
Splicing and regularity

Tom Head and Dennis Pixton

Binghamton University

Splicing Languages and Regularity

1 Introduction

In 1987 a new formalism for the generation of languages was introduced and has since received extensive theoretical development. This new formalism, *splicing*, was inspired by a study of the recombination of DNA molecules carried out in laboratories of molecular biology. For several decades it has been realized that molecules of DNA, RNA, and proteins may be idealized as strings of symbols over finite alphabets with the symbols chosen to denote deoxyribonucleotides, ribonucleotides, and amino acids, respectively. Let us adopt this view and consider how we might represent the making of one cut in each of two molecules m and m' and the construction of a new molecule w by attaching a left portion of m to a right portion of m' : We let $m = pq$ and $m' = uv$ where we plan to cut m between its subsegments p and q and m' between its subsegments u and v . This is easy; the new molecule is represented by the string $w = pv$. It is natural to say that we have cut the molecules represented by m and m' and *spliced* the left segment of m with the right segment of m' , producing the recombined molecule w . There are marvelous tools used in molecular biology called *restriction enzymes*. These tools allow us to cut (double stranded) DNA molecules at precisely specifiable positions. The resulting segments can be attached using an enzyme called a *ligase*. Consequently, in the presence of an appropriate restriction enzyme and a ligase, from such molecules $m = pq$ and $m' = uv$, a new molecule $w = pv$ may be produced. This is a greatly simplified indication of a technology used in *gene splicing*, a fundamental feature of *genetic engineering*. A detailed explanation of the origin of the splicing concept from a study of biomolecular science is the content of Section 2. The present Chapter has been organized so that a reader who does *not* wish to be concerned with the biomolecular roots of splicing can skip Section 2 and hasten to the formal theory of splicing systems and the languages they generate which is resumed in Section 3. Other readers may find Section 2 to be of great interest in motivating the formal concepts introduced in this Chapter. In the remainder of this Section definitions that

are required in *all* further Sections are given. These definitions constitute the beginning of a logical theory that can stand alone independent of the context of its origin in the study of the biomolecular sciences. See [6] for extensive developments and references concerning the theory of splicing systems and languages.

Definition 1. *Let A be a finite set for use as an alphabet. Let A^* be the free monoid consisting of all strings of symbols of A , including the empty string λ . By a splicing rule we mean an ordered quadruple $r = (u, u'; v', v)$ with u, u', v', v in A^* . From any ordered pair of strings that admit factorizations $puu'q$ and $xv'vy$, with p, q, x, y in A^* the string $pvuy$ is said to be generated by r .*

For each of the nine Examples below, the alphabet is the set $A = \{a, c, g, t\}$.

Example 1. Let $r = (g, gatcc; a, gatct)$. From the ordered pair of strings $u = aaaggatccgg$, $v = ttogatctccc$ the string $w = aaaggatctccc$ is generated by r . Note that r generates no string at all from the ordered pair v, u , not even the empty string λ .

If we had chosen the rule $r' = (a, gatct; g, gatcc)$ then from the ordered pair v, u the string $w' = ttogatccgg$ would have been generated by r' and no string would have been generated from the ordered pair u, v by this rule r' .

Example 2. Let $r = (cg, cg; cg, cg)$. From the ordered pair of strings $u = aacgcgaacgcgaa$, $v = ttcgcgtt$ there are two strings that are generated by r , one for each of the two occurrences of the substring $cgcg$ in u . Thus $aacgcgtt$ is generated from this ordered pair, but so is $aacgcgaacgcgtt$. Note that from the ordered pair v, u the strings $ttcgcgaacgcgaa$ and $ttcgcgaa$ are generated by r .

Example 3. Let A, r , and u be as in the Example 2. Note that for the ordered pair u, u there are two choices of the substring $cgcg$ in each member of the pair. Thus four pairs of choices are possible. From two of these choices, r generates a repeat of the original string u . From the remaining two choices r generates the new strings $x = aacgcgaa$ and $y = aacgcgaacgcgaacgcgaa = (aacgcg)^3aa$. From the ordered pair u, y there are several choices of segments $cgcg$, but only one new string, $z = (aacgcg)^4aa$, is generated by r . In fact, from each ordered pair of strings $u, (aacgcg)^i aa$ with $i \geq 2$ one new string $(aacgcg)^{i+1}aa$ is generated by r . Apparently from the ordered pair u, u , each string in the regular language $\{(aacgcg)^i aa : i \geq 1\}$ can be generated by *iterating* the application of the rule r .

Definition 2. *Let A be an alphabet, let L be a language in A^* , and let R be a set of splicing rules. We define $R(L) = \{w \in A^* : \text{There exist strings } u, v \in L \text{ and a rule } r \in R \text{ for which } w \text{ is generated by } r \text{ from the ordered pair } u, v\}$. Thus $R(L)$ consists of all the strings that can be generated from ordered*

pairs of strings in L by a single application of one of the rules belonging to R . Let $R^0(L) = L$ and, for each non-negative integer i , let $R^{i+1}(L) = R^i(L) \cup R(R^i(L))$. The language $R^*(L) = \bigcup \{ R^i(L) : i \geq 0 \}$ is said to be the language generated from L through iterated application of the rule set R .

Example 4. For $L = \{aa, aacgcgaacgcgaa\}$ and the rule set $R = \{r\}$, where $r = (cg, cg; cg, cg)$, $R^*(L) = (aacgcg)^*aa$. Note that $(aacgcg)^0aa$ and $(aacgcg)^2aa$ are in $R^*(L)$ since each is in L and $aacgcgaa$ and each $(aacgcg)^i aa$, with $i \geq 2$, are in $R^*(L)$ by Example 3.

Example 5. For $L = \{c, ac, ca\}$ and $R = \{(a, c; \lambda, ac), (ca, \lambda; c, a)\}$, $R^*(L) = a^*c + ca^*$. Note that if we adjoin aca to L then $R^*(L) = a^*ca^*$.

Example 6. For $L = \{c, caa\}$ and $R = \{r\}$, where $r = (caa, \lambda; c, \lambda)$, $R^*(L) = c(aa)^*$.

Example 7. For $L = \{a, c, g, t\}$ and $r = (\lambda, \lambda; \lambda, \lambda)$ we have: $R(L) = \{\lambda\} \cup L \cup L^2$; $R^0(L) = L$, $R^1(L) = \{\lambda\} \cup L \cup L^2$, $R^2(L) = \{\lambda\} \cup L \cup L^2 \cup L^3 \cup L^4$, and $R^*(L) = \bigcup \{ L^i : i \geq 0 \} = A^*$. We sketch a computation of $R(L)$: Since $a = \lambda\lambda a = a\lambda\lambda$, applying r to the ordered pair consisting of the latter two factorizations, we see that $\lambda\lambda = \lambda$ must be in $R(L)$. Applying r to the ordered pair $a\lambda\lambda$ and $a\lambda\lambda$ we see that $a\lambda\lambda = a$ must be in $R(L)$. Applying r to the ordered pair $a\lambda\lambda$, and $\lambda\lambda a$, we see that $a\lambda\lambda a = aa$ must be in $R(L)$. Applying r to the ordered pair $a\lambda\lambda$ and $\lambda\lambda c$, we see that $a\lambda\lambda c = ac$ must be in $R(L)$. That $R(L) = \{\lambda\} \cup L \cup L^2$ follows by replacing the a and c in all possible ways from the alphabet set $A = \{a, c, g, t\}$.

It seems natural to view the splicing of strings as an operation related to the (iterated) concatenation of strings. In the following example the symbol t is reserved to function as an end marker.

Example 8. For $A = \{a, c, g\}$, $L \subseteq A^*$ and $R = \{(s, t; t, s') : s, s' \in A\}$, $R(tLt) = tLLt$ and $R^*(tt \cup tLt) = tL^*t$.

Definition 3. A splicing system is an ordered pair $S = (R, I)$, where R is a set of splicing rules, $R \subseteq A^* \times A^* \times A^* \times A^*$, and I is an initializing language, $I \subseteq A^*$. The language $R^*(I)$ is said to be the language generated by the splicing system $S = (R, I)$. A language L is a splicing language if $L = R^*(I)$ for some splicing system $S = (R, I)$. A splicing system $S = (R, I)$ is said to be a finite splicing system if both R and I are finite sets. By a finite splicing language we mean a language that is generated by a finite splicing system.

We say that a language L is preserved by a splicing rule r if, for every ordered pair of strings x, y in L and for every string z that can be obtained by splicing x and y using rule r , we have z in L . This allows us to describe the language $L = R^*(I)$ generated by the splicing system $S = (R, I)$ as the

smallest language that contains I and is preserved by each of the splicing rules in the set R .

Each finite language L is a finite splicing language since it is generated by the finite splicing system $S = (R, I)$, where R is empty and $I = L$. Examples 4, 5, 6 and 7 demonstrate that the five *regular* languages $(aacgcg)^*aa$, $a^*c + ca^*$, a^*ca^* , $c(aa)^*$, and A^* itself are all finite splicing languages. That *all finite splicing languages are regular* is a major result of splicing theory. This substantial result is proved in Section 3. However, not all regular languages are finite splicing languages: It is not difficult to prove that the regular language $L = (aa)^*$ is *not* a finite splicing language even though it is a very close relative, even a homomorphic image, of the finite splicing language $c(aa)^*$ of Example 6. In fact, for every regular language L and any symbol that does not occur in any string in L , say c , cL is a splicing language and L is an image of cL under the homomorphism that erases the symbol c and leaves all other symbols unchanged. See Corollary 2 in Section 5.

Example 9. That the regular $L = a^*ca^*ca^*$ is *not* a finite splicing language can be seen as follows: Suppose that $r = (u, u'; v', v)$ is a rule that preserves L . If either uu' or $v'v$ is not a subsegment of any string in L , then r trivially preserves L , but it generates no string at all from any pair of strings in L . Thus if r generates any string at all from a pair of strings in L , then the number of occurrences of the symbol c in each of uu' and $v'v$ is either 0, 1 or 2. If the number of occurrences of c in either uu' or $v'v$ is either 0 or 1, then one can easily specify two strings in L from which r generates a string containing more than two occurrences of the symbol c . The same is true if uu' and $v'v$ each contain 2 occurrences of the symbol c , unless each of the four substrings u , u' , v' , v contains precisely one occurrence of c . In this last case there are at most a finite number of non-negative integers n for which new strings of the form $a^*ca^nca^*$ can be generated by r . Consequently, for every finite set R of rules that preserve L and any finite subset I of L , there is a positive integer N for which the language generated by $S = (R, I)$ is contained in $\{a^*ca^kca^*: 0 \leq k \leq N\}$. Consequently L is not generated by any finite splicing system.

Definition 4. A splicing system $S = (R, I)$, is said to be reflexive if, for each rule $r = (u, u'; v', v)$ in R , the language that S generates is preserved by the rules $\dot{r} = (u, u'; u, u')$ and $\ddot{r} = (v', v; v', v)$.

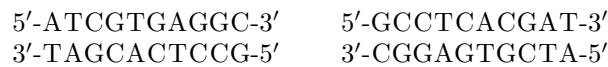
In Section 2 it is explained how biomolecular considerations suggest that the reflexive systems be given special attention. Note that a splicing system (R, I) is surely reflexive if, for any rule r in R , both \dot{r} and \ddot{r} are also in R , although they do not need to be listed in R if they do not directly contribute to the generation of the splicing language. The splicing systems from which the five previously listed splicing languages $(aacgcg)^*aa$, $a^*c + ca^*$, a^*ca^* , $c(aa)^*$, and A^* were generated can be verified to be reflexive.

Exercises

1. Let $A = \{a, c, g, t\}$ serve as the alphabet. For $r = (a, a; c, c)$ and $R = \{r\}$, find the three languages $R^*(\{aacc\})$, $R^*(\{aaa, ccc\})$, and $R^*(\{aaccaacc\})$.
2. Let $A = \{a, c, g, t\}$. For $r = (a, a; c, c)$, $r' = (a, c; a, c)$ and $R = \{r, r'\}$, find the languages $R^*(\{aacc\})$, $R^*(\{aaa, ccc\})$, and $R^*(\{aaccaacc\})$.
3. Let $A = \{0\}$. For $R = \{(00, \lambda; \lambda, 00)\}$, find $R^*(\{\lambda, 00\})$. [Be careful here.]
4. Let $A = \{1\}$. Find three different splicing systems that generate 1^* .
5. Let $A = \{0, 1\}$. Find a splicing system $S = (R, I)$ that generates the language 1^*0^* .
6. Which of the splicing systems in the first five exercises is reflexive?
7. Let $A = \{0\}$. Prove that the language $(00)^*$ is not the language generated by any finite splicing system.

2 The Biomolecular Inspiration for the Splicing Concept

This Section is provided for those readers who wish to be acquainted with the molecular behaviors from which the splicing concept was developed as an abstraction. This Chapter is organized to allow the reader who has completed Section 1, but does *not* wish to be concerned with the relevant biomolecular science, to proceed immediately to Section 3, skipping this entire Section 2. For brevity we focus attention on linear fully double stranded DNA molecules (ds-DNA). Although ds-DNA molecules tend to occur in the famous double helical form discovered by J. Watson and F. Crick, it is adequate for our purposes here to ignore the helical form and represent ds-DNA molecules in the form exemplified in this display:



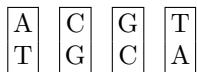
For this paragraph only, consider the display as representing *four* single stranded DNA (ss-DNA) molecules. Each of the four upper case letters, A, C, G, T, represents one of the four deoxyribonucleotides of which each ss-DNA molecule is viewed as a chain. Each deoxyribonucleotide has the structure of a ribose (pentagonal) sugar that contains five carbon atoms that are numbered 1', 2', 3', 4', and 5'. A phosphate is attached at the 5' carbon. At the 1' carbon the essential information is attached in the form of one of the four nuclear bases: either adenine, A, cytosine, C, guanine, G, or thymine, T. The individual nucleotides are linked by the phosphate at the 5' carbon of one ribose ring being attached to the 3' carbon of the next ribose ring. This allows us to speak of an ss-DNA molecule as consisting of a 'sugar-phosphate backbone' having one of the four bases, A, C, G, or T attached to each sugar. One end of an ss-DNA has a 3' carbon to which no phosphate is attached. The

other end has a phosphate protruding from a 5' carbon but not attached to any other carbon. It is very important to understand the distinction between the 5' end and the 3' end of an ss-DNA molecule. Consider the upper left ss-DNA in the display above. The left most nucleotide incorporates adenine and has a phosphate protruding from its 5' end. The right most nucleotide incorporates cytosine and has no phosphate attached at its 3' carbon. Each of the four ss-DNAs in the display can now be understood. All the chemical bonds in the ss-DNA molecules are strong (covalent) chemical bonds. When we wish to break one of these bonds we use special molecular tools (enzymes). Mild changes of conditions, such as moderate increases in temperature, will not break these bonds.

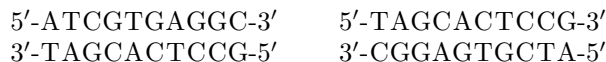
In this paragraph consider the display above as representing *two* ds-DNAs. The first principle concerning the formation of ds-DNA molecules from two ss-DNAs is that the 5' \rightarrow 3' orientations of the two strands must be opposites. Note that this holds in the display above. The bonds that hold the two ss-DNAs in association are weak (hydrogen) bonds. This allows the two strands of a ds-DNA to be separated into its component ss-DNAs by a moderate increase in temperature. In (perfectly formed) ds-DNA, a nucleotide A in one strand bonds only to a T in the other strand and vice versa. Likewise a C in one strand bonds only to a G in the other strand and vice versa. These conditions are the so-called *Watson-Crick pairing* conditions. In the display above each of the two ds-DNAs consists of a pair of adjacent ss-DNAs that are perfectly matched as Watson-Crick hydrogen bonded pairs.

For the reader who progresses to more subtle considerations than are required here, it will be worth noting that A and T are bonded through two hydrogen bonds, but C and G are bonded through three. As a consequence, less energy is required to pull A-T pairs apart than to pull C-G pairs apart.

In modeling there sometimes arise subtle points that must be taken into account that need not arise in the formal theory of splicing. Here is the *first such point*: A formal string of symbols 'lives' in a one-dimensional space (the ordered line). A molecule 'lives' in three-dimensional (not linearly ordered) space. However, consider again the display above. As *formal symbol strings* over the alphabet, D, consisting of the *four* compound *bi-level* symbols:

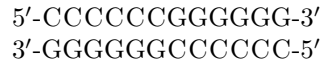


they are *quite different*. However, as representations of *molecules* they are *the same*: If either of these two molecules



is rotated 180° about its mid-point while remaining in the plane of this page, the 'other' molecule is obtained. This means that ds-DNA *molecules* are actually represented by *equivalence classes of strings*, where each equivalence

class consists of either exactly *two strings* or, sometimes only *one string*: The string



when rotated 180° about its center, remains the same string. Chemists say that such a ds-DNA molecule has *dyadic symmetry*. The equivalence class of such a molecule is a *singleton*.

As tools to cut ds-DNA we use enzymes that are called *restriction endonucleases*, or, more casually, *restriction enzymes*. Such enzymes are naturally occurring products produced by bacteria. There are currently over 200 different restriction enzymes available from laboratory supply houses. One such enzyme is *BamH I*. When this enzyme encounters a ds-DNA molecule m in an aqueous (water) solution it can cut m into two pieces if m has a six base pair segment with the sequence:



Suppose that m has the form



where each of the symbols X and Y denotes an unspecified symbol (a 'don't care symbol') from $\{A, C, G, T\}$ and where we require only that each vertically displayed pair is Watson-Crick compatible. A cut by *BamH I* at the indicated site in the molecule m results in two molecules neither of which is fully double stranded:



Notice that the (strong) covalent bonds GG in both the upper strand and the lower strand have been cut by *BamH I*. With these two covalent bonds cut, there is insufficient strength provided by the (weak) hydrogen (vertically displayed) bonds to hold the left and right portions of the molecule together. Thermal agitation in the solution is adequate to disrupt such a four term sequence of hydrogen bonds. It is still true that these two halves are attracted by their potential for forming again these hydrogen bonds. However, if they do form again, without a means to have the covalent GG bonds restored, they will separate once again. The two projecting four term single stranded segments are called *overhangs* or *sticky ends*. If we remove *BamH I* from the solution and add in its place an enzyme called a *ligase* then, when two mutually compatible sticky ends and a ligase enzyme come sufficiently close together the ligase will re-establish the two previously cut strong GG bonds, restoring the original fully double stranded DNA molecule.

Here is the *second point* that must be taken into account in modeling the behavior of molecules in aqueous solution: Suppose that (at least) two molecules are present that have the sequence given above for m . Then after each is cut with *BamH I*, there are two distinct left segments (and two distinct right segments) having sticky ends. Since these segments are in water (in three dimensional space, not on a line on a page), when *BamH I* is replaced by a ligase, two left segments can relate as illustrated:



which allows them to form hydrogen bonds and be ligated to form a ds-DNA molecule. Two right segments can likewise pair together to form a ds-DNA. Consequently the cut operation with *BamH I*, followed by the paste operation provided by a ligase, can potentially yield any one of the three ds-DNA molecules:

1. the original molecule m ,
2. the molecule:



3. the molecule:



In wet lab experiments the number of ds-DNA molecules of a given sequence used is vast (perhaps trillions). Consequently molecules of each sequence that are *enabled to appear* are *expected to appear*. In the present case the three molecules indicated above would surely arise.

When the restriction enzyme *Bgl II* encounters a ds-DNA molecule m' in solution it can cut m' into two pieces if m' has a six base pair segment with the sequence:



Suppose that m' has the form



where each of the symbols U and V denotes an unspecified symbol from $\{A, C, G, T\}$ and each vertically displayed pair is Watson-Crick compatible. A cut by *Bgl II* at the indicated site in m' results in the two molecules



The covalent bonds between A and G (5'-AG-3' in the upper strand and 3'-GA-5' in the lower strand) have been cut by *Bgl* II.

Suppose now that we have a solution containing the molecules m and m' for which the sequences have been suggested in the two preceding paragraphs. Suppose that both *Bam*H I and *Bgl* II are also present in the solution. Then the four segments, two each from m and m' will result. We have chosen this pair of restriction enzymes because they produce *the same sticky ends*. Consequently when these enzymes are replaced by a ligase the following (recombinant) molecule may arise:



There are several other molecules that may arise, but this one immediately above will motivate the precise definition of *the splicing operation* applied to the *ordered pair* m, m' . As a valuable exercise the reader may wish to determine the seven further ds-DNA molecules that may arise, in addition to the two original molecules and the one exhibited above. Note that, when a segment of m is combined with a segment of m' , the result is a molecule which no longer has a site that can be cut by either *Bam*H I or *Bgl* II.

We make one more simplification of the notation for ds-DNA molecules before tying our molecular considerations to formal splicing theory as introduced in Section 1. In a ds-DNA molecule, with fully and perfectly matched single strands, each single strand determines the other. Consequently we need to record only one strand of a ds-DNA when we know that the strands are fully and perfectly matched (with no sticky ends). We therefore make the convention of representing a ds-DNA by listing only one strand. This would require labeling at least one of the two ends as 3' or 5'. We make the further convention that, when no label is given for either end of a string displayed on a line, the 5' end is the *left* end. Finally, when these conventions are used we employ the lower case. These denotations apply also to enzyme sites.

Summarizing these molecular examples:

1. *aaaggatccgg* denotes the molecule



2. *ccggatcctt* denotes *the same molecule* as in part 1. [Be careful here.]
3. *ttagatctcc* denotes the molecule

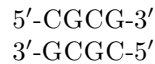


4. *Bam*H I and *Bgl* II cut at segments of the form *ggatcc* and *agatct*, respectively.

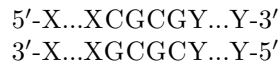
We have seen above that *Bam*H I and *Bgl* II cut in such a way that they produce identical sticky ends. We observed that molecules m , m' when cut by *Bam*H I and *Bgl* II and provided with a ligase may produce a fully and perfectly matched ds-DNA *recombinant* molecule consisting of the left portion of m and the right portion of m' . The splicing concept was developed to provide a context in which strings are spliced (recombined) in simulation of the recombination procedures carried out on ds-DNA molecules in the presence of sets of restriction enzymes and a ligase. The original formulation of splicing [2] was kept very close to the biochemistry. There is still merit in thinking in terms of the original model when considering actual molecular processes. However the precision of the original formulation is maintained and made more elementary to handle in mathematical proofs by employing the improved formulation given by George Păun. For developing formal language theoretic splicing, Păun's notation used here (in Sec. 1 and below) is much more appropriate. The result of cutting with enzymes and recombining with a ligase can be expressed by an ordered quadruple of strings. The example appropriate for representing the action of *Bam*H I, *Bgl* II, and a ligase will make clear the general procedure. The formal representation of the generative capacity of any set of restriction enzymes and a ligase acting on a set of ds-DNA molecules can be represented in the fashion illustrated here:

The compatibility of the sticky ends produced by cuts with *Bam*H I and *Bgl* II is conveyed by listing the sites at which these enzymes act, *ggatcc* and *agatct*, respectively, in this order in an ordered tuple. The extra information that specifies which symbols in the sites yield the sticky ends can be included by further segmenting each of the sites to produce the ordered quadruple $r = (g, gatcc; a, gatct)$ that constitutes the splicing rule r that we associate with the *ordered pair* of enzymes *Bam*H I, *Bgl* II. The reader will now profit from rereading Section 1, Example 1, which is a purely formal version of the present discussion. The two sites relevant for rule r in the ordered pair of strings (molecules) *aaaggatccgg* and *ttagatctccc* [listed as molecular examples 1 and 3 above] are illustrated by segmenting these two strings appropriately: *aaa-g,gatcc-gg* and *tt-a,gatct-ccc*. From the rule r and the segmentations we see that we splice at the commas to produce the recombinant string (molecule) *aaa-g,gatct-ccc = aaaggatctccc*.

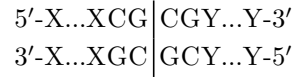
We give one more molecular example. Example 2 of Section 1 was constructed as a purely formal version of this next molecular example. The restriction enzyme *Bst*U I cuts a ds-DNA molecule m in aqueous solution into two pieces if m has a four base pair segment with the sequence:



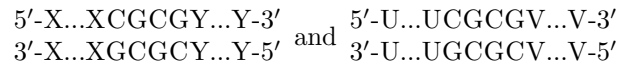
Suppose that m has the form



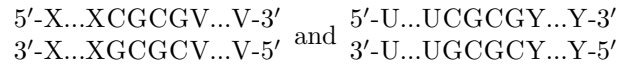
where each of the symbols X and Y denotes an unspecified symbol (a 'don't care symbol') from {A, C, G, T} and where each vertically displayed pair is Watson-Crick compatible. A cut by *Bst*U I at the indicated site in the molecule *m* results in two molecules both of which are fully double stranded:



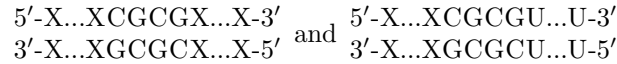
Notice that the (strong) covalent bonds 5'-GC-3' (= 3'-CG-5') in both the upper strand and the lower strand have been cut by *Bst*U I. Note that this enzyme does not provide single stranded overhangs as do the enzymes *Bam*H I and *Bgl* II. Enzymes such as *Bst*U I are said to leave *blunt ends*. Even though cutting leaves blunt ends, re-attachment of the resulting blunt ends can be done using a ligase. Thus even the blunt ends should be regarded as being sticky. Consequently, when molecules



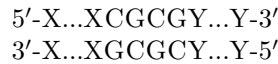
are cut by *Bst*U I, and a ligase is then applied, the following molecules may arise:



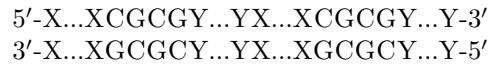
Don't overlook the fact that the two original molecules may be reconstructed and that six additional new molecules may arise, exemplified by the two molecules:



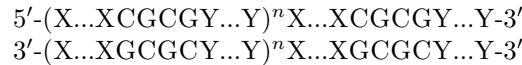
The astute reader may wish to ask whether when a ligase is added to a solution that is *initialized* to contain molecules such as



concatenations of these molecules, such as



and



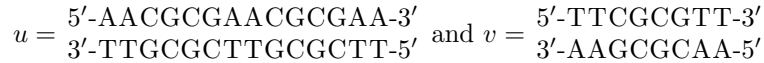
may arise. Would blunt end ligation allow these additional molecules to be produced? *This point is subtle and may be glossed over by readers interested only in the formal theory.* The answer is 'no', but only by the convention

of assuming that the *initially given* DNA molecules are not provided with a phosphate attached at the 5' end. The ligase requires the presence of a phosphate at the 5' end in order to complete the necessary covalent bond. When a restriction enzyme cuts a ds-DNA molecule it leaves the phosphate attached at the freshly produced 5' end. This allows re-ligation at ends created by such cuts. Researchers working with DNA have the biochemical means of either *having* or *not having* phosphates attached at the 5' ends of the *initial* DNA molecules. Thus our 'default' assumption is that our *initial* DNA molecules have no phosphates at their 5' ends. (In short, a blunt end is 'sticky' if and only if a phosphate is present on its 5' strand.) If one wishes to have all possible concatenations to be potentially present, then simply begin with DNA for which phosphates are present at the 5' ends. (Note: blunt end ligation is quite effective in circularizing linear molecules having blunt ends.)

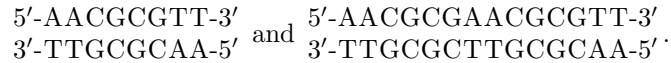
We have noted that the site at which the restriction enzyme *Bst*U I cuts is



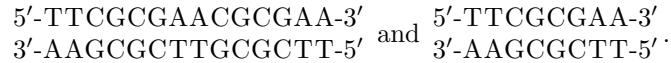
Using our convention for compressing ds-DNA sequences, we specify this site in the short form: *cgcg*. Attending to the fact that *Bst*U I cuts exactly in the middle of its site, it is natural for us to model its recombining capacity (when accompanied by a ligase) with the rule: $r = (cg, cg; cg, cg)$. This is the rule used in Example 2 of Section 1. As Example 2 is re-read now one sees that it is an abstract representation of the fact that when *Bst*U I and a ligase are added to an aqueous solution containing the molecules



the following two recombinant ds-DNA molecules may arise (from this ordered pair u, v):



Moreover (from the ordered pair v, u) these two recombinant molecules may also arise:



One may note that the ordered pairs u, u and v, v provide yet more recombinant molecules.

Examples 3 and 4 of Section 1 should now be thought through again, this time as providing abstract descriptions of recombinant behaviors of ds-DNA molecules made possible by the presence of *Bst*U I and a ligase.

An astute reader may observe that we could just as well use for the rule r any one of several rules that will provide exactly the same generating power.

Two such alternates are $(cgcg, \lambda; cgcg, \lambda)$ and $(\lambda, cgcg; \lambda, cgcg)$. These rules do not suggest the same cutting behavior as $(cg, cg; cg, cg)$. However, if one c is not considered different from another and likewise for g , then each of these rules is exactly equivalent in generative power to another. The rule $(cgcg, \lambda; cgcg, \lambda)$ is said to have only *left context* and the rule $(\lambda, cgcg; \lambda, cgcg)$ is said to have only *right context*. Splicing systems for which all rules have one sided context generate especially simple types of languages [3]. If some bases are labeled (by radioactivity, fluorescence, or otherwise) then a difference in effect of such rules can be detected, but not otherwise.

When one wishes to model with a splicing system the full generative power of a set of ds-DNA molecules under the action of a set of restriction enzymes and a ligase, then an awareness of all the considerations discussed in this Section 2 is required. This is illustrated with a single example: Suppose that it is desired to model the set of fully well formed linear ds-DNA molecules (with no sticky ends) that can potentially arise in an appropriate aqueous solution containing *Bam*H I, *Bgl* II and a ligase as ds-DNA molecules, each having the sequence $m = a^{20}ggatcca^{20}agatcta^{20}ggatcca^{20}$, are added to the solution. Recalling the sites at which *Bam*H I and *Bgl* II act one might too quickly respond by listing the splicing system $S = (R, I)$ where $R = \{r\}$, $r = (g, gatcc; a, gatct)$, and $I = \{m\}$. This would be woefully inadequate. By recalling that molecules live in three-dimensional space (not on a line), we must include the alternate linear representation of m , namely $m' = t^{20}ggatcct^{20}agatctt^{20}ggatcct^{20}$, in the initial language. When two molecules are cut by the same enzyme, the left segment of one can inevitably be ligated to the right segment of the other. Consequently the presence of *Bam*H I enables the rule $r_1 = (g, gatcc; g, gatcc)$ and *Bgl* II enables $r_2 = (a, gatct; a, gatct)$. (*This is the consideration that guarantees that the splicing systems that arise in the modeling discussed here are inevitably reflexive.*) Finally, two molecules in solution, when modeled on a line, must be allowed to occur in either order. Thus the rule $r_3 = (a, gatct; g, gatcc)$ is enabled. The appropriate splicing model for the representation of the generative activity of *Bam*H I, *Bgl* II and a ligase acting on molecules having the sequence $\{m\}$ is: $S = (R, I)$ where $R = \{r, r_1, r_2, r_3\}$ and $I = \{m, m'\}$.

Summary: What precisely is claimed when a splicing system is given as a model of the generative activity of a specified set of restriction enzymes and a ligase acting on ds-DNA molecules having sequences belonging to a specified set? *The answer includes some subtleties:* Let R be the set of rules that express the cutting and rejoining actions of the enzymes (understood as independent of the choice of the initial set). Let I be the set of strings that represent the initial ds-DNA molecules (understood as closed under 180° rotation). The language $R^*(I)$ that is generated consists of the strings that represent sequences of the fully double stranded DNA molecules (*without sticky ends*) which can potentially arise through the action of the given enzymes on *sufficiently* many ds-DNA molecules each having a sequence belonging to I . There is no need to make the unpleasant assumption of the existence of an *actual infinity* of

initial molecules although this is sometimes done informally. (Recall that the set of positive integers may be described as those numbers that can potentially arise through *sufficiently* many additions of one to an initial set consisting of a single one.) Finally, we remark that we have idealized away conditions that are too detailed to merit inclusion here (temperatures, buffering, etc.). For further details see [2] and [4].

Example Application: Given a finite set of restriction enzymes and a finite set of ds-DNA sequences, can a specified ds-DNA molecule, M , be constructed from sufficiently many ds-DNA molecules having sequences in the given set through the application of the given restriction enzymes and a ligase? **General procedure:** Construct the splicing system model for the given data. Section 3 contains an algorithm that produces a finite automaton that recognizes the language generated by the splicing system. M can be constructed by the specified means if and only if the automaton recognizes the string that represents M . In many special cases the general algorithm can be by-passed and simpler procedures can be used; see Section 5 for more discussion and further references. Procedures for contracting long sub-sequences to single auxiliary alphabet symbols may allow the longer ds-DNA molecules to be treated.

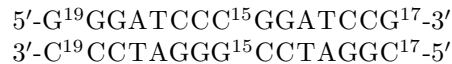
Exercises

1. List all the sequences of the ds-DNA molecules (without sticky ends) that can arise as ds-DNA molecules having the sequence



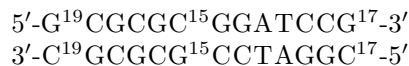
are added to an aqueous solution containing *Bam*H I and a ligase.

2. Find all the sequences of the ds-DNA molecules that can arise as ds-DNA molecules having the sequence



are added to an aqueous solution containing *Bam*H I and a ligase.

3. Find all the sequences of the ds-DNA molecules that can arise as ds-DNA molecules having the sequence



are added to an aqueous solution containing *Bam*H I, *Bst*U I, and a ligase.

4. Give splicing models, with alphabet $A = \{a, c, g, t\}$, appropriate for Exercises 2 and 3.

3 Regularity of the splicing languages

The goal of this section is to prove that finite splicing languages are regular. The proof of regularity requires that the set of rules is finite, but only requires that the initial language is regular. Here is the precise statement:

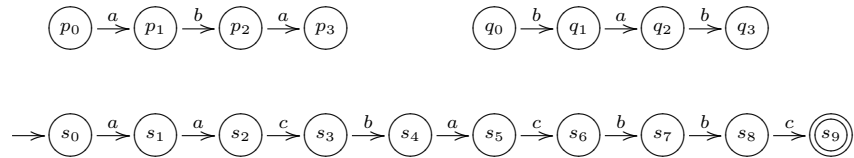
Theorem 1. *If R is a finite set of splicing rules and I is a regular language then the splicing language $R^*(I)$ is regular.*

This was first proved by Culik and Harju [5]; we follow the proof given in [7].

We shall prove the theorem by constructing an automaton which recognizes the splicing language $R^*(I)$. The automaton will be non-deterministic, with null transitions, and will be constructed in a number of stages. We start with an automaton $M = (Q, A, s_0, F, \delta)$ which accepts the initial language I , and we augment this automaton as follows. Suppose $r = (u, u'; v', v)$ is a rule in R , and let B_r be an automaton with initial state i_r and terminal set $\{t_r\}$ which accepts exactly the string uv . The details of B_r are not important; the obvious choice is just a linear graph leading from i_r to t_r with edges labeled by the symbols in uv . We arrange that the states of the automata B_r for different rules are disjoint from each other and from Q . Then we define a new automaton $M_0 = (Q_0, A, s_0, F, \delta_0)$ so that Q_0 is the union of Q and the state sets of the various B_r and δ_0 is the “union” of δ and the transition relations for the various B_r .

Notice that the initial state is not changed, nor is the set of terminal states. These are in Q , and there are no transitions between Q and the new states, so the new states cannot be used in accepting any string. Hence M_0 accepts the language I . We shall refer to the automaton B_r as the r -bridge, and to i_r and t_r as the entry and exit states for this bridge.

To illustrate the construction we use a very simple example. The alphabet is $\{a, b, c\}$, the initial language contains only $aacbaccbbc$, and there are only two rules, $r_1 = (ab, c; a, a)$ and $r_2 = (ba, c; b, b)$. Then we can represent M_0 graphically as follows:



Here the original state sets in Q are s_j , $0 \leq j \leq 9$, with s_9 being the only accepting state. The states p_0, \dots, p_3 are the bridge states for r_1 , so $i_{r_1} = p_0$ and $t_{r_1} = p_3$. Similarly q_0, \dots, q_3 are the bridge states for r_2 , so $i_{r_2} = q_0$ and $t_{r_2} = q_3$.

Now we construct a sequence of automata $M_k = (Q_0, A, s_0, F, \delta_k)$ by recursion, starting with M_0 as above. So we suppose that M_{k-1} has been constructed and we explain the construction of M_k . The only difference is that

M_k may contain certain null transitions that are not present in M_{k-1} . Suppose $r = (u, u'; v', v)$ is a rule in R . If i is a state in M_{k-1} satisfying

1. i is not an exit state of any bridge, and
2. some accepting path in M_{k-1} is in state i immediately before reading uu' as a substring,

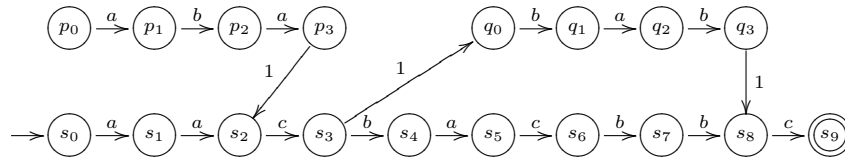
then we add a null transition from i to i_r ; and if t is a state in M_{k-1} satisfying

1. t is not an entry state of any bridge, and
2. some accepting path in M_{k-1} is in state t immediately after reading $v'v$ as a substring,

then we add a null transition from t_r to t . The automaton M_k is the result of adding all such null transitions to the automaton M_{k-1} .

The first type of transition, leading to some entry state i_r , is called an *entry transition*; and the second type, leading from some exit state t_r , is called an *exit transition*. Note that entry transitions cannot start at exit states and exit transitions cannot end at entry states, so this classification is unambiguous. An entry or exit transition is said to be of *level k* if it is not already present in the automaton M_{k-1} . This notion will be very important in the final phase of the proof.

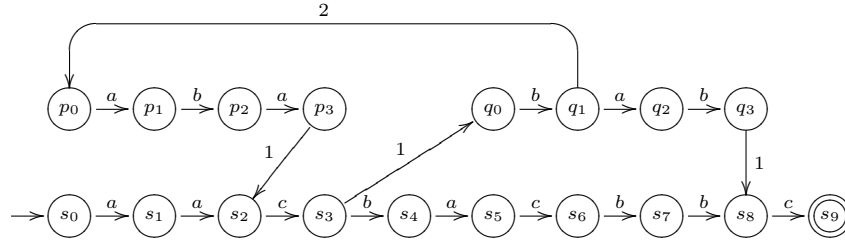
Continuing our example above, the only string accepted by the automaton M_0 is $aacbaccbc$. This contains the string bac , corresponding to the site uu' of rule r_2 . There is only one accepting path for this string, and that path is in the state s_3 just before reading the substring bac , so we add an entry transition from s_3 to q_0 , the r_2 entry state. We also find the string bb in $aacbaccbc$, which is the site $v'v$ of r_2 , and the only accepting path is in the state s_8 just after reading the substring bb . Thus we add an exit transition from q_3 , the r_2 exit state, to s_8 . Examining the other rule r_1 , we see that abc , the uu' site, does not occur in $aacbaccbc$, but that aa , the $v'v$ site, does occur. As above, we add an exit transition from p_3 , the r_1 exit state, to s_2 . Here is the resulting automaton M_1 ; we have labeled the new transitions with 1, but it should be remembered that these are null transitions.



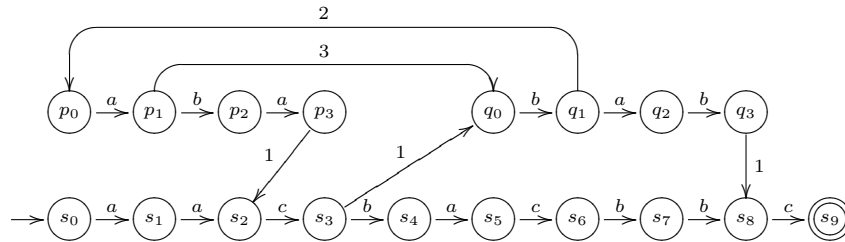
Now consider the language accepted by the automaton M_1 . Of course the initial string $aacbaccbc$ is still accepted. However, there is now another accepting path: From s_0 to s_3 reading aac , along the entry transition to the bridge for r_2 , from q_0 to q_3 along the bridge reading bab , along the exit transition to s_8 , and then from s_8 to s_9 reading c . Hence the string $aacbabc$ is accepted by M_1 . This is the only new string accepted by M_1 , since the states of the r_1 bridge do not participate in an accepting path. Also notice that $aacbabc$ is

the result of splicing two copies of $aacbabc$, using r_2 at the sites (ba, c) and (b, b) .

We now repeat the construction: We must examine the new string $aacbabc$ for any sites which require adding more entry or exit transitions. This string contains abc , the uu' site of r_1 . The only accepting path for this string is in the state q_1 just before reading abc , so we must add an entry transition from q_1 to p_0 , the r_1 entry state. There are no other new transitions, so the next automaton M_2 is as follows, where we have labeled the new transition with 2:



Notice that M_2 now has an accepting path which contains a loop (labeled by the non-empty string $babac$). Hence the language accepted by M_2 is infinite; it corresponds to the regular expression $aac(babac)^*(bacbbc + babc)$. In particular the string $aacbabc$ is accepted by M_2 and contains two copies of the uu' site bac corresponding to r_2 . The only accepting path for this string is in the state p_1 just before reading the first bac , so we need to add an entry transition from p_1 to the r_2 entry state q_0 . This accepting path is in the state s_3 just before the second copy of bac , but we do not need to add an entry transition from s_3 to q_0 since one already exists. Further examination of M_2 does not yield any other sites requiring entry or exit transitions, so our next automaton M_3 appears as follows, where we have labeled the new entry transition with 3:



There is another loop in the graph, following the path $q_0 \rightarrow q_1 \rightarrow p_0 \rightarrow p_1 \rightarrow q_0$ while reading the string ba , and the language accepted by M_3 is larger than the language accepted by M_2 . However, the reader should verify that there are no other opportunities for building entry or exit transitions, so our process stops here. We claim that the language accepted by M_3 is the splicing language defined by the splicing system (R, I) .

Rather than verifying that M_3 recognizes the splicing language in the example, we now explain why the process works in general. There are three steps in the proof:

1. The successive construction of automata terminates after a finite number of steps with an automaton M_n which does not allow further construction of entry or exit transitions.
2. The language L_n accepted by M_n contains I and is closed under splicing by rules in R , so L_n contains the splicing language $R^*(I)$.
3. Every string accepted by M_n is in the splicing language $R^*(I)$.

For the first step, consider that M_k differs from M_{k-1} only in the transition relation. Specifically, M_k allows the same transitions as M_{k-1} plus a number of null transitions that were not allowed by M_{k-1} . The total number of such null transitions is limited: Any such null transition represents a connection between a state in Q_0 and either i_r or t_r for some r in R ; so the total number of such null transitions is bounded by $2N_Q N_R$ where N_Q is the number of states in Q_0 and N_R is the number of rules in R . Hence it is impossible to continue forever finding such null transitions to add to the automaton so, for some n , the automata M_n and M_{n+1} are the same.

For the second step, first notice that M_n still contains all the states and transitions of the original automaton M , so M_n accepts every string that is accepted by M . That is, L_n , the language accepted by M_n , contains I , the language accepted by M . To show that L_n is closed under splicing by rules in R suppose that $r = (u, u'; v', v)$ is a rule in R and suppose that $w = xuu'y'$ and $w' = x'v'vy$ are both accepted by M_n . We must show that the result of splicing w and w' using r at the indicated sites is accepted by M_n . To do this we first analyze w . Choose an accepting path for $w = xuu'y'$ in M_n and follow the *longest* prefix of this path which reads the string x ; this leaves the automaton in the state i . If i is the exit state of any bridge then i is not a terminal state of M_n so our accepting path continues beyond i . But the only transitions leading from exit states are null transitions, so we can continue the path at least one more step while still reading just x , contradicting the choice of i . Hence i is not the exit state of any bridge, so, according to the construction of M_{n+1} from M_n , there is an entry transition in M_{n+1} from i to the r entry state i_r . Of course, since $M_{n+1} = M_n$, this entry transition is already available in M_n . Similarly, choose an accepting path for $w' = x'v'vy$ and follow the *shortest* prefix of this path which reads the string $x'v'v$. Arguing as above (but in the reverse direction) we see that t cannot be the initial state of any bridge, and so there is an exit transition from t_r , the r exit state, to t . Now we prescribe a new path in M_n : Follow the accepting path for $w = xuu'y'$ until x has been read and the automaton is in state i . Next follow the entry transition from i to i_r , follow the r bridge from i_r to t_r while reading uv , and then follow the exit transition from t_r to t . Finally, follow the rest of the accepting path for $w' = x'v'vy$ from t to an accepting state while reading y . In following this composite path the automaton reads $z = xuvy$, so z is in

L_n . This finishes the argument since z is the result of splicing $w = xuu'y'$ and $w' = x'v'vy$ using r at the indicated sites.

The third step is more difficult. We need to show that every accepting path in M_n accepts a string in the splicing language. The proof works by induction, but it is not clear at first glance what property of the path can be used as a basis for the induction. Obvious things to try are the length of the path, or the number of entry and exit transitions in the path, or the smallest subscript k so that the path lies in M_k . None of these seem to work. We shall, in fact, base the proof on the *complexity* of an accepting path, so we first define this notion and explain how to base an induction proof on it.

Recall that the level of an entry or exit transition is the subscript k so that the transition is in M_k but not in M_{k-1} . If π is a path in the automaton M_n and $1 \leq k \leq n$ we let $c_k(\pi)$ be the number of entry or exit transitions of level k in π . Then define the *complexity* of π to be the n -tuple $c(\pi) = \langle c_1(\pi), c_2(\pi), \dots, c_n(\pi) \rangle$. For an example, refer to the diagram for M_3 on page 19, where $n = 3$ and the entry and exit transitions are labeled with their levels. The only accepting path for the string $aacbabc$ has complexity $\langle 2, 0, 0 \rangle$ and the only accepting path for $aacbabababc$ has complexity $\langle 4, 2, 1 \rangle$.

In order to base an induction on complexity we need an order relation, and we use a variant of *lexicographic order*. If c and d are n -tuples of non-negative integers we say c is smaller than d , and write $c \prec d$, if, for some k , we have $c_j = d_j$ for all $j > k$ and, moreover, $c_k < d_k$. In other words, $c \prec d$ means that, when reading the n -tuples from the right, c is less than d in the first index in which they differ. For example, $\langle 9, 4, 3 \rangle$ is smaller than $\langle 1, 5, 3 \rangle$. It is easy to check that this relation is transitive and satisfies the trichotomy law: For any two n -tuples c and d , exactly one of $c \prec d$, $c = d$ or $c \succ d$ holds. In fact, this relation is a *well ordering*, which is the key idea in the proof of the following principle of induction.

Lemma 1. *Suppose P_c is a set of statements indexed by the set of n -tuples of non-negative integers c . Suppose*

If P_c is true for all $c \prec d$ then P_d is true.

Then P_d is true for all n -tuples d .

Proof. Let S be the set of all n -tuples d for which the statement P_d is *false*. The statement of the lemma is equivalent to the assertion that S is empty, so assume that it is *not* empty. The set of all n^{th} components d_n of members d of S is a non-empty set of non-negative integers; let s_n be the minimum of this set of non-negative integers, and let S_n be the set of members d of S for which $d_n = s_n$. Then S_n is not empty, and all the elements of S_n have the same n^{th} component. However, as above, the set of all $(n-1)^{\text{th}}$ components d_{n-1} of elements of S_n has a minimum element s_{n-1} .

After n steps of this process we are left with non-negative integers s_k for $1 \leq k \leq n$, so that the corresponding n -tuple $s = \langle s_1, s_2, \dots, s_n \rangle$ is in S and

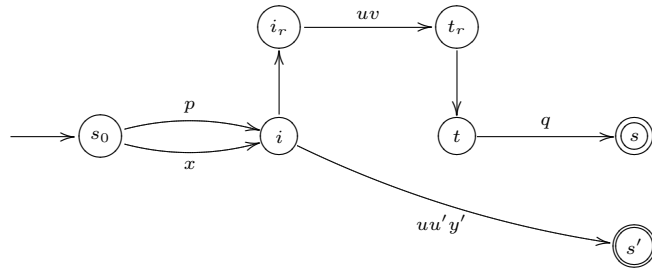
satisfies $s \prec d$ for all other n -tuples d in S . Hence for all c with $c \prec s$ we must have $c \notin S$, and so P_c is true. But from the assumption of the lemma we must then have P_s true, contradicting $s \in S$. This contradiction finishes the proof of Lemma 1.

So now we are ready to establish the third step of the proof. Suppose π is an accepting path in M_n , and assume that any string accepted by an accepting path of smaller complexity is in the splicing language. We must show that the string accepted by π is in the splicing language.

First consider the possibility that π never traverses an entry transition. Since π starts at s_0 in Q it can never reach a bridge, and so it can never reach an exit transition. Thus π has complexity $\langle 0, 0, \dots, 0 \rangle$ and it never leaves the automaton M . So the string accepted by π is in the language I accepted by M . Since I is contained in the splicing language, there is nothing more to say.

Alternatively suppose π contains an entry transition. Let the *last* entry transition in the path π be from the state i to the entry state i_r of the r bridge corresponding to some rule $r = (u, u'; v', v)$. Now the only way to leave this bridge is by an entry transition or by an exit transition from the exit state t_r of this bridge to some state t ; but we have assumed there are no following entry transitions on the path. Hence we can divide our path π into segments: From s_0 to i while reading some string p ; from i to i_r via an entry transition; from i_r to t_r along the r bridge while reading uv ; from t_r to t via an exit transition; and finally from t to some state in F while reading some string q . Thus the string accepted by the path is $z = puvq$. We need to identify two strings which are accepted by paths in M_n with smaller complexity and which, when spliced together, produce z .

Suppose the entry transition from i to i_r has level k . Remember the justification for creating this entry transition: It was constructed because there is an accepting path τ in M_{k-1} which accepts a string of the form $xuu'y'$ and is in state i after reading x . We define another path σ in M_n by following the path π from s_0 to i while reading p , and then following the path τ from i to an accepting state while reading $uu'y'$. That is, the string $w = puu'y'$ is read by the accepting path σ in M_n . We can visualize the three paths π , τ and σ (labeled with the strings they accept) as follows:



(This is just a schematic diagram; the path labeled p , for example, may contain loops and entry or exit transitions, and may intersect the other paths.)

We denote the subpath of π from s_0 to i as π_1 and the remaining subpath from i to an accepting state as π_2 . Similarly we denote the subpaths of τ before and after i as τ_1 and τ_2 . Thus σ consists of π_1 followed by τ_2 . Now we compare $c(\sigma)$ and $c(\pi)$. We have $c_j(\pi) = c_j(\pi_1) + c_j(\pi_2)$ and $c_j(\sigma) = c_j(\pi_1) + c_j(\tau_2)$. However, if $j \geq k$ we have $c_j(\tau_2) = 0$ since τ is an accepting path in M_{k-1} , so it does not contain any entry or exit transitions of level j . Hence $c_j(\sigma) \leq c_j(\pi)$. Moreover, π_2 contains the entry transition from i to i_r which is of level k , so $c_k(\pi_2) > 0$ and we conclude $c_k(\sigma) < c_k(\pi)$. So the first index j (reading from the right) at which $c(\sigma)$ and $c(\pi)$ differ must satisfy $j \geq k$ and $c_j(\sigma) < c_j(\pi)$; that is, $c(\sigma) \prec c(\pi)$.

Consider also the exit transition from t_r to t . Reasoning as above we first find an accepting path τ' in some $M_{k'}$ which accepts a string $x'v'vy$ and is in state t after reading $x'v'v$; then we connect the first part of this path with the last segment of the path π to form a path σ' in M_n . This path accepts the string $w' = x'v'vq$ and $c(\sigma') \prec c(\pi)$.

Since σ and σ' have smaller complexity than π the inductive hypothesis implies that w and w' are in the splicing language. If we splice $w = puu'y'$ and $w' = x'v'vq$ using the rule r at the indicated sites we get z , so z is in the splicing language.

Now the induction principle of Lemma 1 finishes step 3 of the proof, and so finishes the proof of Theorem 1.

Exercises

1. Following the proof of Theorem 1, construct an automaton which accepts the splicing language generated from the initial string $bbaaac$ using the rules $(b, a; ba, a)$, $(b, ac; ba, c)$ and $(bb, c; b, bc)$. [You will need to construct M_5 .] As a check on your construction you should see that the splicing language is infinite and consists of words of the form $b^m a^n c$. What are the restrictions on m and n ?
2. Find the complexity of each accepting path in the automaton of Exercise 1.
3. Suppose every rule in R has the form $(a, \lambda; a, \lambda)$, where a is in the alphabet. Show that in this case the splicing language is the language accepted by M_1 . In fact it may not be true that $M_2 = M_1$, since there may be “null loops” introduced in the second stage.
4. Find an example to show that M_1 does not necessarily define the splicing language if all rules have the form $(a, \lambda; b, \lambda)$, where a and b are in the alphabet.
5. Suppose the alphabet has cardinality N . Is there an integer n (in terms of N , but independent of I and R) so that $M_n = M_{n+1}$ for all splicing systems of the form described in Exercise 4? [This is a fairly hard problem.] Note that Exercise 1 can be adapted, by replacing the initial string with $bba^p c$, to show that no such bound exists in general, even with a fixed choice of R .

4 Sets of splicing rules

The regularity theorem leads to two natural questions: What happens if the initial language I is not finite? What happens when the set of splicing rules R is not finite?

The first question has a simple answer. We already remarked that the proof of the regularity theorem does not require that the initial language be finite, but just that it be regular. A generalization is that if the initial language I belongs to a *full AFL* \mathcal{A} and the rule set is finite then the splicing language $R^*(I)$ also belongs to \mathcal{A} . We do not pursue this further but refer the reader to [8]. Note that this implies, for example, that if I is context-free then so is $R^*(I)$.

The second question is more interesting. To formulate it properly we need to consider a classification of infinite rule sets. For this we introduce an alternative notation for splicing rules: We identify a rule $(u, u'; v', v)$ with the string $u\#u'\$v'\#v$, where $\#$ and $\$$ are two new symbols not in our original alphabet A . This correspondence between rules and strings in $A^*\#A^*\$A^*\#A^*$ is obviously a bijection, and we can use it to define, for example, what it means for a set of rules to be regular.

One reason we might want to look at infinite rule sets is in a type of recognition problem, in which we ask whether a given language is a splicing language. Thus, for a language L , we might look at $\mathcal{R}(L)$, the set of all rules r that preserve L . Clearly, if $L = R^*(I)$ is the splicing language generated by the rule set R and the initial language I then $R \subseteq \mathcal{R}(L)$. If we are working on the recognition problem for regular languages then the following is very helpful.

Theorem 2. *If L is a regular language then so is $\mathcal{R}(L)$.*

We shall prove this theorem as an introduction to the use of the *syntactic monoid* in splicing. The definitions and basic properties of the syntactic monoid are given in the appendix, Section 6. We shall use the notation $\text{Syn } L$ for the syntactic monoid of the language L .

Since we are proving the regularity of a set of rules we shall use the string representation in the proof.

Suppose $r = u\#u'\$v'\#v$ is in $\mathcal{R}(L)$. We first notice the following: If u is syntactically congruent to \tilde{u} then $\tilde{r} = \tilde{u}\#u'\$v'\#v$ is in $\mathcal{R}(L)$. To prove this suppose \tilde{z} and z' are two words in L which can be spliced using the rule \tilde{r} ; we must show that any result \tilde{w} of such splicing must be in L . Thus we have factorizations $\tilde{z} = x\tilde{u}u'y$ and $z' = x'v'vy'$, so $\tilde{w} = x\tilde{u}vy'$. Since \tilde{z} is in L and \tilde{u} is syntactically congruent to u we can replace \tilde{u} with u in \tilde{z} and the result, $z = xu'u'y$, is in L . Now z and z' are in L and r is in $\mathcal{R}(L)$, so the result $w = xuvy'$ of splicing z and z' using r is also in L . Now we can repeat the argument above, in the opposite direction: Since w is in L and u is syntactically congruent to \tilde{u} we can replace \tilde{u} with u in w and conclude that $x\tilde{u}vy' = \tilde{w}$ is in L , which is what we wanted.

The same observation, with essentially the same proof, applies to the other components u' , v' , and v of r . Thus we have shown that, whenever r is in $\mathcal{R}(L)$, we must have $C_r = U\#U'\$V'\#V \subseteq \mathcal{R}(L)$, where U is the syntactic class containing u , U' is the syntactic class containing u' , and so on. That is, $\mathcal{R}(L)$ is the union of all such concatenations C_r , for $r \in \mathcal{R}(L)$. By Theorem 7 and closure under concatenation, each C_r is a regular language; and by Theorem 6 there are only finitely many such concatenations, so $\mathcal{R}(L)$ is regular.

In light of Theorem 2 it is natural to conjecture that the splicing language generated by a regular set of rules and a regular (or finite) initial language should be regular. This conjecture fails spectacularly:

Theorem 3. *If L is any RE language then there are a finite set I , a regular set of rules R , and a regular language L_0 so that $L = L_0 \cap R^*(I)$.*

We do not prove this here (see [4]), but we give a simple example to illustrate what can happen:

The alphabet is $A = \{a, b, X, X', Y, Y', Z\}$ and the initial set is $I = \{XY, ZbY', X'aZ\}$. The rule set R consists of $(\lambda, Y; Y, \lambda)$, together with all rules of the form $(\lambda, Z; X, wY)$, $(X'w, Y; Z, bY')$, $(X, \lambda; X', wY')$, or $(Xw, Y'; \lambda, Y)$, where w varies over all words in $\{a, b\}^*$. Then $\{a, b\}^* \cap R^*(I)$ is the non-regular language $\{a^n b^n : n \geq 0\}$.

Exercises

1. Show that $\mathcal{R}(L) = \emptyset$ where $L = (a^2)^*$ and $A = \{a\}$.
2. A rule r is *useful* for a language L if there are words w_1 and w_2 in L which may be spliced using r . Let $\mathcal{R}_0(L)$ denote the set of rules in $\mathcal{R}(L)$ which are useful for L . Show that $\mathcal{R}_0(L)$ is a regular set of rules if L is regular.
3. Find $\mathcal{R}_0(L)$ where $L = b(a^2)^*$ and $A = \{a, b\}$.
4. Provide the details for the example at the end of the section.

5 Constants and reflexive splicing systems

Recall that a splicing system (R, I) is *reflexive* if the splicing language $R^*(I)$ is preserved by both $\dot{r} = (u, u'; u, u')$ and $\ddot{r} = (v', v; v', v)$ whenever $r = (u, u'; v', v)$ is in R . A language L is called a *reflexive* splicing language if it is generated by a finite reflexive splicing system. Our understanding of such languages is based on a notion introduced (in a very different context) by Schützenberger: We say a word z is a *constant* of the language L if, for any strings x , y , x' and y' , if xzy and $x'zy'$ are in L then xzy' is in L .

The following is the basic connection between constants and splicing:

Proposition 1. *A rule of the form $(u, v; u, v)$ preserves the language L if and only if the string uv is a constant of L .*

Using this and the regularity of $\mathcal{R}(L)$ from section 4 it is easy to deduce the (well-known) result that the set of constants of a regular language is regular. We shall need a simple fact about constants:

Lemma 2. *If c is a constant of L and c is a factor of a string c' then c' is a constant of L .*

The proofs of Proposition 1 and Lemma 2 are exercises for the reader.

Suppose (R, I) is a reflexive splicing system which generates the reflexive splicing language L . Write S for the set of all sites uv of rules in R . This is a finite set, and, by Proposition 1, every word of S is a constant of S .

The words of L can be divided into two classes. The words which contain a site of some rule of R are called “live”: they can participate in further splicing operations. The ones which do not contain such a site are called “dead”: they are inert as far as further splicing goes. The set of live words L_0 is just $L \cap A^*SA^*$, so it is regular; and so the set of dead words L_1 , being the complement of L_0 in L , is also regular. If a word in L is not in the initial set I then it is the result of a splicing operation, and the inputs to such a splicing operation contain sites so they are in L_0 . Hence each word of L_1 is either in I or is the result of splicing two elements of L_0 .

In fact this picture of L can be converted into a characterization of reflexive splicing languages. The key is to characterize languages with only finitely many “dead” words, in the following sense:

Theorem 4. *Suppose L is a regular language and S is a finite set of constants of L . If every element of L , with finitely many exceptions, has a factor in S then L is a reflexive splicing language.*

To prove this we must construct a reflexive splicing system (R, I) from L and S , and then we must show that $L = R^*(I)$. We write $L_0 = L \cap A^*SA^*$ and $L_1 = L - L_0$; by hypothesis L_1 is finite. We first define (R, I) as follows.

Let K be the cardinality of the syntactic monoid $\text{Syn } L$ and let N be the maximum length of a word in S . Our definitions are:

1. R is the set of all rules of the form $(u, \lambda; v, \lambda)$ or $(\lambda, u; \lambda, v)$, where u and v are syntactically congruent constants of L of length less than $2K + N$.
2. I is the set of words of L of length less than $4K + N$, together with the words of L_1 .

It is clear that R is reflexive, since the condition that u and v be syntactically congruent is satisfied if $u = v$.

Now we need to show that $L = R^*(I)$; we start with showing that any z in $R^*(I)$ is in L . This is immediate if $z \in I$, since $I \subseteq L$. We proceed by induction, so we may assume z is in $R^k(I)$ for some $k > 0$ but not in $R^{k-1}(I)$, and that every word in $R^{k-1}(I)$ is in L . But then z is the result of splicing two elements z_1 and z_2 of $R^{k-1}(I)$ using a rule r of R . Hence z_1 and z_2 are in L . Consider the case that r has the form $(u, \lambda; v, \lambda)$ (the alternate form is

handled similarly). We can factor $z_1 = x_1uy_1$ and $z_2 = x_2vy_2$, so $z = x_1uy_2$. Since u and v are syntactically congruent and $z_2 = x_2vy_2$ is in L we conclude that $z'_2 = x_2uy_2$ is in L . Since both $z_1 = x_1uy_1$ and $z'_2 = x_2uy_2$ are in L and u is a constant we conclude that $x_1uy_2 = z$ is in L .

The rest of the proof relies on the following:

Lemma 3. *Suppose L is a regular language and K is the cardinality of the syntactic monoid $\text{Syn } L$. If z is any string of length at least $2K$ then z has three distinct syntactically congruent prefixes and three distinct syntactically congruent suffixes, each of length at most $2K$.*

Proof. For $0 \leq k \leq 2K$ let p_k be the prefix of z of length k ; this is well defined since $2K \leq |z|$. Let η be the quotient map from A^* to $\text{Syn } L$. Since there are $2K + 1$ prefixes and only K elements of $\text{Syn } L$ we can apply the pigeonhole principle to find three indices $i < j < k$ so that $\eta(p_i) = \eta(p_j) = \eta(p_k)$. That is, the three prefixes p_i , p_j and p_k are syntactically congruent, and they have length at most $2K$.

A similar argument works for suffixes.

Now to conclude the proof of Theorem 4 we start with $z \in L$ and prove that z is in $R^*(I)$. We proceed by induction on the length of z . If $|z| < 4K + N$ or $z \in L_1$ then $z \in I \subseteq R^*(I)$, so we only need to consider the case that z is not in L_1 and has length at least $4K + N$. Since $z \notin L_1$ it must be in L_0 , so it has an element s of S as a factor: $z = xsy$. Since $|z| \geq 4K + N$ and $|s| \leq N$ we must have either $|x| \geq 2K$ or $|y| \geq 2K$. We give the rest of the proof assuming that $|y| \geq 2K$; the alternative is handled similarly.

We now apply Lemma 3 to y , so $y = tuvw$ where the three prefixes t , tu and tuv are distinct and syntactically congruent, with lengths at most $2K$. In particular, u and v cannot be the empty word. Now, using the facts that tuv is syntactically congruent to tu and that $z = xstuvw$ is in L we conclude that $z_1 = xstuw$ is in L , and it is shorter than z since $|v| > 0$.

We need another string to splice with z_1 . Let η be the projection of A^* onto $\text{Syn } L$. Since t and tu are syntactically congruent we have $\eta(t) = \eta(tu)$. Then, since η is a homomorphism, $\eta(tv) = \eta(t)\eta(v) = \eta(tu)\eta(v) = \eta(tuv)$, and so tv and tuv are syntactically congruent. Hence, by the same argument as above, $z_2 = xstvw$ is in L and is shorter than z .

By the induction hypothesis, z_1 and z_2 are in $R^*(I)$. Moreover, both stu and st are constants (by Lemma 2), they have length less than $N + 2K$ since $|s| \leq N$ and $|tu| < |tuv| \leq 2K$, and they are syntactically congruent since $\eta(stu) = \eta(s)\eta(tu) = \eta(s)\eta(t) = \eta(st)$. Therefore $r = (stu, \lambda; st, \lambda)$ is in R , and z is the result of splicing z_1 and z_2 using r , so z is in $R^*(I)$.

This completes the induction argument, and hence the proof of Theorem 4.

The splicing system constructed in the proof is, generally, enormous. Following the details of the proof shows that we only need to consider u and v which have a prefix or suffix in S . As a practical matter, much smaller sets

of rules and initial strings are usually sufficient. In any case it is worthwhile stating the form of the rules explicitly:

Corollary 1. *The language L of Theorem 4 may be written in the form $R^*(I)$ where each rule of R has one of the forms $(sp, \lambda; sq, \lambda)$ or $(\lambda, ps; \lambda, qs)$ with s in S .*

As an example of the power of Theorem 4 we mention the following:

Corollary 2. *If L_1 and L_2 are regular languages in A^* and c is a symbol which is not in A then L_1cL_2 is a reflexive splicing language.*

We have now done all the work to prove the following characterization theorem for reflexive splicing languages.

Theorem 5. *Suppose L is a regular language. Then L is a reflexive splicing language if and only if there are finite sets S and I of strings and a finite set of rules R so that:*

1. *Each element of S is a constant of L .*
2. *Each site of each rule of R is in S .*
3. *$L = L_0 \cup R(L_0) \cup I$ where $L_0 = L \cap A^*SA^*$.*

Proof. Suppose (R, I) is a reflexive splicing system and $L = R^*(I)$. We define S to be the set of sites of rules in R . Then, following the discussion before the statement of Theorem 4, S is a finite set of constants of L and every “dead” word in $L_1 = L - L_0$ is in I or in $R(L_0)$.

For the converse, assume we have such sets S , I and R . Clearly the words in S are constants of L_0 , so we can apply Theorem 4 to produce a reflexive splicing system (R_0, I_0) so that $L_0 = R_0^*(I_0)$. Now define the splicing system (R_1, I_1) by $R_1 = R_0 \cup R$ and $I_1 = I_0 \cup I$. According to Corollary 1, we may arrange that each site of a rule of R_0 has a factor in S and, by condition 2, the same is true for R . Hence the rules of R_1 can only operate on words of L_0 . So we have $R_0(L) = R_0(L_0) \subseteq L_0 \subseteq L$ and, by condition 3, $R(L) = R(L_0) \subseteq L$. Hence the rules of R_1 preserve L and $I_1 \subseteq L$, and it follows that $R_1^*(I_1) \subseteq L$. But $L_0 = R_0^*(I_0) \subseteq R_1^*(I_1)$, so, again using condition 3, $L \subseteq R_1^*(I_1) \cup R(R_1^*(I_1)) \cup I \subseteq R_1^*(I_1)$. So we have shown $L = R_1^*(I_1)$.

This characterization theorem points the way to an algorithm for determining whether a given regular language L is a reflexive splicing language. We fix an integer N and let S be the set of all constants of L of length at most N . Next, for each pair s and t of constants in S and for each choice of factorizations $s = uu'$, $t = v'v$ we decide whether the rule $(u, u'; v', v)$ is in $R(L)$. This is algorithmically feasible since the proof of Theorem 2 determines the regular set $R(L)$ explicitly in terms of the syntactic monoid (and a similar argument applies to determine the set of constants). We define R to be the set of all rules selected in this way. We also define I to be the set of all “short”

words in L ; for example, the set of words of length at most N . It is easy to construct an automaton to accept $R(L_0)$ for regular L_0 and finite R , so we can calculate the regular set $L_0 \cup R(L_0) \cup I$ and check whether it is equal to L . If the answer to this is “Yes” then we have determined that L is a reflexive splicing language. The problem, of course, is that if the answer is “No” then we cannot conclude that L is *not* a reflexive splicing language, since we might not have chosen N “large enough”. To make this procedure into a decidability algorithm we must determine a value of N which is guaranteed to be large enough. This determination is carried out in [1], which gives a sufficiently large value of N in terms of the size of the syntactic monoid of L .

The more general question of deciding whether a given regular language is a splicing language remains open.

Exercises

1. Prove Proposition 1.
2. Prove Lemma 2.
3. Exhibit a reflexive splicing system (R, I) so that $a(b^2)^* = R^*(I)$. (Note that a is a constant.)
4. Same question for $a(b^2)^*(c^2)^*$.
5. Let L be the language of words in $\{a, b\}^*$ which contain at most 2 b 's. Show that L is a splicing language but not a reflexive splicing language.

6 Appendix: The intrinsic automaton and the syntactic monoid of an arbitrary language.

As usual: A is the alphabet; A^* is the set of all strings over A and L is a language over A , i.e., a subset of A^* . Relative to the language L we have two constructions, of the automaton recognizing L and the syntactic monoid of L , defined as follows.

The (minimal) automaton M that recognizes L : The *states* of M are the equivalence classes defined, for each x in A^* , by $[x] = \{y \in A^* : R(y) = R(x)\}$ where $R(x)$ is the set of *right contexts* accepted by x relative to L . Specifically: $R(x) = \{v \in A^* : xv \in L\}$. Each symbol a in A *acts on* the state $[x]$ by $[x]a = [xa]$. M has a specified start state, $[1]$, and a specified set of terminal states, $\{[x] : x \in L\}$. M may be viewed as a directed graph having the states of L as its vertices, having directed edges $([x], a, [xa])$, each considered to be labeled by the alphabet symbol a . The automaton M is considered to *recognize* each string in A^* which is the label of a path from a start state to a terminal state. This M recognizes precisely those strings that are in L . A language L is regular if its automaton has only finitely many states.

The syntactic monoid S of L : The elements of S are the equivalence classes defined, for each x in A^* , by $\llbracket x \rrbracket = \{y \in A^* : LR(y) = LR(x)\}$, where $LR(x)$

is the set of *two-sided contexts* accepted by x relative to the language L . Specifically: $LR(x) = \{ \langle u, v \rangle \in A^* \times A^* : uxv \in L \}$. S has an associative binary operation that is *well defined* by setting $\llbracket x \rrbracket \llbracket y \rrbracket = \llbracket xy \rrbracket$. Since $\llbracket 1 \rrbracket$ serves as a two-sided identity for this operation, S has the structure of a *monoid*, i.e., a *semigroup* with an identity element. The partition of A^* into the classes $\llbracket x \rrbracket$ refines the partition of A^* into the classes $[x]$ since $LR(x) = LR(y)$ implies $R(x) = R(y)$. Consequently when S is finite L is regular.

The action of A on the states of M extends, inductively, to an action of A^* on the states of M . Consequently each string y in A^* determines a function from the state set of L into itself defined by $[u]x = [ux]$. Two strings x and y determine the same function precisely if, for every u in A^* , $[ux] = [uy]$. But this holds precisely if, for all v in A^* , uxv is in L if and only if uyv is in L . Thus x and y determine the same function precisely if $\llbracket x \rrbracket = \llbracket y \rrbracket$, and this construction defines a faithful representation of S as self-maps of the states of M . When L is regular there can be only a finite number of functions from the state set of L into itself. Consequently when L is regular S is finite.

Summary:

Theorem 6. *For every language L we have an intrinsically associated automaton that recognizes the language and we have an associated syntactic monoid. The following are equivalent:*

1. L is a regular language;
2. The intrinsic automaton for L has only finitely many states; and
3. The syntactic monoid of L is finite.

In the same spirit, construct a directed graph with states given by the elements of S and directed edges $(\llbracket x \rrbracket, a, \llbracket xa \rrbracket)$, each considered to be labeled by the alphabet symbol a . Then $\llbracket x \rrbracket$ is recognized by the automaton based on this graph with start state $\llbracket 1 \rrbracket$ and a single terminal state, $\llbracket x \rrbracket$. Hence:

Theorem 7. *If L is regular then each syntactic class $\llbracket x \rrbracket$ is regular.*

References

1. E. Goode and D. Pixton. Recognizing splicing languages: syntactic monoids and simultaneous pumping. *to appear*.
2. T. Head. Formal language theory and DNA: an analysis of the generative capacity of specific recombinant behaviors. *Bulletin of Mathematical Biology*, 49(6):737–759, 1987.
3. T. Head. Splicing languages generated with one sided context. In G. Paun, editor, *Computing With Bio-molecules—Theory and Experiments*, pages 269–282. Springer-Verlag, Singapore, 1998.
4. T. Head, G. Păun, and D. Pixton. Generative mechanisms suggested by DNA recombination. In G. Rozenberg and A. Salomaa, editors, *Handbook of Formal Languages*, volume 2. Springer Verlag, Berlin, Heidelberg, New York, October 1996.

5. K. Culik II and T. Harju. Splicing semigroups of dominoes and DNA. *Discrete Applied Mathematics*, 31:261–277, 1991.
6. G. Paun, G. Rozenberg, and A. Salomaa. *DNA Computing - New Computing Paradigms*. Springer-Verlag, Berlin Heidelberg, 1998.
7. D. Pixton. Regularity of splicing languages. *Discrete Appl. Math.*, 69(1–2):99–122, August 1996.
8. D. Pixton. Splicing in abstract families of languages. *Theoretical Computer Science*, 234:135–166, 2000.

Index

- constant 25–29
- DNA 3–16
- intrinsic automaton 29
- ligase 3, 9–16
- minimal automaton 29
- restriction enzyme 3, 9–12, 14–16
- splicing system 3–30
 - reflexive 6, 7, 15, 25, 26, 28, 29
- sticky end 9–16
- syntactic congruence 24, 26, 27
- syntactic monoid 24, 26, 27, 29–30
- Watson-Crick 7–10, 13

