MIT/LCS/TR-338

TOWARDS A PROBLEM SOLVING SYSTEM FOR MOLECULAR GENETICS

Phyllis Koton

This blank page was inserted to preserve pagination.

Towards a Problem Solving System for Molecular Genetics

by

Phyllis A. Koton

May 1985

This research was supported in part by the National Institutes of Health Grant No. 1 P01 LM 03374-04 from the National Library of Medicine, and in part by National Institutes of Health Grant No. 1 P41 RR 01096-05 from the Division of Research Resources.

Laboratory for Computer Science

Massachusetts Institute of Technology

Cambridge, Massachusetts 02139

Towards a Problem Solving System for Molecular Genetics

by

Phyllis A. Koton

ABSTRACT

This paper describes a program called GENEX that reasons about the behavior of bacterial operons. It is the first step towards a generalized system that will reason about genetic control mechanisms. The system is easily extensible and able to produce detailed explanations without relying on canned text. Problems in molecular genetics are complicated by uncertainty introduced when reasoning about conformations. GENEX can reduce the number of possible solutions that must be verified by formulating likely models from the behavior of the system it is examining. Future work is outlined that will extend the program presented here by improving the reasoning mechanisms, utilizing several AI methods.

Keywords: Expert Systems, Genetics

Acknowledgments

Prof. Peter Szolovits suggested this topic, and was exceptionally helpful and patient in the long process from idea to thesis to this report.

Dr. Allan Maxam of Harvard Medical School provided the initial impetus for this project.

Dr. Benjamin Neel of Beth Israel Hospital served as the molecular genetics expert for this project.

Prof. Ethan Signer provided the test questions for the program.

I'd like to thank the members of the Clinical Decision Making Group for providing a pleasant and supportive work environment, and I owe special thanks to Tom Russ for some very helpful suggestions, Prof. Ramesh Patil for the many discussions we had which gave me a lot of new things to think about, and Prof. Benjamin Kuipers for all-around good advice.

To my parents, who once again gave their time so that I might complete my work, I am truly thankful and indebted.

The work described in this report is based in part on my SM Thesis [Koton 83].

Contents

Chapter 1

Introduction

Humans are limited in their ability to manage complex collections of information. For example, the field of molecular genetics is advancing so rapidly that it is becoming difficult for any one researcher to assimilate all the new information being reported. Every month, over 500 new papers are published in the leading journals of the field. Artificial intelligence techniques can provide a framework for organizing and managing this information. In addition, expert systems technology allows us to model the behavior of the expert in using and manipulating this data.

These techniques could be used to build an intelligent knowledge base for the domain of molecular genetics. The knowledge base would be a record of the current knowledge in the field. It could be used by molecular biologists to interpret experimental data and to propose hypotheses for unexplained phenomena [1]. By saying that the system will "propose a hypothesis," I mean proposing some *mechanism* to explain an observation or set of observations.

The system would not be "smarter" than a molecular biologist, but it would be extremely methodical and consistent in applying the information it has. The same cannot always be said of human experts. By thorough examination of its knowledge base, the system may be able to suggest possibilities that the researcher has overlooked.

This paper describes a program called GENEX II that reasons about the behavior of bacterial operons. It is the first step towards a more generalized system that reasons about genetic control mechanisms.

GENEX II models the operon on what I term a *micro* level: it reasons about the behavior of the operon over one cycle (or at most, a few cycles) of replication, transcription, and translation. It does not do a *macro*-level simulation of the operon, i.e. it does not model the operon's behavior over an extended period of repeated cycles.

[1] Another use would be retrieving information relevant to their research.

While the ability to do macro reasoning is clearly important for some problems, it is not necessary for the types of problems that GENEX II was intended to solve.

GENEX II takes as input a description of an operon (real or imaginary) and information on whether or not the genes of that operon are expressed, then attempts to explain the observed behavior of the operon. The specificity of the result is dependent on how much information the program is given as input. If GENEX II is told only that the genes of an operon are not expressed, it will suggest every possible reason why the end product is not made, consistent with the information about the operon it has. If given the nucleotide sequence of the operon and the location and nature of any mutations, it can give a highly specific assessment of the cause of the operon's behavior. In order to reduce the number of possible explanations that the system must examine (which could number in the thousands), GENEX II first reduces the space of possible solutions by determining likely models of the operon's behavior using the observed behavior. It then constrains the explanations it considers to those that fit the likely model.

GENEX II represents an initial attempt at reasoning about genetic control systems by computer. An earlier version, GENEX I, was is described here and in (Koton, 1983). GENEX I performed well on test questions and demonstrated that such a system is feasible. However, it encoded the domain knowledge in highly-compiled rules, which limited the types of problems it could solve and made it difficult to produce explanations or extend the knowledge base. The GENEX II system has a more detailed representation of the domain knowledge and a different reasoning method which eliminates many of the problems of the previous system. Further work is outlined which will extend the program presented here by improving the reasoning mechanisms, utilizing a number of AI methods.

Chapter 2

A Brief Overview of DNA Replication and Protein Synthesis

The hereditary properties of all cells are controlled by structures known as genes, whose principal component is deoxyribonucleic acid (DNA). Each protein produced by a cell is coded for by a gene specific for that protein. A protein is simply a chain of amino acids, and a gene carries the information that determines the order of the amino acids.

The DNA consists of two nucleotide chains which form a double helix. There are four main nucleotides: adenine (A), guanine (G), cytosine (C), and thymine (T). The two chains are linked together by hydrogen bonds between the pairs of bases. Adenine is always paired with thymine and cytosine is always paired with guanine. Thus, there is a complementary relationship between the nucleotide sequences of the two chains. During DNA replication, the two strands partially unravel, and each single strand serves as the template for the formation of its complement. DNA polymerase, the enzyme which links individual nucleotides together into a linear chain, binds to the single strands and promotes this reaction.

All genetic information is carried in the sequence of the four nucleotides that make up the gene. The average gene contains about 1000 nucleotide pairs. A group of three adjacent nucleotides codes for an amino acid. The relationship between nucleotide triplets and amino acids is termed the *genetic code*. Mutations involve changes in the nucleotide sequence of the gene; this may result in a different amino acid sequence and therefore a different protein being produced.

DNA is not the direct template for amino acid synthesis. Instead, the sequence information from DNA is transferred into molecules of ribonucleic acid (RNA) by the enzyme RNA polymerase. This process is known as transcription. The relationship between DNA, RNA, and protein is expressed by the *central dogma* shown in figure 2.1, which summarizes the transfer of genetic information.

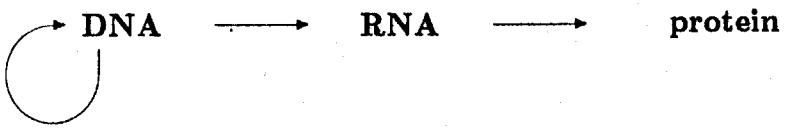

Figure 2.1: the central dogma. DNA serves as the template for its own replication. RNA molecules are made on DNA templates, and all protein sequences are made on messenger RNA (mRNA) templates.

Messenger RNA (mRNA) is a single stranded molecule of RNA whose sequence is identical to one strand of the DNA duplex from which it is made and complementary to the other [1]. The process by which mRNA directs the formation of proteins is called translation, and also involves ribosomes and transfer RNA (tRNA). In prokaryotes, [2] translation begins as soon as a segment of mRNA has been synthesized, while transcription of the rest of the gene is still underway. During translation, ribosomes, which are particles made up of ribosomal RNA (rRNA) and proteins, bind to the transcript and move across it, "reading" the nucleotide triplets. For each nucleotide triplet (called a *codon*) in the mRNA, there is a corresponding complementary triplet (an *anticodon*) present in a particular tRNA. Each tRNA is capable of binding only one specific amino acid and also is capable of binding to only one codon on the mRNA template. Protein synthesis occurs in the ribosome, where the formation of bonds between the amino acids assembled by the tRNA is catalyzed, forming the polypeptide chain.

The sequence of nucleotides or amino acids that make up a molecule is known as the primary structure of a molecule. In its native state, a protein tends to fold into a three-dimensional configuration that maximizes the number of favorable interactions between the protein and its environment. The 3-D configuration of a molecule is called its conformation. It is determined by the primary structure of the molecule, however the specific relationship between sequence and conformation is not known. Mutations, which change the primary structure of a molecule, can change the conformation of the molecule, which may result in a change in the behavior of the molecule.

---

[1] except that the nucleotide thymine in DNA is replace by the nucleotide uracil in RNA.

[2] simple one-celled organisms which do not have nuclei, e.g. bacteria. As compared to *eukaryotes*, organisms with cells that have nuclei, e.g. yeast, man.

Chapter 3

Control of Expression of Bacterial Operons

Different proteins are produced in different amounts within a given cell. Some proteins, for example those which are involved in synthesizing nucleotides, are needed in large amounts. Other proteins are needed in much smaller amounts. Cellular mechanisms exist to selectively synthesize those proteins that are needed in large amounts. Similarly, not every protein that the organism can make is needed all the time. For instance, the proteins that digest an energy source such as lactose are useless in the absence of lactose. Organisms have therefore evolved mechanisms to regulate the amount of protein made in response to differing environmental conditions.

The amount of a given protein produced by a cell can be expressed as follows:

$$\begin{aligned} amt\ of\ protein &= (amt\ of\ protein\ made) - (amt\ of\ protein\ degraded) \\ amt\ of\ protein\ made &= K \times amt\ of\ message, K > 1 \\ amt\ of\ message &= (amt\ of\ message\ made) - (amt\ of\ message\ degraded) \end{aligned}$$

Many of the mechanisms for controlling the amount of protein and message made and degraded are known, and these can be examined systematically.

Most known bacterial proteins, once made, are relatively stable. Differences in the amount of protein present are most likely related to the amount of protein made rather than the amount degraded [1]. Control of the amount of protein made can be divided into transcriptional control and translational control. The amount of protein which is degraded is governed by *post-translational control.*

[1] Watson, p. 382. Note, however, that only about 10% of the E. Coli genome has been defined, and it remains possible that mechanisms involving differential stability of proteins may play an important regulatory role for the remaining 90% of the gene products.

In bacteria, most control of gene expression is effected at the transcriptional level. Consequently, GENEX concentrates on transcriptional control.

3.1 Control of Transcription

Synthesis of the mRNA message of a gene or set of genes is regulated by signals encoded in nucleotide sequences in the DNA. Sequences called *promoters* specify start sites for the transcription of mRNA. *Terminator* sequences specify transcription stop sites. The sequences of specific promoters (and terminators) can differ among organisms, and from gene to gene in the same organism, but there are homologies among the known sequences[2]. Some sequences interact more effectively with RNA polymerase in their roles as promoter (or terminator). Thus regulation of transcription can occur by varying the efficiency with which polymerase interacts with these sites.

Another form of transcriptional control, termed *attenuation*, is exerted during the *elongation* phase of transcription. Attenuation refers to the regulation of whether or not a mRNA molecule that has been initiated is transcribed to completion. This is also dependent on the DNA sequence.

3.1.1 Control of Initiation

The initial interaction of RNA polymerase with a specific region of DNA is known as *initiation*. If all regions of DNA were equally attractive to RNA polymerase, transcription would be able to initiate anywhere along the strand, irrespective of gene boundaries. This would result in a large number of "nonsense" mRNAs[3] being produced and would be very wasteful of the cell's energy. Therefore cells have developed mechanisms to ensure that initiation only occurs at specific sites along the DNA strand.

Characteristics of the promoter

Initiation occurs at the promoter. It is thought to consist of three steps: recognition of the promoter site by RNA polymerase; "firm binding" of the polymerase to the DNA; and the actual initiation of transcription. By convention, the nucleotide at which transcription begins is numbered 1. The promoter region consists

[2]See Rosenberg and Court.

[3]i.e. those which do not code for any protein.

of approximately 40 nucleotides preceding the transcription start site (i.e. position -40).[4]

A region located about 35 nucleotides upstream of the transcription start site is thought to function as the RNA polymerase recognition site. It consists of the highly conserved sequence TTG followed by the slightly less conserved sequence ACA [5]. In some systems it has been shown that the efficiency of the promoter correlates with how well the -35 region matches the prototype hexanucleotide TTGACA [6]. The sequences surrounding the -35 region do not show definite homology but tend to be AT-rich. Since AT pairs are held together by only two phosphodiester bonds, while GC pairs contain three bonds, it is possible that local melting also plays a role, allowing the polymerase to unwind the DNA after it has recognized the TTGACA sequence.

The RNA firm binding site is located in a region approximately ten nucleotides upstream of the transcription start site. It is a seven base pair sequence homologous to the sequence TATAATG. The T residue in the sixth position is invariant among known promoters and it is likely that this nucleotide plays a necessary role in the positioning of RNA polymerase on the DNA strand. Again, promoter efficiency appears to correlate with how well the sequence matches the prototype [7].

Transcription of mRNA initiates at a position six to nine base pairs downstream of the highly conserved T residue in the -10 region. The start nucleotide is usually an A or a G. However, it is impossible to locate the exact start site without *in vivo* or *in vitro* data.

A gene whose initiation is normally only controlled by a promoter is said to be *constitutive*. Its end product is produced in fixed amounts, regardless of environmental conditions. There are other genes whose synthesis can be adapted to reflect growth conditions by virtue of the interaction of specific *regulatory proteins* with a sequence adjacent to the genes known as an *operator*. A set of genes transcribed from a single promoter, and whose joint function is controlled by an operator and a regulatory protein is known as an *operon* (see figure 3.1).

Role of the Regulatory Protein

A class of proteins called *regulatory proteins* is involved in the control of operons. The genes which code for regulatory proteins are called *regulatory genes*. All

[4] If the CAP site is included as part of the promoter, then the region extends to -80.

[5] "conserved" refers to presence across species lines, i.e. conserved in evolution. It is a general molecular biological principle that conserved sequences are functionally important.

[6] Rosenberg and Court, p. 325.

[7] Rosenberg and Court, p. 330.

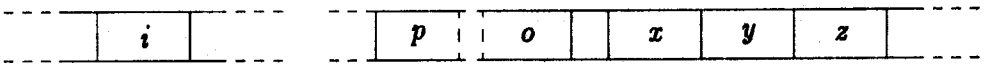

Figure 3.1: a typical operon. i represents the regulatory gene; p, the promoter; o, the operator; and x, y and z, the structural genes of the operon. Relative sizes of the parts are not drawn to scale.

known regulatory proteins are either produced constitutively or regulate their own production[8].

Regulatory proteins in bacteria can exhibit both positive and negative control mechanisms. In positive control (PC) systems, the presence of the regulatory protein (the *activator*) is necessary for the expression of the operon. In negative control (NC) systems, the operon is expressed unless it is inhibited by the presence of the regulatory protein (the *repressor*). Most known bacterial operons are NC. Some operons are under both positive and negative control.

Regulatory proteins by themselves are not always functional. If they were, the operons that they control would be permanently on or off. Instead, the activity of a regulatory protein is dependent on its combining with a specific small molecule. The binding of the small molecule to the regulatory protein changes the conformation of the protein, thus affecting its ability to bind to the operator (see below).

Regulatory proteins are either *inducible* or *repressible*. If the binding of the small molecule to the regulatory protein turns the operon on, then the system is inducible (the small molecule is called an *inducer*). If the binding of the small molecule turns the operon off, the system is repressible and the small molecule is called a *corepressor*.

Certain general rules determine whether a system is inducible or repressible and what the small molecule is likely to be. An operon whose genes code for enzymes to digest a certain substance is likely to be inducible by that substance, since the digestive enzymes are needed only when the substance is present. On the other hand, an operon whose genes code for enzymes to synthesize a certain product is likely to be repressed by that product, since if the product is present, there is no need to synthesize it.

The class of operons known as *glucose sensitive operons* operate under both positive and negative control. Each of these operons controls the breakdown of a specific sugar. In the presence of glucose, none of the glucose sensitive operon's

[8]A possible explanation for this is that if all regulatory genes were themselves regulated by other regulatory proteins, it would require an infinite chain of control.

proteins are synthesized, because the catabolism (breakdown) of glucose lowers the amount of cyclic AMP (cAMP) within the cell. cAMP is necessary for the transcription of all glucose sensitive operons. It works by binding the *catabolite activator protein (CAP)*, which then binds to a specific site on DNA (the CAP site). This acts to increase the rate at which RNA polymerase binds to the promoter. Thus the CAP-cAMP complex is the positive control element for all glucose sensitive operons.

The Role of the Operator

All regulatory proteins act by binding to DNA at specific sites called *operators*. The operator region is located at or near the transcription start site, slightly overlapping the promoter. The operator sequence must be at least 10-12 nucleotides in length to avoid the possibility of the same sequence randomly occurring elsewhere on the chromosome.

The mechanism by which an active repressor molecule inhibits expression of its associated operon is well understood for several operons. Since the operator region includes the transcription start site, the presence of a bound repressor at the operator physically blocks the binding of RNA polymerase to the DNA strand. The action of activator molecules is not as well understood. It is thought that they act by somehow "opening up" the DNA helix for the polymerase.

There are thus four models for operon control: PC-inducible, PC-repressible, NC-inducible, and NC-repressible. How the operon reacts to its external environment is dependent on which one of these four types it is.

3.1.2 Control of Elongation

Transcription termination sites can occur early within, as well as at the end of, an operon. These internal termination sites are known as *attenuators*. In bacterial systems, attenuators have been found in operons whose end products are involved in the synthesis of amino acids. Attenuators are located in a *leader region* between the transcription start site and the structural genes of the operon. Attenuators regulate gene expression by controlling transcription of the operon downstream from the attenuator. The mechanism by which this occurs takes advantage of the fact that in prokaryotic systems, translation of the initial segment of the peptide begins while the later regions are being transcribed.

Analysis of the sequence of the leader region of several operons revealed that the region contains a potential transcription start site and termination site. The termination site is preceded by a segment rich in the amino acid that is the end product of the corresponding biosynthetic pathway. In a bacterial cell lacking this

amino acid, the ribosome translating the leader sequence becomes stalled at these amino acid codons (due to a lack of charged tRNA molecules) [9]. The mRNA sequence within the leader region is capable of forming two alternative secondary structures. One of these structures is more stable (and thus thermodynamically favored) when ribosomes have stalled in the leader region. This structure does not contain a functional transcription termination signal. The other secondary structure contains a function transcription terminator sequence. Since transcription and translation proceed cotemporally in prokaryotic systems, it is thought that the secondary structure in the stalled message directs the RNA polymerase to continue transcribing the operon. Thus transcription of the structural genes of the operon can be completed, leading to the synthesis of the deficient amino acid.

3.1.3 Control of Termination

Transcription termination occurs at the end of genes and operons and is necessary to prevent transcription in one part of the chromosome from reading through to other genes.

Comparison of transcription termination site sequences reveals several common features [10]. All contain inverted repeat sequences (palindromes). These sequences have the ability to form stable hairpin-loop structures by base-pairing between nucleotides of the same strand of DNA or RNA, and might serve for steric recognition. The termination site is preceded by a GC-rich region of variable length (from 3-11 consecutive GC pairs). Since GC bonds are more stable than AT bonds, this segment may assist termination by hampering the movement of RNA polymerase along the template. Finally, the termination sequences end in a series of four to eight T residues. These may enhance release of the transcript.

As discussed above, most of the control of gene expression in prokaryotes exists at the transcriptional level. Other levels of control are possible, however. The amount of gene product made can be affected by the stability of the message (*post-transcriptional control*), by mutations which change the reading of the message (*translational control*), and by stability of the final product (*post-translational control*).

Mutations which affect the stability of the message can only be hypothesized, since the exact relationship between conformation and stability is not known. Similarly, the steady-state level of a particular protein can be influenced by post-translational modification (e.g. proteolytic cleavage) which affects its stability.

[9] tRNA molecules are said to be charged if they are bound to their corresponding amino acid, otherwise they are said to be uncharged.

[10] All the termination data is from Rosenberg and Court.

However, not enough is currently known about these areas for GENEX to make any significant use of them.

3.2 Control of Translation

The amount of protein made from a given message could be influenced by the relative affinity of ribosomes for that message compared with other cellular messages. Thus, a mutation in the ribosome binding site could theoretically alter gene expression by increasing or decreasing the rate at which ribosomes attach to and begin translation of a specific message. The most extreme example of this is an mRNA molecule from which the ribosome binding site has been deleted–such a message would almost never be translated into protein.

Each codon in an mRNA molecule codes for either an amino acid or chain termination. Of the 64 possible three-letter codons, three code for chain termination, and 61 code for amino acids (of which there are only 20). Thus several different codons may correspond to the same amino acid. However, a mutation which changes even a single nucleotide of a message may change the protein made from that message. Ribosomes can only initiate translation at a specific sequence, AUG [11]. Mutations which alter the nucleotide sequence of the translated region of the gene can result in the wrong protein being made (missense mutations), or, if the mutation replaces a codon for an amino acid with a termination codon, in no protein being made (nonsense mutations).

[11]Much less frequently, GUG.

Chapter 4

The Genex I Program

The following represents a brief description of the GENEX I program. Readers interested in further implementational details are referred to the appendix.

4.1 Overview

The GENEX I program represented an operon as consisting of a promoter, operator, possible leader sequence, structural genes, and an associated regulatory gene. The operator was assumed to be located spatially between the promoter and the structural genes. The promoter included the CAP site in glucose-sensitive operons, but other positive regulators were assumed to function at the operator. The regulatory gene was represented as having a promoter but no other control sites. There was no constraint on the location of the regulatory gene relative to the operon.

To use the GENEX I program, the user entered the name of the operon to be examined, the nature of the protein for which it codes–whether it is involved in an anabolic or catabolic pathway, and what its substrates or end products are–and whether or not the genes of the operon are expressed. Additional information could also be given to assist the program in determining a model for the operon's behavior. This included the nucleotide sequence of all or part of the operon, the existence and location of any mutations in the operon or its regulatory gene, and the phenotype of any known diploids.

The more information that was given to the program initially, the more precise its solution. Told only that the operon's genes are or are not expressed, it would suggest every model consistent with the protein product of the operon. Given the nucleotide sequence of the operon, it could give a solution as detailed as pinpointing the exact nucleotide mutation which was the likely cause of the behavior.

The "expert knowledge" in the GENEX I program consisted of the information

in the preceding chapter encoded as 62 heuristic rules relating observed phenomena to possible causes. In each rule, the information was highly compiled and generally incorporated several inference steps.

The rules were grouped into procedures based on their categorization as being relative to transcriptional or translational control mechanisms, and further subdivided according to the location of the mutation which might be affecting the expression of the operon. Within these procedures, the information was further divided into two types: factors which enhance gene expression and factors which inhibit it. The procedures, which resembled decision trees, were then systematically applied to the operon given as input.

The flow of control in the original GENEX I program generally followed the protocol for determining the control of gene expression in bacterial operons shown in figure 4.1.

If there was a possibility that some aspect of the regulatory gene was causing the observed behavior, the program would call itself recursively using the regulatory gene as its input, since it too can be subject to transcriptional and translational control (such as promoter or structural mutations). The program would ignore procedures that do not apply to regulatory genes, for instance, those that look for positive and negative control structures [1].

GENEX I could terminate execution under two circumstances: when it had generated all possible solutions, or when it had found a solution that was an extremely likely cause of the observed behavior.

The primary data structure in the GENEX I program was a two-element list whose first element was an atom representing the operon being examined and whose second element represented the product of the operon (i.e. the protein coded for by the structural genes). Properties of the operon and gene product were attached to the property list of the corresponding element. This representation was chosen purely for the sake of expediency.

As an example of the way that the original system represented and used the domain information, consider the regulatory protein. GENEX I used the rule that anabolic processes are repressible and catabolic processes are inducible to determine the interaction of small molecules with the regulatory protein. There were also procedures to determine if any of the substances known to be present are a potential inducer (or corepressor) of the operon. Finally, there were rules which explained the relationship between the presence or absence of an inducer (or corepressor) and protein synthesis, shown in the FIND-INDUCERS procedure in figure 4.2. As can be seen in the above program, GENEX I could analyze the operon and

[1] Since, as mentioned above, regulatory genes are not thought to be themselves regulated by repressors or activators.

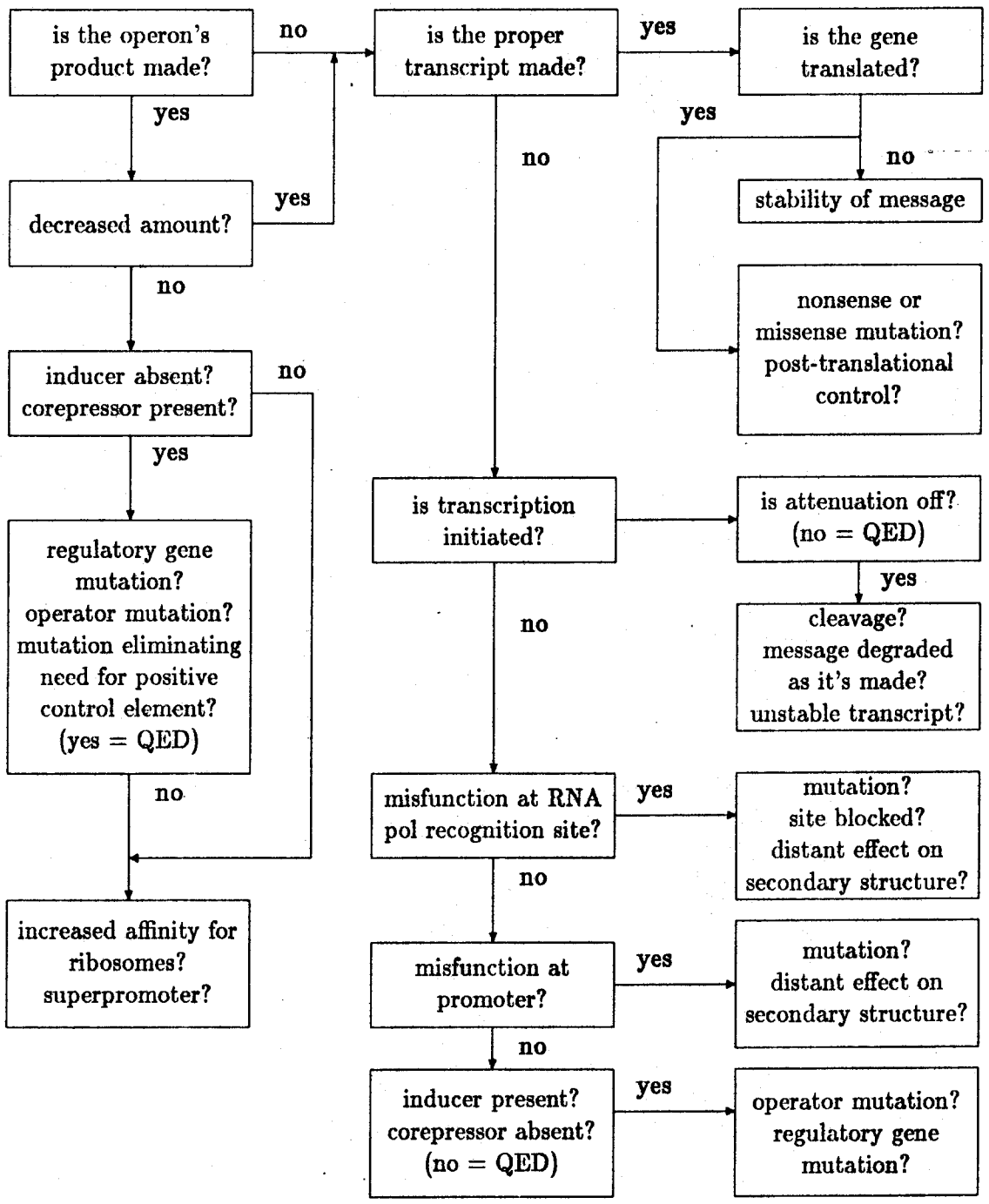

Figure 4.1: Protocol for modeling gene control in bacterial operons.

```
(defun find-inducers (x)
  (cond ((null (get (car x) 'inducer))
         ;we have to figure out the inducer
         (let ((inducer
                (do ((possible-inducers things-present
                                        (cdr possible-inducers)))
                    ((null possible-inducers) nil)
                    (cond ((inducer? x (car possible-inducers))
                           (return (car possible-inducers)))))))
           (cond ((and (null inducer)(not protein-made?))
           ;it seems that nothing present is the inducer, so...
                  (format t
                   "~%~A possibly inactive due to missing inducer"
                         (car x)))
                 (t (format t
                            "~%assuming ~A is inducer for ~A~@
                             ~A is active in presence of ~A"
                            inducer (car x) (car x) inducer)))))
;if we get to here, we were told what the gene's inducer is.
;first check if it's present, and if not,
;then maybe that explains bug.
        (t (cond ((member (get (car x) 'inducer) things-present)
                  (format t "~%~A is active in presence of ~A"
                          (car x) (get (car x) 'inducer)))
                 ((not protein-made?)
                  (format t
                   "~%~A inactive due to missing inducer ~A"
                          (car x) (get (car x) 'inducer))
                          (*throw 'exit 'qed))))))
```

Figure 4.2: the FIND-INDUCERS procedure.

determine whether or not it would be active, but the consequences that it reported were "canned." This is because GENEX I's rules contained highly-compiled knowledge and the program had no access to the lower-level mechanisms that justify the information contained in the rules.

The GENEX I program was tested on three examples taken from the final exam of an MIT undergraduate genetics course. These were all the questions appropriate for the program (dealing with not more than one mutation, and not requiring calculation of quantities. See discussion below). GENEX I was *not* designed and tested using these questions as a guide. In order to present a true picture of the capabilities of the program, it was evaluated using questions that were not available at the time the program was designed and implemented. A representative question from the exam and the program's output are shown below. (Comments on the example are given in italics.)

The enzyme *uncine synthetase* (USase) of the hypothetical bacterium *Altacoccus profundii* synthesizes the essential amino acid uncine. When no uncine is present in the growth medium USase is made at a high level, allowing cells to grow. When uncine is added to the medium, however, very little USase is made (the cells use the added uncine to grow).

This information can be represented to the program as follows. Let the operon be called **unc**. *The gene coding for USase is labeled* **unc-gene**.

```
unc
(UNC-GENE USASE)
```

This is what we the program has been told about USase:

```
(plist 'usase)
(PURPOSE SYNTHESIS END-PRODUCT UNCINE INV-PROPORTIONAL-TO (UNCINE))
```

THINGS-PRESENT *is a list of the substances present in the medium at the time of interest. The substances besides uncine are acting as red herrings.*

```
things-present
(LACTOSE TRYP UNCINE ARABINOSE)
```

```
(initialize unc)
is UNC-GENE product made? NO
```

The gene product, USase, is not made in the presence of uncine.

```
is there a mutation in UNC-GENE? (yes, no, unknown) NO
is there an unlinked mutation? (yes, no, unknown) NO
was diploid constructed? NO
DONE
```

The program now has all the information stated in the above paragraph.

Outline briefly, at the molecular level, two simple but different models that would explain why very little USASE is made in the presence of uncine. In each, be sure to indicate the role of uncine.

```
(genex unc)
assuming no misfunction at UNC-GENE promoter
is UNC-GENE a regulatory gene? NO
assuming UNC-GENE can be repressed
assuming UNC-GENE repressor is repressible
assuming UNCINE is corepressor for UNC-GENE--
        UNC-GENE is inactive in presence of UNCINE
```

GENEX I *has found one model for the behavior of the* unc *operon–that it is negatively controlled and corepressed by uncine. It used a rule which states that genes which code for biosynthetic pathways are repressible by their end-products.*

```
assuming no misfunction at UNC-GENE operator
checking positive control of UNC-GENE
assuming UNCINE is corepressor for UNC-GENE--
        UNC-GENE is inactive in presence of UNCINE
done checking positive control of UNC-GENE
```

A second model has been found, that unc *is positively controlled and the activator is inactivated by the presence of the co-repressor, uncine.*

```
message possibly attenuated due to presence of UNCINE
NIL
```

In fact, GENEX I *has found a third model for the behavior of* unc*, that uncine acts as an attenuator on the transcription of the* unc *genes. It used a rule which states that genes which code for the synthesis of amino acids can be attenuated.*

Certain mutants of *A. profundii* make USase even in the presence of added uncine (which can be shown to penetrate the cell as effectively as in the wild type). In each of your models from part A, indicate two different ways mutants could allow this to happen.

In order to answer this question, GENEX I *is run twice. The first time, it is told to assume that there is a mutation in the operon itself, the second time it is told to assume that the mutation is in the regulatory gene.*

First examine a potential mutation in the operon:

```
(initialize unc)
is UNC-GENE product made? YES
in what amount?  (increased or decreased) INCREASED
is there a mutation in UNC-GENE? (yes, no, unknown) YES
Where is it located? (promoter operator xc-region linked) LINKED
was diploid constructed? NO
DONE

(genex unc)
is UNC-GENE a regulatory gene? NO
assuming UNC-GENE can be repressed
assuming UNC-GENE repressor is repressible
assuming UNCINE is corepressor for UNC-GENE--
        UNC-GENE is inactive in presence of UNCINE
possible structural mutation in UNC-GENE operator allowing activator
        to bind despite presence of corepressor
```

This is one way our second model could be mutated which would explain the observed behavior.

```
mutation possibly affecting attenuator
NIL
```

GENEX I *also suggests that the mutation could be affecting the attenuator region, consistent with its third model of how the* unc *operon is regulated.*

Now for the regulatory gene mutation:

```
(initialize unc)
is UNC-GENE product made? YES
```

```
in what amount?  (increased or decreased) INCREASED
is there a mutation in UNC-GENE? (yes, no, unknown) NO
is there an unlinked mutation? (yes, no, unknown) YES
was diploid constructed? NO
DONE

(genex unc)
assuming no misfunction at UNC-GENE promoter
assuming UNC-GENE can be repressed
assuming UNCINE is corepressor for UNC-GENE--
        UNC-GENE is inactive in presence of UNCINE
examining UNC-GENE-REGULATORY-GENE
Is the regulatory protein made? (yes, no, unknown) UNKNOWN
assume UNC-GENE-REGULATORY-PROTEIN is not made
```

First assume that it's a nonsense mutation.

```
operon cannot be switched off without UNC-GENE-REGULATORY-PROTEIN
```

If the repressor is not made, the operon cannot be turned off. This is one mutation that may occur in our first model.

```
now assuming UNC-GENE-REGULATORY-PROTEIN is made
possible structural mutation in UNC-GENE-REGULATORY-PROTEIN
        eliminating ability to bind corepressor
possible structural mutation in UNC-GENE-REGULATORY-PROTEIN which
        prevents it from binding to operator
done examining UNC-GENE-REGULATORY-GENE
```

These are two more possible mutations which would alter the behavior of our first model in a way which would explain the observations.

```
assuming no misfunction at UNC-GENE operator
checking positive control of UNC-GENE
assuming UNCINE is corepressor for UNC-GENE--
        UNC-GENE is inactive in presence of UNCINE
examining UNC-GENE-REGULATORY-GENE
possible structural mutation in UNC-GENE-REGULATORY-PROTEIN
        eliminating ability to bind corepressor
```

This is a second mutation that could occur in our second model which would explain the observed behavior.

```
done examining UNC-GENE-REGULATORY-GENE
done checking positive control of UNC-GENE
NIL
```

In its limited domain, GENEX I performed very well. The goal of the system was to give every probable model for the behavior of the operon (within the limits of its knowledge). For all the problems on which it was tested, it gave all the correct models, sometimes more than the problem called for, and did not give any incorrect models. In this sense, the program was considered a success.

4.2 Limitations of the Genex I Program

GENEX I has certain limitations which prevent it from answering all types of questions even within the domain of bacterial operons.

The program is modeled on the investigative course of human experts. However, this results in a poorly modularized program, because the domain knowledge is incorporated into the reasoning mechanism. This makes it difficult to extend or modify the domain knowledge, because it would require rewriting all the procedures dealing with the new or changed information. This problem is relatively simple to correct, by rewriting the rules and inference engine as separate entities.

A more critical problem that GENEX I has is that its knowledge base consists solely of empirical associations and compiled knowledge, and the system's reasoning process consists of matching the right rules to the observed phenomena. Because the system solves a problem by matching the current situation against a set of predetermined situations, the set of rules must *anticipate* situations that may arise. In a small domain, it may be possible to enumerate all possible situations and write a rule to handle each one.[2] In a large domain, it may not be feasible to predict every possible occurrence. If the current situation does not match any rule, then the system fails.

Take as an example the observation that GENEX I could only handle problems involving one mutation in the operon. Many real-world problems in molecular genetics require reasoning about the combined effect of multiple mutations on gene expression.. In order to expand the system to consider multiple mutations, it would have been necessary to add a large number of rules with complicated antecedents,

[2]This technique was used in Puff (Kunz, 1978).

each rule corresponding to each possible combination of mutation location, gene product production, and control model.[3] These rules would no longer resemble the simple "rules of thumb" that the expert uses and on which the program was based.

Another limitation of the way that the domain knowledge is represented in the original system is that it is implicit in the rules, and thus inaccessible. For example, the original version of GENEX can only give "canned" explanations of its conclusions. The information that GENEX has is already compiled into high-level rules. GENEX has a rule which states that deleting the promoter causes a decrease in transcription of the operon. If given a problem that included a promoter mutation and a decrease in gene product, GENEX applies this rule and suggests that the promoter mutation caused the decrease in gene product. However, GENEX has no knowledge of the mechanism which makes this true, i.e. that the promoter is where the RNA polymerase binds, if the promoter is deleted the polymerase cannot bind, and if the polymerase cannot bind the operon cannot be transcribed. The details of how lower-level mechanisms combine to produce the more complex observed behavior was not included because the program could solve the problems without them. As a result of this, the system cannot be extended to explain its conclusions. The best it can do is to state which rules it applies. The rules that GENEX uses are not wrong, but each is a high-level compilation of a more detailed collection of knowledge about objects and their interactions. This lower-level information is not available to GENEX for use in justifying its conclusions.

Furthermore, because the knowledge is contained implicitly, it is inflexible. One piece of knowledge may be contained in any number of rules in the system, so in order to change that knowledge it must be changed in every rule that uses it. Highly compiled rules do not permit simple, local changes which are then reflected system-wide. For example, the original version of GENEX has an inflexible model of the operon. It is constrained to think of the operon as having exactly one promoter, operator, and regulatory gene. It cannot be presented with a problem where, for instance, a second promoter had been inserted into the operon. In operons having more than one structural gene, GENEX is constrained to deal with the genes as a group—it cannot reason about them as separate entities. The reason for this rigid model is that GENEX does not actually have or use any structure which represents "the operon." Instead, the representation is implicit in the procedures which reason about the operon, and therefore changing the way that GENEX models the operon requires changing most of the rules in the program. A variable model of the operon, which could change with each invocation of the program, is impossible in the initial version of the program.

[3] To see why the number of rules necessary would be so large, consider that if GENEX I had been given all the information necessary to solve problems involving two mutations in compiled form, this still would not have allowed it to solve problems involving three mutations.

A final problem with the empirical "rules of thumb" used by the original GENEX I program is that they often contain implicit preconditions. For instance, the heuristic that states that "deleting the operator results in increased expression of the operon" contains the implicit precondition "all other things being equal." If all other things are *not* equal, the validity of the heuristic's conclusion is unknown [4]. Implicit preconditions make it difficult, if not impossible, to determine the conditions under which a heuristic can be applied correctly. However, explicitly stating every precondition may be tedious – consider having to explicitly encode "all other things being equal" in a system with hundreds of components.

GENEX has no mechanisms for reasoning about spatial relationships, for instance, those between the structural genes, the promoter, and the operator. There are types of problems where the ability to do this kind of reasoning is important. For example, there is a phenomenon called the *polarity effect* whereby genes closer to the promoter are transcribed with higher efficiency. The ability to do spatial reasoning is essential for dealing with information about tertiary structure.

GENEX I makes no distinction between an increase in the amount of protein and constitutive production of the protein (or between a decrease in the amount of protein made vs. no protein made at all). This is because GENEX I has no concept of quantity. For many problems, relative quantities of substances are not significant to the solution of the problem. However, it would be useful to be able to deal with gross level changes. Also, the distinction between increased and constitutive production (and between decreased production and none at all) is important and should be expressible.

[4] This is the difference between an abstraction and a heuristic. An abstracted rule $X \longrightarrow Y$ holds for any x which can be abstracted to X via the abstraction relation. The same is not necessarily true for a heuristic.

Chapter 5

The Genex II System

The performance of the GENEX I was probably as good as it could be considering the simple design of the system. There were a few additions that would have been easy to implement and that would have fit in with the format of the program. For instance, it would have been useful to have a procedure which, given a DNA sequence, located any hairpin loops [1]. The program could then have been given new rules that depended on knowing the secondary structure of the DNA (such as those that might have been written for the conditions of transcription termination). GENEX I could also have used a procedure to determine if a sequence was AT- or GC-rich. Again, it could then have been given new rules which used this information.

These additions could have fine-tuned the performance of GENEX I, but would not have significantly increased its reasoning power. Any substantial improvements in the capabilities of the system required a major redesign of the program control structure and knowledge representation format. The improved version would have a model for each object and mechanism involved in operon control. The models would include such information as what the different parts of the object were, the role that each part plays in the functioning of the operon, how the parts interact, and how lower-level mechanisms combine into higher-level behavior. When presented with a problem, the system would pass the information through the models in a form of simulation to reach its conclusions (rather than doing a match between the observed phenomena and the knowledge in the system, as the original program did). The advantage of this model-driven approach is that the system would have more "understanding" of the processes with which it deals, because more of the details would be captured by the program (rather than being compiled out as rules by the system designer, as was done in the original program). The system would

[1] In fact, most commercially available DNA sequencing programs include such a procedure, which is why one was not included.

have access to the details, so it could generate more detailed solutions, and have the capability to explain the reasoning behind its conclusions. Finally, since the models would no longer be incorporated into the reasoning mechanism, the system would be more easily extensible. These significant improvements were implemented in the GENEX II program.

GENEX II was designed using the performance criteria described above as its goal. The domain knowledge was separated from the reasoning mechanisms, the rules used in the original system were rewritten to capture the underlying mechanistic details, and a primitive explanation capability was added. GENEX II is currently implemented in Prolog [2].

5.1 Representation of Objects in Genex II

Since the domain of the system is molecular genetics, the objects that are manipulated by the system are those that exist inside a cell. Some of the objects that are represented are shown in figure 5.1.

The representation of objects is independent of the representation of cell operations and the reasoning mechanism of the program. The representation of an object includes properties of the object, a description of its parts (which are themselves described) and its relationship with its environment. Part of the definition of a repressible regulatory protein is given in figure 5.2. For readers unfamiliar with Prolog, a detailed explanation of the description of a regulatory protein follows.

The description of the regulatory protein is made up of eleven *clauses*. The clause is made up of a *head* and a *body*. The head consists of a single *goal* and the body consists of a sequence of zero or more goals [3]. In the first clause shown in figure 5.2, "protein(X)" is the head and "reg-protein(X)" is the body. This can be interpreted to mean "to satisfy the goal *protein(X)*, satisfy the goal *reg-protein(X)*," or, in English, "any X is a protein if X is a reg-protein." Where there is more than one goal with the same head clause, Prolog will attempt to satisfy each one in turn, until one succeeds, or they all fail. Variables in Prolog are denoted by names beginning with a capital letter. The scope of the variable is the clause in which it appears, so the variable X in the first clause is independent of the variable named X in the next three clauses.

The second and third clauses state that the activator or repressor of an operon is a regulatory protein, i.e. that there are two types of regulatory protein. The

[2] For an introduction to the Prolog language, see Clocksin and Mellish, 1981.

[3] In this example, no clause has an empty body. A clause with no body is called a *unit clause* and is written in the form *P.* where *P* is the head goal, and is interpreted to mean "*P* is true."

```
operon
promoter
operator
terminator
binding-site
regulatory-site
gene
structural-gene
regulatory-gene
nucleotide
nucleotide-sequence
codon
DNA
RNA
amino-acid
amino-acid-sequence
protein
structural-protein
regulatory-protein
sugar
repressor
activator
inducer
corepressor
enzyme
polymerase
```

Figure 5.1: The objects represented in GENEX II

```
protein(X) :- reg-protein(X).
reg-protein(X) :- activator(Operon,X).
reg-protein(X) :- repressor(Operon,X).
reg-protein(X) :- product(Reg-gene,X),
                  reg-gene(Reg-gene).
part-of(R, operator-binding-site(R)) :-
                  reg-protein(R), normal(R).
part-of(R, operator-binding-site(R)) :-
                  reg-protein(R),
                  assuming(part-of(R, operator-binding-site(R))).
part-of(R, small-mol-binding-site(R)) :-
                  reg-protein(R), normal(R).
part-of(R, small-mol-binding-site(R)) :-
                  reg-protein(R),
                  assuming(part-of(R, small-mol-binding-site(R))).
active(Reg_protein) :- reg_protein(Reg_protein),
                  active_smmol_conform(Reg_protein).
active(Reg_protein)d :- reg_protein(Reg_protein),
                  assuming(active_smmol_conform(Reg_protein)).
active-smmol-conform(R) :- repressor(Op,R),
                  normal(R),
                  repressible(Op),
                  corepressor(Op,C),
                  bindable(C,small-mol-binding-site(R)).
```

Figure 5.2: Part of the description of a regulatory protein

fourth clause defines a regulatory protein to be the product of a regulatory gene.

The next four clauses define the important parts of a regulatory protein: its operator binding site and its small molecule binding site. GENEX II assumes that if the regulatory gene is not mutated, then the product regulatory protein is normal and has all its parts. If the regulatory gene is mutated, then the goal is satisfied if we are currently assuming [4] that its parts are intact.

Finally there are clauses for determining whether a regulatory protein is active, that is, able to bind to the operator. A normal regulatory protein is active if it interacts correctly with the associated small molecule (GENEX II calls this state `active-smmol-conform`). In the case of a mutated regulatory protein, the goal is satisfied if we are currently assuming that `active-smmol-conform` is true. GENEX II contains twelve rules[5] for determining whether a regulatory protein is in an active conformation. Because of space considerations, only one is shown in figure 5.2 [6]. This rule states that a normal repressor is in the active conformation if the corepressor of the operon is able to bind to the small molecule binding site of the repressor.

In the GENEX I program, an operon was predefined, and there was only an implicit representation of the relationship between its parts, and its interaction with its environment. In GENEX II, this is all explicit. For instance, the *lac* operon would be represented as shown in figure 5.3 [7].

5.2 Representation of Operations in Genex II

The system must also be able to manipulate models of the operations that can be performed on the objects. These operations correspond to the processes that go on inside the cell. The operations that the system can use to model the processes of a cell are *express*, *replicate*, *transcribe*, *translate*, *bind*, and *break*. A future system might include the operations insert, react, cut, ligate, polymerize, add-functional-group, remove-functional-group, replace-functional-group, denature, renature, and unwind. The addition of these operations would be very simple.

GENEX II simulates an operation by proving that all the preconditions of that operation are satisfied. If they are, it asserts that any postconditions of the opera-

[4] Assumptions are discussed in section 5.3.1.

[5] The use of the word *rule* in this chapter should be interpreted as referring to a Prolog clause, and should not be confused with the rules of a "rule-based system."

[6] Interested readers can consult Appendix B, which contains a complete listing of the GENEX II code.

[7] The user does not have to enter all this information by hand. There are rules for deriving some of the relationships based on given information.

```
operon(lac-operon).

part-of(lac-operon, lac-operon-promoter).
promoter(lac-operon-promoter).

part-of(lac-operon, lac-operon-operator).
operator(lac-operon-operator).

part-of(lac-operon, lac-operon-reg-gene).
reg-gene(lac-operon-reg-gene).

part-of(lac-operon, b-galactosidase-gene).
struct-gene(lac-operon, b-galactosidase-gene).

part-of(lac-operon, b-galactoside-permease-gene).
struct-gene(lac-operon, b-galactoside-permease-gene).

part-of(lac-operon, b-galactoside-transacetylase-gene).
struct-gene(lac-operon, b-galactoside-transacetylase-gene).

part-of(P,overlap-region(P,O)) :- promoter(P),operator(O).
part-of(O,overlap-region(P,O)) :- operator(O),promoter(P).
next-to(lac-operon-promoter,b-galactosidase-gene).
next-to(b-galactosidase-gene,b-galactoside-permease-gene).
next-to(b-galactoside-permease-gene,
        b-galactoside-transacetylase-gene).

product(b-galactosidase-gene, b-galactosidase).
enzyme(b-galactosidase).
substrate(b-galactosidase, lactose).
purpose(b-galactosidase, catabolic-proc).

...etc
```

Figure 5.3: Representation of the *lac* operon in GENEX II.

```
bind(Mol, Site) :- binding-site(Mol, Site),
                   complementary-conform(Mol,Site),
                   \+ bound(X,Site),
                   free-to-bind(Mol),
                   assert(bound(Mol,Site)).
```

Figure 5.4: The BIND operation.

tion are true.

GENEX II's representation of the bind operation shown in figure 5.4. This operation involves two objects, the molecule being bound, and the site to which it is being bound. The operation has several preconditions:

First, the site must be the binding site for that molecule. In general, the binding site is an abstract label corresponding to a specific region of the operon, for instance, the binding site of the regulatory protein is the operator. GENEX II also has rules to determine whether a nucleotide sequence can be a binding site for a molecule. For example, if the molecule is RNA polymerase, it can check to see if the site has properties which would qualify it as a promoter sequence.

The molecule and the site to which it is being bound must have complementary conformations. Since the details of protein folding are not yet understood, GENEX II uses the assumption that in their normal (i.e. non-mutated) state, the molecule and the site have complementary conformations. This may be dependent on the presence or absence of a small molecule for regulatory protein binding operations. If either of the objects are mutated, GENEX II will check to see whether it is currently assuming that they have complementary conformations, and proceed accordingly.

Next, the bind operation requires that the site be unoccupied. In this respect, GENEX II uses a simplification of the actual biological process. It ignores the possibility that a molecule that it is attempting to bind has a higher affinity for a site than a molecule that is already there, in which case the bound molecule would eventually be displaced. Since the program works on the *micro* level, reasoning about only one operon through only one cycle of gene expression, it can avoid dealing with the equilibrium situation by always binding the molecule with the higher affinity for the site first.

Finally, the molecule must be free to bind to the site. This clause represents the possibility that there is a limited amount of a molecule present in the system, and there might be none available for this binding operation. Other substances are available in essentially unlimited quantities, and this clause would always be true for bind operations involving that substance. This feature is not currently imple-

mented in GENEX II, because, as explained in the previous paragraph, GENEX II currently only models a single operon at a time, and therefore does not deal with the quantitative aspects of the problem.

If the preconditions are satisfied, GENEX II will then assert that the molecule is bound to the site, which is the postcondition of the bind operation.

5.3 Program Structure

The GENEX II program is divided into three parts: a front end, called the model-maker which gets the information for the current simulation from the user; an interpreter, which simulates the behavior of the operon on the input, and the database.

The model-maker asks the user to enter information about the operon, and constructs the system's physical model of the operon from this information. Depending on the input, the model-maker can also decide that certain variables are *uncertain* (discussed below). The model-maker sets up all the system variables needed to run the program.

The second level of the GENEX II program is the GENEX II interpreter. The clauses that contain the system's domain knowledge are not run directly by the Prolog interpreter, but rather through this intermediary, thus allowing more control over the program's execution. Explanations are generated by attempting to prove that the operon is expressed (or not expressed, depending on the observed behavior), using rules corresponding to the operations that take place in the cell. The GENEX II interpreter decides when a result is nontrivial and should be reported to the user. This serves as a primitive explanation capability in the current system.

The bottom level of the GENEX II system is the knowledge base containing facts about prokaryotic operons and rules for reasoning about their behavior, as described in the previous section. It consists of just over 250 clauses, of which about 150 are definitions (e.g. `protein(x) :- reg_protein(x).`). The remaining 100 clauses encode rules for simulating the objects in the domain.

5.4 Dealing with Uncertainty

Certain concepts with which GENEX II deals are uncertain to some degree, due to incomplete input given to the system. For example, suppose GENEX II is told that an operon has a promoter mutation, but is not given the promoter sequence. Its hypotheses about the effect of that mutation have a lower degree of certainty than if the sequence had been given, since in that case, the program could have compared

it with known promoter up- and down-mutations [8]. Other rules in GENEX II's knowledge base are what I term *ultimately uncertain* from the system's point of view: that is, the system has no way to determine the truth or falsity of that rule, due to an incomplete understanding within the domain itself (rather than incomplete information within the system). An example of ultimate uncertainty is whether a mutation in the small molecule binding site of the regulatory protein will leave the protein able to assume an active conformation. The system can't tell[9] if this will cause the regulatory protein to be unable to bind the small molecule and thus never to be able to assume an active conformation, or if it makes the small molecule unnecessary, causing the regulatory protein to always be in an active conformation. However, these two cases can be distinguished by their effect on the *behavior* of the operon. This observation led to the introduction of *likely models*, discussed below.

The number of uncertain variables depends on the number and location of mutations given in the problem statement. GENEX II decides that a given variable is uncertain before running the main program. If any part of the operon is mutated, then any rules relating to the conformation of that part are uncertain, as are any rules relating to whether or not that part, or any of its subparts, even is part of the operon (as might be the case in a deletion mutation). Mutations in the transcribed portion of the operon are dealt with by the transcribe and translate operations. The worst case (in terms of number of uncertain variables) occurs when it is not even known whether or not the operon contains a mutation. Then every uncertain variable for every part of the operon is activated, as well as an uncertain variable corresponding to the presumed intactness of each operon part.

GENEX II initially dealt with ultimately uncertain clauses by initiating two paths of computation *a priori* for each ultimately uncertain variable: one for possible value *true*, and one for possible value *false*. It then simulated the behavior of the operon for each possible set of values. Of course, not every possible state yielded a simulation result consistent with the observed behavior of the operon. Only those which did were proposed as possible explanations to the user [10]. However, the total number of computations that would be generated is exponential in the number of uncertain variables, and for any but the most trivial problems, this is an unmanageable number. For instance, a problem containing a regulatory gene mutation and an operator mutation would require seven uncertain variables, and

[8] An up-mutation is one which increases the efficiency of the promoter. A down-mutation decreases the efficiency of the promoter.

[9] nor can any expert: the relationship between sequence and conformation is by no means completely understood by biologists.

[10] GENEX II does not rank the potential solutions in order of plausibility. This requires knowledge of molecular biology that the author has not yet acquired.

thus 128 distinct possible states [11].

In order to reduce the number of possible states that the system had to examine, several new rules were added to the knowledge base. Before GENEX II begins to search for possible explanations, these rules use the *observed behavior* of the operon to derive a *likely model* of its behavior. GENEX II restricts its search for explanations to those which fit with the likely model. This model is one of the following mutually exclusive, collectively exhaustive possibilities: inducible, repressible, constitutive, or uninducible [12]. Each of these models requires certain conditions to be true. For example, in order for an operon's product to be produced constitutively, either the regulatory protein cannot bind to the operator, or the small molecule cannot bind to the regulatory protein. GENEX II compares the conditions of the model with the known structure of the operon and the uncertain variables. A condition for a model can be satisfied in one of two ways. The condition may be known to be true – for instance, the user could tell the system that a regulatory protein does not bind to the operator based on experimental evidence, or may be provable based on known information. If a condition is not provable based on known information, but is possibly true (depending on the state of one or more uncertain variables) then the system will assume that the uncertain variables are in the proper state to make the model work. GENEX II then simulates the behavior of the operon under the combination of known and assumed conditions.

In many cases there are still a number of possible states which must be checked, since there is more than one assignment of uncertain variables that will satisfy the model. The number of states is much smaller than in the unrestricted case, however. It is not possible for the system to overlook a solution which fits the model, since it will try every state which is consistent with the model, and exactly one model is consistent with the observed behavior of the operon. If it happens that a model exists with which the system is unfamiliar, of course, the system will be unable to find a correct solution.

The use of likely models reduces the number of possible states that the program must verify by reducing the number of uncertain variables generated from the initial problem statement. Each likely model has a list of sets associated with it. Each set is an assignment of uncertain variables that would make the likely model true. The number of possible states which must be examined is halved for each uncertain variable for which we can assign a value. The result of using likely models to control

[11] While this may not seem like an unmanageable number, the reader must consider that we are still discussing relatively simple problems in a simplified problem space. In a more general molecular genetics problem solver, an exponential growth rate would be overwhelming.

[12] The system does not use these terms in the literal biological sense. For instance, an operon which was lacking a promoter would be considered "uninducible" because the gene product was not made despite the presence or absence of the small molecule.

genotype	no. of uncertain variables	no. of possible states	no. of states generated
i^c, o^c	7	128	32
i^c	5	32	8
i^s	5	32	8

Table 5.1: Reduction in possible states using likely models.

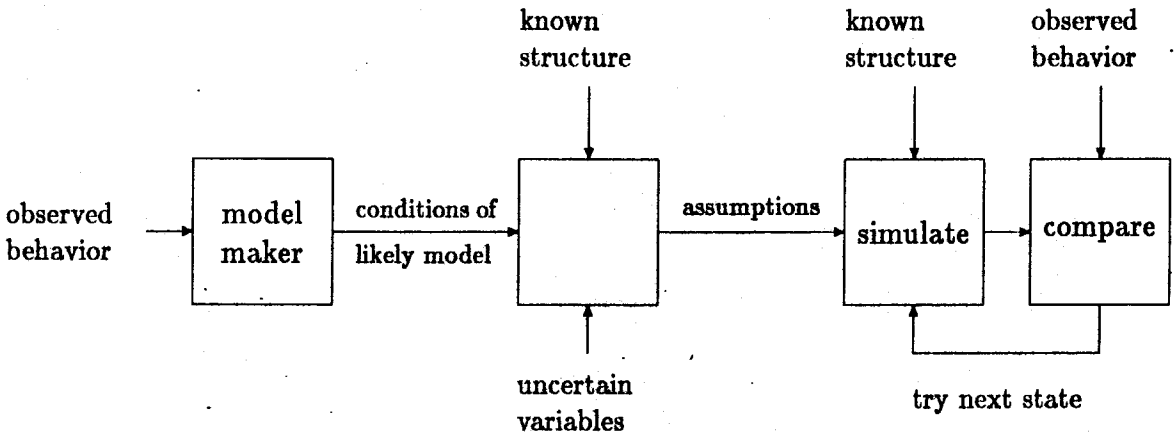

Figure 5.5: A flowchart for the GENEX II system.

the size of the search space is shown in table 5.1.

For the problems tested the use of likely models reduces the number of states that must be searched by 75 per cent.

The flow of control and information in the GENEX II is summarized in figure 5.5. I believe that this is the same way in which molecular biologists approach problems. A hypothesis is proposed based on previous problems which had similar behavior (corresponding to the model-making phase of the program). The biologist attempts to fit the details of the current problem to the hypothesis. The program does the same by making a set of assumptions from the uncertain variables. Finally, the biologist attempts to verify the hypothesis. GENEX II also attempts to verify the hypothesis, by simulating the behavior of the operon, and checking if it agrees with the observed behavior.

In some of the possible states, certain variables may be unimportant, i.e. the result of the simulation is the same whether they are true or false. For example,

given an unexpressed operon with a mutation in the promoter and a mutation in the structural genes, the latter mutation may be unimportant if we assume that the promoter is unable to bind RNA polymerase. If the program were able to recognize this, it could generate a more informative explanation.

GENEX II's model-maker is similar to heuristic DENDRAL's Planner. DENDRAL had a Structure Generator which could generate all chemically plausible isomers, given a chemical formula, and a Predictor which determined if a candidate structure's mass spectrum fit the given data. Because of the enormous size of the problem space, the designers wanted to reduce the number of candidate structures generated. They achieved this by requiring the generator to generate only those classes of structures which met some criteria obtained from the problem data. The Planner consists of a group of *specialists*, one for each chemical family. Each specialist determines the class of allowable candidates within its family by using a collection of "special facts and special-purpose heuristics" [13] applied to the given data. GENEX II's four likely models correspond to DENDRAL's specialists.

5.5 Generating Explanations and Justifications

The GENEX II interpreter recognizes certain operations and circumstances as important. When important operations are simulated, or important circumstances are noted, the system reports this to the user. The designation of operations as important is done by the system, and can change with each invocation of the program.

A rule is said to be important if it depends on an uncertain variable, or if it is non-trivial. All unit clauses are trivial (this inhibits the system from continually informing the user that it has determined that the input operon is indeed an operon, for instance). A rule is also trivial if it relates to a model of the operon that is not the one that the system has assumed is true. For instance, the system will not inform the user of any attempts to prove that the system has some inducer X if the model of the system is repressible.

In addition, the procedures EXPRESSED, INITIATED, ATTENUATED, TRANSCRIBE, TRANSLATE, TERMINATE, and PROCESS are permanently designated as being important. These seven procedures serve as milestones during execution.

GENEX II currently explains its reasoning to the user by informing the user of important results during the course of execution. This is essentially a trace of the program, and is meaningful simply because all rules have been given mnenonic names. Since the system now has a finely detailed model of operons and operon behavior, this trace is informative without resorting to the use of canned text,

[13] Feigenbaum, p. 175.

and has the potential to be upgraded, using techniques such as those described by Swartout (1981).

At the present time, GENEX II cannot respond to requests for justification from the user; however, the design of the system permits this as a possible future modification.

5.6 Types of Problems that Genex II Can Solve

GENEX II can solve all the problems that were solved by the original program. The new encoding of the domain knowledge allows the system to represent more objects and relationships, so the system is more powerful than it was before.

It was difficult to expand GENEX I to reason about problems involving more than one mutation in the operon, because it would have required adding a complicated conditional statement corresponding to each combination of mutation sites, gene expression, and control model in the system. GENEX II can now reason about the combined effect of multiple mutations on gene expression. The model-maker computes the set of all possible states corresponding to the mutation sites, etc., and then simulates the operon on each of the possible states. This allows the system to be easily expanded, since the set of possible states is dynamic: it is created by the system when the program is invoked, rather than being written into the program.

GENEX II now has an explicit model of the operon, which allows the system to represent non-standard operons. This allows it to solve another large class of problems. For instance, it can now model an operon which has two promoters. GENEX II can also model diploids, [14] allowing the system to determine, for instance, whether a mutation is *trans*-dominant or *trans*-recessive.

The system still has no concept of quantity. GENEX II cannot solve any problem relating to equilibrium states, enzyme assays, or any abstract type of quantitative measurement, such as *less than* or *a small amount*. It was decided at the inception of this project by the molecular biology expert that the more interesting genetics research questions involved micro-level reasoning rather than reasoning about quantities, and the system reflects that bias.

The system currently takes in a description of behavior and structure, and explains the behavior of the operon. With a minor modification to the system, it could also take in a structure and predict the behavior.

[14] a operon in which some parts are represented twice.

5.7 "Understanding" Molecular Genetics

This section contains two examples which demonstrate the increased capabilities of the system. The first example deals with a double mutant containing mutations in both the regulatory gene and the operator. This example is again taken from the MIT undergraduate genetics exam [15].

Bacterium pedantia **produces a protein, Sporulin, that makes other bacteria sporulate...Sporulin synthesis is inducible by chalk. In the absence of chalk a repressor produced by gene *spoR* prevents expression of the Sporulin structural gene spoS.**

Haploids carrying two mutations can be made in *B. pedantia* by recombination. Most haploid double mutants that carry both a constitutive *spoR* and a constitutive *spoO* mutation are constitutive, as expected. However, one particular double — the haploid carrying *spoR*43 and *spoO*97 — is anomalous: it produces Sporulin *only* when chalk is *absent*. *spoR*43 is known to be missense. Explain briefly how the repressor and operator are functioning in the double mutant.

The program is started up and the information stated in the problem is given as input.

```
| ?- genex.
end every answer with a period followed by a CR!

solution mode (terse or verbose): verbose.
name of operon: spo.
spo structural genes: enter one at a time, 5prime - 3prime order
 -- end with ''done.''
structural gene:
|: spos.
what is product of spos
|: sporulin.
what is purpose of sporulin: (synthesis, digestion, or other)
|: other.
structural gene:
|: done.
```

[15] For this example, the program generated 32 possible states, two of which were shown to be consistent with the model. The explanation for only one state is shown here, due to space considerations. The output of the program has been edited to omit redundancy in the proofs. Places where editing has occurred and comments inserted are marked by the symbol "⟩⟩."

```
enter sequence of spo or [ ]
|: [ ].
are there any mutations in spo or reg-gene (yes, no, or unknown) ?
|: yes.
mutation site known (yes or no) ?
|: yes.
mutation in promoter (yes or no) ?
|: no.
mutation in reg-gene (yes or no) ?
|: yes.
mutation in operator (yes or no) ?
|: yes.
mutation in struc-genes (yes or no) ?
|: no.
is spo gene product made?
|: no.
enter any other information available
e.g. proportional-to(X,Y), inv-proportional-to(X,Y),
present(Substance), etc.
-- end with ''done.''
|: present(chalk).
|: inv-rel(chalk,sporulin).
|: done.
```

The program now lists the assumptions that it is making based on the input. "smbs" stands for "small molecule binding site."

```
assuming: mutated(operator_specific(spo_operator))
assuming: mutated(overlap_region(spo_promoter,spo_operator))
assuming: active_smmol_conform(spo_reg_protein)
assuming: mutated(spo_reg_protein)
assuming: part_of(spo_reg_protein,smbs(spo_reg_protein))
assuming: complementary_conform(_1145,smbs(spo_reg_protein))
assuming: smmol(_1858)
assuming: reg_protein(spo_reg_protein)
assuming: operator(spo_operator)
assuming: complementary_conform(_1858,smbs(spo_reg_protein))
assuming: complementary_conform(spo_reg_protein,
                                overlap_region(spo_promoter,spo_operator))
```

The following trace is printed out by the program. Each time a new goal is encountered, the program prints out the rule or rules it uses to try to prove that goal.

```
goal: expressed(spo)
  attempting to prove:
  [operon(spo),
   initiated(spo),
   \+attenuated(spo),
   xcribe(get_gene_seq(spo),_1721),
   xlate(_1721,_1722)]
   goal: initiated(spo)
     attempting to prove:
     [operon(spo),
      part_of(spo,_1772),
      promoter(_1772),
      bindable(rna_pol,_1772)]
```

Expression initiates when RNA polymerase binds to the promoter. The system first checks to see if polymerase is already bound to the promoter, otherwise it checks if RNA polymerase is able to bind.

```
      goal: bindable(rna_pol,spo_promoter)
        attempting to prove:
        [bound(rna_pol,spo_promoter)]
        goal: bound(rna_pol,spo_promoter)
          attempting to prove:
          [promoter(spo_promoter),
           part_of(spo_promoter,_1951),
           bound(rna_pol,_1951)]
          goal: bound(rna_pol,overlap_region(spo_promoter,
                                             spo_operator))
          >>failed goal: bound(rna_pol,
                               overlap_region(spo_promoter,
                                              spo_operator))
        >>failed goal: bound(rna_pol,spo_promoter)
      goal: bindable(rna_pol,spo_promoter)
        attempting to prove:
        [bind(rna_pol,spo_promoter)]
        goal: bind(rna_pol,spo_promoter)
          attempting to prove:
          [binding_site(rna_pol,spo_promoter),
```

```
complementary_conform(rna_pol,spo_promoter),
\+bound(_1951,spo_promoter),
asserta(genex_clause(bound(rna_pol,spo_promoter)))]
goal: binding_site(rna_pol,spo_promoter)
  attempting to prove:
  [promoter(spo_promoter)]
proved: binding_site(rna_pol,spo_promoter)
goal: complementary_conform(rna_pol,spo_promoter)
  attempting to prove:
  [operon(_2039),
   assoc(_2039,_2040),
   product(_2040,spo_promoter),
   smmol(_2039,rna_pol),
   present(rna_pol),
   normal(spo_promoter)]
```

This rule is inapplicable because the spo promoter is not the product of any operon.

```
goal: complementary_conform(rna_pol,spo_promoter)
  attempting to prove:
    [promoter(spo_promoter),
     \+bad_rna_conform(spo_promoter)]
```

"Bad rna conform" is GENEX*'s term for a promoter not in the proper conformation to bind RNA polymerase. An active regulatory protein bound to the operator prevents the polymerase from binding to the promoter, in negative-control systems.* GENEX *will try to determine if the regulatory protein can bind to the operator.*

```
goal: normal(spo_promoter)
  attempting to prove:
  [\+mutated(spo_promoter),
   \+assuming(mutated(spo_promoter))]
proved: normal(spo_promoter)
goal: bindable
      (spo_reg_protein,
       overlap_region(spo_promoter,spo_operator))
  attempting to prove:
  [bound(spo_reg_protein,
         overlap_region(spo_promoter,spo_operator))]
  >>failed goal: bound(spo_reg_protein,
                       overlap_region(spo_promoter,
```

```
                                                spo_operator))
goal: bindable
      (spo_reg_protein,
       overlap_region(spo_promoter,spo_operator))
  attempting to prove:
  [bind(spo_reg_protein,
        overlap_region(spo_promoter,spo_operator))]
  goal: bind(spo_reg_protein,
             overlap_region(spo_promoter,spo_operator))
    attempting to prove:
    [binding_site
          (spo_reg_protein,
           overlap_region(spo_promoter,spo_operator)),
     complementary_conform
          (spo_reg_protein,
           overlap_region(spo_promoter,spo_operator)),
     \+bound(_2718,
             overlap_region(spo_promoter,spo_operator)),
     asserta(genex_clause
          (bound(spo_reg_protein,
                 overlap_region(spo_promoter,
                                spo_operator))))]
    goal: binding_site(spo_reg_protein,
                       overlap_region(spo_promoter,
                                      spo_operator))
      attempting to prove:
      [reg_protein(spo_reg_protein),
       operator(spo_operator),
       promoter(spo_promoter)]
    proved: binding_site(spo_reg_protein,
                         overlap_region(spo_promoter,
                                        spo_operator))
    goal: complementary_conform
            (spo_reg_protein,
             overlap_region(spo_promoter,spo_operator))
```

Based on its likely model of the operon, the system is already assuming that the regulatory protein and the overlap region have complementary conformations. In order to see the details of the solution, we have told the program to ignore its assumption by telling it to use "verbose" mode. ("Terse" mode would be used to quickly ver-

ify a solution without having to see all the details). It will have to show that the regulatory protein and the overlap region have complementary conformations using model-based reasoning.

```
attempting to prove:
[operator(spo_operator),
 promoter(spo_promoter),
 reg_protein(spo_reg_protein),
 normal(overlap_region(spo_promoter,spo_operator)),
 active(spo_reg_protein)]
```

The system assumes that if the overlap region is normal, then the regulatory protein can bind to it. This goal fails because the system is assuming that the overlap region is mutated.

```
goal: normal(overlap_region(spo_promoter,
                            spo_operator))
  attempting to prove:
  [\+mutated(overlap_region(spo_promoter,
                            spo_operator)),
   \+assuming(mutated(overlap_region(spo_promoter,
                                     spo_operator)))]
  goal: normal(overlap_region(spo_promoter,
                              spo_operator))
    attempting to prove:
    [gene(_3432),
     product(overlap_region
               (spo_promoter,spo_operator),_3432),
     normal(_3432)]
  >>failed goal:normal(overlap_region(spo_promoter,
                                      spo_operator))
```

The system now tries an empirical rule, which states that if the regulatory protein demonstrates an increased affinity for the operator, the two must have complementary conformations, without the need for the small molecule to "turn on" the repressor.

```
goal: complementary_conform
        (spo_reg_protein,
         overlap_region(spo_promoter,spo_operator))
  attempting to prove:
  [reg_protein(spo_reg_protein),
```

```
promoter(spo_promoter),
operator(spo_operator),
inc_affinity(spo_reg_protein,spo_operator)]
goal: inc_affinity(spo_reg_protein,spo_operator)
  attempting to prove:
  [operon(_3433),
   part_of(_3433,spo_operator),
   operator(spo_operator),
   mutated(spo_operator),
   assoc(_3433,_3434),
   reg_gene(_3434),
   product(_3434,spo_reg_protein),
   phenotype(_3433,uninducible)]
  goal: phenotype(spo,uninducible)
    attempting to prove:
    [operon(spo),
     assoc(spo,_3740),
     reg_gene(_3740),
     mutated(_3740),
     observed_not_expressed(spo),
     no_influence(_3741,spo),
     asserta(genex_clause(phenotype
                          (spo,uninducible)))]
    goal: no_influence(_3741,spo)
      attempting to prove:
      [operon(spo),
       structural_gene(spo,_3945),
       product(_3945,_3946),
       \+dir_rel(_3741,_3946),
       \+inv_rel(_3741,_3946)]
    >>failed goal: no_influence(_3741,spo)
```

This goal failed because the system was told that chalk inhibits the production of sporulin.

```
>>failed goal: phenotype(spo,uninducible)
goal: inc_affinity(spo_reg_protein,
                   spo_operator)
  attempting to prove:
  [operon(_3433),
   part_of(_3433,spo_operator),
```

```
                operator(spo_operator),
                mutated(spo_operator),
                assoc(_3433,_3434),
                reg_gene(_3434),
                product(_3434,spo_reg_protein),
                phenotype(_3433,superrepressed)]
              goal: phenotype(spo,superrepressed)
                attempting to prove:
                [operon(spo),
                 assoc(spo,_3740),
                 reg_gene(_3740),
                 mutated(_3740),
                 observed_not_expressed(spo),
                 no_influence(_3741,spo),
                 asserta(genex_clause
                           (phenotype(spo,superrepressed)))]
                goal: no_influence(_3741,spo)
                >>failed goal: no_influence(_3741,spo)
            >>failed goal: phenotype(spo,superrepressed)
          >>failed goal: inc_affinity(spo_reg_protein,
                                      spo_operator)
goal: active(spo_reg_protein)
  attempting to prove:
  [reg_protein(Reg_protein),
   active_smmol_conform(Reg_protein)])
```

GENEX *now tries several rules in an attempt to show that the regulatory protein is in an active conformation. They will fail because they either depend on a normal regulatory protein, or because they require a different control model than the one exhibited by SPO (i.e. repressible).*

```
goal: active_smmol_conform(spo_reg_protein)
  attempting to prove:
  [repressor(_3823,spo_reg_protein),
  normal(spo_reg_protein),
   inducible(_3823),
   \+inducer(_3823,_3824)]
  goal: normal(spo_reg_protein)
    attempting to prove:
    [\+mutated(spo_reg_protein),
    \+assuming(mutated(spo_reg_protein))]
```

```
  goal: normal(spo_reg_protein)
    attempting to prove:
    [gene(_4083),
     product(spo_reg_protein,_4083),
     normal(_4083)]
  >>failed goal: normal(spo_reg_protein)
goal: active_smmol_conform(spo_reg_protein)
  attempting to prove:
  [repressor(_3823,spo_reg_protein),
  normal(spo_reg_protein),
  inducible(_3823),
  inducer(_3823,_3824),
  \+bindable(_3824,smbs(spo_reg_protein))]
  goal: normal(spo_reg_protein)
  >>failed goal: normal(spo_reg_protein)
goal: active_smmol_conform(spo_reg_protein)
  attempting to prove:
  [repressor(_3823,spo_reg_protein),
  phenotype(_3823,inducible),
  \+inducer(_3823,_3824)]
  goal: phenotype(spo,inducible)
    attempting to prove:
    [operon(spo),
     structural_gene(spo,_4083),
     product(_4083,_4084),
     dir_rel(_4085,_4084),
     asserta(genex_clause(phenotype(spo,inducible)))]
  >>failed goal: phenotype(spo,inducible)
goal: active_smmol_conform(spo_reg_protein)
  attempting to prove:
  [repressor(_3823,spo_reg_protein),
   phenotype(_3823,inducible),
   inducer(_3823,_3824),
   \+bindable(_3824,smbs(spo_reg_protein))]
  goal: phenotype(spo,inducible)
  >>failed goal: phenotype(spo,inducible)
goal: active_smmol_conform(spo_reg_protein)
  attempting to prove:
  [repressor(_3823,spo_reg_protein),
   normal(spo_reg_protein),
```

```
 repressible(_3823),
 corepressor(_3823,_3824),
 bindable(_3824,smbs(spo_reg_protein))]
goal: normal(spo_reg_protein)
>>failed goal: normal(spo_reg_protein)
goal: active_smmol_conform(spo_reg_protein)
attempting to prove:
[repressor(_3823,spo_reg_protein),
 phenotype(_3823,repressible),
 corepressor(_3823,_3824),
 bindable(_3824,smbs(spo_reg_protein))]
```

This rule will succeed.

```
goal: phenotype(spo,repressible)
  attempting to prove:
  [operon(spo),
   structural_gene(spo,_4083),
   product(_4083,_4084),
   inv_rel(_4085,_4084),
   asserta(genex_clause
            (phenotype(spo,repressible)))]
proved: phenotype(spo,repressible)
goal: corepressor(spo,_3824)
  attempting to prove:
  [operon(spo),
   structural_gene(spo,_4231),
   product(_4231,_4232),
   end_product(_4232,_3824),
   inv_proportional_to(_4232,_3824),
   repressor(spo,_4233),
   bindable(_3824,_4233)]
```

The previous rule encoded a real-world heuristic for determining the corepressor of a repressible operon, however it fails on this "toy" example. The system uses a more general, simplified rule: if the gene product is made only in the absence of some substance, then that substance is a likely corepressor of the operon.

```
goal: corepressor(spo,_3824)
attempting to prove:
[operon(spo),
```

```
    structural_gene(spo,_4231),
    product(_4231,_4232),
    inv_rel(_3824,_4232)]
  proved: corepressor(spo,chalk)
goal: bindable(chalk,smbs(spo_reg_protein))
```

The system tries to determine if chalk is binding to the regulatory protein and putting it in an active conformation.

```
  attempting to prove:
  [bound(chalk,smbs(spo_reg_protein))]
>>failed goal: bound(chalk,smbs(spo_reg_protein))
goal: bindable(chalk,smbs(spo_reg_protein))
  attempting to prove:
  [bind(chalk,smbs(spo_reg_protein))]
  goal: bind(chalk,smbs(spo_reg_protein))
    attempting to prove:
    [binding_site(chalk,smbs(spo_reg_protein)),
     complementary_conform(chalk,
                           smbs(spo_reg_protein)),
     \+bound(_4396,smbs(spo_reg_protein)),
     asserta(genex_clause
               (bound(chalk,
                      smbs(spo_reg_protein))))]
    goal: binding_site(chalk,smbs(spo_reg_protein))
      attempting to prove:
      [reg_protein(spo_reg_protein),
       part_of(spo_reg_protein,smbs(spo_reg_protein)),
       assoc(_4422,_4423),
       reg_gene(_4423),
       product(_4423,spo_reg_protein),
       inducible(_4422),
       inducer(_4422,chalk)]
      goal: inducible(spo)
      >>failed goal: inducible(spo,_4863)
```

This rule failed because the SPO operon is not inducible. This rule and the following state that small molecule of an operon binds to the small molecule binding site of the operon's regulatory protein.

```
goal: binding_site(chalk,smbs(spo_reg_protein))
```

```
attempting to prove:
[reg_protein(spo_reg_protein),
 part_of(spo_reg_protein,
         smbs(spo_reg_protein)),
 assoc(_4422,_4423),
 reg_gene(_4423),
 product(_4423,spo_reg_protein),
 repressible(_4422),
 corepressor(_4422,chalk)]
goal: repressible(spo)
  attempting to prove:
  [operon(spo),
   structural_gene(spo,_4863),
   product(_4863,_4864),
   purpose(_4864,_4865),
   anabolic_proc(_4865)]
  goal: repressible(spo)
    attempting to prove:
    [operon(spo),
     corepressor(spo,_4863)]
    goal: corepressor(spo,_4863)
      attempting to prove:
      [operon(spo),
       structural_gene(spo,_4914),
       product(_4914,_4915),
       end_product(_4915,_4863),
       inv_proportional_to(_4915,_4863),
       repressor(spo,_4916),
       bindable(_4863,_4916)]
      goal: corepressor(spo,_4863)
        attempting to prove:
        [operon(spo),
         structural_gene(spo,_4914),
         product(_4914,_4915),
         inv_rel(_4863,_4915)]
      proved: corepressor(spo,chalk)
  proved: repressible(spo)
proved: binding_site(chalk,
                     smbs(spo_reg_protein))
goal: complementary_conform
```

```
                    (chalk,smbs(spo_reg_protein))
                attempting to prove:
                [operon(_5220),
                 assoc(_5220,_5221),
                 reg_gene(_5221),
                 product(_5221,spo_reg_protein),
                 reg_protein(spo_reg_protein),
                 smmol(_5220,chalk),
                 phenotype(_5220,repressible)]
```

An empirical rule. The corepressor must be in the proper conformation to bind to the regulatory protein if the operon is observed to be repressed in the presence of the corepressor.

```
                goal: smmol(spo,chalk)
                  attempting to prove:
                  [corepressor(spo,chalk)]
                  goal: corepressor(spo,chalk)
                  proved: corepressor(spo,chalk)
                proved: smmol(spo,chalk)
                goal: phenotype(spo,repressible)
                  attempting to prove:
                  [operon(spo),
                   structural_gene(spo,_5146),
                   product(_5146,_5147),
                   inv_rel(_5148,_5147),
                   asserta(genex_clause
                             (phenotype(spo,repressible)))]
                  proved: phenotype(spo,repressible)
              proved: complementary_conform
                         (chalk,smbs(spo_reg_protein))
              proved: bind(chalk,smbs(spo_reg_protein))
          proved: bindable(chalk,smbs(spo_reg_protein))
      proved: active_smmol_conform(spo_reg_protein)
    proved: active(spo_reg_protein)
  proved: complementary_conform
              (spo_reg_protein,
               overlap_region(spo_promoter,spo_operator))
proved: bind(spo_reg_protein,
             overlap_region(spo_promoter,spo_operator))
proved: bindable(spo_reg_protein,
```

```
                                overlap_region(spo_promoter,spo_operator))
        >>failed goal: bind(rna_pol,spo_promoter)
transcription cannot initiate
>>failed goal: expressed(spo)
checking other explanations
no sequence given...
no message to translate...
done.
```

Briefly, the system has decided that the spo *operon is no longer inducible, but is now repressible by chalk. When chalk is bound to the mutated regulatory protein, it is in the proper conformation to bind to the mutated operator, thus RNA polymerase cannot bind. Transcription cannot initiate, and the product is not made.*

The following is another example of the system's more detailed knowledge of the mechanisms of molecular genetics. In this example, GENEX has been told that the promoter has been deleted,and is then asked to explain why the operon is not expressed.

```
name of operon: op.

op structural genes: enter one at a time, 5prime - 3prime order
 -- end with "done"
structural gene:
|: done.
enter sequence of op or [ ]
|: [ ].
are there any mutations in op or reg_gene (yes, no, or unknown) ?
|: yes.
mutation site known (yes or no) ?
|: yes.
mutation in promoter (yes or no) ?
|: deletion.
mutation in reg_gene (yes or no) ?
|: no.
mutation in operator (yes or no) ?
|: no.
mutation in struc_genes (yes or no) ?
|: no.
is op gene product made?
|: no.
```

```
enter any other information available
e.g. proportional_to(X,Y), inv_proportional_to(X,Y), present(Substance),
etc.
-- end with "done"
|: done.
goal: expressed(op)
  attempting to prove:
  [operon(op),
   initiated(op),
   \+attenuated(op),
   xcribe(get_gene_seq(op),_853),
   xlate(_853,_854)]
  goal: initiated(op)
    attempting to prove:
    [operon(op),
     part_of(op,_904),
     promoter(_904),
     bindable(rna_pol,_904)]
    goal: promoter(_904)
      attempting to prove:
      [sequence(_904,_999),
       \+sequence(_904,[]),
       length(_999,_1000),approx(_1000,40),
       contains_subseq(_904,[t,t,g,a,c,a]),
       contains_subseq(_904,[t,a,t,a,a,t,g])]
```

Since the system does not have any part of the operon labeled as the promoter, it checks the sequence to see if any section of the sequence qualifies as the promoter. In this example, no sequence was given so it fails. It then checks to see if any part of the operon that it knows about is the promoter. This will also fail.

```
      goal: promoter(op_operator)
      >>failed goal: promoter(op_operator)
      goal: promoter(op_terminator)
      >>failed goal: promoter(op_terminator)
>>failed goal: promoter(_904)
>>failed goal: initiated(op)
transcription cannot initiate
>>failed goal: expressed(op)
checking other explanations
checking other explanations
```

```
goal: xcribe([],[])
no sequence given...
goal: xlate([],[])
no message to translate...
done.
```

With no promoter there is no place for the RNA polymerase to bind so the operon cannot be transcribed.

5.8 What Makes a Problem Complex?

In many domains, some interesting problems cannot be solved using heuristics alone. In the domain of bacterial operons, only problems involving prototypical operons (e.g. those whose structure corresponds to that shown in figure 3.1) with a single mutation are guaranteed to be solved correctly by the original GENEX system using the heuristics given. Some might claim that this is the fault of the system designer, for giving the wrong heuristics, or an insufficient number of heuristics.

Certainly, any given problem which is currently unsolvable could be solved by giving a new heuristic (or set of heuristics). This does not improve the performance of the system in general, but only on that one type of problem. Only if we could give the system a heuristic for every type of problem would its overall performance improve. This is not always possible, as discussed in the previous chapter.

The original version of GENEX was given heuristics to deal with operons having no more than one mutation. Supposed we tried to improve the system by giving the heuristics sufficient to deal with all pairs of mutations. Then there would still be the problem of three mutations, etc., until we find that we have created a set of heuristics whose size is the exponential of the size of our original heuristics. We have transformed the problem to one of choosing the correct heuristic–not always so easy, and possibly intractable now, due to the number of heuristics.

When model-based reasoning is used, the system creates its solution anew for each problem, based on its model of the domain, rather than relying on pre-compiled solutions. There cannot be a situation which the system cannot handle as long as it is derivable from the model. We trade off time for space[16]. Model-based reasoning has several other advantages, which were discussed in this chapter.

The problem of when to use heuristics and when to use model-based reasoning is not restricted to the case given above, i.e. which problem can be solved using *only* heuristics. For some problems heuristics may work for part of the solution, and the

[16]The ideas presented in this and the preceding paragraphs are due to Ramesh Patil, personal communication.

remainder of the problem is solved using model-based reasoning. In the solution of other problems, heuristic and model-based reasoning may be interspersed. This is an issue in expert systems: when is the use of empirical or heuristic knowledge appropriate, and when is it necessary to return to an approach based on more general principles?

As discussed in the previous chapter, heuristic rules often contain implicit preconditions which make it difficult to determine the exact conditions under which a heuristic can be applied correctly. The human expert can usually recognize a situation as being one to which a certain heuristic applies. Furthermore, a human expert can often *reduce* a complicated problem to a more simple one to which a known heuristic applies. If the rule was applied mistakenly, the expert recognizes that subsequent conditions are incorrect or inconsistent. This is certainly a topic for further study. The experiment described in this paper demonstrates the need for a system which can use both heuristics and model-based reasoning.

Chapter 6

Future Work

This chapter outlines the broad requirements and approaches for a generalized problem solving system in molecular genetics. It describes several AI mechanisms that may prove useful in developing later versions of the GENEX II program. It assumes that in later versions of the system, a major concern will be limiting the complexity of the program and the enormous amount of information with which it must deal.

6.1 Meta-Reasoning

GENEX II reasons about uncertainty by using the necessary conditions of the model it has made to assign values to the uncertain variables. In the current implementation, the necessary conditions for each model are given to the program. It should be possible for the program itself to derive these conditions, since all the necessary information is contained in the knowledge base. The following example illustrates why this is difficult.

Assume the system has decided that an operon is repressible. It does this by noting an inverse relationship between the product of the operon and some small molecule, which it assumes is the corepressor. The necessary conditions for repressibility are (1) that the repressor can bind to the operator, and (2) that the small molecule can bind to the repressor. If the knowledge base contained rules such as "an active repressor must bind at the operator to switch the operon off," and "if the operon is repressible and the small molecule is bindable to the repressor, then the repressor is active," then the derivation of the necessary conditions would be trivial. The second rule is in the knowledge base already. The first rule could be derived from existing rules by collapsing a chain of inferences, e.g. R1: an operon is not expressed if it is not initiated, R2: it is not initiated if the RNA polymerase cannot bind to the promoter, R3: a molecule cannot bind to a site if something

is already bound there, R4: the repressor binding site of the operator overlaps the promoter, and finally, R5: an active repressor binds at the repressor binding site of the operator.

Unfortunately, there is no easy way to extract these inferences from the encoding of the rules. For instance, R2 is the last clause of the rule to prove initiation can occur, and the preceding clauses are only there to restrict the variable in the call to BIND to a promoter. Many of GENEX II's rules mix clauses containing knowledge about operon structure, about the function of operon subparts, and about processes involving the operon, instead of representing each of these separately. If the different types of knowledge were represented explicitly, it would be much simpler to recognize necessary conditions by analysis of the knowledge base.

6.2 Analogy and Learning

The system could decrease searching the problem space by determining whether the observations presented fit any known mechanism. If so, the system can attempt to transfer properties from the known mechanism to the new situation. I call this hypothesis formation by analogy. Indeed, the original goal of this project was to design a system that would hypothesize about eukaryotic gene regulation based on information it had about prokaryotic gene regulation. The system would do this by attempting to draw analogies between the prokaryotic and eukaryotic systems.[1] The ability to reason by analogy would also enable the system to explain the structure and behavior of a new system by showing its similarity to an existing system.

Patrick Winston has done work in the area of learning and reasoning by analogy. In an early paper, he introduces the *transfer frame* as a means of acquiring information from simile-like statements. For example, from the statement "John is like a fox," the system would conclude "John is clever." The simile determines a *source frame* and a *destination frame*. For the example given above, *John* is the destination and *fox* is the source. The transfer frame determines which property-value pairs are allowed to pass from the source to the destination. Winston proposes several criteria for determining which properties should be transferred: those which are exhibited to an unusual degree (very high or very low), those which are deemed globally important (such as *purpose*), and those which are filled in an unusual way relative to other descriptions in the same class as the source. If more than one transfer frame is generated, the frames are filtered according to the following method: prefer transfer frames that have properties which are present in the typical instance

[1] Of course, there is no reason why the system has to be restricted to prokaryotic regulation as a source for analogy. It could use information about regulation in lower forms of eukaryotes, or even information about systems from a totally different fields.

or in some sibling of the destination, or prefer those that are in the context of the last transfer (if all else fails).

A mechanism for hypothesizing by analogy would be very useful to a molecular genetics expert system. It might use a mechanism similar to Winston's transfer frames. Every process in the prokaryotic system could have a *purpose* or *action*. Then when an explanation was needed in the eukaryotic domain, the system would look for a process in the prokaryotic domain that had a purpose or action that matched the need. The system would then attempt to match the objects in the prokaryotic domain to objects in the eukaryotic domain, possibly using *transformations* (for instance, one transformation might state that it is permissible to substitute one enzyme for another if they have similar functions) to aid the correspondence. The role of some object in the prokaryotic domain might be played by several objects in the eukaryotic domain; in this case the system would have to perform some transformation that would enable it to think of the group as a single entity for the purpose of translating the action of the mechanism.

Research by Tom Mitchell in the areas of learning and problem solving also has application to the area of hypothesis formation by analogy. In particular, his recent work on generalization (wherein the problem solver attempts to formulate a heuristic given an application of an operator to a rule) describes a means by which the problem solver can create new terms to use in defining its heuristics. Generalization would be used when formulating analogies to produce the transfer frames described above.

The new work by Mitchell is particularly interesting because it claims to give the program the capability to formulate new relationships, instead of being restricted to those about which it has been told. The program creates new terms by propagating and combining constraints on the use of a sequence of operators. The domain of Mitchell's program, called LEX, is integral calculus, so the operators consist of approximately 50 transformations that can be applied to integrals, such as:

$$\int rf(x)dx \longrightarrow r\int f(x)dx$$

where r indicates a real number.

Mitchell gives the following example in which the program could create the new term *odd integer*. Consider the solution path shown in figure 6.1.

State5 is a polynomial raised to some *integer* power. By propagation of this constraint backwards through the sequence of operators used, the program could determine that state1 must match an instance of $\cos(x)dx$, where c is constrained to satisfy the predicate $real(c) and\ integer((c-1)/2)$, which is the definition of *odd integer*.

Using Mitchell's method, the new terms are created out of terms which appear in the operator definitions. There seems to be an *essential set* of terms which must be

$$
\begin{array}{ll}
state1: & \int \cos^7(x)dx \\
\downarrow & op1: f^r(x) \Rightarrow f^{r-1}(x)f(x) \\
state2: & \int \cos^6(x)\cos(x)dx \\
\downarrow & op2: f^r(x) \Rightarrow (f^2(x))^{\lfloor r/2 \rfloor} \\
state3: & \int (\cos^2(x))^3\cos(x)dx \\
\downarrow & op3: \cos^2(x) \Rightarrow (1-\sin^2(x)) \\
state4: & \int (1-\sin^2(x))^3\cos(x)dx \\
\downarrow & op4: \int g(f(x))f'(x)dx \Rightarrow \int g(u)du, u = f(x) \\
state5: & \int (1-u^2)^3 du, u = \sin(end)
\end{array}
$$

Figure 6.1: Solution path for $\int \cos^7(x)dx$

predefined before any new terms can be created. For LEX's domain, this set would include integer, real number, function, etc. Any new terms which the system creates must derive from the basic set of terms provided. If a potential new term is truly novel, i.e. not derivable in whole from any pre-existing definitions, the program will be unable to define it. There is not even a way to ensure that a complete set of essential terms has been provided, for the same reason. This would limit the power of a hypothesis-forming system if all the essential terms have not yet been defined. However, even a combination of previously-known terms, such as Mitchell's method could generate, may provide new insight into a problem.

6.3 Abstraction Spaces and Multiple Levels of Representation

An abstraction space is a simplification of a given problem space in which unimportant details are ignored. After a problem is solved in the abstraction space, all that is needed to complete the solution is to account for those details in the original problem space, if this is possible. A hierarchy of abstraction spaces can be constructed if the system is able to discriminate among several levels of detail. Objects appearing in the highest abstraction level are those most critical to the solution of the problem. Other objects appear in lower levels as they become necessary to fill in the gaps in the partial solutions created in the preceding levels. If a problem cannot be solved in a given abstraction space, control reverts to a higher abstraction space. The problem solver eliminates the troublesome step and attempts to complete a new plan starting at that point in the higher space. Only then does control return.

to the lower space.

In his 1974 paper on ABSTRIPS, Sacerdoti introduced the concept of a hierarchy of abstraction spaces. ABSTRIPS solves a problem in one abstraction space before beginning to plan in a lower space. In this way, steps that don't lead to the problem solution or that yield very inefficient plans can be eliminated early, before a great deal of effort has gone into planning the details of their execution. This saves the problem solver a lot of wasted work. The difficulty lies in determining which aspects of a problem are important and which are details.

The concept of multiple levels of representation is complementary to abstraction spaces. If the user presents the system with a problem, the system would first try to solve it using the least detailed descriptions possible. If the details are not needed, they will only complicate the problem-solving process. If they are needed, then the system can always choose to use them. The system would not always use the least detailed representation it has available, but it would use the least amount of detail that is sufficient to solve the problem. If the user asks for an elaboration of a solution, it might become necessary for the system to use a more detailed representation.

Ramesh Patil's 1981 work on electrolyte and acid-base disorders examined multiple levels of representation. Patil argued that knowledge represented at several levels of detail is necessary for complex reasoning. Patil implemented a system which used a multi-level model for representing diseases and causal phenomena. The deepest level consisted of pathophysiological knowledge about disease, while higher levels contained more syndromic knowledge. Patil states

> The aggregate syndromic knowledge provides us with a concise global perspective and helps in the efficient exploration of diagnostic alternatives. The physiological knowledge, on the other hand, provides us the capabilities of handling complex clinical situations ..., evaluating the physiological validity of the diagnostic possibilities being explored, and organizing a number of fragmented and seemingly unrelated facts into a coherent causal description [2].

Each level of description can be viewed as a collection of nodes, each representing some state of a physiological parameter, and links, representing the relationships between different nodes. Some nodes can be defined in terms of a network of states, called the elaboration structure, at the next lower (more detailed) level of description. The multi-level description scheme also allows a single link in a higher level to represent a chain of links in a lower level. These are useful mechanisms for controlling the amount of detail that is dealt with at any given time.

[2]Patil 1981, p. 44.

Three levels of representation – *genetic*, *molecular*, and *atomic* – should be sufficient for an expert system in molecular genetics. For example, the action of enzymes can be understood on several levels. On the least detailed level, the function of an enzyme is to catalyze the reaction of two chemicals (called *substrates*). If both substrates are present, it is assumed that the reaction will take place. On a more detailed level, the way an enzyme works is that each of the substrates binds to the enzyme, and then since they are physically close to each other, the reaction is more likely to occur. Thus the preconditions for the enzyme working at this level of detail are not only that the substrates are present, but that they are able to bind to the enzyme (i.e. their binding site is not blocked by anything). On the atomic level the conformations of all three elements involved and the thermodynamics of the reaction are important [3]. In attempting to explain some problem involving enzymes, the system will first use the least detailed model, and if it can complete the solution on that level, it will attempt to work out the solution in the more detailed model. GENEX II currently makes models and does simulation on the genetic level, except in the case of the BIND operation which is simulated on the molecular level. This is because GENEX II has no means of switching between levels, and most interesting events (i.e. those that the system would want to describe in more detail) occur during the bind operation.

PEPTIDE (Weld, 1984) is a system which predicts genetic activity by simulating a combination of five basic processes (BIND, FOLD, SLIDE, DROP, and REACT). The five processes are discrete; "they either happen completely or not at all." A continuous process is generated from a repeated cycle of discrete processes. PEPTIDE can determine what will happen when a continuous process is run to the limit, i.e. until "a currently active discrete process will stop or an inactive one will start." For instance, PEPTIDE would represent transcription as a BIND operation, followed by a continuous simulation of a REACT-SLIDE cycle, followed by a DROP.

PEPTIDE represents processes on the molecular level described above. This level of detail is suitable for simulating a subpart of some larger process within the cell. It would not be appropriate for GENEX II to solve problems entirely on this level [4] for several reasons. First, the molecular level often is too finely-grained to produce a satisfactory solution. Consider a problem in which a molecule is bound to DNA and blocks the movement *along* the DNA strand of an enzyme such as RNA polymerase. The exact location of binding is probably not essential to the solution of the problem; the general region of the DNA strand (e.g. transcribed region) is.

[3] Much of the information required to model molecular genetics on the atomic level is still unknown in the domain.

[4] but note that Weld never claims that PEPTIDE should be used to solve any entire real problems.

A program which reasoned *only* on the molecular level might have to simulate the consequences of blocking the RNA polymerase at every one of 1000 specific sites along the DNA strand [5].

The second reason why it is difficult to reason entirely on the molecular level is that there is not enough fundamental knowledge about the physical chemistry of these processes to predict the interaction of two macromolecules. Therefore any solution reached by simulation alone would have to simulate every possible interaction.

Finally, many of the heuristics used by molecular biologists (and by GENEX II's model-maker) to select a likely model for the operon's behavior deal with objects on the genetic level. A higher-level view of the system is needed to reduce the complexity of the solution space, and also to resolve the potentially conflicting results of the multiple simulations. The lower-level view would allow the system to elaborate on its solutions.

6.4 Constraints

Constraints can sometimes be used to cut off a particular branch of speculation. Suppose the system hypothesized that a protein was preventing DNA from being transcribed by binding to the DNA and blocking the binding site for RNA Polymerase II. The typical binding site is between 15 and 40 nucleotides in length, so the binding site of the protein would have to be approximately that size to cover it. Now perhaps the system had previously constrained the protein to be smaller than this, to support the hypothesis that the protein got into the nucleus by passing through the nuclear membrane. These two constraints are not simultaneously satisfiable, so the system can abandon this line of reasoning.

Constraints are a very common device in artificial intelligence reasoning systems. MOLGEN (Stefik, 1981), a program that plans gene cloning experiments in molecular genetics [6] uses constraints to coordinate separate subproblems because MOLGEN breaks up a problem into subproblems that are solved independently. Constraints that represent the interactions of these subproblems ensure that the various solutions interface correctly. A typical constraint might state that the endonuclease used in step six should not cut the gene inserted in step two. MOLGEN

[5] PEPTIDE shifts its focus to the larger scale in this instance, too: its "qualitative representation of quantities (including positions on a chain) only contains information like 'a repressor is bound between the promoter and the gene.' "

[6] In addition to the work reported by Stefik, there has been a considerable amount of additional work done in the areas of representing the knowledge needed to do planning in molecular genetics, building up knowledge bases (Friedland, 1982), and experiment planning.

identifies three operations on constraints: formulation, propagation, and satisfaction. Constraint formulation is the adding of new constraints as the design process progresses. Usually, the constraints become more detailed during the solution process. Constraint propagation brings together constraints from the separate subproblems. MOLGEN uses a least commitment strategy–it avoids instantiating variables in the constraints for as long as possible. This gives it the greatest number of options for constraint satisfaction, finding values that satisfy the constraints on the problem.

6.5 Representation

There are two problems of knowledge representation that must be considered in any expert system: how the knowledge is represented internally and how the knowledge is represented for the user. In a molecular genetics expert system, the internal representation must facilitate the representation of several kinds of knowledge: facts about molecular biology (which can be as diverse as strings of letters representing nucleotide sequences or a procedure that encodes the behavior of a polymerase), knowledge about processes and how to reason about them, and knowledge of how to formulate hypotheses.

As discussed in the section on meta-knowledge, GENEX II's reasoning power is restricted by a representation which does not distinguish between different types of knowledge (about structure, processes, and interactions). A knowledge representation system proposed by Brian Smith in 1978 seems suited to encoding knowledge about the objects and operations of molecular genetics. The representation was based on a division of knowledge between the "anatomical" (how is x structured?) and "physiological" (how does x work?). The representation could also be used to encode processes. Smith classified potential questions that might be asked of the system into requests for information (e.g. "what is X?" and "how do X and Y relate?"), verification of suggestions ("is it true that ...?"), and hypotheticals ("what would happen if ...?").

Smith's proposed knowledge representation system seems to clarify the distinction between the different types of knowledge in the system. The capability to answer the above types of questions could serve as the framework for the hypothesis-forming mechanism in a molecular genetics expert system.

A representation such as the one described by Smith would also facilitate the representation of spatial and quantitative information.

How an expert system's knowledge is represented to the user is a serious consideration. I believe that new domain information should be entered by the experts themselves. Molecular biology knowledge that must first pass through the com-

puter scientist is subject to possible oversimplification, misinterpretation and plain errors. A related problem is that in a field where new information is being discovered rapidly, the knowledge base of a system such as the one proposed can easily become obsolete. Facilitating addition of new information by the experts themselves should be a primary goal of the knowledge-acquisition mechanism of a molecular genetics expert system.

6.6 Control Structure

In a system as complex as this, there is an advantage to separating solving the problem from planning the strategy of how to solve the problem. This is called a *layered control structure.*

Stefik described a layered control structure that separated decisions about planning steps in the laboratory from decisions about what strategy to use in planning the problem solving process. MOLGEN divided the problem solver into three levels, called planning spaces.

The lowest level, the domain space, in a molecular genetics system contains knowledge about the objects and operations inside the nucleus of a cell. The operations are actions that would be carried out in a real cell (for example, REPLICATE). The domain space is not a control level. The GENEX II knowledge base corresponds to MOLGEN's domain space.

The design space is a control level that contains knowledge about how to solve a problem. The operators (such as PROPOSE-OPERATOR and PROPAGATE-CONSTRAINT) create and arrange steps in domain space.

The highest control level is strategy space. The strategizer controls the design operators that solve the problem. It contains operators such as USE-ANALOGY and USE-MORE-DETAIL. This level also controls the use of abstraction spaces. The model-maker in GENEX II is a simplified version of MOLGEN's strategy space. If GENEX II encoded some expert techniques for formulating hypotheses, these would be used at the strategy level.

An advantage of this approach is that by dividing the different levels of the planning process into separate domains, the complexity of the problem solving process is reduced. Another advantage is that being able to reason about the planning process makes the problem solver better at setting priorities and scheduling subgoals.

6.7 The Role of Complexity Reducing Mechanisms in Genex

It would be useful for a molecular genetics expert system to include features from the systems described above. I assume that the knowledge base will be sufficiently complex that attempting to solve the problem in a very detailed fashion right away is not efficient–there are just too many possibilities at each point in the process. Rather, a molecular genetics expert system would have to work out the broad outlines of the solution first, and add the details gradually. I think that this is the way that human experts work, too. Such a system would use techniques from truth maintenance to keep track of what things it believed to be true while it examined different possible explanations for a problem. None of the systems described above contain all the mechanisms that the molecular genetics system should incorporate. For instance, neither ABSTRIPS nor MOLGEN use the idea of truth maintenance which would allow them to reason about a projected course of action or events. This is necessary to examine the consequences of proposed hypotheses. ABEL has the limitation that all relationships between states are represented by causal links. This is not sufficient to represent all possible interactions. For example, it has been observed that regions of DNA that are hypermethylated tend not to be transcribed. It is not known whether the relationship between hypermethylation and reduced gene activity is causal, but in Patil's system we would have no choice but to represent it as such.

Issues that should be addressed when designing a molecular genetics expert system include:

- What knowledge do experts use to select the most promising refinements (and rule out the least likely) at each step in the problem solving process.
- How to control the amount of detail. When do you need more detail and when do you need to step back and look at the larger picture?
- How to limit the search space of possible hypotheses. Can some possibilities be eliminated *a priori*, and if so, how do you identify them? How to "zero in" on an answer.
- How can analogies help in formulating hypotheses?

6.8 Analyzing Laboratory Data

It may become desirable to extend GENEX II so that it would be capable of understanding biological data; e.g, interpretation of gel patterns and enzyme activity

studies, etc. While the inability to interpret laboratory data does necessarily prevent GENEX II from solving a given problem, it makes it necessary for the problem to be pre-processed into a form that GENEX II will understand. This is time-consuming and might limit the system's usefulness.

In order to analyze laboratory data, GENEX II must have the ability to do macro-level reasoning, as described above. Weld (1984) introduces a process called aggregation which could give GENEX II this capability. As Weld describes it,

> The system starts by simulating the effects of discrete processes until it recognizes a cycle...where all discrete processes are repeated. It then aggregates the trace of a single iteration of the cycle of discrete process executions, producing a continuous process.

Although Weld demonstrates aggregation on his molecular-level operations, there appears to be no reason why the same technique could not be applied to GENEX II's genetic-level operations, so that the system could reason about approximate amounts of gene product made, etc. Some technique for reasoning about the results of aggregation is necessary for GENEX II to do macro-level reasoning. For example, the aggregation of a continuous process of DNA replications would result in all the resulting DNAs being identical, if the original replication (the one that started the cycle going) produced two identical strands. Given a DNA molecule with both strands methylated [7] and if told that there was no maintainance methylase, the system could correctly conclude that the net result would be unmethylated DNA, since the original replication would produce one methylated and one non-methylated strand in each copy. The actual implementation of Weld's aggregation system in GENEX II would not require major system redesign, and is a possible future improvement to the system.

[7] having methyl groups attached to the cytosine residues.

Chapter 7

Conclusion

In this paper we examined two programs, both designed to solve the same types of problems in the same domain. The first program used compiled knowledge and empirical associations, and the second, model-based reasoning. The first program, GENEX I, performed well, but because the domain knowledge was compiled into the rules, the knowledge base was difficult to extend, and the system could not access the knowledge to produce explanations. In the second system, GENEX II, the domain knowledge is represented explicitly, and is thus easily extensible and available for generating explanations.

Because rules *match* the current problem against a set of predetermined situations, the number of problems that GENEX I could solve was limited to those which match the situations predetermined by the rules. In the GENEX II, the system creates possible models of the current problem and tests each one for consistency with the observed behavior of the operon [1]. Theoretically, any problem derivable from the model can be handled by the second program.

As always, there are tradeoffs. GENEX II required a more complicated control structure to reduce the amount of search. Its more detailed knowledge created more steps in the inference chains, resulting in a slower-executing program. The performance of the system would be improved if it could use both heuristic and model-based reasoning.

The GENEX programs demonstrate that it is possible to apply artificial intelligence successfully to the solution of certain types of problems in molecular genetics. Problems in molecular genetics are complicated by uncertainty introduced when reasoning about conformations. Although a definitive solution to this problem must await advances in protein chemistry, GENEX II can reduce the number of possible solutions that must be verified by formulating likely models from the behavior of

[1] This method is known as *generate and test.*

the system it is examining.

GENEX II now reasons about bacterial operons, a small subpart of molecular genetics. If a future version of GENEX is to represent other systems, such as eukaryotic operons, and use more of the available knowledge in the field, it will require the use of the AI techniques discussed here in order to reduce the complexity of the system.

Chapter 8

References

1. Barnett, J. Some Issues of Control in Expert Systems. Proceedings of the International Conference on Cybernetics and Society (1982).

2. Clocksin, W.F. and C.S. Mellish. Programming in Prolog. (Springer-Verlag: 1981).

3. Davis, R. Applications of Meta-Level Knowledge to the Construction, Maintainance, and Use of Large Knowledge Bases. Stanford University Artificial Intelligence Laboratory Technical Memo 283 (1976).

4. Doyle, J. A Glimpse of Truth Maintenance. MIT Artificial Intelligence Laboratory Technical Memo 461-a (1978).

5. Feigenbaum, E., B. Buchanan, and J. Lederberg. On Generality and Problem Solving: A Case Study Using the DENDRAL Program. In *Machine Intelligence 6*, B. Meltzer and D. Mitchie, eds. (American Elsevier, New York:1971).

6. Friedland, P. et al. GENESIS: a knowledge-based genetic engineering simulation system for representation of genetic data and experiment planning. Nucleic Acids Research, vol. 10, no. 1, (1982).

7. Hood, L., Wilson, J., and Wood, W. Molecular Biology of Eukaryotic Cells (W. A. Benjamin, Menlo Park:1975).

8. Koton, P. A System to Aid in the Solution of Problems in Molecular Genetics. SM Thesis, MIT Laboratory for Computer Science, May 1983.

9. Kunz, J. A Physiologic Rule-based System for Interpreting Pulmonary Function Test Results. Stanford University Computer Science Dept Heuristic Programming Project Report HPP-78-19 (1978).

10. Lewin, B. Gene Expression (John Wiley and Sons, New York:1980).

11. Mitchell,T. M. Learning and Problem Solving. Proceedings of the Eighth International Joint Conference on Artificial Intelligence, 1983.

12. Patil, R. S. Causal Representation of Patient Illness for Electrolyte and Acid-Base Diagnosis. MIT Laboratory for Computer Science TR-267 (1981).

13. Rosenberg, M. and Court, D. Regulatory sequences involved in the promotion and termination of RNA transcription. Ann. Rev. Genetics 13:319-353 (1979).

14. Sacerdoti, E. Planning in a Hierarchy of Abstraction Spaces, Artificial Intelligence, vol. 5, no. 2, (1974).

15. Stefik, M. Planning with Constraints (MOLGEN: Part 1), Artificial Intelligence, 16:111-140 (1981).

16. —. Planning and Meta-Planning (MOLGEN: Part 2), Artificial Intelligence, 16:141-170 (1981).

17. Smith, B. A Proposal for a Computational Model of Anatomical and Physiological Reasoning. MIT Artificial Intelligence Laboratory Memo 493 (1978).

18. Stewart, A. and Hunt, D. The Genetic Basis of Development (Blackie and Son, Ltd. Glasgow:1982).

19. Sussman, G.J. A Computer Model of Skill Acquisition. American Elsevier (New York: 1975).

20. Swartout, William R. Producing Explanations and Justifications of Expert Consulting Programs. MIT Laboratory for Computer Science Technical Report 251 (1981).

21. Watson, J. D. Molecular Biology of the Gene, 3rd ed. W. A. Benjamin (Menlo Park:1970).

22. Weld, D. S. Switching Between Discrete and Continuous Process Models to Predict Genetic Activity. SM Thesis, MIT Artificial Intelligence Laboratory, May 1984 (pending).

23. Winston, Patrick H. Learning by Creating Transfer Frames, Artificial Intelligence 10:147-172 (1978).

24. —. Learning and Reasoning by Analogy: The Details. MIT Artificial Intelligence Laboratory Memo 520 (1980).

25. —. Learning New Principles from Precedents and Exercises: The Details. MIT Artificial Intelligence Laboratory Memo 632 (1981).

26. —. Learning by Augmenting Rules and Accumulating Censors. MIT Artificial Intelligence Laboratory Memo 678 (1982).

27. Yanofsky, C. Attenuation in the control of expression of bacterial operons. Nature 26:751-758 (1981).

Appendix A

Details of the Genex I Program

A.1 Data Structures

The primary data structure in GENEX I was a two-element list whose first element was an atom representing the physical structure of the operon being examined and whose second element represented the product of the operon (i.e. the protein coded for by the structural genes). Properties of the structure and gene product were attached to the property list of the corresponding element. This representation was chosen purely for the sake of expediency.

One property of the operon structure that GENEX I used is its sequence. GENEX I stored the sequence of only one strand (preferably the one that is transcribed, if this is known) since the sequence of the complementary strand can easily be obtained from the known sequence. The sequence was stored as a list of characters (either A, T, G, or C). The reason for storing the sequence as a list rather that as a single multi-character atom (i.e. as "(A T T G C)" rather than as "ATTGC") was to facilitate access to and manipulation of the individual nucleotides of the sequence.

The structure property list also contained the properties `INDUCIBLE` (which has value *t* if the operon is inducible) and, similarly, `REPRESSIBLE`. The identity of the inducer or corepressor was stored under the property `INDUCER` or `COREPRESSOR`.

Other properties of the operon which might have been stored on its structure property list include its tertiary structure (e.g. the location of any hairpin loops or cloverleaves), the location of any AT-rich or GC-rich segments, and the location of any repeat sequences.

Properties of the gene product that GENEX I used are its purpose (i.e. the function it serves for the organism, usually either synthesis or digestion), its substrates, its end-product (if the gene product is used in a synthesis reaction), and whether production of the gene product is influenced by the presence of some other sub-

stance in the cell (either proportionally or inversely). This information was used to determine the control structure of the operon–if it is inducible or repressible, what the inducer or co-repressor is, whether or not it is subject to attenuation. Not all these properties are applicable to all operons.

If the program determined that it should examine the regulatory gene of the operon, it constructed a representation for the regulatory gene. This was very similar to the representation of an operon. It was a list with the form (*operon-name*-REGULATORY-GENE *operon-name*-REGULATORY-PROTEIN). The sequence of the regulatory gene could be stored and used in the same way as that of an operon. The original GENEX did not use any properties of the regulatory protein. In any case, they would not have been the same properties as those used for the operon product. The latter were used to determine control structure, and since regulatory genes are not themselves regulated, those properties do not apply. A property of the regulatory protein which might have been used would be some aspect of its quaternary structure (e.g. monomeric or dimeric). This would have assisted the program in determining how the regulatory protein interacts with the operator.

A.2 System Variables

Before running the GENEX I program, the user had to set up certain variables that the program used. First, the program had to be given the operon, in the form of a two-element list as described above. Any properties of the operon discussed above might also have been given. The variable THINGS-PRESENT, which is a list of the substances present in the medium, was set. Finally, the user ran the INITIALIZE program. This asked the user if the gene product was made and whether there were any known mutations. It used the responses to set the following global variables:

PROTEIN-MADE. This has value *t* if the operon's product is made, otherwise it has the value *nil*.

INCREASED-AMOUNT-PROTEIN, DECREASED-AMOUNT-PROTEIN. If PROTEIN-MADE is *t*, the user is asked if it is made in a greater or lesser amount than normal. If made in a greater amount, INCREASED-AMOUNT-PROTEIN is set to *t*. If lesser, DECREASED-AMOUNT-PROTEIN is set to *t*, but PROTEIN-MADE is set to *nil*, since the causes of a decrease in amount made are, in general, the same as the causes of no production at all.

KNOWN-MUTATION. This has value *t* if there is a known mutation in the operon or regulatory gene, *nil* if it is definitely known that there is no mutation, and has the value unknown otherwise.

MUTATION-LOCATION. If KNOWN-MUTATION is *t*, this can be set to indicate a mutation in the promoter, operator, or transcribed region of the operon, a mutation in

the operon whose exact location is unknown (a "*linked*" mutation), or a mutation in the regulatory gene (an "*unlinked*" mutation).

Finally, if the system was told about the behavior of pseudodiploids,[1] it used this information to help determine whether the system is under positive or negative control. Having determined this, the INITIALIZE program could set the variables PC (positive control) or NC (negative control) or both to *t*.

A.3 Control Structure of the Genex I Program

All of the procedures described below are called with one argument, which is the operon being examined, unless otherwise stated.

The user started up the system by invoking the genex program. This program passed control to one of the procedures XPLN-NO-PROTEIN (if PROTEIN-MADE is *nil* or if DECREASED-AMOUNT-PROTEIN is *t*) or XPLN-INC-PROTEIN (if PROTEIN-MADE is *t*).

XPLN-NO-PROTEIN divided the solution of the problem into three parts, corresponding to the procedures it called: transcription-related causes (examined in the procedure XC-BUG), translation-related causes (examined in XL-BUG), and post-translational causes [2]. XPLN-INC-PROTEIN considered two possible ways to explain the behavior of the operon, one in terms of transcription (again using the procedure XC-BUG), and the other in terms of attenuation, using the procedure ATTENUATOR-BUG.

XC-BUG checked to see if the behavior of the operon could be explained by some mechanism of transcription. It called the two procedures INITIATION-BUG and CHECK-ATTENUATION.

The CHECK-ATTENUATION procedure encoded the knowledge that the system had about attenuation of transcription. If the sequence of the operon was not given, the procedure could go no further than suggesting the possibility of attenuation in operons involved in the synthesis of amino acids. If the leader sequence was given, however, a procedure called FIND-ATTEN-REG was called by CHECK-ATTENUATION. This procedure translated the nucleotide sequence into amino acids and could identify a likely attenuator, if one exists. The criterion used for an attenuator was the occurrence of five codons for the end product amino acid in a sequence of eight adjacent codons following a start codon. This procedure would correctly identify

[1] A pseudodiploid is an operon of which some parts are represented twice.

[2] GENEX had no information about post-translational control, but this could have been added in a new procedure at some later date, and it would have been called from the XPLN-NO-PROTEIN procedure.

the attenuator region in five of the six known attenuated operons [3].

The ATTENUATOR-BUG procedure called by XPLN-INC-PROTEIN used the FIND-ATTEN-REG procedure to check whether the increased protein production might be due to a deleted attenuator region. It did this by comparing the original sequence with the mutated sequence of the operon.

Effects which occur at the time of initiation of transcription were divided by the INITIATION-BUG procedure into an enumeration of promoter-related control (in the procedure PROMOTER-BUG), operator-related control (in the procedure OPERATOR-BUG), control due to the presence or absence of the small molecule (examined in the procedure CHECK-REPRESSOR), and solutions which could be attributed to some positive control element (examined in the procedure POSITIVE-CONTROL-BUG.

The XL-BUG procedure was responsible for reporting the possible effects of mutations in the translated region of the operon (or, if the system was examining the regulatory gene, mutations in that gene). If the sequence of the operon was known, the CHECK-XL-MUTATION2 procedure was called which compared the original sequence with the mutated sequence, looking for nonsense and missense mutations.

The system examined the regulatory gene when the variable MUTATION-REGION was set to *unlinked*, or when the variable KNOWN-MUTATION was set to unknown. The procedure REG-GENE-MUTATION, which was called by XL-BUG and XPLN-INC-PROTEIN, examined the effect of regulatory gene mutations on the production of the regulated protein.

CHECK-REPRESSOR, which was called by INITIATION-BUG as noted above, in turn called the procedures INDUCIBLE? and REPRESSIBLE? which determined if an operon is inducible or repressible, and INDUCER? and COREPRESSOR? which determined whether a given substance is the inducer or corepressor of the operon. If the inducer or corepressor of the operon was not given by the user, CHECK-REPRESSOR attempted to determine what the inducer or corepressor is, using the procedures FIND-INDUCERS and FIND-COREPRESSORS.

CHECK-REPRESSOR also called the procedure MUTATED-REPRESSOR. This procedure recursively invoked GENEX using the regulatory gene as the argument. CHECK-REPRESSOR reset many of the system variables to maintain consistency during the recursive call. The variable **protein-made**, which referred to the operon's product before the call, was stored in the variable **regulated-protein-made** for the duration of the recursive call, while **protein-made** referred to the regulatory protein, the product of the regulatory gene. Since the system did not know whether or not the regulatory protein was made, CHECK-REPRESSOR called GENEX twice, once with

[3] It would not recognize the *trp* attenuator, which contains only two adjacent *trp* codons. However, tryptophan is a rare amino acid, usually occurring only once in every 100 amino acids. An improved version of this procedure would take into account the relative frequencies of the amino acid residues when deciding how many adjacent residues constitute an attenuator.

protein-made set to *t* and once with it set to *nil*. mutation-location, which previously held the value *unlinked*, indicating a mutation outside the physical boundaries of the object under consideration, was reset to *linked*, indicating a mutation in the object currently being examined. There was no chance of CHECK-REPRESSOR getting caught in an infinite loop since regulatory genes are not themselves represented as having regulatory genes.

The procedure POSITIVE-CONTROL-BUG examined two mechanisms of positive control: the cAMP-CAP complex, using the procedure CAP-BUG; and activators, using the procedure CHECK-ACTIVATOR. CHECK-ACTIVATOR was very similar to CHECK-REPRESSOR, and called all the same procedures.

A.4 Knowledge Encoding in Genex I

This section gives some examples of how the information presented in the previous chapter was encoded into the GENEX I program.

A.4.1 Mutations

GENEX I was guaranteed to give a correct solution only for problems involving a single mutation in the operon or its regulatory gene [4].

There are several aspects of a mutation that are important with regard to its effect on gene expression.

- Is the mutation located in the operon itself or in its regulatory gene?
- If the mutation is in the operon, in what region–promoter, operator, transcribed region, or translated region?
- Does the mutation increase or decrease gene expression?
- Is the mutant dominant or recessive in a diploid?

GENEX I used this information to help build a model of the operon's behavior. Of course, not all this information is always available to the program.

GENEX I had rules (encoded as decision trees) to help it model the behavior of the operon when a mutation is present. For example, it had a rule that says that dominant operator mutants cause constitutive synthesis in PC systems and are uninducible in NC systems.

[4] By "single mutation," I do not mean a point mutation, but rather a mutation in one region of the operon. Thus GENEX I cannot reason about the combined effect of mutations in both the regulatory gene and the operator, for example.

If the nucleotide sequence of the operon was given, GENE I gave a very specific analysis of the cause of the operon's behavior. It would identify a mutation in the leader sequence that affects attenuation. If the mutation was in the promoter, GENEX I would compare the mutated sequence and normal sequence with the template sequence, checking for the better match (since promoter strength correlates with how well the promoter matches the template). If it determined that the mutation was in the translated region of the gene, the CHECK-XL-MUTATION procedure would compare the mutated sequence with the normal one to find any nonsense or missense mutation.

A.4.2 Determining Positive and Negative Control

Since most known operons function under negative control, GENEX I assumed that the input operon is subject to negative control and tried to model the behavior using that assumption. It had rules for recognizing when an operon is also under positive control: if the operon codes for enzymes to digest a sugar other than glucose (in which case it is under the control of the cAMP-CAP complex), or if the deletion of the regulatory gene makes the protein uninducible. The behavior of pseudodiploids could also cause the program to consider positive control.

A.5 Another Example

Bacterium pedantia **produces a protein, sporulin, that makes other bacteria sporulate...Sporulin synthesis is inducible by chalk. In the absence of chalk a repressor produced by the gene *spoR* prevents expression of the Sporulin structural gene *spoS*.**

This is what the program has been told about the Sporulin gene:

```
spo
(SPO-GENE SPORULIN)

(plist 'spo-gene)
(INDUCER CHALK INDUCIBLE T)
```

Two *spoR* mutations are available. Strains carrying the mutation *spoR*1 secrete sporulin whether or not chalk is present. Strains carrying *spoR*2 fail to produce Sporulin even when chalk is present. One of the two mutations is known to be missense, and the other is known to be nonsense. Which mutation is the missense one, and why?

First give GENEX I *the phenotype of the spoR1 mutation.*

```
(initialize spo)
is SPO-GENE product made? YES
in what amount?  (increased or decreased) INCREASED
is there a mutation in SPO-GENE? (yes, no, unknown) NO
is there an unlinked mutation? (yes, no, unknown) YES
was diploid constructed? NO
DONE

(GENEX spo)
is SPO-GENE a regulatory gene? NO
assuming no misfunction at SPO-GENE promoter
assuming SPO-GENE can be repressed
examining SPO-GENE-REGULATORY-GENE
Is the regulatory protein made? (yes, no, unknown) UNKNOWN
assume SPO-GENE-REGULATORY-PROTEIN is not made
possible mutation at SPO-GENE-REGULATORY-GENE promoter preventing RNA
        Pol from binding
operon cannot be switched off without SPO-GENE-REGULATORY-PROTEIN
QED
```

GENEX I *claims that if the regulatory protein is not made, i.e. if spoR1 is a nonsense mutation, the operon cannot be switched off, which explains the constitutive behavior of the Sporulin gene.*

Now give the phenotype of the spoR2 mutation.

```
(setq things-present '(arabinose lactose chalk))
(ARABINOSE LACTOSE CHALK)
```

We tell it that chalk is present since strains with the spoR*2 mutation don't produce sporulin even in the presence of the inducer.*

```
(initialize spo)
is SPO-GENE product made? NO
is there a mutation in SPO-GENE? (yes, no, unknown) NO
is there an unlinked mutation? (yes, no, unknown) YES
was diploid constructed? NO
```

```
DONE

(genex spo)

is SPO-GENE a regulatory gene? NO
assuming no misfunction at SPO-GENE promoter
assuming SPO-GENE can be repressed
SPO-GENE is active in presence of CHALK
examining SPO-GENE-REGULATORY-GENE
Is the regulatory protein made? (yes, no, unknown) UNKNOWN
assume SPO-GENE-REGULATORY-PROTEIN is not made
now assuming SPO-GENE-REGULATORY-PROTEIN is made
possible mutation in SPO-GENE-REGULATORY-GENE  promoter causing
        superpromoter
possible structural mutation in SPO-GENE-REGULATORY-PROTEIN preventing
        inducer from binding
done examining SPO-GENE-REGULATORY-GENE
```

GENEX I *found no reasons for Sporulin not to be made if the regulatory protein is not made (implying that the mutation cannot be nonsense). It found two types of mutations in spoR that might cause Sporulin to be made in decreased amounts or not at all. One is a missense mutation that prevents the repressor from binding the inducer (possibly by eliminating the inducer binding site) which would prevent the repressor from attaining an inactive conformation. The other mutation it proposes is in the spoR promoter causing greater amounts of the regulatory protein to be made (a so-called "superpromoter" because it has a higher affinity for RNA polymerase). This would generally not be an explanation for the observed phenomenon unless there was only a limited amount of inducer present.*

The rest of the examination revealed no other possible explanations and is omitted.

Appendix B

Genex II Program Listing

This chapter contains the GENEX database as well as the code for the GENEX interpreter and the front end. Clauses encoding domain knowledge are expressed as arguments to a binary function called **genex_clause** to facilitate the use of the interpreter. For example, if the original Prolog clause encoding a piece of domain knowledge was

```
rule1(X,Y) :- a(X),b(Y).,
```

it would be written as

```
genex_clause(rule1(X,Y),[a(X),b(Y)]).
```

to run under the GENEX interpreter.

```
genex_clause(nt(t),[]).
genex_clause(nt(g),[]).
genex_clause(nt(c),[]).
genex_clause(nt(u),[]).

genex_clause(complement(a,t),[]).
genex_clause(complement(g,c),[]).
genex_clause(complement(t,a),[]).
genex_clause(complement(c,g),[]).

%facts about operons%
genex_clause(part_of(Operon, Promoter),
                              [operon(Operon), promoter(Promoter)]).
genex_clause(part_of(Operon, Operator),
                              [operon(Operon), operator(Operator)]).
```

```
genex_clause(assoc(Operon, Reg_gene),
                          [operon(Operon), reg_gene(Reg_gene)]).
genex_clause(part_of(Operon, Struc_genes),
                           [operon(Operon),
                           setof(X, structural_gene(Operon,X),
                           Struc_genes)]).
genex_clause(part_of(Operon, T),
                           [operon(Operon), terminator(T)]).

genex_clause(ntseq([]),[]).
genex_clause(ntseq([X|Y]),[nt(X),ntseq(Y)]).

genex_clause(enzyme(rna-pol),[]).
genex_clause(enzyme(dna-pol),[]).
genex_clause(protein(X),[enzyme(X)]).
genex_clause(part_of(Enz,active_site1(Enz)),[enzyme(Enz)]).
genex_clause(part_of(Enz,active_site2(Enz)),[enzyme(Enz)]).
genex_clause(binding_site(Mol, active_site1(Enz)),
                             [substrate1(Enz,Mol)]).
genex_clause(binding_site(Mol, active_site2(Enz)),
                             [substrate2(Enz,Mol)]).

genex_clause(promoter(P),
        [sequence(P,Seq),\+ sequence(P,[]),
        length(Seq,N),
        approx(N,40),
        contains_subseq(P,[t,t,g,a,c,a]),
        contains_subseq(P,[t,a,t,a,a,t,g])]).


genex_clause(contains_subseq(X,[],0),[X=[]]).
genex_clause(contains_subseq(X,[],0),[X=[_|_]]).
genex_clause(contains_subseq([Head|P_tail],[Head|S_tail],1),
                              [S_tail=[]]).
genex_clause(contains_subseq([Head|P_tail],[Head|S_tail],1),
                              [prefix_of(P_tail,S_tail)]).

genex_clause(contains_subseq([_|P_tail],Subseq,Offset),
                             [contains_subseq(P_tail,Subseq,N),
                              Offset is N+1]).
```

```
genex_clause(header([X|Y],[X|Z]), [Y=[]]).
genex_clause(header([X|Y],[X|Z]), [prefix(Z,Y)]).
%the string X...Y appears at the head of string X...Y...Z %

genex_clause(prefix_of([X|Y],[X]),[]).
genex_clause(prefix_of([X|Y],[X|Z]),[prefix_of(Y,Z)]).

genex_clause(approx(N,M),[Tolerance is M/10,
                High is M+Tolerance,
                Low is M-Tolerance,
                N > Low,
                N < High]).

genex_clause(length([],0),[]).
genex_clause(length([H|T],N),[length(T,M),N is M+1]).

genex_clause(member(X,[X|_]),[]).
genex_clause(member(X,[_|List]),[member(X,List)]).

genex_clause(gene(X),[reg_gene(X)]).
genex_clause(gene(X),[structural_gene(X)]).

genex_clause(protein(X),[reg_protein(X)]).
genex_clause(reg_protein(X),[activator(Operon,X)]).
genex_clause(reg_protein(X),[repressor(Operon,X)]).
genex_clause(reg_protein(X),[product(Reg_gene,X),
                reg_gene(Reg_gene)]).
genex_clause(present(Reg_pro),[reg_protein(Reg_pro)]).
genex_clause(protein(X),[struct_protein(X)]).
genex_clause(part_of(R,smbs(R)),
                [reg_protein(R),
                 normal(R)]).
genex_clause(part_of(R, smbs(R)),
          [reg_protein(R),assuming(part_of(R,smbs(R)))]).

genex_clause(amino_acid_seq([]),[]).
genex_clause(amino_acid_seq([Y|Z]),[protein(Y)]).
genex_clause(amino_acid_seq([Y|Z]),[amino_acid(Y), amino_acid_seq(Z)]).
genex_clause(amino_acid(gly),[]).
```

```
genex_clause(amino_acid(ala),[]).
genex_clause(amino_acid(val),[]).
genex_clause(amino_acid(ileu),[]).
genex_clause(amino_acid(leu),[]).
genex_clause(amino_acid(ser),[]).
genex_clause(amino_acid(thr),[]).
genex_clause(amino_acid(pro),[]).
genex_clause(amino_acid(asp),[]).
genex_clause(amino_acid(glu),[]).
genex_clause(amino_acid(lys),[]).
genex_clause(amino_acid(arg),[]).
genex_clause(amino_acid(asn),[]).
genex_clause(amino_acid(gln),[]).
genex_clause(amino_acid(cys),[]).
genex_clause(amino_acid(met),[]).
genex_clause(amino_acid(trp),[]).
genex_clause(amino_acid(phe),[]).
genex_clause(amino_acid(tyr),[]).
genex_clause(amino_acid(his),[]).

genex_clause(codon([u,u,u],phe),[]).
genex_clause(codon([u,u,c],phe),[]).
genex_clause(codon([u,u,a],leu),[]).
genex_clause(codon([u,u,g],leu),[]).
genex_clause(codon([u,c,u],ser),[]).
genex_clause(codon([u,c,c],ser),[]).
genex_clause(codon([u,c,a],ser),[]).
genex_clause(codon([u,c,g],ser),[]).
genex_clause(codon([u,a,u],tyr),[]).
genex_clause(codon([u,a,c],tyr),[]).
genex_clause(codon([u,a,a],term),[]).
genex_clause(codon([u,a,g],term),[]).
genex_clause(codon([u,g,u],cys),[]).
genex_clause(codon([u,g,c],cys),[]).
genex_clause(codon([u,g,a],term),[]).
genex_clause(codon([u,g,g],trp),[]).
genex_clause(codon([c,u,u],leu),[]).
genex_clause(codon([c,u,c],leu),[]).
genex_clause(codon([c,u,a],leu),[]).
genex_clause(codon([c,u,g],leu),[]).
```

```
genex_clause(codon([c,c,u],pro),[]).
genex_clause(codon([c,c,c],pro),[]).
genex_clause(codon([c,c,a],pro),[]).
genex_clause(codon([c,c,g],pro),[]).
genex_clause(codon([c,a,u],his),[]).
genex_clause(codon([c,a,c],his),[]).
genex_clause(codon([c,a,a],gln),[]).
genex_clause(codon([c,a,g],gln),[]).
genex_clause(codon([c,g,u],arg),[]).
genex_clause(codon([c,g,c],arg),[]).
genex_clause(codon([c,g,a],arg),[]).
genex_clause(codon([c,g,g],arg),[]).
genex_clause(codon([a,u,u],ileu),[]).
genex_clause(codon([a,u,c],ileu),[]).
genex_clause(codon([a,u,a],ileu),[]).
genex_clause(codon([a,u,g],met),[]).
genex_clause(codon([a,c,u],thr),[]).
genex_clause(codon([a,c,c],thr),[]).
genex_clause(codon([a,c,a],thr),[]).
genex_clause(codon([a,c,g],thr),[]).
genex_clause(codon([a,a,u],asn),[]).
genex_clause(codon([a,a,c],asn),[]).
genex_clause(codon([a,a,a],lys),[]).
genex_clause(codon([a,a,g],lys),[]).
genex_clause(codon([a,g,u],ser),[]).
genex_clause(codon([a,g,c],ser),[]).
genex_clause(codon([a,g,a],arg),[]).
genex_clause(codon([a,g,g],arg),[]).
genex_clause(codon([g,u,u],val),[]).
genex_clause(codon([g,u,c],val),[]).
genex_clause(codon([g,u,a],val),[]).
genex_clause(codon([g,u,g],val),[]).
genex_clause(codon([g,c,u],ala),[]).
genex_clause(codon([g,c,c],ala),[]).
genex_clause(codon([g,c,a],ala),[]).
genex_clause(codon([g,c,g],ala),[]).
genex_clause(codon([g,a,u],asp),[]).
genex_clause(codon([g,a,c],asp),[]).
genex_clause(codon([g,a,a],glu),[]).
genex_clause(codon([g,a,g],glu),[]).
```

```
genex_clause(codon([g,g,u],gly),[]).
genex_clause(codon([g,g,c],gly),[]).
genex_clause(codon([g,g,a],gly),[]).
genex_clause(codon([g,g,g],gly),[]).

genex_clause(binding_site(X,overlap_region(P,O)),
                [reg_protein(X), operator(O),promoter(P)]).
genex_clause(binding_site(rna_pol,P),[promoter(P)]).
genex_clause(binding_site(I,smbs(R)),
                        [reg_protein(R),
                        part_of(R,smbs(R)),
                        assoc(O,RG),
                        reg_gene(RG),
                        product(RG,R),
                        inducible(O),
                        inducer(O,I)]).

genex_clause(binding_site(C,smbs(R)),
                        [reg_protein(R),
                        part_of(R,smbs(R)),
                        assoc(O,RG),
                        reg_gene(RG),
                        product(RG,R),
                        repressible(O),
                        corepressor(O,C)]).

genex_clause(bind(Mol, Site),
                        [binding_site(Mol, Site), %preconditions%
                        complementary_conform(Mol,Site),
                        \+ bound(X,Site),
                      % free_to_bind(Mol), %
                        asserta(genex_clause(bound(Mol,Site)))]).
genex_clause(bind(Mol, Site),
                        [binding_site(Mol, Site), %preconditions%
                        assuming(complementary_conform(Mol,Site)),
                        \+ bound(X,Site),
                      % free_to_bind(Mol), %
                        asserta(genex_clause(bound(Mol,Site)))]).

genex_clause(part_of(P,overlap_region(P,O)),[promoter(P),operator(O)]).
```

```
genex_clause(part_of(P,promoter_specific(P)),[promoter(P)]).
genex_clause(part_of(O,overlap_region(P,O)),[operator(O),promoter(P)]).
genex_clause(part_of(O,operator_specific(O)),[operator(O)]).

genex_clause(same(X,X),[]).

genex_clause(bound(Mol, P),[promoter(P),
                            part_of(P, Promoter_part),
                            bound(Mol, Promoter_part)]).

genex_clause(bound(Mol, O),[operator(O),
                 part_of(O, Operator_part),
                 bound(Mol, Operator_part)]).


genex_clause(bindable(X,Y),[bound(X,Y)]).
genex_clause(bindable(X,Y),[bind(X,Y)]).

genex_clause(complementary_conform(Reg_pro,overlap_region(P,Oper)),
                          [operator(Oper),
                          promoter(P),
                          reg_protein(Reg_pro),
                          normal(overlap_region(P,Oper)),
                          active(Reg_pro)]).
genex_clause(complementary_conform(Mol, Regpro),[operon(O),
                          assoc(O,RG),
                          product(RG,Regpro),
                          smmol(O,Mol),
                          present(Mol),
                          normal(Regpro)]).
genex_clause(smmol(O,Mol),[inducer(O,Mol)]).
genex_clause(smmol(O,Mol),[corepressor(O,Mol)]).
genex_clause(complementary_conform(rna_pol,P),[promoter(P),
                          \+ bad_rna_conform(P)]).

genex_clause(unbind(Mol,Site),[retract(bound(Mol,Site))]).

%%% this clause states the requirements for an operon to be expressed
%%%
genex_clause(expressed(Operon),[operon(Operon),
```

```
                initiated(Operon),!,
                not_attenuated(Operon),!,
                xcribe(get_gene_seq(Operon),Xcripts),!,
                xlate(Xcripts,Protein)]).
%               processed(Protein).     %
% processed includes termination_ok (if not msg could be unstable) %

%%% if operon is not initiated, the above will fail, but we want it %%%
%%% to keep looking for other reasons, so run it from the next step %%%
%%% if we have gotten this far and no sequence is given, we cannot con-
clude %%%
%%% anything about transcription or translation, so just stop and fail.
%%%

genex_clause(expressed(Operon),[print('transcription cannot initiate'),nl,
print('checking other explanations'),nl,!,
operon(Operon),
not_attenuated(Operon),!,
get_gene_seq(Operon,[]),!,fail]).

%%% otherwise continue to see if you can find problems in transcription
%%%
%%% or translation.                                                    %%%
genex_clause(expressed(Operon),[print('transcription cannot initiate'),nl,
print('checking other explanations'),nl,!,
operon(Operon),
not_attenuated(Operon),!,
get_gene_seq(Operon,[Seq|Rest]),!,
xcribe([Seq|Rest],Xcripts),!,
xlate(Xcripts,Protein)]).
% processed(Protein). %

genex_clause(expressed(Operon),[operon(Operon),
print('checking other explanations'),nl,
get_gene_seq(Operon,[Seq|Rest]),!,
xcribe([Seq|Rest],Xcripts),!,
xlate(Xcripts,Protein)]).
% processed(Protein). %

%% if you get this far, you can't prove operon expressed, so fail.  %%
```

```
genex_clause(expressed(Operon),[operon(Operon),!,fail]).

% If there is no sequence given for the operon, it should be %
% entered as [] %

genex_clause(get_gene_seq(0,[]),[sequence(0,[])]).
%%% get_gene_seq gets structural gene sequence from operon sequence. %%%
%%% it is not implemented yet. %%%

genex_clause(xcribe([],[]),[print('no sequence given...'),nl]).
genex_clause(xcribe([Gene|Rest],[Msg|Xcripts]),
                        [sequence(Gene,Seq),
                         transc(Seq,Msg),
                         xcribe(Rest,Xcripts)]).

genex_clause(transc([],[]),[print('no transcript created...'),nl]).
genex_clause(transc([First|Rest],[Mfirst|Mrest]),
                         [xc(First,Mfirst),transc(Rest,Mrest)]).
genex_clause(xc(a,u),[]).
genex_clause(xc(t,a),[]).
genex_clause(xc(g,c),[]).
genex_clause(xc(c,g),[]).

genex_clause(xlate([],[]),[print('no message to translate...'),nl]).
genex_clause(xlate([Msg|Rest],[Prot1|Proteins]),
                [transl(Msg,Prot1),xlate(Rest,Proteins)]).

% have to look for the ribosome binding site for each transcribed gene
in msg %

genex_clause(transl(M,Protein),
             [contains_subseq(M,[a,g,g,a],N), % the rib. bind site %
              index(M,N,Substr), %ignore seq upstream of rib bind site%
              get_start_codon(Substr,0,XL),
              xl(XL,Protein)]).

genex_clause(get_start_codon(Msg,20,_),[fail]).
% start codon is within 13 nt of bind site %

genex_clause(get_start_codon(Msg,Offset,XL),
```

```
                    [Start_point is Offset+10,
                    length(Msg,L),
                    L > Start_point,
                    index(Msg,Start_point,Head),!,
                    prefix_of(Head,[a,u,g]), XL=Head]).

genex_clause(get_start_codon(Msg,Offset,XL),
                    [Start_point is Offset+10,
                    length(Msg,L),
                    L > Start_point,
                    index(Msg,Start_point,Head),!,
                    New_offset is Offset+1,
                    get_start_codon(Msg,New_offset,XL)]).

genex_clause(index([],_,[]),[]).
genex_clause(index(String,1,String),[]).
genex_clause(index([_|Tail],I,Substr),
                    [J is I-1, index(Tail,J,Substr)]).

genex_clause(xl([],[]),[]).
genex_clause(xl([Nt1,Nt2,Nt3|Rest],[Amino1|Aminos]),
                    [codon([Nt1,Nt2,Nt3],Amino1),
                    Amino1=term,
                    Aminos=[]]).  % termination case %
genex_clause(xl([Nt1,Nt2,Nt3|Rest],[Amino1|Aminos]),
                    [codon([Nt1,Nt2,Nt3],Amino1),
                    xl(Rest,Aminos)]).

% have to take care of error cases %
genex_clause(proper_xcript(M),
                   [\+ nonsense_mut(M),! ,\+ missense_mut(M), !]).


genex_clause(anabolic_proc(digestion),[]).
genex_clause(catabolic_proc(synthesis),[]).

% how to tell if an operon is inducible %
% if the operon doesn't fit either of these rules, %
% then guess either or both %
```

```
genex_clause(inducible(O),[operon(O),
                structural_gene(O,Gene),
                product(Gene,Prod),
                purpose(Prod,Proc),
                catabolic_proc(Proc)]).
genex_clause(inducible(O),[operon(O),inducer(O,I)]).

genex_clause(repressible(O),[operon(O),
                structural_gene(O,Gene),
                product(Gene,Prod),
                purpose(Prod,Proc),
                anabolic_proc(Proc)]).
genex_clause(repressible(O),[operon(O),corepressor(O,C)]).

genex_clause(inducer(O,I),[operon(O),
                structural_gene(O,Gene),
                product(Gene,Prod),
                substrate(Prod,I),
                proportional_to(Prod,I),
                repressor(O,R),
                bindable(I,R)]).
genex_clause(inducer(O,I),
                [operon(O),
                structural_gene(O,Gene),
                product(Gene,P),
                dir_rel(I,P)]).
genex_clause(corepressor(O,C),[operon(O),
                structural_gene(O,Gene),
                product(Gene,Prod),
                end_product(Prod,C),
                inv_proportional_to(Prod,C),
                repressor(O,R),
                bindable(C,R)]).
genex_clause(corepressor(O,C),[operon(O),
                structural_gene(O,Gene),product(Gene,P),
                inv_rel(C,P)]).

genex_clause(initiated(Op),[operon(Op),
                part_of(Op,P),
                promoter(P),
```

```
                bindable(rna_pol,P)]).

genex_clause(bad_rna_conform(P),[promoter(P),
                normal(P),
                operon(Op),
                part_of(Op,P),
                neg_control(Op),
                repressor(Op,R),
                part_of(Op,O),
                operator(O),
                bindable(R,overlap_region(P,O))]).

genex_clause(bad_rna_conform(P),[promoter(P),
                normal(P),
                operon(Op),
                part_of(Op,P),
                possible_pos_control(Op),
                activator(Op,A),
                part_of(Op,O),
                operator(O),
                inactive_activator(A,O,P)]).
genex_clause(inactive_activator(A,O,P),[\+ active(A)]).
genex_clause(inactive_activator(A,O,P),
                 [\+ bindable(A,overlap_region(P,O))]).

genex_clause(neg_control(Op),[operon(Op)]).
% most operons are under negative control %

genex_clause(possible_pos_control(Op),[glucose_sensitive(Op)]).
genex_clause(possible_pos_control(Op),[assuming(pos_control(Op))]).

genex_clause(glucose_sensitive(Op),[operon(Op),
                        structural_gene(Op,X),
                        product(X,P),
                        purpose(P, catabolism(S)),
                        sugar(S)]).

genex_clause(sugar(lactose),[]).
genex_clause(sugar(galactose),[]).
genex_clause(sugar(arabinose),[]).
```

```
genex_clause(sugar(maltose),[]).

genex_clause(activator(Op,cap),[operon(Op),glucose_sensitive(Op)]).
genex_clause(inducible(cap),[]).
genex_clause(inducer(cap,cAMP),[]).
genex_clause(present(cAMP),[\+ present(glucose)]).

genex_clause(active(Reg_protein),
             [reg_protein(Reg_protein),
              active_smmol_conform(Reg_protein)]).
genex_clause(active(Reg_protein),
             [reg_protein(Reg_protein),
              assuming(active_smmol_conform(Reg_protein))]).

genex_clause(active_smmol_conform(R),
             [repressor(Op,R),
              normal(R),
              inducible(Op),
              \+ inducer(Op,I)]).
genex_clause(active_smmol_conform(R),
             [repressor(Op,R),
              normal(R),
              inducible(Op),
              inducer(Op,I),
              \+ bindable(I,smbs(R))]).
genex_clause(active_smmol_conform(R),
             [repressor(Op,R),
              phenotype(Op,inducible),
              \+ inducer(Op,I)]).
genex_clause(active_smmol_conform(R),
             [repressor(Op,R),
              phenotype(Op,inducible),
              inducer(Op,I),
              \+ bindable(I,smbs(R))]).
genex_clause(active_smmol_conform(R),
             [repressor(Op,R),
              normal(R),
              repressible(Op),
              corepressor(Op,C),
              bindable(C,smbs(R))]).
```

```
genex_clause(active_smmol_conform(R),
                [repressor(Op,R),
                 phenotype(Op,repressible),
                 corepressor(Op,C),
                 bindable(C,smbs(R))]).
genex_clause(active_smmol_conform(A),
                [activator(Op,A),
                 normal(A),
                 inducible(Op),
                 inducer(Op,I),
                 bindable(I,smbs(A))]).
genex_clause(active_smmol_conform(A),
                [activator(Op,A),
                 phenotype(Op,inducible),
                 inducer(Op,I),
                 bindable(I,smbs(A))]).
genex_clause(active_smmol_conform(A),
                [activator(Op,A),
                 normal(A),
                 repressible(Op),
                 \+ corepressor(Op,C)]).
genex_clause(active_smmol_conform(A),
                [activator(Op,A),
                 normal(A),
                 repressible(Op),
                 corepressor(Op,C),
                 \+ bindable(C,smbs(A))]).
genex_clause(active_smmol_conform(A),
                [activator(Op,A),
                 phenotype(Op,repressible),
                 \+ corepressor(Op,C)]).
genex_clause(active_smmol_conform(A),
                [activator(Op,A),
                 phenotype(Op,repressible),
                 corepressor(Op,C),
                 \+ bindable(C,smbs(A))]).

genex_clause(normal(X),[\+ mutated(X),\+ assuming(mutated(X))]).
genex_clause(normal(Y),[gene(X),product(Y,X),normal(X)]).
genex_clause(repressor(Op,R),
```

```
                    [operon(Op),
                    neg_control(Op),
                    assoc(Op,Reg_gene),
                    reg_gene(Reg_gene),
                    product(Reg_gene,R)]).

genex_clause(activator(Op,A),
                    [operon(Op),
                    possible_pos_control(Op),
                    assoc(Op,Reg_gene),
                    reg_gene(Reg_gene),
                    product(Reg_gene,A)]).

genex_clause(not_attenuated(Op),
 [seq(Op,Seq),\+ Seq = [],
  structural_gene(Op,X),
  product(X,Prod),
  purpose(Prod,synthesis),
  product(Prod,Amino_acid),
  amino_acid(Amino_acid),
  present(Amino_acid),
  codons(Amino_acid,Codons),
  find_atten_reg(Seq,Codons),!,fail]).
genex_clause(not_attenuated(Op),
 [seq(Op,[]),
  structural_gene(Op,X),
  product(X,Prod),
  purpose(Prod,synthesis),
  product(Prod,Amino_acid),
  amino_acid(Amino_acid),
  present(Amino_acid),
  print(Op),print('  is possibly attenuated'),nl,!,fail]).
                    present(Amino_acid),
                    print(Op),print('  is possibly attenuated'),nl]).

genex_clause(find_atten(Seq,Codons),
                   [find_atten2(Seq,Codons,0,0,[])]).

genex_clause(find_atten2([],C,length,clength,atten_seq),[fail]).
```

```
%  the following four clauses determine the likely model  of %
% the operon's behavior. %

%  the following four clauses determine the likely model  of %
% the operon's behavior. %

create_model(O,Remove,Assumptions,Mode) :- genex_clause(operon(O),[]),
    genex_clause(observed_expressed(O),[]),
    no_influence(Smmol,O),
    ((genex_clause(reg_gene(RG),[]),
      genex_clause(mutated(RG),[]));
     (genex_clause(part_of(O,Op),[]),
      genex_clause(operator(Op),[]),
      genex_clause(mutated(Op),[]))),
    precond(constitutive,O,Remove,Assumptions,Mode).

create_model(O,Remove,Assumptions,Mode) :- genex_clause(operon(O),[]),
    genex_clause(observed_not_expressed(O),[]),
    no_influence(Smmol,O),
    ((genex_clause(assoc(O,RG),[]),
      genex_clause(reg_gene(RG),[]),
      genex_clause(mutated(RG),[]));
     (genex_clause(part_of(O,Op),[]),
      genex_clause(operator(Op),[]),
      genex_clause(mutated(Op),[]))),
   precond(uninducible,O,Remove,Assumptions,Mode).

create_model(O,Remove,Assumptions,Mode) :- genex_clause(operon(O),[]),
genex_clause(structural_gene(O,Gene),[]),
genex_clause(product(Gene,Prod),[]),
genex_clause(dir_rel(Smmol,Prod),[]),
precond(inducible,O,Remove,Assumptions,Mode).

create_model(O,Remove,Assumptions,Mode) :- genex_clause(operon(O),[]),
genex_clause(structural_gene(O,Gene),[]),
genex_clause(product(Gene,Prod),[]),
genex_clause(inv_rel(Smmol,Prod),[]),
precond(repressible,O,Remove,Assumptions,Mode).

no_influence(Smmol,O) :-
```

```
genex_clause(operon(O),[]),
genex_clause(structural_gene(O,Gene),[]),
genex_clause(product(Gene,Prod),[]),
 \+ genex_clause(dir_rel(Smmol,Prod),[]),
 \+ genex_clause(inv_rel(Smmol,Prod),[]).

% the following four clauses return a list of sets of conditions %
% necessary for the model to be true. Each  set separately is    %
% sufficient. These are used in terse mode.  %

precond(repressible,O,
        [[complementary_conform(X,smbs(RP)), % uncertain vars to remove
%
  complementary_conform(RP,overlap_region(P,Op))]],
        [[smmol(X),reg_protein(RP),operator(Op), % make these %
 complementary_conform(X,smbs(RP)),      % assumptions %
 complementary_conform(RP,overlap_region(P,Op))]],terse)
:- name(O, Oname),
append(Oname,"_promoter",Pname),
name(P,Pname),
append(Oname,"_operator",Opname),
name(Op,Opname),
append(Oname,"_reg_gene",Rname),
name(R,Rname),
append(Oname,"_reg_protein",RPname),
name(RP,RPname).

precond(inducible,O,
        [[complementary_conform(X,smbs(RP)),
  complementary_conform(RP,overlap_region(P,Op))]],
[[smmol(X),reg_protein(RP),operator(Op),
   complementary_conform(X,smbs(RP)),
 complementary_conform(RP,overlap_region(P,Op))]],terse)
:- name(O, Oname),
append(Oname,"_promoter",Pname),
name(P,Pname),
append(Oname,"_operator",Opname),
name(Op,Opname),
append(Oname,"_reg_gene",Rname),
name(R,Rname),
```

```
append(Oname,"_reg_protein",RPname),
name(RP,RPname).
precond(constitutive,O,
        [[complementary_conform(RP,overlap_region(P,Op))],
         [complementary_conform(X,smbs(RP))]],
        [[not_complementary_conform(RP,overlap_region(P,Op))],
         [not_complementary_conform(X,smbs(RP))]],terse)
:- name(O, Oname),
append(Oname,"_promoter",Pname),
name(P,Pname),
append(Oname,"_operator",Opname),
name(Op,Opname),
append(Oname,"_reg_gene",Rname),
name(R,Rname),
append(Oname,"_reg_protein",RPname),
name(RP,RPname).

precond(uninducible,O,
        [[complementary_conform(rna_pol,P)],
         [complementary_conform(Smmol,smbs(RP))],
         [complementary_conform(RP,overlap_region(P,Op))]],
        [[not_complementary_conform(rna_pol,P)],
         [not_complementary_conform(Smmol,smbs(RP))],
         [reg_protein(RP),operator(Op),
          complementary_conform(RP,overlap_region(P,Op))],
         [not_complementary_conform(RP,overlap_region(P,Op))]],terse)
:- name(O, Oname),
append(Oname,"_promoter",Pname),
name(P,Pname),
append(Oname,"_operator",Opname),
name(Op,Opname),
append(Oname,"_reg_gene",Rname),
name(R,Rname),
append(Oname,"_reg_protein",RPname),
name(RP,RPname).

% The following four clauses are used in verbose mode.  They do not %
% include the assumptions about conformation so that the program %
% will give a detailed description of the mechanism of the operon %
% using model-based reasoning. %
```

```
precond(repressible,O,
        [],
        [[smmol(X),reg_protein(RP),operator(Op),
 complementary_conform(X,smbs(RP)),
 complementary_conform(RP,overlap_region(P,Op))]],verbose)
:- name(O, Oname),
append(Oname,"_promoter",Pname),
name(P,Pname),
append(Oname,"_operator",Opname),
name(Op,Opname),
append(Oname,"_reg_gene",Rname),
name(R,Rname),
append(Oname,"_reg_protein",RPname),
name(RP,RPname).

precond(inducible,O,
        [],
[[smmol(X),reg_protein(RP),operator(Op),
   complementary_conform(X,smbs(RP)),
 complementary_conform(RP,overlap_region(P,Op))]],verbose)
:- name(O, Oname),
append(Oname,"_promoter",Pname),
name(P,Pname),
append(Oname,"_operator",Opname),
name(Op,Opname),
append(Oname,"_reg_gene",Rname),
name(R,Rname),
append(Oname,"_reg_protein",RPname),
name(RP,RPname).
precond(constitutive,O,
        [],
        [[not_complementary_conform(RP,overlap_region(P,Op))],
         [not_complementary_conform(X,smbs(RP))]],verbose)
:- name(O, Oname),
append(Oname,"_promoter",Pname),
name(P,Pname),
append(Oname,"_operator",Opname),
name(Op,Opname),
append(Oname,"_reg_gene",Rname),
```

```
name(R,Rname),
append(Oname,"_reg_protein",RPname),
name(RP,RPname).

precond(uninducible,O,
        [],
        [[not_complementary_conform(rna_pol,P)],
         [not_complementary_conform(Smmol,smbs(RP))],
         [reg_protein(RP),operator(Op),
          complementary_conform(RP,overlap_region(P,Op))],
         [not_complementary_conform(RP,overlap_region(P,Op))]],verbose)
:- name(O, Oname),
append(Oname,"_promoter",Pname),
name(P,Pname),
append(Oname,"_operator",Opname),
name(Op,Opname),
append(Oname,"_reg_gene",Rname),
name(R,Rname),
append(Oname,"_reg_protein",RPname),
name(RP,RPname).


%%% given the behavior, deduce a model of the behavior.  Use the %%%
%%% preconditions for the model to tell you what variables to  %%%
%%% assume %%%


genex_clause(phenotype(O,derepressed),
                            [operon(O),
                             assoc(O,RG),
                             reg_gene(RG),
                             mutated(RG),
                             observed_expressed(O),
                             no_influence(Smmol,O),
                             asserta
                               (genex_clause(phenotype(O,derepressed)))]).
genex_clause(phenotype(O,derepressed),
                            [operon(O),
                             part_of(O,Op),
```

```
                                operator(Op),
                                mutated(Op),
                                observed_expressed(O),
                                no_influence(Smmol,O),
                                asserta
                                  (genex_clause(phenotype(O,derepressed)))]).

genex_clause(phenotype(O,superrepressed),
                               [operon(O),
                                assoc(O,RG),
                                reg_gene(RG),
                                mutated(RG),
                                observed_not_expressed(O),
                                no_influence(Smmol,O),
                                asserta
                                  (genex_clause(phenotype(O,superrepressed)))]).
genex_clause(phenotype(O,superrepressed),
                               [operon(O),
                                part_of(O,Op),
                                operator(Op),
                                mutated(Op),
                                observed_not_expressed(O),
                                no_influence(Smmol,O),
                                asserta
                                  (genex_clause(phenotype(O,superrepressed)))]).

genex_clause(phenotype(O,constitutive),
                               [operon(O),
                                assoc(O,RG),
                                reg_gene(RG),
                                mutated(RG),
                                observed_expressed(O),
                                no_influence(Smmol,O),
                                asserta
                                  (genex_clause(phenotype(O,constitutive)))]).
genex_clause(phenotype(O,constitutive),
                               [operon(O),
                                part_of(O,Op),
                                operator(Op),
                                mutated(Op),
```

```
                              observed_expressed(O),
                              no_influence(Smmol,O),
                              asserta
                                (genex_clause(phenotype(O,constitutive)))]).

genex_clause(phenotype(O,uninducible),
                             [operon(O),
                              assoc(O,RG),
                              reg_gene(RG),
                              mutated(RG),
                              observed_not_expressed(O),
                              no_influence(Smmol,O),
                              asserta
                                (genex_clause(phenotype(O,uninducible)))]).
genex_clause(phenotype(O,uninducible),
                             [operon(O),
                              part_of(O,Op),
                              operator(Op),
                              mutated(Op),
                              observed_not_expressed(O),
                              no_influence(Smmol,O),
                              asserta
                                (genex_clause(phenotype(O,uninducible)))]).

genex_clause(phenotype(O,inducible),
                            [operon(O),
                            structural_gene(O,Gene),
                            product(Gene,Prod),
                            dir_rel(Smmol,Prod),
                            asserta(genex_clause(phenotype(O,inducible)))]).

genex_clause(phenotype(O,repressible),
                            [operon(O),
                            structural_gene(O,Gene),
                            product(Gene,Prod),
                            inv_rel(Smmol,Prod),
                            asserta(genex_clause(phenotype(O,repressible)))]).

genex_clause(no_influence(Smmol,O),
                  [operon(O),
```

```
                    structural_gene(O,Gene),
                    product(Gene,Prod),
                    \+ dir_rel(Smmol,Prod),
                    \+ inv_rel(Smmol,Prod)]).

genex_clause(inc_affinity(Reg_pro,Oper),
                          [operon(O),
                           part_of(O,Oper),
                           operator(Oper),
                           mutated(Oper),
                           assoc(O,RG),
                           reg_gene(RG),
                           product(RG,Reg_pro),
                           phenotype(O,uninducible)]).
genex_clause(inc_affinity(Reg_pro,Oper),
                          [operon(O),
                           part_of(O,Oper),
                           operator(Oper),
                           mutated(Oper),
                           assoc(O,RG),
                           reg_gene(RG),
                           product(RG,Reg_pro),
                           phenotype(O,superrepressed)]).
genex_clause(inc_affinity(Reg_pro,Oper),
                          [operon(O),
                           part_of(O,Oper),
                           operator(Oper),
                           assoc(O,RG),
                           reg_gene(RG),
                           mutated(RG),
                           product(RG,Reg_pro),
                           phenotype(O,uninducible)]).
genex_clause(inc_affinity(Reg_pro,Oper),
                          [operon(O),
                           part_of(O,Oper),
                           operator(Oper),
                           assoc(O,RG),
                           reg_gene(RG),
                           mutated(RG),
                           product(RG,Reg_pro),
```

```
                        phenotype(O,superrepressed)]).

genex_clause(dec_affinity(Reg_pro,Oper),
                       [operon(O),
                        part_of(O,Oper),
                        operator(Oper),
                        mutated(Oper),
                        assoc(O,RG),
                        reg_gene(RG),
                        product(RG,Reg_pro),
                        phenotype(O,derepressed)]).
genex_clause(dec_affinity(Reg_pro,Oper),
                       [operon(O),
                        part_of(O,Oper),
                        operator(Oper),
                        mutated(Oper),
                        assoc(O,RG),
                        reg_gene(RG),
                        product(RG,Reg_pro),
                        phenotype(O,constitutive)]).
genex_clause(dec_affinity(Reg_pro,Oper),
                       [operon(O),
                        part_of(O,Oper),
                        operator(Oper),
                        assoc(O,RG),
                        reg_gene(RG),
                        mutated(RG),
                        product(RG,Reg_pro),
                        phenotype(O,derepressed)]).
genex_clause(dec_affinity(Reg_pro,Oper),
                       [operon(O),
                        part_of(O,Oper),
                        operator(Oper),
                        assoc(O,RG),
                        reg_gene(RG),
                        mutated(RG),
                        product(RG,Reg_pro),
                        phenotype(O,constitutive)]).

genex_clause(norm_affinity(Reg_pro,Oper),
```

```
                    [operon(O),
                     part_of(O,Oper), operator(Oper),
                     assoc(O,RG), reg_gene(RG),
                     product(RG,Reg_pro),
                     \+ phenotype(O,derepressed),
                     \+ phenotype(O,superrepressed),
                     \+ phenotype(O,constitutive),
                     \+ phenotype(O,uninducible)]).

genex_clause(complementary_conform(Reg_pro,overlap_region(P,Oper)),
                [reg_protein(Reg_pro),promoter(P),operator(Oper),
                inc_affinity(Reg_pro,Oper)]).
genex_clause(complementary_conform(Reg_pro,overlap_region(P,Oper)),
                [operon(O),assoc(O,Reg_gene),reg_gene(Reg_gene),
                product(Reg_gene,Reg_pro),
                reg_protein(Reg_pro),active(Reg_pro),
                part_of(O,P),promoter(P),
                part_of(O,Oper),operator(Oper),
                phenotype(O,repressible)]).
genex_clause(complementary_conform(Reg_pro,overlap_region(P,Oper)),
                [operon(O),assoc(O,Reg_gene),reg_gene(Reg_gene),
                product(Reg_gene,Reg_pro),
                reg_protein(Reg_pro),active(Reg_pro),
                part_of(O,P),promoter(P),
                part_of(O,Oper),operator(Oper),
                phenotype(O,inducible)]).
genex_clause(complementary_conform(X,smbs(RP)),
                [operon(O),assoc(O,RG),reg_gene(RG),product(RG,RP),
                reg_protein(RP),smmol(O,X),phenotype(O,repressible)]).
genex_clause(complementary_conform(X,smbs(RP)),
                [operon(O),assoc(O,RG),reg_gene(RG),product(RG,RP),
                reg_protein(RP),smmol(O,X),phenotype(O,inducible)]).

genex_clause(part_of(RP,smbs(RP)),
                [operon(O),assoc(O,RG),reg_gene(RG),product(RG,RP),
                reg_protein(RP),phenotype(O,repressible)]).
genex_clause(part_of(RP,smbs(RP)),
                [operon(O),assoc(O,RG),reg_gene(RG),product(RG,RP),
                reg_protein(RP),phenotype(O,inducible)]).
```

The following code comprises the GENEX interpreter.

```
member(X,[X|_]).
member(X,[_|L]) :- member(X,L).

clause_uses(G,G).
clause_uses(G,Clause) :- genex_clause(G,Body),
                         (member(Clause,Body);
                          member(\+ Clause,Body)).

trivial(G) :- repressible(O),clause_uses(G,inducible(O)).
trivial(G) :- inducible(O),clause_uses(G,repressible(O)).
trivial(G) :- inducer(O,I),clause_uses(G,corepressor(O,C)).
trivial(G) :- corepressor(O,C),clause_uses(G,inducer(O,I)).
trivial(G) :- \+ pos_control(O),clause_uses(G,pos_control(O)).
trivial(G) :- \+ neg_control(O),clause_uses(G,neg_control(O)).
trivial(G) :- genex_clause(G,[]).
trivial(part_of(O,X)).
trivial(assoc(O,X)).
trivial(neg_control(O)).
trivial(pos_control(O)).
trivial(reg_protein(RP)).
trivial(repressor(O,RP)).
trivial(activator(O,RP)).
trivial(gene(X)).
trivial(neg_control(O)).
trivial(print(X)).


% The following program runs the possible states. The 1ST ARG. %
% is a list of lists of UNCERTAIN VARIABLE COMBINATIONS, %
% representing the possible states that might be in effect. %
% The 2ND ARG. is the OPERON that we want to simulate.  All %
% possible states in the list of states are tried.  If the %
% result of the simulation matches the observed behavior, then %
% the list of assumptions which were made for the current run of %
% the simulation is printed out, along with highlights of the %
% simulation of the program.  This program assumes that system  %
% variables which are not uncertain will be set up before this %
% program is run. %
try_possible_state([],Operon) :-   % CASE: NO ASSUMPTIONS      %
(observed_expressed(Operon) -> prove(expressed(Operon)));
```

```
 \+ prove(expressed(Operon)).  % if it's not observed expressed %

try_possible_state([A|B], Operon) :-  % CASE: ASSUMPTIONS MADE    %
try_possible_states([A|B], Operon).

try_possible_states([],_).

try_possible_states([Newstate|Rest],Operon) :-
retract_assumptions,
make_assumptions(Newstate),
((genex_clause(observed_expressed(Operon),[]),
  try_prove_expressed(Operon));
 (genex_clause(observed_not_expressed(Operon),[]),
  try_prove_not_expressed(Operon))),
try_possible_states(Rest,Observed).


try_prove_expressed(O) :- prove(expressed(O)),nl,print('yes'),nl.
try_prove_expressed(O) :- nl,print('no'),nl.

try_prove_not_expressed(O) :- \+ prove(expressed(O)),nl,print('yes'),nl.
try_prove_not_expressed(O) :- nl,print('no'),nl.

% The following program actually interprets the genex_clauses %
% in PROLOG %

prove([]).
prove([Goal|Rest]) :-
   genex_clause(Goal,Body),
   (\+ trivial(Goal) -> (print('goal: '),print(Goal),nl,
                         print('attempting to prove: '),print(Body),nl);true),
   prove(Body),
   (\+ trivial(Goal) -> (print('proved: '),print(Goal),nl); true),
   prove(Rest). % this is for goals with more than one clause %
prove([\+ Goal|Rest]) :-
    (genex_clause(Goal,Body)  -> (\+ prove(Body), prove(Rest)));
    prove(Rest).
prove([asserta(Assertion)]) :- asserta(Assertion).
prove([print(X)|Rest]) :- print(X),prove(Rest).
prove([nl|Rest]) :- nl,prove(Rest).
```

```
prove(Goal) :-
   \+ functor(Goal,'.',_), % if goal is not a list %
   prove([Goal]).

% MAKE_ASSUMPTIONS asserts assuming(_) for each uncertain variable %
% in the list passed to it.  Must be given a LIST.  Note, however, %
% that GENEX only looks for positive assumptions, the negative ones %
% are used only for explaining the programs results (because in  %
% prolog, if there is no way to prove foo(X), it is the same as if %
% we had explicitly stated  \+ foo(X)...) %

make_assumptions([]).
make_assumptions([First|Rest]) :-
                asserta(genex_clause(assuming(First),[])),
                print('assuming: '),print(First),nl,
                make_assumptions(Rest).


% RETRACT-ASSUMPTIONS removes every instance of a clause of the %
% form assuming(_) %

retract_assumptions :- genex_clause(assuming(_)),
                       retract(genex_clause(assuming(_))),fail.
retract_assumptions.

insert(Item,List,[Item|List]).
remove(Item,[Item|List],List).

% make a list of uncertain variables %

% make states set from list of uncertain variables %
make_uncertain_sets([],[]).
make_uncertain_sets([First|Rest],[New|Newrest]) :-
make_uncertain_set(First,New),
make_uncertain_sets(Rest,Newrest),!.

make_uncertain_set([],[]).
make_uncertain_set([X],Set) :-
X =.. [Func|Args],
name(Func,N),
```

```
append("not_",N,Neg),
name(N1,Neg),
Z =.. [N1|Args],
Set=[[X],[Z]].
make_uncertain_set([X|Rest],Uncertain) :-
make_uncertain_set([X],[X1,X2]),
make_uncertain_set(Rest,[S1|S2]),
add_to_set([S1|S2],X1,New1),
add_to_set([S1|S2],X2,New2),
append(New1,New2,Uncertain),!.

% add-to-set takes as input a bigset of sets and a newset.  It returns
%
% the result of appending newset to each set in bigset. %

add_to_set([],List1,[]).
add_to_set([First|S2],X,[New|Rest]) :-
append(First,X,New),
add_to_set(S2,X,Rest),!.


subtract_sets([],Var_set,[Var_set]).
subtract_sets([X],Var_set,[Newset]) :- subtract(Var_set,X,Newset).
subtract_sets([Subtractee|Rest],Var_set,[Newset|Newrest]) :-
        subtract(Var_set,Subtractee,Newset),
        subtract_sets(Rest,Var_set,Newrest),!.

make_states([],[],[]).
make_states([Uncerts1|UncertsN],[],States) :- [Uncerts1|UncertsN] = States.
make_states([Uncerts1|UncertsN],[Conds1|CondsN],States) :-
add_to_set(Uncerts1,Conds1,States1),
make_states(UncertsN,CondsN,StatesN),
append(States1,StatesN,States),!.

append([A|X],Y,[A|Z]) :- append(X,Y,Z).
append([],Y,Y).

same(X,X).

genex :- print('end every answer with a period followed by a CR!'),
```

```
nl,nl,
        print('solution mode (terse or verbose): '),read(Mode),
print('name of operon: '),read(O), asserta(genex_clause(operon(O),[])),
make_parts(O),
print('are there any mutations in '),print(O),print(' or reg_gene'),
print(' (yes, no, or unknown) ?'), nl, read(Mut),
do_mutations(Mut,O,Sites),
        set_uncertain_vars(O,Sites,Uncert_vars),
        listtoset(Uncert_vars,Var_set), %no dup.elts in var_set%
print('is '),print(O),print(' gene product made?'),nl,
read(Expressed),
((same(Expressed,'yes') ->
asserta(genex_clause(observed_expressed(O),[]));
(same(Expressed,'no') ->
asserta(genex_clause(observed_not_expressed(O),[]));
(same(Expressed,'predict') ->
asserta(genex_clause(predict_expressed(O),[])))),
print('enter any other information available '),
nl,
print('e.g. proportional_to(X,Y), inv_proportional_to(X,Y)'),
print(', present(Substance), etc.'), nl,
print('-- end with "done"'), nl,
any_other_info(O),
% if we are not in predict mode, can use observed behavior of operon to
%
% decide on the likely model of its behavior. %
(( \+ same(Expressed,'predict') ->
% return uncertain vars to be removed and sets of conditions for model
%
       (create_model(O,Remove_these,Conditions,Mode),
% the next clause subtracts each of the condition sets from the uncert.
%
% var.set in turn, yielding a number of uncert.var sets in which the  %
% remaining vars do not depend on some set of conditions %
        subtract_sets(Remove_these,Var_set,Newsets),
        listtoset(Newsets,Newsets1), % get rid of dupl. sets %
% make uncertain sets from the remaining uncertain var. sets and merge
%
% the conditions into their corresponding uncertain sets %
        make_uncertain_sets(Newsets1,Uncert_sets),
```

```
        make_states(Uncert_sets,Conditions ,States),nl));
% in predict mode, we don't have the observed behavior and so cannot %
% reduce the number of uncertain variables, so just use the original set.
%
 (same(Expressed,'predict') ->
  make_uncertain_set(Var_set, Uncert_sets))),
try_possible_state(States,0),
print('do you want to try positive control? (yes or no)'),nl,
read(PC),
do_pos_control(PC,Set,0).

do_pos_control('no',_,_) :- print('end of GENEX session.'),nl.
do_pos_control('yes',Set,0) :-
retract(genex_clause(neg_control(Operon))),
assert(genex_clause(assuming(pos_control(0)),[])),
print('assuming positive control...'),nl,
try_possible_state(0),
assert(genex_clause(neg_control(0),[])),
print('assuming positive and negative control...'),nl,
try_possible_state(0).

do_mutations('no',0,[]) :- assert(genex_clause(normal(0),[])).
do_mutations('unknown',0,[promoter,operator,reg_gene,struct_genes,unknown]).
do_mutations('yes',0, Sites) :-
print('mutation site known (yes or no) ?'),nl,
read(Ans),
(same(Ans,'no') ->
   (Sites = [promoter,reg_gene,operator,struct_genes,unknown]));
(same(Ans,'yes') ->
   (get_mutation_sites(Sites,
[promoter,reg_gene,operator,struct_genes]))).

make_parts(0) :- name(0, Oname),
append(Oname,"_promoter",Pname),
name(P,Pname),
asserta(genex_clause(promoter(P),[])),
append(Oname,"_operator",Opname),
name(Op,Opname),
asserta(genex_clause(operator(Op),[])),
append(Oname,"_reg_gene",Rname),
```

```
name(R,Rname),
asserta(genex_clause(reg_gene(R),[])),
append(Oname,"_reg_protein",RPname),
name(RP,RPname), % hack so we can use reg-protein %
asserta(genex_clause(product(R,RP),[])),  % in mutation rules %
append(Oname,"_terminator",Tname),
name(T,Tname),
asserta(genex_clause(terminator(T),[])),
print(O),
print(' structural genes: '),
print('enter one at a time, 5prime - 3prime order'),
nl,
print(' -- end with "done"'),
nl,
make_struct_genes(O),
print('enter sequence of '),print(O),print(' or []'),
nl,read(Seq),
asserta(genex_clause(sequence(O,Seq),[])).

make_struct_genes(O) :- print('structural gene:'), nl,
read(X), make_struct_gene(O,X).
make_struct_gene(O,'done').
make_struct_gene(O,X) :-
 asserta(genex_clause(structural_gene(O,X),[])),
 print('what is product of '),print(X),nl,
 read(Y),
 asserta(genex_clause(product(X,Y),[])),
 print('what is purpose of '),print(Y),
 print(': (synthesis, digestion, or other)'),nl,
 read(P),
 asserta(genex_clause(purpose(Y,P),[])),
 make_purpose(Y,P),
 make_struct_genes(O).

make_purpose(Y,'synthesis') :- print('what does '),print(Y),
print(' synthesize?'),nl,
read(End_prod),
asserta(genex_clause(end_product(Y,End_prod))).
make_purpose(Y,'digestion') :- print('what does '),print(Y),
print(' digest?'),nl,
```

```
read(Substrate),
asserta(genex_clause(substrate(Y,Substrate))).
make_purpose(Y,'other').
get_mutation_sites([],[]).
get_mutation_sites(Sites, [Site|Rest]) :-
print('mutation in '),print(Site),print(' (yes or no) ?'),
nl,read(Ans),
analyse(Ans,Site,Add),
get_mutation_sites(Rest_of_sites, Rest),
append(Add,Rest_of_sites,Sites).


analyse('yes',Site,[Site]) :-  genex_clause(X,[]),
functor(X,Site,_),
arg(1,X,Op_site),
asserta(genex_clause(mutated(Op_site),[])).
analyse('no',Site,[]).
analyse('deletion',promoter,[]) :- retract(genex_clause(promoter(X),[])).
analyse('deletion',operator,[]) :- retract(genex_clause(operator(X),[])).
analyse('deletion',reg_gene,[]) :- retract(genex_clause(reg_gene(X),[])).
analyse('deletion',struct_genes,[]) :-
                                retract(genex_clause(struct_genes(X),[])).

append([],Y,Y).
append([A|X],Y,[A|Z]) :- append(X,Y,Z).

% the following program makes a list of Uncertain_vars (arg3) for the
%
% operon O using the list of mutated regions of the operon (arg2) %

set_uncertain_vars(_,[],[]).
set_uncertain_vars(O, [Site|Rest], Uncertain_vars) :-
same(Site,promoter) ->
(genex_clause(promoter(P),[]),
 set_uncertain_vars(O,Rest,Rest_of_vars),
 append([complementary_conform(rna_pol,P)],
 Rest_of_vars,
 Uncertain_vars));
same(Site,operator) ->
(genex_clause(operator(Op),[]),
```

```
 genex_clause(promoter(P),[]),
 genex_clause(reg_gene(RG),[]),
 genex_clause(product(RG,RP),[]),
 set_uncertain_vars(0,Rest,Rest_of_vars),
                append([complementary_conform(RP,overlap_region(P,Op)),
  mutated(overlap_region(P,Op)),
  mutated(operator_specific(Op))],
 Rest_of_vars,
 Uncertain_vars));
same(Site,struct_genes) -> true;
same(Site,reg_gene) ->
(genex_clause(reg_gene(RG),[]),
 genex_clause(product(RG,RP),[]),
 genex_clause(operator(Op),[]),
 genex_clause(promoter(P),[]),
 set_uncertain_vars(0,Rest,Rest_of_vars),
 append([complementary_conform(RP,overlap_region(P,Op)),
 complementary_conform(X,smbs(RP)),
 part_of(RP,smbs(RP)),
 mutated(RP),
 active_smmol_conform(RP)],
 Rest_of_vars,
 Uncertain_vars));
same(Site,unknown) -> % if mutation site is not known %
(genex_clause(promoter(P),[]),
 genex_clause(operator(Op),[]),
 genex_clause(reg_gene(RG),[]),
 genex_clause(product(RG,RP),[]),
 genex_clause(structural_gene(0,X),[]),
 set_uncertain_vars(0,Rest,Rest_of_vars),
 append([mutated(P),mutated(Op),mutated(RG),mutated(X),
complementary_conform(rna_pol,P),
complementary_conform(RP,overlap_region(P,Op)),
complementary_conform(Smmol,smbs(RP)),
mutated(overlap_region(P,Op)),
mutated(operator_specific(Op)),
 active_smmol_conform(RP)],
 Rest_of_vars,
 Uncertain_vars)).
```

```
any_other_info(0) :- read(X), do_other_info(0,X).
do_other_info(0,'done').
do_other_info(0, Info) :- asserta(genex_clause(Info,[])),any_other_info(0).
```

GENEX also uses a Prolog set manipulation utility package written by Lawrence Byrd and R.A. O'Keefe.