> cmCase1and.c3I11I22;
a3formulas.p35 <-> cmCase1and.c3I61I82;
a3formulas.p36 <-> cmCase1and.c3I42I13;
a3formulas.p37 <-> cmCase1and.c3I93I24;
a3formulas.p38 <-> cmCase1andandnot.c3I22I63I11;
a3formulas.p39 <-> cmCase1andandnot.c3I22I93I11;
a3formulas.p310 <-> cmCase1andandnot.c3I22I13I11;
a3formulas.p311 <-> cmCase1andandnot.c3I82I63I61;
a3formulas.p312 <-> cmCase1andandnot.c3I82I43I61;
a3formulas.p313 <-> cmCase1andandnot.c3I82I13I61;
```

```
a4formulas.p40 <-> cmCase1recog.c4I41;
a4formulas.p41 <-> cmCase1recog.c4I42;
a4formulas.p42 <-> cmCase1recog.c4I43;
a4formulas.p43 <-> cmCase1recog.c4I44;
a4formulas.p44 <-> cmCase1recog.c4I21;
a4formulas.p46 <-> cmCase1and.c4I22I73;
a4formulas.p47 <-> cmCase1and.c4I33I14;
a4formulas.p48 <-> cmCase1andandnot.c4I81I22I42;
a4formulas.p49 <-> cmCase1andandnot.c4I81I92I42;
a4formulas.p410 <-> cmCase1andandnot.c4I81I72I42;
a4formulas.p411 <-> cmCase1andandnot.c4I81I32I42;
a4formulas.p412 <-> cmCase1andandnot.c4I81I12I42;
a4formulas.p413 <-> cmCase1andandnot.c4I71I92I42;
a4formulas.p414 <-> cmCase1andandnot.c4I71I82I42;
a4formulas.p415 <-> cmCase1andandnot.c4I71I32I42;
a4formulas.p416 <-> cmCase1andandnot.c4I71I22I42;
a4formulas.p417 <-> cmCase1andandnot.c4I71I12I42;

a6formulas.p60 <-> cmCase1recog.c6I61;
a6formulas.p61 <-> cmCase1recog.c6I62;
a6formulas.p62 <-> cmCase1recog.c6I63;
a6formulas.p63 <-> cmCase1recog.c6I64;
a6formulas.p64 <-> cmCase1recog.c6I71;
a6formulas.p65 <-> cmCase1recog.c6I81;
a6formulas.p66 <-> cmCase1and.c6I73I94;
a6formulas.p67 <-> cmCase1and.c6I82I33;
a6formulas.p68 <-> cmCase1andandnot.c6I21I12I62;
a6formulas.p69 <-> cmCase1andandnot.c6I21I32I62;
a6formulas.p610 <-> cmCase1andandnot.c6I21I72I62;
a6formulas.p611 <-> cmCase1andandnot.c6I21I82I62;
a6formulas.p612 <-> cmCase1andandnot.c6I21I92I62;
a6formulas.p613 <-> cmCase1andandnot.c6I31I12I62;
a6formulas.p614 <-> cmCase1andandnot.c6I31I22I62;
a6formulas.p615 <-> cmCase1andandnot.c6I31I72I62;
a6formulas.p616 <-> cmCase1andandnot.c6I31I82I62;
a6formulas.p617 <-> cmCase1andandnot.c6I31I92I62;

a7formulas.p70 <-> cmCase1recog.c7I71;
a7formulas.p71 <-> cmCase1recog.c7I72;
a7formulas.p72 <-> cmCase1recog.c7I73;
```

```
a7formulas.p73 <-> cmCase1recog.c7I74;
a7formulas.p74 <-> cmCase1and.c7I41I22;
a7formulas.p75 <-> cmCase1and.c7I62I93;
a7formulas.p76 <-> cmCase1and.c7I13I84;
a7formulas.p77 <-> cmCase1and.c7I91I82;
a7formulas.p78 <-> cmCase1andandnot.c7I22I63I41;
a7formulas.p79 <-> cmCase1andandnot.c7I22I93I41;
a7formulas.p710 <-> cmCase1andandnot.c7I22I43I41;
a7formulas.p711 <-> cmCase1andandnot.c7I82I93I91;
a7formulas.p712 <-> cmCase1andandnot.c7I82I43I91;
a7formulas.p713 <-> cmCase1andandnot.c7I82I13I91;

a8formulas.p80 <-> cmCase1recog.c8I81;
a8formulas.p81 <-> cmCase1recog.c8I82;
a8formulas.p82 <-> cmCase1recog.c8I83;
a8formulas.p83 <-> cmCase1recog.c8I84;
a8formulas.p84 <-> cmCase1recog.c8I11;
a8formulas.p85 <-> cmCase1recog.c8I41;
a8formulas.p86 <-> cmCase1and.c8I13I74;
a8formulas.p87 <-> cmCase1and.c8I42I93;
a8formulas.p88 <-> cmCase1andandnot.c8I91I72I82;
a8formulas.p89 <-> cmCase1andandnot.c8I91I62I82;
a8formulas.p810 <-> cmCase1andandnot.c8I91I42I82;
a8formulas.p811 <-> cmCase1andandnot.c8I91I32I82;
a8formulas.p812 <-> cmCase1andandnot.c8I91I12I82;
a8formulas.p813 <-> cmCase1andandnot.c8I61I92I82;
a8formulas.p814 <-> cmCase1andandnot.c8I61I72I82;
a8formulas.p815 <-> cmCase1andandnot.c8I61I42I82;
a8formulas.p816 <-> cmCase1andandnot.c8I61I32I82;
a8formulas.p817 <-> cmCase1andandnot.c8I61I12I82;

a9formulas.p90 <-> cmCase1recog.c9I91;
a9formulas.p91 <-> cmCase1recog.c9I92;
a9formulas.p92 <-> cmCase1recog.c9I93;
a9formulas.p93 <-> cmCase1recog.c9I94;
a9formulas.p94 <-> cmCase1and.c9I73I64;
a9formulas.p95 <-> cmCase1and.c9I22I33;
a9formulas.p96 <-> cmCase1and.c9I31I62;
a9formulas.p97 <-> cmCase1and.c9I81I42;
a9formulas.p98 <-> cmCase1andandnot.c9I62I73I31;
a9formulas.p99 <-> cmCase1andandnot.c9I62I83I31;
```

```
a9formulas.p910 <-> cmCase1andandnot.c9I62I33I31;
a9formulas.p911 <-> cmCase1andandnot.c9I42I83I81;
a9formulas.p912 <-> cmCase1andandnot.c9I42I33I81;
a9formulas.p913 <-> cmCase1andandnot.c9I42I23I81;

/* literal maps */

i1.gameA1 <-> en1I61;
i1.gameA2 <-> en1I61;
i1.gameA3 <-> en1I61;
i1.gameA4 <-> en1I61;
i1.gameA5 <-> en1I61;
i1.gameA6 <-> en1I91;
i1.gameA7 <-> en1I91;
i1.gameA8 <-> en1I91;
i1.gameA9 <-> en1I91;
i1.gameA10 <-> en1I91;
i1.gameA11 <-> en1I61;
i1.gameA12 <-> en1I61;
i1.gameA13 <-> en1I61;
i1.gameA14 <-> en1I61;
i1.gameA15 <-> en1I91;
i1.gameA16 <-> en1I91;
i1.gameA17 <-> en1I91;
i1.gameA18 <-> en1I91;
i1.gameA19 <-> en1I91;

i1.gameB1 <-> en1I81;
i1.gameB2 <-> en1I81;
i1.gameB3 <-> en1I81;
i1.gameB4 <-> en1I81;
i1.gameB5 <-> en1I81;
i1.gameB6 <-> en1I71;
i1.gameB7 <-> en1I71;
i1.gameB8 <-> en1I71;
i1.gameB9 <-> en1I71;
i1.gameB10 <-> en1I71;
i1.gameB11 <-> en1I81;
i1.gameB12 <-> en1I81;
i1.gameB13 <-> en1I81;
i1.gameB14 <-> en1I81;
```

```
i1.gameB15 <-> en1I71;
i1.gameB16 <-> en1I71;
i1.gameB17 <-> en1I71;
i1.gameB18 <-> en1I71;
i1.gameB19 <-> en1I71;

i1.gameC1 <-> en1I11;
i1.gameC2 <-> en1I11;
i1.gameC3 <-> en1I11;
i1.gameC4 <-> en1I11;
i1.gameC5 <-> en1I11;
i1.gameC6 <-> en1I41;
i1.gameC7 <-> en1I41;
i1.gameC8 <-> en1I41;
i1.gameC9 <-> en1I41;
i1.gameC10 <-> en1I41;
i1.gameC11 <-> en1I11;
i1.gameC12 <-> en1I11;
i1.gameC13 <-> en1I11;
i1.gameC14 <-> en1I41;
i1.gameC15 <-> en1I41;
i1.gameC16 <-> en1I41;
i1.gameC17 <-> en1I41;
i1.gameC18 <-> en1I11;
i1.gameC19 <-> en1I11;

i1.gameD1 <-> en1I21;
i1.gameD2 <-> en1I21;
i1.gameD3 <-> en1I21;
i1.gameD4 <-> en1I21;
i1.gameD5 <-> en1I21;
i1.gameD6 <-> en1I31;
i1.gameD7 <-> en1I31;
i1.gameD8 <-> en1I31;
i1.gameD9 <-> en1I31;
i1.gameD10 <-> en1I31;
i1.gameD11 <-> en1I21;
i1.gameD12 <-> en1I21;
i1.gameD13 <-> en1I21;
i1.gameD14 <-> en1I21;
i1.gameD15 <-> en1I31;
```

```
i1.gameD16 <-> en1I31;
i1.gameD17 <-> en1I31;
i1.gameD18 <-> en1I31;
i1.gameD19 <-> en1I31;

i2.gameA1 <-> en2I12;
i2.gameA2 <-> en2I32;
i2.gameA3 <-> en2I42;
i2.gameA4 <-> en2I72;
i2.gameA5 <-> en2I92;
i2.gameA6 <-> en2I12;
i2.gameA7 <-> en2I32;
i2.gameA8 <-> en2I42;
i2.gameA9 <-> en2I62;
i2.gameA10 <-> en2I72;
i2.gameA11 <-> en2I82;
i2.gameA12 <-> en2I82;
i2.gameA13 <-> en2I82;
i2.gameA14 <-> en2I82;
i2.gameA15 <-> en2I82;
i2.gameA16 <-> en2I82;
i2.gameA17 <-> en2I82;
i2.gameA18 <-> en2I82;
i2.gameA19 <-> en2I82;

i2.gameB1 <-> en2I12;
i2.gameB2 <-> en2I22;
i2.gameB3 <-> en2I32;
i2.gameB4 <-> en2I72;
i2.gameB5 <-> en2I92;
i2.gameB6 <-> en2I12;
i2.gameB7 <-> en2I22;
i2.gameB8 <-> en2I32;
i2.gameB9 <-> en2I82;
i2.gameB10 <-> en2I92;
i2.gameB11 <-> en2I42;
i2.gameB12 <-> en2I42;
i2.gameB13 <-> en2I42;
i2.gameB14 <-> en2I42;
i2.gameB15 <-> en2I42;
i2.gameB16 <-> en2I42;
```

```
i2.gameB17 <-> en2I42;
i2.gameB18 <-> en2I42;
i2.gameB19 <-> en2I42;

i2.gameC1 <-> en2I32;
i2.gameC2 <-> en2I42;
i2.gameC3 <-> en2I62;
i2.gameC4 <-> en2I72;
i2.gameC5 <-> en2I92;
i2.gameC6 <-> en2I12;
i2.gameC7 <-> en2I32;
i2.gameC8 <-> en2I62;
i2.gameC9 <-> en2I72;
i2.gameC10 <-> en2I92;
i2.gameC11 <-> en2I22;
i2.gameC12 <-> en2I22;
i2.gameC13 <-> en2I22;
i2.gameC14 <-> en2I22;
i2.gameC15 <-> en2I22;
i2.gameC16 <-> en2I22;
i2.gameC17 <-> en2I22;
i2.gameC18 <-> en2I22;
i2.gameC19 <-> en2I22;

i2.gameD1 <-> en2I12;
i2.gameD2 <-> en2I32;
i2.gameD3 <-> en2I72;
i2.gameD4 <-> en2I82;
i2.gameD5 <-> en2I92;
i2.gameD6 <-> en2I12;
i2.gameD7 <-> en2I22;
i2.gameD8 <-> en2I72;
i2.gameD9 <-> en2I82;
i2.gameD10 <-> en2I92;
i2.gameD11 <-> en2I62;
i2.gameD12 <-> en2I62;
i2.gameD13 <-> en2I62;
i2.gameD14 <-> en2I62;
i2.gameD15 <-> en2I62;
i2.gameD16 <-> en2I62;
i2.gameD17 <-> en2I62;
```

```
i2.gameD18 <-> en2I62;
i2.gameD19 <-> en2I62;

i3.gameA11 <-> en3I13;
i3.gameA12 <-> en3I43;
i3.gameA13 <-> en3I73;
i3.gameA14 <-> en3I93;
i3.gameA15 <-> en3I13;
i3.gameA16 <-> en3I43;
i3.gameA17 <-> en3I63;
i3.gameA18 <-> en3I33;
i3.gameA19 <-> en3I33;

i3.gameB11 <-> en3I23;
i3.gameB12 <-> en3I33;
i3.gameB13 <-> en3I73;
i3.gameB14 <-> en3I13;
i3.gameB15 <-> en3I23;
i3.gameB16 <-> en3I33;
i3.gameB17 <-> en3I83;
i3.gameB18 <-> en3I93;
i3.gameB19 <-> en3I93;

i3.gameC11 <-> en3I43;
i3.gameC12 <-> en3I63;
i3.gameC13 <-> en3I93;
i3.gameC14 <-> en3I13;
i3.gameC15 <-> en3I33;
i3.gameC16 <-> en3I63;
i3.gameC17 <-> en3I93;
i3.gameC18 <-> en3I73;
i3.gameC19 <-> en3I73;

i3.gameD11 <-> en3I33;
i3.gameD12 <-> en3I73;
i3.gameD13 <-> en3I83;
i3.gameD14 <-> en3I93;
i3.gameD15 <-> en3I23;
i3.gameD16 <-> en3I73;
i3.gameD17 <-> en3I83;
i3.gameD18 <-> en3I13;
```

```
i3.gameD19 <-> en3I13;

i4.gameA18 <-> en4I14;
i4.gameA19 <-> en4I44;

i4.gameB18 <-> en4I24;
i4.gameB19 <-> en4I34;

i4.gameC18 <-> en4I64;
i4.gameC19 <-> en4I94;

i4.gameD18 <-> en4I74;
i4.gameD19 <-> en4I84;

/* conclusion maps */

o1.gameA1.conclusion6 <-> emCase1recog.e6I61;
o1.gameA1.conclusion2 <-> emCase1recog.e2I61;
o2.gameA1.conclusion1 <-> emCase1recog.e1I12;
o2.gameA1.conclusion8 <-> emCase1andandnot.e8I61I12I82;

o1.gameA2.conclusion6 <-> emCase1recog.e6I61;
o1.gameA2.conclusion2 <-> emCase1recog.e2I61;
o2.gameA2.conclusion3 <-> emCase1recog.e3I32;
o2.gameA2.conclusion8 <-> emCase1andandnot.e8I61I32I82;

o1.gameA3.conclusion6 <-> emCase1recog.e6I61;
o1.gameA3.conclusion2 <-> emCase1recog.e2I61;
o2.gameA3.conclusion4 <-> emCase1recog.e4I42;
o2.gameA3.conclusion8 <-> emCase1andandnot.e8I61I42I82;

o1.gameA4.conclusion6 <-> emCase1recog.e6I61;
o1.gameA4.conclusion2 <-> emCase1recog.e2I61;
o2.gameA4.conclusion7 <-> emCase1recog.e7I72;
o2.gameA4.conclusion8 <-> emCase1andandnot.e8I61I72I82;

o1.gameA5.conclusion6 <-> emCase1recog.e6I61;
o1.gameA5.conclusion2 <-> emCase1recog.e2I61;
o2.gameA5.conclusion9 <-> emCase1recog.e9I92;
o2.gameA5.conclusion8 <-> emCase1andandnot.e8I61I92I82;
```

```
o1.gameA6.conclusion9 <-> emCase1recog.e9I91;
o1.gameA6.conclusion2 <-> emCase1recog.e2I91;
o2.gameA6.conclusion1 <-> emCase1recog.e1I12;
o2.gameA6.conclusion8 <-> emCase1andandnot.e8I61I12I82;

o1.gameA7.conclusion9 <-> emCase1recog.e9I91;
o1.gameA7.conclusion2 <-> emCase1recog.e2I91;
o2.gameA7.conclusion3 <-> emCase1recog.e3I32;
o2.gameA7.conclusion8 <-> emCase1andandnot.e8I91I32I82;

o1.gameA8.conclusion9 <-> emCase1recog.e9I91;
o1.gameA8.conclusion2 <-> emCase1recog.e2I91;
o2.gameA8.conclusion4 <-> emCase1recog.e4I42;
o2.gameA8.conclusion8 <-> emCase1andandnot.e8I91I42I82;

o1.gameA9.conclusion9 <-> emCase1recog.e9I91;
o1.gameA9.conclusion2 <-> emCase1recog.e2I91;
o2.gameA9.conclusion6 <-> emCase1recog.e6I62;
o2.gameA9.conclusion8 <-> emCase1andandnot.e8I91I62I82;

o1.gameA10.conclusion9 <-> emCase1recog.e9I91;
o1.gameA10.conclusion2 <-> emCase1recog.e2I91;
o2.gameA10.conclusion7 <-> emCase1recog.e7I72;
o2.gameA10.conclusion8 <-> emCase1andandnot.e8I91I72I82;

o1.gameA11.conclusion6 <-> emCase1recog.e6I61;
o1.gameA11.conclusion2 <-> emCase1recog.e2I61;
o2.gameA11.conclusion2 <-> emCase1recog.e8I82;
o2.gameA11.conclusion3 <-> emCase1and.e3I61I82;
o3.gameA11.conclusion1 <-> emCase1recog.e1I13;
o3.gameA11.conclusion7 <-> emCase1andandnot.e7I82I13I91;

o1.gameA12.conclusion6 <-> emCase1recog.e6I61;
o1.gameA12.conclusion2 <-> emCase1recog.e2I61;
o2.gameA12.conclusion2 <-> emCase1recog.e8I82;
o2.gameA12.conclusion3 <-> emCase1and.e3I61I82;
o3.gameA12.conclusion4 <-> emCase1recog.e4I43;
o3.gameA12.conclusion7 <-> emCase1andandnot.e7I82I43I91;

o1.gameA13.conclusion6 <-> emCase1recog.e6I61;
o1.gameA13.conclusion2 <-> emCase1recog.e2I61;
```

```
o2.gameA13.conclusion2 <-> emCase1recog.e2I82;
o2.gameA13.conclusion3 <-> emCase1and.e3I61I82;
o3.gameA13.conclusion7 <-> emCase1recog.e7I73;
o3.gameA13.conclusion1 <-> emCase1and.e1I82I73;

o1.gameA14.conclusion6 <-> emCase1recog.e6I61;
o1.gameA14.conclusion2 <-> emCase1recog.e2I61;
o2.gameA14.conclusion8 <-> emCase1recog.e8I82;
o2.gameA14.conclusion3 <-> emCase1and.e3I61I82;
o3.gameA14.conclusion9 <-> emCase1recog.e9I93;
o3.gameA14.conclusion7 <-> emCase1andandnot.e7I82I93I91;

o1.gameA15.conclusion6 <-> emCase1recog.e6I61;
o1.gameA15.conclusion2 <-> emCase1recog.e2I61;
o2.gameA15.conclusion2 <-> emCase1recog.e2I82;
o2.gameA15.conclusion7 <-> emCase1and.e7I91I82;
o3.gameA15.conclusion1 <-> emCase1recog.e1I13;
o3.gameA15.conclusion3 <-> emCase1andandnot.e3I82I13I61;

o1.gameA16.conclusion9 <-> emCase1recog.e9I91;
o1.gameA16.conclusion2 <-> emCase1recog.e2I91;
o2.gameA16.conclusion2 <-> emCase1recog.e2I82;
o2.gameA16.conclusion7 <-> emCase1and.e7I91I82;
o3.gameA16.conclusion4 <-> emCase1recog.e4I43;
o3.gameA16.conclusion3 <-> emCase1andandnot.e3I82I43I61;

o1.gameA17.conclusion9 <-> emCase1recog.e9I91;
o1.gameA17.conclusion2 <-> emCase1recog.e2I91;
o2.gameA17.conclusion2 <-> emCase1recog.e2I82;
o2.gameA17.conclusion7 <-> emCase1and.e7I91I82;
o3.gameA17.conclusion6 <-> emCase1recog.e6I63;
o3.gameA17.conclusion3 <-> emCase1andandnot.e3I82I63I61;

o1.gameA18.conclusion9 <-> emCase1recog.e9I91;
o1.gameA18.conclusion2 <-> emCase1recog.e2I91;
o2.gameA18.conclusion2 <-> emCase1recog.e2I82;
o2.gameA18.conclusion7 <-> emCase1and.e7I91I82;
o3.gameA18.conclusion3 <-> emCase1recog.e3I33;
o3.gameA18.conclusion6 <-> emCase1and.e6I82I33;
o4.gameA18.conclusion1 <-> emCase1recog.e1I14;
o4.gameA18.conclusion4 <-> emCase1and.e4I33I14;
```

```
o1.gameA19.conclusion9 <-> emCase1recog.e9I91;
o1.gameA19.conclusion2 <-> emCase1recog.e2I91;
o2.gameA19.conclusion2 <-> emCase1recog.e8I82;
o2.gameA19.conclusion7 <-> emCase1and.e7I91I82;
o3.gameA19.conclusion3 <-> emCase1recog.e3I33;
o3.gameA19.conclusion6 <-> emCase1and.e6I82I33;
o4.gameA19.conclusion4 <-> emCase1recog.e4I44;
o4.gameA19.conclusion1 <-> emCase1and.e1I33I44;

o1.gameB1.conclusion8 <-> emCase1recog.e8I81;
o1.gameB1.conclusion6 <-> emCase1recog.e6I81;
o2.gameB1.conclusion1 <-> emCase1recog.e1I12;
o2.gameB1.conclusion4 <-> emCase1andandnot.e4I81I12I42;

o1.gameB2.conclusion8 <-> emCase1recog.e8I81;
o1.gameB2.conclusion6 <-> emCase1recog.e6I81;
o2.gameB2.conclusion2 <-> emCase1recog.e2I22;
o2.gameB2.conclusion4 <-> emCase1andandnot.e4I81I22I42;

o1.gameB3.conclusion8 <-> emCase1recog.e8I81;
o1.gameB3.conclusion6 <-> emCase1recog.e6I81;
o2.gameB3.conclusion3 <-> emCase1recog.e3I32;
o2.gameB3.conclusion4 <-> emCase1andandnot.e4I81I32I42;

o1.gameB4.conclusion8 <-> emCase1recog.e8I81;
o1.gameB4.conclusion6 <-> emCase1recog.e6I81;
o2.gameB4.conclusion7 <-> emCase1recog.e7I72;
o2.gameB4.conclusion4 <-> emCase1andandnot.e4I81I72I42;

o1.gameB5.conclusion8 <-> emCase1recog.e8I81;
o1.gameB5.conclusion6 <-> emCase1recog.e6I81;
o2.gameB5.conclusion9 <-> emCase1recog.e9I92;
o2.gameB5.conclusion4 <-> emCase1andandnot.e4I81I92I42;

o1.gameB6.conclusion7 <-> emCase1recog.e7I71;
o1.gameB6.conclusion6 <-> emCase1recog.e6I71;
o2.gameB6.conclusion1 <-> emCase1recog.e1I12;
o2.gameB6.conclusion4 <-> emCase1andandnot.e4I71I12I42;

o1.gameB7.conclusion7 <-> emCase1recog.e7I71;
```

```
o1.gameB7.conclusion6 <-> emCase1recog.e6I71;
o2.gameB7.conclusion2 <-> emCase1recog.e2I22;
o2.gameB7.conclusion4 <-> emCase1andandnot.e4I71I22I42;

o1.gameB8.conclusion7 <-> emCase1recog.e7I71;
o1.gameB8.conclusion6 <-> emCase1recog.e6I71;
o2.gameB8.conclusion3 <-> emCase1recog.e3I32;
o2.gameB8.conclusion4 <-> emCase1andandnot.e4I71I32I42;

o1.gameB9.conclusion7 <-> emCase1recog.e7I71;
o1.gameB9.conclusion6 <-> emCase1recog.e6I71;
o2.gameB9.conclusion2 <-> emCase1recog.e2I82;
o2.gameB9.conclusion4 <-> emCase1andandnot.e4I71I82I42;

o1.gameB10.conclusion7 <-> emCase1recog.e7I71;
o1.gameB10.conclusion6 <-> emCase1recog.e6I71;
o2.gameB10.conclusion9 <-> emCase1recog.emCase1recog.e9I92;
o2.gameB10.conclusion4 <-> emCase1andandnot.e4I71I92I42;

o1.gameB11.conclusion8 <-> emCase1recog.e8I81;
o1.gameB11.conclusion6 <-> emCase1recog.e6I81;
o2.gameB11.conclusion4 <-> emCase1recog.e4I42;
o2.gameB11.conclusion9 <-> emCase1and.e9I81I42;
o3.gameB11.conclusion2 <-> emCase1recog.e2I23;
o3.gameB11.conclusion1 <-> emCase1andandnot.e1I42I23I71;

o1.gameB12.conclusion8 <-> emCase1recog.e8I81;
o1.gameB12.conclusion6 <-> emCase1recog.e6I81;
o2.gameB12.conclusion4 <-> emCase1recog.e4I42;
o2.gameB12.conclusion9 <-> emCase1and.e9I81I42;
o3.gameB12.conclusion3 <-> emCase1recog.e3I33;
o3.gameB12.conclusion1 <-> emCase1andandnot.e1I42I33I71;

o1.gameB13.conclusion8 <-> emCase1recog.e8I81;
o1.gameB13.conclusion6 <-> emCase1recog.e6I81;
o2.gameB13.conclusion4 <-> emCase1recog.e4I42;
o2.gameB13.conclusion9 <-> emCase1and.e9I81I42;
o3.gameB13.conclusion7 <-> emCase1recog.e7I73;
o3.gameB13.conclusion1 <-> emCase1andandnot.e1I42I73I71;

o1.gameB14.conclusion8 <-> emCase1recog.e8I81;
```

```
o1.gameB14.conclusion6 <-> emCase1recog.e6I81;
o2.gameB14.conclusion4 <-> emCase1recog.e4I42;
o2.gameB14.conclusion9 <-> emCase1and.e9I81I42;
o3.gameB14.conclusion1 <-> emCase1recog.e1I13;
o3.gameB14.conclusion3 <-> emCase1and.e3I42I13;

o1.gameB15.conclusion7 <-> emCase1recog.e7I71;
o1.gameB15.conclusion6 <-> emCase1recog.e6I71;
o2.gameB15.conclusion4 <-> emCase1recog.e4I42;
o2.gameB15.conclusion1 <-> emCase1and.e1I71I42;
o3.gameB15.conclusion2 <-> emCase1recog.e2I23;
o3.gameB15.conclusion9 <-> emCase1andandnot.e9I42I23I81;

o1.gameB16.conclusion7 <-> emCase1recog.e7I71;
o1.gameB16.conclusion6 <-> emCase1recog.e6I71;
o2.gameB16.conclusion4 <-> emCase1recog.e4I42;
o2.gameB16.conclusion1 <-> emCase1and.e1I71I42;
o3.gameB16.conclusion3 <-> emCase1recog.e3I33;
o3.gameB16.conclusion9 <-> emCase1andandnot.e9I42I33I91;

o1.gameB17.conclusion7 <-> emCase1recog.e7I71;
o1.gameB17.conclusion6 <-> emCase1recog.e6I71;
o2.gameB17.conclusion4 <-> emCase1recog.e4I42;
o2.gameB17.conclusion1 <-> emCase1and.e1I71I42;
o3.gameB17.conclusion3 <-> emCase1recog.e3I83;
o3.gameB17.conclusion9 <-> emCase1andandnot.e9I42I83I81;

o1.gameB18.conclusion7 <-> emCase1recog.e7I71;
o1.gameB18.conclusion6 <-> emCase1recog.e6I71;
o2.gameB18.conclusion4 <-> emCase1recog.e4I42;
o2.gameB18.conclusion1 <-> emCase1and.e1I71I42;
o3.gameB18.conclusion9 <-> emCase1recog.e9I93;
o3.gameB18.conclusion8 <-> emCase1and.e8I42I93;
o4.gameB18.conclusion2 <-> emCase1recog.e2I24;
o4.gameB18.conclusion3 <-> emCase1and.e3I93I24;

o1.gameB19.conclusion7 <-> emCase1recog.e7I71;
o1.gameB19.conclusion6 <-> emCase1recog.e6I71;
o2.gameB19.conclusion4 <-> emCase1recog.e4I42;
o2.gameB19.conclusion1 <-> emCase1and.e1I71I42;
o3.gameB19.conclusion9 <-> emCase1recog.e9I93;
```

```
o3.gameB19.conclusion8 <-> emCase1and.e8I42I93;
o4.gameB19.conclusion3 <-> emCase1recog.e3I34;
o4.gameB19.conclusion2 <-> emCase1and.e2I93I34;

o1.gameC1.conclusion1 <-> emCase1recog.e1I11;
o1.gameC1.conclusion8 <-> emCase1recog.e8I11;
o2.gameC1.conclusion3 <-> emCase1recog.e3I32;
o2.gameC1.conclusion2 <-> emCase1andandnot.e2I11I32I22;

o1.gameC2.conclusion1 <-> emCase1recog.e1I11;
o1.gameC2.conclusion8 <-> emCase1recog.e8I11;
o2.gameC2.conclusion4 <-> emCase1recog.e4I42;
o2.gameC2.conclusion2 <-> emCase1andandnot.e2I11I42I22;

o1.gameC3.conclusion1 <-> emCase1recog.e1I11;
o1.gameC3.conclusion8 <-> emCase1recog.e8I11;
o2.gameC3.conclusion6 <-> emCase1recog.e6I62;
o2.gameC3.conclusion2 <-> emCase1andandnot.e2I11I62I22;

o1.gameC4.conclusion1 <-> emCase1recog.e1I11;
o1.gameC4.conclusion8 <-> emCase1recog.e8I11;
o2.gameC4.conclusion7 <-> emCase1recog.e7I72;
o2.gameC4.conclusion2 <-> emCase1andandnot.e2I11I72I22;

o1.gameC5.conclusion1 <-> emCase1recog.e1I11;
o1.gameC5.conclusion8 <-> emCase1recog.e8I11;
o2.gameC5.conclusion9 <-> emCase1recog.e9I92;
o2.gameC5.conclusion2 <-> emCase1andandnot.e2I11I92I22;

o1.gameC6.conclusion4 <-> emCase1recog.e4I41;
o1.gameC6.conclusion8 <-> emCase1recog.e8I41;
o2.gameC6.conclusion1 <-> emCase1recog.e1I12;
o2.gameC6.conclusion2 <-> emCase1andandnot.e2I41I12I22;

o1.gameC7.conclusion4 <-> emCase1recog.e4I41;
o1.gameC7.conclusion8 <-> emCase1recog.e8I41;
o2.gameC7.conclusion3 <-> emCase1recog.e3I32;
o2.gameC7.conclusion2 <-> emCase1andandnot.e2I41I32I22;

o1.gameC8.conclusion4 <-> emCase1recog.e4I41;
o1.gameC8.conclusion8 <-> emCase1recog.e8I41;
```

```
o2.gameC8.conclusion6 <-> emCase1recog.e6I62;
o2.gameC8.conclusion2 <-> emCase1andandnot.e2I41I62I22;

o1.gameC9.conclusion4 <-> emCase1recog.e4I41;
o1.gameC9.conclusion8 <-> emCase1recog.e8I41;
o2.gameC9.conclusion7 <-> emCase1recog.e7I72;
o2.gameC9.conclusion2 <-> emCase1andandnot.e2I41I72I22;

o1.gameC10.conclusion4 <-> emCase1recog.e4I41;
o1.gameC10.conclusion8 <-> emCase1recog.e8I41;
o2.gameC10.conclusion9 <-> emCase1recog.e9I92;
o2.gameC10.conclusion2 <-> emCase1andandnot.e2I41I92I22;

o1.gameC11.conclusion1 <-> emCase1recog.e1I11;
o1.gameC11.conclusion8 <-> emCase1recog.e8I11;
o2.gameC11.conclusion2 <-> emCase1recog.e2I22;
o2.gameC11.conclusion3 <-> emCase1and.e3I11I22;
o3.gameC11.conclusion4 <-> emCase1recog.e4I43;
o3.gameC11.conclusion7 <-> emCase1andandnot.e7I22I43I41;

o1.gameC12.conclusion1 <-> emCase1recog.e1I11;
o1.gameC12.conclusion8 <-> emCase1recog.e8I11;
o2.gameC12.conclusion2 <-> emCase1recog.e2I22;
o2.gameC12.conclusion3 <-> emCase1and.e3I11I22;
o3.gameC12.conclusion6 <-> emCase1recog.e6I63;
o3.gameC12.conclusion7 <-> emCase1andandnot.e7I22I63I41;

o1.gameC13.conclusion1 <-> emCase1recog.e1I11;
o1.gameC13.conclusion8 <-> emCase1recog.e8I11;
o2.gameC13.conclusion2 <-> emCase1recog.e2I22;
o2.gameC13.conclusion3 <-> emCase1and.e3I11I22;
o3.gameC13.conclusion9 <-> emCase1recog.e9I93;
o3.gameC13.conclusion7 <-> emCase1andandnot.e7I22I93I41;

o1.gameC14.conclusion4 <-> emCase1recog.e4I41;
o1.gameC14.conclusion8 <-> emCase1recog.e8I41;
o2.gameC14.conclusion2 <-> emCase1recog.e2I22;
o2.gameC14.conclusion7 <-> emCase1and.e7I41I22;
o3.gameC14.conclusion1 <-> emCase1recog.e1I13;
o3.gameC14.conclusion3 <-> emCase1andandnot.e3I22I13I11;
```

```
o1.gameC15.conclusion4 <-> emCase1recog.e4I41;
o1.gameC15.conclusion8 <-> emCase1recog.e8I41;
o2.gameC15.conclusion2 <-> emCase1recog.e2I22;
o2.gameC15.conclusion7 <-> emCase1and.e7I41I22;
o3.gameC15.conclusion3 <-> emCase1recog.e3I33;
o3.gameC15.conclusion9 <-> emCase1and.e9I22I33;

o1.gameC16.conclusion4 <-> emCase1recog.e4I41;
o1.gameC16.conclusion8 <-> emCase1recog.e8I41;
o2.gameC16.conclusion2 <-> emCase1recog.e2I22;
o2.gameC16.conclusion7 <-> emCase1and.e7I41I22;
o3.gameC16.conclusion6 <-> emCase1recog.e6I63;
o3.gameC16.conclusion3 <-> emCase1andandnot.e3I22I63I11;

o1.gameC17.conclusion4 <-> emCase1recog.e4I41;
o1.gameC17.conclusion8 <-> emCase1recog.e8I41;
o2.gameC17.conclusion2 <-> emCase1recog.e2I22;
o2.gameC17.conclusion7 <-> emCase1and.e7I41I22;
o3.gameC17.conclusion9 <-> emCase1recog.e9I93;
o3.gameC17.conclusion3 <-> emCase1andandnot.e3I22I93I11;

o1.gameC18.conclusion1 <-> emCase1recog.e1I11;
o1.gameC18.conclusion8 <-> emCase1recog.e8I11;
o2.gameC18.conclusion2 <-> emCase1recog.e2I22;
o2.gameC18.conclusion3 <-> emCase1and.e3I11I22;
o3.gameC18.conclusion7 <-> emCase1recog.e7I73;
o3.gameC18.conclusion4 <-> emCase1and.e4I22I73;
o4.gameC18.conclusion6 <-> emCase1recog.e6I64;
o4.gameC18.conclusion9 <-> emCase1and.e9I73I64;

o1.gameC19.conclusion1 <-> emCase1recog.e1I11;
o1.gameC19.conclusion8 <-> emCase1recog.e8I11;
o2.gameC19.conclusion2 <-> emCase1recog.e2I22;
o2.gameC19.conclusion3 <-> emCase1and.e3I11I22;
o3.gameC19.conclusion7 <-> emCase1recog.e7I73;
o3.gameC19.conclusion4 <-> emCase1and.e4I22I73;
o4.gameC19.conclusion9 <-> emCase1recog.e9I94;
o4.gameC19.conclusion6 <-> emCase1and.e6I73I94;

o1.gameD1.conclusion2 <-> emCase1recog.e2I21;
o1.gameD1.conclusion4 <-> emCase1recog.e4I21;
```

```
o2.gameD1.conclusion1 <-> emCase1recog.e1I12;
o2.gameD1.conclusion6 <-> emCase1andandnot.e6I21I12I62;

o1.gameD2.conclusion2 <-> emCase1recog.e2I21;
o1.gameD2.conclusion4 <-> emCase1recog.e4I21;
o2.gameD2.conclusion3 <-> emCase1recog.e3I32;
o2.gameD2.conclusion6 <-> emCase1andandnot.e2I21I32I62;

o1.gameD3.conclusion2 <-> emCase1recog.e2I21;
o1.gameD3.conclusion4 <-> emCase1recog.e4I21;
o2.gameD3.conclusion7 <-> emCase1recog.e7I72;
o2.gameD3.conclusion6 <-> emCase1andandnot.e6I21I72I62;

o1.gameD4.conclusion2 <-> emCase1recog.e2I21;
o1.gameD4.conclusion4 <-> emCase1recog.e4I21;
o2.gameD4.conclusion2 <-> emCase1recog.e2I82;
o2.gameD4.conclusion6 <-> emCase1andandnot.e6I21I82I62;

o1.gameD5.conclusion2 <-> emCase1recog.e2I21;
o1.gameD5.conclusion4 <-> emCase1recog.e4I21;
o2.gameD5.conclusion9 <-> emCase1recog.e9I92;
o2.gameD5.conclusion6 <-> emCase1andandnot.e6I21I92I22;

o1.gameD6.conclusion3 <-> emCase1recog.e3I31;
o1.gameD6.conclusion4 <-> emCase1recog.e4I31;
o2.gameD6.conclusion1 <-> emCase1recog.e1I12;
o2.gameD6.conclusion6 <-> emCase1andandnot.e6I31I12I62;

o1.gameD7.conclusion3 <-> emCase1recog.e3I31;
o1.gameD7.conclusion4 <-> emCase1recog.e4I31;
o2.gameD7.conclusion2 <-> emCase1recog.e2I22;
o2.gameD7.conclusion6 <-> emCase1andandnot.e6I31I22I62;

o1.gameD8.conclusion3 <-> emCase1recog.e3I31;
o1.gameD8.conclusion4 <-> emCase1recog.e4I31;
o2.gameD8.conclusion7 <-> emCase1recog.e7I72;
o2.gameD8.conclusion6 <-> emCase1andandnot.e6I31I72I62;

o1.gameD9.conclusion3 <-> emCase1recog.e3I31;
o1.gameD9.conclusion4 <-> emCase1recog.e4I31;
o2.gameD9.conclusion2 <-> emCase1recog.e2I82;
```

```
o2.gameD9.conclusion6 <-> emCase1andandnot.e6I31I82I62;

o1.gameD10.conclusion3 <-> emCase1recog.e3I31;
o1.gameD10.conclusion4 <-> emCase1recog.e4I31;
o2.gameD10.conclusion9 <-> emCase1recog.e9I92;
o2.gameD10.conclusion6 <-> emCase1andandnot.e6I31I92I22;

o1.gameD11.conclusion2 <-> emCase1recog.e2I21;
o1.gameD11.conclusion4 <-> emCase1recog.e4I21;
o2.gameD11.conclusion6 <-> emCase1recog.e6I62;
o2.gameD11.conclusion1 <-> emCase1and.e1I21I62;
o3.gameD11.conclusion3 <-> emCase1recog.e3I33;
o3.gameD11.conclusion9 <-> emCase1andandnot.e9I62I33I31;

o1.gameD12.conclusion2 <-> emCase1recog.e2I21;
o1.gameD12.conclusion4 <-> emCase1recog.e4I21;
o2.gameD12.conclusion6 <-> emCase1recog.e6I62;
o2.gameD12.conclusion1 <-> emCase1and.e1I21I62;
o3.gameD12.conclusion7 <-> emCase1recog.e7I73;
o3.gameD12.conclusion9 <-> emCase1andandnot.e9I62I73I31;

o1.gameD13.conclusion2 <-> emCase1recog.e2I21;
o1.gameD13.conclusion4 <-> emCase1recog.e4I21;
o2.gameD13.conclusion6 <-> emCase1recog.e6I62;
o2.gameD13.conclusion1 <-> emCase1and.e1I21I62;
o3.gameD13.conclusion3 <-> emCase1recog.e3I83;
o3.gameD13.conclusion9 <-> emCase1andandnot.e1I62I83I31;

o1.gameD14.conclusion2 <-> emCase1recog.e2I21;
o1.gameD14.conclusion4 <-> emCase1recog.e4I21;
o2.gameD14.conclusion6 <-> emCase1recog.e6I62;
o2.gameD14.conclusion1 <-> emCase1and.e1I21I62;
o3.gameD14.conclusion9 <-> emCase1recog.e9I93;
o3.gameD14.conclusion7 <-> emCase1and.e7I62I93;

o1.gameD15.conclusion3 <-> emCase1recog.e3I31;
o1.gameD15.conclusion4 <-> emCase1recog.e4I31;
o2.gameD15.conclusion6 <-> emCase1recog.e6I62;
o2.gameD15.conclusion9 <-> emCase1and.e9I31I62;
o3.gameD15.conclusion2 <-> emCase1recog.e2I23;
o3.gameD15.conclusion1 <-> emCase1andandnot.e1I62I23I21;
```

```
o1.gameD16.conclusion3 <-> emCase1recog.e3I31;
o1.gameD16.conclusion4 <-> emCase1recog.e4I31;
o2.gameD16.conclusion6 <-> emCase1recog.e6I62;
o2.gameD16.conclusion9 <-> emCase1and.e9I31I62;
o3.gameD16.conclusion7 <-> emCase1recog.e7I73;
o3.gameD16.conclusion1 <-> emCase1andandnot.e1I62I73I21;

o1.gameD17.conclusion3 <-> emCase1recog.e3I31;
o1.gameD17.conclusion4 <-> emCase1recog.e4I31;
o2.gameD17.conclusion6 <-> emCase1recog.e6I62;
o2.gameD17.conclusion9 <-> emCase1and.e9I31I62;
o3.gameD17.conclusion3 <-> emCase1recog.e3I83;
o3.gameD17.conclusion1 <-> emCase1andandnot.e1I62I83I21;

o1.gameD18.conclusion3 <-> emCase1recog.e3I31;
o1.gameD18.conclusion4 <-> emCase1recog.e4I31;
o2.gameD18.conclusion6 <-> emCase1recog.e6I62;
o2.gameD18.conclusion9 <-> emCase1and.e9I31I62;
o3.gameD18.conclusion1 <-> emCase1recog.e1I13;
o3.gameD18.conclusion2 <-> emCase1and.e2I62I13;
o4.gameD18.conclusion7 <-> emCase1recog.e7I74;
o4.gameD18.conclusion8 <-> emCase1and.e8I13I74;

o1.gameD19.conclusion3 <-> emCase1recog.e3I31;
o1.gameD19.conclusion4 <-> emCase1recog.e4I31;
o2.gameD19.conclusion6 <-> emCase1recog.e6I62;
o2.gameD19.conclusion9 <-> emCase1and.e9I31I62;
o3.gameD19.conclusion1 <-> emCase1recog.e1I13;
o3.gameD19.conclusion2 <-> emCase1and.e2I62I13;
o4.gameD19.conclusion8 <-> emCase1recog.e8I84;
o4.gameD19.conclusion7 <-> emCase1and.e7I13I84;
```

Appendix D

Deoxyribozyme Gate Evaluation Rules

Version 1.5 secondary structure evaluation rules and associated scores for observed secondary structure are given. ISO is used to formulate simple arithmetic relations that focus on different aspects which are experimentally known to confer good performance. For the scenario of the gate alone, we check whether or not stems have adequately formed, and how much structure is present in the loops. Perfect form is a complete stem with no missing or shifted base-pairs, and a loop devoid of any internal binding. For the scenario of the gate and input, we check how completely the input has bound to the gate loop region. Perfect form is a full 15 nt binding.

In each table, the first column denotes the evaluation rule identifier, the second column denotes the check for the first ISO triple, the third column denotes the check for any subsequent triples, and the fourth column gives the score. These rules are used together with the results of NUPACK suboptimal modeling, where the entire flood of reported secondary structures is converted into ISO, and then evaluated against the rules. Each resulting score, for each reported structure, is used in determining the overall utility expectation.

Critical Structure Stem-Loop Location Evaluation Rules		
Rule	**Triple** $(i,s,o)_j$	**Score**
gate-loc-1 location within footprint	$i_j \geq StartIndex$, $i_j + 2s_j + o_j \leq StopIndex$	1.00
gate-loc-2 incorrect starting location	$i_j < StartIndex$	0.00
gate-loc-3 incorrect stopping location	$i_j + 2s_j + o_j > StopIndex$	0.00

Critical Structure Stem Evaluation Rules		
Rule	**Triple** $(i,s,o)_j$	**Score**
gate-stem-1 perfect form	$i_j = StartIndex$, $s_j > StemLength - 1$	1.00
gate-stem-2 shifted by 1 nt	$i_j = StartIndex + 1$, $s_j > StemLength - 2$	0.90
gate-stem-3 too short by 1 nt	$i_j = StartIndex$, $s_j = StemLength - 1$	0.90
gate-stem-4 any other stem pattern		0.00

Critical Structure Loop Evaluation Rules 1-7			
Rule	**Triple** $(i,s,o)_j$	**Triple(s)** $(i,s,o)_k$	**Score**
gate-loop-1 perfect form	$o_j > LoopOpening$	case $k > j$: $i_k > StopIndex$ case $k < j$: $i_k + 2s_k + o_k < i_j$	1.00
gate-loop-2 stem encroaching by 1 bp	$s_j = StemLength + 1$, $o_j = LoopOpening - 2$	case $k > j$: $i_k > StopIndex$ case $k < j$: $i_k + 2s_k + o_k < i_j$	0.80
gate-loop-3 stem encroaching by 2 bp	$s_j = StemLength + 2$, $o_j = LoopOpening - 4$	case $k > j$: $i_k > StopIndex$ case $k < j$: $i_k + 2s_k + o_k < i_j$	0.60
gate-loop-4 stem encroaching by $>$ 2 bp	$s_j > StemLength + 2$, $o_j < LoopOpening - 4$	case $k > j$: $i_k > StopIndex$ case $k < j$: $i_k + 2s_k + o_k < i_j$	0.00
gate-loop-5 1 bp intermediate	$o_j > LoopOpening$	case $k > j$: $i_j + s_j - 1 < i_k <$ $StopIndex - s_j - (2 + o_k)$, $s_k = 1$, $\lvert(i,s,o)_k\rvert = 1$ case $k < j$: $i_k + 2s_k + o_k < i_j$	0.90
gate-loop-6 2 bp intermediate	$o_j > LoopOpening$	case $k > j$: $i_j + s_j - 1 < i_k <$ $StopIndex - s_j - (4 + o_k)$, $s_k = 2$, $\lvert(i,s,o)_k\rvert = 1$ case $k < j$: $i_k + 2s_k + o_k < i_j$	0.80
gate-loop-7 3 bp intermediate	$o_j > LoopOpening$	case $k > j$: $i_j + s_j - 1 < i_k <$ $StopIndex - s_j - (6 + o_k)$, $s_k = 3$, $\lvert(i,s,o)_k\rvert = 1$ case $k < j$: $i_k + 2s_k + o_k < i_j$	0.70

Critical Structure Loop Evaluation Rules 8-12			
Rule	**Triple** $(i,s,o)_j$	**Triple(s)** $(i,s,o)_k$	**Score**
gate-loop-8 >3 bp intermediate	$o_j > LoopOpening$	case $k > j$: $i_j + s_j - 1 < i_k <$ $StopIndex - s_j - (2s_k + o_k)$, $s_k > 3$, $\|(i,s,o)_k\| = 1$ case $k < j$: $i_k + 2s_k + o_k < i_j$	0.00
gate-loop-9 stem encroaching by 1 bp and 1 bp intermediate	$s_j = StemLength + 1$, $o_j = LoopOpening - 2$	case $k > j$: $i_j + s_j - 1 < i_k <$ $StopIndex - s_j - (2 + o_k)$, $s_k = 1$, $\|(i,s,o)_k\| = 1$ case $k < j$: $i_k + 2s_k + o_k < i_j$	0.70
gate-loop-10 stem encroaching by 1 bp and 2 bp intermediate	$s_j = StemLength + 1$, $o_j = LoopOpening - 2$	case $k > j$: $i_j + s_j - 1 < i_k <$ $StopIndex - s_j - (4 + o_k)$, $s_k = 2$, $\|(i,s,o)_k\| = 1$ case $k < j$: $i_k + 2s_k + o_k < i_j$	0.50
gate-loop-11 stem encroaching by 1 bp and >2 bp intermediate	$s_j = StemLength + 1$, $o_j = LoopOpening - 2$	case $k > j$: $i_j + s_j - 1 < i_k <$ $StopIndex - s_j - (2s_k + o_k)$, $s_k > 2$, $\|(i,s,o)_k\| = 1$ case $k < j$: $i_k + 2s_k + o_k < i_j$	0.00
gate-loop-12 stem encroaching by 2 bp and 1 bp intermediate	$s_j = StemLength + 2$, $o_j = LoopOpening - 4$	case $k > j$: $i_j + s_j - 1 < i_k <$ $StopIndex - s_j - (2 + o_k)$, $s_k = 1$, $\|(i,s,o)_k\| = 1$ case $k < j$: $i_k + 2s_k + o_k < i_j$	0.50

Critical Structure Loop Evaluation Rules 13-14			
Rule	**Triple** $(i,s,o)_j$	**Triple(s)** $(i,s,o)_k$	**Score**
gate-loop-13 stem encroaching by 2 bp and >1 bp intermediate	$s_j = StemLength+2$, $o_j = LoopOpening-4$	case $k > j$: $i_j + s_j - 1 < i_k <$ $StopIndex - s_j - (2s_k + o_k)$, $s_k > 1$, $\lvert(i,s,o)_k\rvert = 1$ case $k < j$: $i_k + 2s_k + o_k < i_j$	0.00
gate-loop-14 any other loop pattern			0.00

Input Binding Location Evaluation Rules		
Rule	**Triple** $(i,s,o)_j$	**Score**
gateinput-loc-1 location within footprint	$i_j \geq StartIndex$, $i_j + 2s_j + o_j \leq StopIndex$	1.00
gateinput-loc-2 incorrect starting location	$i_j < StartIndex$	0.00
gateinput-loc-3 incorrect stopping location	$i_j + 2s_j + o_j > StopIndex$	0.00

Input Binding Formation Evaluation Rules			
Rule	**Triple** $(i,s,o)_j$	**Triple(s)** $(i,s,o)_k$ $k > j, l > k$	**Score**
gateinput-bind-1 perfect form	$i_j = StartIndex$, $s_j = StemLength$, $o_j = LoopOpening$		1.00
gateinput-bind-2 missed input binding by 1 nt at 3' end	$i_j = StartIndex$, $s_j = StemLength - 1$		0.70
gateinput-bind-3 missed input binding by 1 nt at 5' end	$i_j = StartIndex + 1$, $s_j = StemLength - 1$		0.70
gateinput-bind-4 missed input binding by 1 nt within interior	$i_j = StartIndex$	$s_j + s_k = StemLength - 1$, $o_k = LoopOpening$	0.70
gateinput-bind-5 missed input binding by 2 nt at 3' end	$i_j = StartIndex$, $s_j = StemLength - 2$		0.40
gateinput-bind-6 missed input binding by 2 nt at 5' end	$i_j = StartIndex + 2$, $s_j = StemLength - 2$		0.40
gateinput-bind-7 missed input binding by 1 nt at 3' and 5' ends	$i_j = StartIndex + 1$, $s_j = StemLength - 2$		0.40
gateinput-bind-8 missed input binding by 1 nt at 3' end and by 1 nt within interior	$i_j = StartIndex$	$s_j + s_k = StemLength - 2$, $o_k = LoopOpening + 2$	0.40
gateinput-bind-9 missed input binding by 1 nt at 5' end and by 1 nt within interior	$i_j = StartIndex + 1$	$s_j + s_k = StemLength - 2$, $o_k = LoopOpening$	0.40
gateinput-bind-10 missed input binding by 1 nt within interior twice	$i_j = StartIndex$	$s_j + s_k + s_l = StemLength - 2$, $o_1 = LoopOpening$	0.40
gateinput-bind-11 any other binding pattern			0.00

Substrate Binding Location Evaluation Rules		
Rule	**Triple** $(i,s,o)_j$	**Score**
gatesubstrate-loc-1 location within footprint	$i_j \geq StartIndex$, $i_j + 2s_j + o_j \leq StopIndex$	1.00
gatesubstrate-loc-2 incorrect starting location	$i_j < StartIndex$	0.00
gatesubstrate-loc-3 incorrect stopping location	$i_j + 2s_j + o_j > StopIndex$	0.00

Substrate Binding Formation Evaluation Rules			
Rule	**Triple** $(i,s,o)_j$	**Triple(s)** $(i,s,o)_k$**,** $k > j$	**Score**
gatesubstrate-bind-1 perfect form	$i_j = StartIndex$, $s_j = StemLength$, $o_j = LoopOpening$		1.00
gatesubstrate-bind-2 missed substrate binding by 1 nt at 3' end	$i_j = StartIndex$, $s_j = StemLength - 1$		0.40
gatesubstrate-bind-3 missed substrate binding by 1 nt at 5' end	$i_j = StartIndex + 1$, $s_j = StemLength - 1$		0.40
gatesubstrate-bind-4 missed substrate binding by 1 nt within interior	$i_j = StartIndex$	$s_j + s_k =$ $StemLength - 1$ $o_k = LoopOpening$	0.40
gatesubstrate-bind-5 any other binding pattern			0.00

References

[1] Biopython. http://biopython.org.

[2] NCBI GenBank. http://www.ncbi.nih.gov/genbank/.

[3] PostgreSQL. http://www.postgresql.org.

[4] The R Project for Statistical Computing. http://r-project.org.

[5] Ádám Halász, Vijay Kumar, Marcin Imieliński, Calin Belta, Oleg Sokolsky, Sen Pathak, and Harvey Rubin. Analysis of lactose metabolism in E. Coli using reachability analysis of hybrid systems. *IET Systems Biology*, 1(2):130–140, 2007.

[6] Leonard M. Adleman. Molecular computation of solutions to combinatorial problems. *Science*, 266(5187):1021–1024, 1994.

[7] Mirela Andronescu, Rosalia Aguirre-Hernandez, Anne Condon, and Holger Hoos. RNAsoft: a suite of RNA secondary structure prediction and design software tools. *Nucleic Acids Research*, 31(13):3414–3422, 2003.

[8] Mirela Andronescu, Danielle Dees, Laura Slaybaugh, Yinglei Zhao, Anne Condon, Barry Cohen, and Steven Skiena. Algorithms for testing that sets of DNA word design avoid unwanted secondary structure. In Masami Hagiya and Azuma Ohuchi, editors, *DNA Computing: 8th International Workshop on DNA-Based Computers*, volume 2568 of *Lecture Notes in Computer Science*, pages 182–195. Springer, 2003.

[9] Yaakov Benenson, Binyamin Gil, Uri Ben-Dor, Rivka Adar, and Ehud Shapiro. An autonomous molecular computer for logical control of gene expression. *Nature*, 429:423–429, 2004.

[10] Dennis A. Benson, Ilene Karsch-Mizrachi, David J. Lipman, James Ostell, and Eric W. Sayers. GenBank. *Nucleic Acids Research*, 39:D32–D37, 2011.

[11] Staš Bevc, Janez Konc, Jure Stojan, Milan Hodošček, Matej Penca, Matej Praprotnik, and Dušanka Janežič. ENZO: a Web Tool for Derivation and Evaluation of Kinetic Models of Enzyme Catalyzed Reactions. *PLoS ONE*, 6(7), 2011.

[12] Guy E. Blelloch and Bruce M. Maggs. *Parallel Algorithms*. The Computer Science Handbook. Chapman and Hall, 2004.

[13] Max Born and Herbert S. Green. A kinetic theory of liquids. *Nature*, 159(4034):251–254, 1947.

[14] James E. Brady and Gerald E. Humiston. *General Chemistry Principles and Structures*. John Wiley and Sons, 1975.

[15] Ronald R. Breaker. DNA enzymes. *Nature Biotechnology*, 15:427–431, 1997.

[16] Ronald R. Breaker and Gerald F. Joyce. A DNA enzyme with Mg^{2+}-dependent RNA phosphoesterase activity. *Chemistry and Biology*, 2:655–660, 1995.

[17] Ian Brierley, Simon Pennell, and Robert Gilbert. Viral RNA pseudoknots: versatile motifs in gene expression and replication. *Nature Reviews Microbiology*, 5:598–610, 2007.

[18] Carl W. Brown, III, Matthew R. Lakin, Eli K. Horwitz, M. Leigh Fanning, Hannah E. West, Darko Stefanovic, and Steven W. Graves. Signal Propogation in Multi-Layer DNAzyme Cascades Using Structured Chimeric Substrates. *Angewandte Chemie International Edition*, 53(28):7183–7187, 2014.

[19] Michael Burrows and David J. Wheeler. A block sorting lossless data compression algorithm. *Technical Report 124, Digital Equipment Corporation*, 1994.

[20] Luca Cardelli. Strand algebras for DNA computing. In Russell J. Deaton and Akira Suyama, editors, *DNA Computing and Molecular Programming*, volume 5877 of *Lecture Notes in Computer Science*, pages 12–24. Springer, 2009.

[21] Thomas R. Cech. The chemistry of self-splicing RNA and RNA enzymes. *Science*, 236(4808):1532–1539, 1987.

[22] George M. Church, Yuan Gao, and Sriram Kosuri. Next-generation digital information storage in DNA. *Science*, 28:1628, 2012.

[23] Richard F. Clippinger. A logical coding system applied to the ENIAC (Electronic Numerical Integrator and Computer). *Ballistic Research Laboratories*, Report No. 673, Project No. TB3-0007, Research and Development Division, Ordnance Department, Aberdeen Proving Ground, 1948.

[24] B. Jack Copeland. *Colossus, The Secrets of Bletchley Park's Codebreaking Computers*. Oxford University Press, Great Britain, 2006.

[25] Diana K. Darnell, Simran Kaur, Stacey Stanislaw, Jay K. Konieczka, Tatiana A. Yatskievych, and Parker B. Antin. MicroRNA expression during chick embryo development. *Developmental Dynamics*, 235(11):3156–3165, 2006.

[26] Robert M. Dirks, Justin S. Bois, Joseph M. Schaeffer, Erik Winfree, and Niles A. Pierce. Thermodynamic analysis of interacting nucleic acid strands. *SIAM Review*, 49(1):65–88, 2007.

[27] Robert M. Dirks, Milo Lin, Erik Winfree, and Niles A. Pierce. Paradigms for computational nucleic acid design. *Nucleic Acids Research*, 32(4):1392–1403, 2004.

[28] Robert M. Dirks and Niles A. Pierce. A partition function algorithm for nucleic acid secondary structure including pseudoknots. *Journal of Computational Chemistry*, 24:1664–1677, 2003.

[29] Robert M. Dirks and Niles A. Pierce. An algorithm for computing nucleic acid base-pairing probabilities including pseudoknots. *Journal of Computational Chemistry*, 25:1295–1304, 2004.

[30] Shawn M. Douglas, Hendrik Dietz, Tim Liedl, Bjorn Hogberg, Franziska Graf, and William M. Shih. Self-assembly of DNA into nanoscale three-dimensional shapes. *Nature*, 459:414–418, 2009.

[31] Shawn M. Douglas, Adam H. Marblestone, Surat Teerapittayanon, Alejandro Vasquez, George M. Church, and William M. Shih. Rapid prototyping of 3D DNA-origami shapes with caDNAno. *Nucleic Acids Research*, 37(15):5001–5006, 2009.

[32] K. Eric Drexler. *Molecular Machinery and Manufacturing With Applications to Computation*. Massachusetts Institute of Technology, 1991.

[33] Johann Elbaz, Michael Moshe, and Itamar Wilner. Coherent activation of DNA tweezers: A "SET-RESET" logic system. *Angewandte Chemie International Edition in English*, 48(21):3834–3837, 2009.

[34] M. Leigh Fanning, Joanne Macdonald, and Darko Stefanovic. Advancing the Deoxyribozyme-Based Logic Gate Design Process. In Russell J. Deaton and Akira Suyama, editors, *DNA Computing and Molecular Programming*, volume 5877 of *Lecture Notes in Computer Science*, pages 45–54. Springer, 2009.

[35] M. Leigh Fanning, Joanne Macdonald, and Darko Stefanovic. ISO: Numeric representation of nucleic acid forms. In *Proceedings of the ACM International Conference on Bioinformatics and Computational BIology, ACM-BCB*, 2011.

[36] Udo Feldkamp. CANADA: Designing nucleic acid sequences for nanobiotechnology applications. *Journal of Computational Chemistry*, 31(3):660–663, 2010.

[37] Udo Feldkamp, Hilmar Rauhe, and Wolfgang Banzhaf. Software tools for DNA sequence design. *Genetic Programming and Evolvable Machines*, 4:153–171, 2003.

[38] Daniela Fera, Nahmee Kim, Nahum Shiffeldrim, Julie Zorn, Uri Laserson, Hark Hin Gan, and Tamar Schlick. RAG: RNA-as-graphs web resource. *BMC Bioinformatics*, 5:88, 2004.

[39] Richard P. Feynman. There's Plenty of Room at the Bottom. *Journal of Microelectromechanical Systems*, 1(1):60–66, 1992.

[40] Carl A. J. M. Firth and Dennis Bray. Stochastic Simulation of Cell Cycle Regulation. In James M. Bower and Hamid Bolouri, editors, *Computational Modeling of Genetic and Biochemical Networks*, pages 263–286. MIT Press, 2001.

[41] Gary William Flake. *The Computational Beauty of Nature*. MIT Press, Cambridge, Massachusetts, 1998.

[42] Walter Fontana, Danielle A. M. Konings, Peter Stadler, and Peter Schuster. Statistics of RNA secondary structures. *Biopolymers*, 33:1389–1404, 1993.

[43] Andrei Gabrielian and Alexander Bolshoy. Sequence complexity and DNA curvature. *Computers and Chemistry*, 23(3-4):263–274, 1999.

[44] Yang Gao, Lauren K. Wolf, and Rosina M. Georgiadis. Secondary structure effects of DNA hybridization kinetics: a solution versus surface comparison. *Nucleic Acids Research*, 34(11):3370–3377, 2006.

[45] Max H. Garzon and Kiran C. Bobba. A Geometric Approach to Gibbs Energies and Optimal DNA Codeword Design. In Andrew Turberfield and Darko Stefanovic, editors, *DNA18-DNA Computing and Molecular Programming*, volume 7433 of *Lecture Notes in Computer Science*, pages 73–85. Springer, 2012.

[46] Martin Gerhardt and Heike Schuster. A Cellular Automaton Describing The Formation Of Spatially Ordered Structures In Chemical Systems. *Physica D*, 36(3):209–221, 1989.

[47] Robert Giegerich, Björn Voß, and Marc Rehmsmeier. Abstract shapes of RNA. *Nucleic Acids Research*, 32(16):4843–4851, 2004.

[48] Daniel T. Gillespie. A general method for numerically simulating the stochastic time evolution of coupled chemical reactions. *Journal of Computational Physics*, 22:403–434, 1976.

[49] Daniel T. Gillespie. The Chemical Langevin Equation. *The Journal of Chemical Physics*, 113:297–306, 2000.

[50] Daniel T. Gillespie, Andreas Hellander, and Linda R. Petzold. Perspective: Stochastic algorithms for chemical kinetics. *The Journal of Chemical Physics*, 138(170901):1–14, 2013.

[51] Nick Goldman, Paul Bertone, Siyuan Chen, Christophe Dessimoz, Emily M. LeProust, Botond Sipos, and Ewan Birney. Towards practical, high-capacity, low-maintenance information storage in synthesized DNA. *Nature*, 494:77–80, 2013.

[52] Jan Gorodkin, Ivo L. Hofacker, Elfar Torarinsson, Zizhen Yao, Jakob H. Havgaard, and Walter L. Ruzzo. *De novo* prediction of structured RNAs from genomic sequences. *Trends in Biotechnology*, 28(1):9–19, 2009.

[53] Elton Graugnard, Amber Cox, Jeunghoon Lee, Cheryl Jorcyk, Bernard Yurke, and William L. Hughes. Operation of a DNA-based autocatalytic network in serum. In *DNA Computing and Molecular Programming*, volume 6518 of *Lecture Notes in Computer Science*, pages 83–88. Springer, 2011.

[54] Hongzhou Gu, Jie Chao, Shou-Jun Xiao, and Nadrian Seeman. A proximity-based programmable DNA nanoscale assembly line. *Nature*, 465:202–205, 2010.

[55] John L. Hennessy and David R. Patterson. *Computer Architecture A Quantitative Approach*. Morgan Kauffman Publishers, Inc., 2nd edition, 1995.

[56] Desmond J. Higham. Modeling and Simulating Chemical Reactions. *University of Strathclyde Mathematics Research Report*, 2007.

[57] Mikio Hirabayashi. Kyoto cabinet. `http://fallabs.com`.

[58] Ivo L. Hofacker, Walter Fontana, Peter F. Stadler, Sebastian Bonhoeffer, Manfred Tacker, and Peter Schuster. Fast folding and comparison of RNA secondary structures (The Vienna RNA package). *Monatshefte für Chemie*, 255:279–284, 1994.

[59] Paulien Hogeweg and Burt Hesper. Energy directed folding of RNA sequences. *Nucleic Acids Research*, 12(1):67–74, 1984.

[60] Stefan Hoops, Sven Sahle, Ralph Gauges, Christine Lee, Jürgen Pahle, Natalia Simus, Mudita Singhal, Liang Xu, Pedro Mendes, and Ursula Kummer. COPASI – a COmplex PAthway SImulator. *Bioinformatics*, 22:3067–3074, 2006.

[61] Ann B. Jacobson and Michael Zuker. Structural Analysis by Energy Dot Plot of a Large mRNA. *Journal of Molecular Biology*, 233(2):261–269, 1993.

[62] Emma Y. Jin and Christian M. Reidys. Asymptotic enumeration of RNA structures with pseudoknots. *Bulletin of Mathematical Biology*, 70:951–970, 2008.

[63] Zhang Kai, Qiang Xiao Li, Zhao Dong Ming, and Xu Jin. General nucleic acid sequence design using implicit enumeration. Bio-Inspired Computing, pages 1–10, 2009.

[64] Yan Karklin, Richard F. Meraz, and Stephen R. Holbrook. Classification of non-coding RNA using graph representations of secondary structure. *Pacific Symposium on Biocomputing*, pages 4–15, 2005.

[65] Stuart A. Kauffman. Emergent properties in random complex automata. *Physica D*, 10(1-2):145–156, 1984.

[66] Mugdha Khaladkar, Vivian Bellofatto, Jason Wang, Bin Tian, and Bruce Shapiro. RADAR: a web server for RNA data analysis and research. *Nucleic Acids Research*, 35(2):W300–W304, 2007.

[67] Dmitry M. Kolpashchikov and Milan N. Stojanovic. Boolean control of aptamer binding states. *Journal of the American Chemical Society*, 127:11348–11351, 2005.

[68] Andrzej K. Konopka. Sequence complexity and composition. `http://www.els.net`, 2005.

[69] Harvey Lederman, Joanne Macdonald, Darko Stefanovic, and Milan Stojanovic. Deoxyribozyme-based three-input logic gates and construction of a molecular full adder. *Biochemistry*, 45:1194–1199, 2006.

[70] Neocles B. Leontis, Aurelie Lescoute, and Eric Westhof. The building blocks and motifs or RNA architecture. *Current Opinion in Structural Biology*, 16(xx):279–287, 2006.

[71] Vladimir I. Levenshtein. Binary codes capable of correcting deletions, insertions and reversals. *Doklady Akademii Nauk SSSR*, 163(4):845–848, 1965.

[72] Chunhua Liu, Natasha Jonoska, and Nadrian Seeman. Reciprocal DNA nanomechanical devices controlled by the same set strands. *Nano Letters*, 9(7):2641–2647, 2009.

[73] Joanne Macdonald, Yang Li, Marko Sutovic, Harvey Lederman, Kiran Pendri, Wanhong Lu, Benjamin Andrews, Darko Stefanovic, and Milan Stojanovic. Medium scale integration of molecular logic gates in an automaton. *Nano Letters*, 6(11):2598–2603, 2006.

[74] Amit Marathe, Anne E. Condon, and Robert M. Corn. On combinatorial DNA word design. *Journal of Computational Biology*, 8:201–220, 1999.

[75] Laerke B. Marcussen, Morten L. Jepson, Emil L. Kristofferson, Oskar Franch, Joanna Proszek, and Yi-Ping Ho. DNA-based sensor for real-time measurement of the enzymatic activity of human topoisomerase I. *Sensors*, 13(4):4017–4028, 2013.

[76] Adam A. Margolin and Milan N. Stojanovic. Boolean calculations made easy (for ribozymes). *Nature Biotechnology*, 23:1374–1376, 2005.

[77] Nicholas Markham and Michael Zuker. DINAMelt web server for nucleic acid melting prediction. *Nucleic Acids Research*, 33:W577–W581, 2005.

[78] William C. McBee, Amy S. Gardiner, Robert P. Edwards, Jamie L. Lesnock, Rohit Bhargava, R. Marshall Austin, Richard S. Guido, and Saleem A. Khan.

[79] Donald A. McQuarrie. Stochastic approach to chemical kinetics. *Journal of Applied Probability*, 4:413–478, 1967.

[80] Bence Mélykúti, Kevin Burrage, and Konstantinos C. Zygalakis. Fast stochastic simulation of biochemical reaction systems by alternative formulations of the chemical Langevin equation. *The Journal of Chemical Physics*, 132(164109):1–12, 2010.

[81] Bernard M. Moret. *The Theory of Computation*. The Computer Science Handbook. Addison-Wesley, 1998.

[82] Clint Morgan, Darko Stefanovic, Christopher Moore, and Milan N. Stojanovic. Building the components for a biomolecular computer. In C. Ferretti, G. Mauri, and C. Zandron, editors, *DNA Computing: 10th International Workshop on DNA-Based Computers*, volume 3384 of *Lecture Notes in Computer Science*, pages 247–257. Springer, 2005.

[83] Rajesh K. Nayak, Olve B. Peersen, Kathleen B. Hall, and Alan Van Orden. Millisecond Time-Scale Folding and Unfolding of DNA Hairpins Using Rapid-Mixing Stopped-Flow Kinetics. *Journal of the American Chemical Society*, 134:2453–2456, 2012.

[84] Jacques Ninio. Properties of nucleic acid representations I. Topology. *Biochemie*, 5:485–494, 1971.

[85] Ruth Nussinov, George Pieczenik, Jerrold R. Griggs, and Daniel J. Kleitman. Algorithms For Loop Matchings. *SIAM Journal on Applied Mathematics*, 35(1):68–82, 1978.

[86] Amy E. Pasquinelli. MicroRNAs and their targets: recognition, regulation and an emerging reciprocal relationship. *Nature Reviews Genetics*, 13(4):271–282, 2012.

[87] Giulio Pavesi, Giancarlo Mauri, Marco Stefani, and Graziano Pesole. RNAprofile: an algorithm for finding conserved secondary structure motifs in unaligned RNA sequences. *Nucleic Acids Research*, 32(10):3258–3269, 2004.

[88] Hai Pei, Na Lu, Yanli Wen, Shiping Song, Yan Liu, Hao Yan, and Chunhai Fan. A DNA nanostructure-based biomolecular probe carrier platform for electrochemical biosensing. *Advanced Materials*, 22(42):4754–4758, 2010.

[89] Renjun Pei, Elizabeth Matamoros, Manhong Liu, Darko Stefanovic, and Milan Stojanovic. Training a molecular automaton to play a game. *Nature Nanotechnology*, 5:773–777, 2010.

[90] Andrew Philips and Luca Cardelli. A programming language for composable DNA circuits. *Journal of the Royal Society Interface*, 6(4):S419–S436, 2009.

[91] Andrew Phillips, Matthew Lakin, and Loic Pauleve. Stochastic simulation of process calculi for biology. *Electronic Proceeedings in Theoretical Computer Science*, 40:1–5, 2010.

[92] Julia E. Poje, Tamara Kastratovic, Andrew R. Macdonald, Ana C. Guillermo, Steven E. Troetti, Omar J. Jabado, M. Leigh Fanning, Darko Stefanovic, and Joanne

Macdonald. Visual Displays that Directly Interface and Provide Read-Outs of Molecular States Via Molecular Graphics Processing Units. *Angewandte Chemie International Edition*, 2014.

[93] Lulu Qian and Erik Winfree. A simple DNA gate motif for synthesizing large-scale circuits. In Ashish Goel, Friedrich C. Simmel, and Petr Sosík, editors, *DNA Computing*, pages 70–89. Springer-Verlag, Berlin, Heidelberg, 2009.

[94] Lulu Qian and Erik Winfree. Scaling up digital circuit computation with DNA strand displacement cascades. *Science*, 332(6034):1196–1201, 2011.

[95] Lulu Qian and Erik Winfree. A simple DNA gate motif for synthesizing large-scale circuits. *The Journal of the Royal Society Interface*, 8(62):1281–1297, 2011.

[96] Lulu Qian, Erik Winfree, and Jehoshua Bruck. Neural network computation with DNA strand displacement cascades. *Nature*, 475:368–372, 2011.

[97] Brittany Rauzan, Elizabeth McMichael, Rachel Cave, Lesley R. Sevcik, Kara Ostrosky, Elisabeth Whitman, Rachel Stegemann, Audra L. Sinclair, Martin J. Serra, and Alice A. Deckert. Kinetics and Thermodynamics of DNA, RNA, and Hybrid Duplex Formation. *Biochemistry*, 52:765–772, 2013.

[98] Frederick Reif. *Fundamentals of Statistical and Thermal Physics*. McGraw Hill, 1965.

[99] Rae M. Robertson, Stephan Laib, and Douglas E. Smith. Diffusion of isolate DNA molecules: Dependence on length and topology. *Proceedings of the National Academy of Sciences*, 103(19):7310–7314, 2006.

[100] Alberto Rodriguez-Pulido, Aline I. Kondrachuk, Deepak K. Prusty, Jia Gao, Maria A. Loi, and Andreas Herrmann. Light-triggered sequence-specific cargo release from DNA block copolymer-lipid vesicles. *Angewandte Chemie*, 125(3):1042–1046, 2013.

[101] Robert Rosen. The Polarity Between Structure and Function. The Center for Theoretical Biology Dialogue Discussion Paper, November 22, 1971.

[102] Paul W. K. Rothemund, Nick Papadakis, and Erik Winfree. Algorithmic self-assembly of DNA sierpinski triangles. *PLoS Biology*, 2(12):2041–2053, 2004.

[103] Rick Russell, Ian S. Millett, Mark W. Tate, Lisa W. Kwok, Bradley Nakatani, Sol M. Gruner, Simon G. J. Mochrie, Vijay Pande, Sebastian Doniach, Daniel Herschlag,

and Lois Pollack. Rapid compaction during RNA folding. *Proceedings of the National Academy of Science*, 99(7):4266–4271, 2002.

[104] John SantaLucia, Jr. A unified view of polymer, dumbbell, and oligonucleotide DNA nearest-neighbor thermodynamics. *Proceedings of the National Academy of Sciences (PNAS)*, 95(4):1460–1465, 1998.

[105] Stephen W. Santoro and Gerald F. Joyce. A general purpose RNA-cleaving DNA enzyme. *Proceedings of the National Academy of Science*, 94:4262–4266, 1997.

[106] Yonatan Savir and Tsvi Tlusty. Optimal design of a molecular recognizer: Molecular recognition as a bayesian signal detection problem. *IEEE Journal of Selected Topics in Signal Processing*, 2(3):390–399, 2008.

[107] Georg Seelig, David Soloveichik, David Zhang, and Erik Winfree. Enzyme-free nucleic acid logic circuits. *Science*, 314:1585–1588, 2006.

[108] Nadrian Seeman. Nanomaterials based on DNA. *Annual Review of Biochemistry*, 79:65–87, 2010.

[109] Nadrian C. Seeman. Biomolecular stereodynamics. In R. H. Sarma, editor, *Nucleic Acid Junctions: Building Blocks for Genetic Engineering in Three Dimensions*, pages 269–277. Adenine Press, 1981.

[110] Nadrian C. Seeman. De novo design of sequences for nucleic acid structural engineering. *Journal of Biomolecular Structure and Dynamics*, 8:573–581, 1990.

[111] Nadrian C. Seeman and Neville R. Kallenbach. Design of immobile nucleic acid junctions. *Biophysical Journal*, 44:201–209, 1983.

[112] Bruce Shapiro. An algorithm for comparing multiple RNA secondary structures. *Bioinformatics*, 4(3):387–393, 1988.

[113] Bruce Shapiro and Kaizhong Zhang. Comparing multiple RNA secondary structures using tree comparisons. *Computer Applications in Biosciences*, 6(4):309–318, 1990.

[114] Bruce A. Shapiro, Yaroslava G. Yingling, Wojciech Kasprzak, and Eckart Bindewald. Bridging the gap in RNA structure prediction. *Current Opinion in Structural Biology*, 17:157–165, 2007.

[115] Adam Shea, Brian Fett, Marc D. Riedel, and Keshab Parhi. Writing and compiling code into biochemistry. *Pacific Symposium on Biocomputing*, 15:456–464, 2010.

[116] Seung Woo Shin. Compiling and verifying DNA-based chemical reaction network implementations. *M. S. Thesis, California Institute of Technology*, 2011.

[117] Scott K. Silverman. DNA as a versatile chemical component for catalysis, encoding, and stereocontrol. *Angewandte Chemie International Edition*, 49:7180–7201, 2010.

[118] David W. Staple and Sam E. Butcher. Pseudoknots: RNA structures with diverse functions. *PLoS Biology*, 3(6), 2005.

[119] Milan Stojanovic, Tiffany Elizabeth Mitchell, and Darko Stefanovic. Deoxyribozyme-based logic gates. *Journal of the American Chemical Society*, 124:3555–3561, 2002.

[120] Milan Stojanovic and Darko Stefanovic. A deoxribozyme-based molecular automaton. *Nature Biotechnology*, 21(9):1069–1074, 2003.

[121] Milan Stojanovic and Darko Stefanovic. Deoxyribozyme-based half-adder. *Journal of the American Chemical Society*, 125:6673–6676, 2003.

[122] Milan N. Stojanovic, Paloma de Prada, and Donald W. Landry. Fluorescent sensors based on aptamer self-assembly. *Journal of the American Chemical Society*, 122:11547–11548, 2000.

[123] Milan N. Stojanovic and Dmitry M. Kolpashchikov. Modular aptameric sensors. *Journal of the American Chemical Society*, 126:9266–9270, 2003.

[124] Guinevere Strack, Marcos Pita, Maryna Ornatska, and Evgeny Katz. Boolean logic gates that use enzymes as input signals. *ChemBioChem*, 9(8):1260–1266, 2008.

[125] Lubert Stryer. A fluorescence energy transfer as a spectroscopic ruler. *Annual Review Biochemistry*, 47:819–846, 1978.

[126] Oleg N. Temkin, Andrew V. Zeigarnik, and Danail Bonchev. *Chemical Reaction Networks: A Graph Theoretical Approach*. CRC Press, 1996.

[127] Gilbert Thill, J. Marc Vasseur, and N. Kyle Tanner. Structural and sequence elements required for the self-cleaving activity of the hepatitis delta virus ribozyme. *Biochemistry*, 32:4254–4262, 1993.

[128] Ram S. Verma. *Genes and Genomes*. Elsevier, 1998.

[129] Virgile Viasnoff, Amit Meller, and Herve Isambert. DNA nanomechanical switches under folding kinetics control. *Nano Letters*, 6(1):101–104, 2006.

[130] Michael Waterman and Temple F. Smith. RNA Secondary Structure: A Complete Mathematical Analysis. *Mathematical Biosciences*, pages 257–266, 1978.

[131] Thomas Way, Tao Tao, and Bryan Wagner. Compiling mechanical nanocomputer components. *Global Journal of Computer Science and Technology*, 10(2):36–42, 2010.

[132] Bryan Wei, Mingjie Dai, and Peng Yin. Complex shapes self-assembled from single-stranded DNA tiles. *Nature*, 485:623–626, 2012.

[133] Jorg R. Weimar. Cellular Automata Approaches to Enzymatic Reaction Networks. In Stefania Bandini, Bastien Chopard, and Marco Tomassini, editors, *Cellular Automata for Research and Industry (ACRI)*, volume 2493 of *Lecture Notes in Computer Science*, pages 294–303. Springer-Verlag, 2002.

[134] Shelley F. J. Wickham, Jonathan Bath, Yousuke Katsuda, Masayuki Endo, Kumi Hidaka, Hiroshi Sugiyama, and Andrew J. Turberfield. A DNA-based molecular motor that can navigate a network of tracks. *Nature Nanotechnology*, 7(3):169–173, 2012.

[135] Sarah A. Woodson. Compact intermediates in RNA folding. *Annual Reviews of Biophysics*, 39:61–77, 2010.

[136] Sarah A. Woodson. Taming free energy landscapes with RNA chaperones. *RNA Biology*, 7(6):677–686, 2010.

[137] Gang Wu, Natasha Jonoska, and Nadrian Seeman. Construction of a DNA nanoobject directly demonstrates computation. *Biosystems*, 98(2):80–84, 2009.

[138] Bernard Yurke, Andrew J. Turberfield, Allen P. Mills Jr., Friedrich C. Simmel, and Jennifer Neumann. A DNA-fuelled molecular machine made of DNA. *Nature*, 406:605–608, 2000.

[139] Joseph N. Zadeh, Conrad Steenberg, Justin S. Bois, Brian R. Wolfe, Marshall B. Pierce, Asif R. Khan, Robert M. Dirks, and Niles A. Pierce. NUPACK: analysis and design of nucleic acid systems. *Journal of Computational Chemistry*, 32:179–173, 2011.

[140] David Yu Zhang, Andrew J. Turberfield, Bernard Yurke, and Eric Winfree. Engineering entropy-driven reactions and networks catalyzed by DNA. *Science*, 318:1121–1125, 2007.

[141] Geoffrey Zubay. *Biochemistry*. Macmillan Publishing Company, 1988.

[142] Michael Zuker. Mfold web server for nucleic acid folding and hybridization prediction. *Nucleic Acids Research*, 31(13):3406–3415, 2003.

[143] Michael Zuker and David Sankoff. RNA secondary structures and their prediction. *Bulletin of Mathematical Biology*, 46(4):591–621, 1984.