The handle http://hdl.handle.net/1887/37052 holds various files of this Leiden University dissertation.

**Author:** Vliet, Rudy van  
**Title:** DNA expressions : a formal notation for DNA  
**Issue Date:** 2015-12-10
Chapter 4

DNA Expressions

The formal DNA molecules constitute the basis of our DNA language. They allow us to define the actual elements of the language: the DNA expressions. DNA expressions are strings that denote (formal) DNA molecules, in a similar way that arithmetic expressions denote numbers. They are the central concept of this thesis, and are introduced in this chapter.

After defining the DNA expressions, we examine how one can reconstruct their structure, i.e., how they are built up, from their appearance as flat strings. We subsequently explain how to decide whether or not a given string is a DNA expression. We show that the set of all DNA expressions is a context-free language, by means of a proper context-free grammar. DNA expressions may be represented by their derivation trees in this grammar, but these trees are very large. Therefore, we define another, more concise tree representation: the structure tree of a DNA expression. Finally, we introduce several notions of equivalence, for DNA expressions that denote (almost) the same formal DNA molecule.

4.1 Operators and DNA expressions

The basic building blocks of DNA expressions are $N$-words. DNA expressions result by applying operators to $N$-words. The operators we consider in this thesis are $\uparrow$, $\downarrow$ and $\updownarrow$, to be pronounced as uparrow, downarrow and updownarrow, respectively. DNA expressions also contain opening and closing brackets: $\langle$ and $\rangle$, which delimit the scope of the operators – each (occurrence of an) operator acts only on the part of the expression that is contained between its opening and closing brackets. Hence, the set of all DNA expressions, denoted by $D$, is a language over the alphabet $\Sigma_D$, where $\Sigma_D = N \cup \{\uparrow, \downarrow, \updownarrow, \langle, \rangle\} = \{A, C, G, T, \uparrow, \downarrow, \updownarrow, \langle, \rangle\}$.

We will use the symbol $E$ (possibly with annotations like subscripts) to denote a DNA expression. If a string can be either an $N$-word or a DNA expression, then we use $\varepsilon$ (possibly with annotations like subscripts) to denote it.

Informally, a DNA expression is a string of the form $\langle \varepsilon_1 \varepsilon_2 \ldots \varepsilon_n \rangle$, $\langle \downarrow \varepsilon_1 \varepsilon_2 \ldots \varepsilon_n \rangle$ or $\langle \updownarrow \varepsilon_1 \rangle$, where $n \geq 1$ and the $\varepsilon_i$’s are either $N$-words or DNA expressions themselves. The $\varepsilon_i$’s are called the arguments of the operator involved. We say that an operator is applied to its arguments. The arguments of the operators $\uparrow$ and $\downarrow$ must satisfy certain conditions, which will be explained shortly.

Clearly, not every string over $\Sigma_D$ is a DNA expression. In particular, every DNA expression contains brackets and at least one operator, which implies that $N$-words are not DNA expressions.
If \( E \) is a DNA expression, then the semantics of \( E \), denoted by \( S(E) \), is the formal DNA molecule represented by \( E \). For every DNA expression, there is exactly one such formal DNA molecule, so \( S \) is a mapping from our language of DNA expressions \( D \) into \( F \). When we precisely define the DNA expressions, we will also describe the corresponding semantics. We do not define DNA expressions and their semantics separately, because there are restrictions on the DNA expressions we can construct (the syntax) that are explained best in terms of the molecules denoted (the semantics).

In fact, it is possible to rephrase the semantic restrictions in syntactic terms. That would, however, make the definition far more tedious. In Section 4.3, we discuss how to check whether or not a given string over \( \Sigma_D \) is a DNA expression. We will see then, that in order to verify the semantic restrictions, we do not have to compute the complete semantics of (parts of) the DNA expression. In Section 4.5, we give a context-free grammar generating the language of all DNA expressions. This may be considered as a purely syntactic definition of the DNA expressions. The official definition, however, will make use of semantic terms, because that makes the definition easier to understand.

Properties of formal DNA molecules carry over in a natural way to DNA expressions by the following convention:

\[
\text{property } P \text{ holds for a DNA expression } E_1 \text{ (DNA expressions } E_1 \text{ and } E_2) \iff \\
\text{property } P \text{ holds for } S(E_1) \text{ (} S(E_1) \text{ and } S(E_2) \text{, respectively).}
\]

Thus, e.g., we may say that the upper strand of DNA expression \( E_1 \) strictly covers the lower strand to the right, or that DNA expression \( E_1 \) prefits DNA expression \( E_2 \) by upper strands.

Before we present the formal definition of a DNA expression, we want to provide some intuition for the action of the three operators and for the restrictions that are imposed onto their arguments.

The most elementary expressions in our DNA language are the applications of the operators to a (single) \( \mathcal{N} \)-word \( \alpha \): \( \langle \uparrow \alpha \rangle \), \( \langle \downarrow \alpha \rangle \) and \( \langle \updownarrow \alpha \rangle \). The expression \( \langle \uparrow \alpha \rangle \) denotes the upper \( \mathcal{A} \)-word \( \langle \alpha \rangle \) (which, in turn, denotes the strand \( 5'\alpha-3' \)), \( \langle \downarrow \alpha \rangle \) denotes the lower \( \mathcal{A} \)-word \( \langle \alpha \rangle \) (the strand \( 3'-\alpha-5' \)), and \( \langle \updownarrow \alpha \rangle \) denotes the double \( \mathcal{A} \)-word \( \langle \alpha \rangle \) with upper strand \( \alpha \) (the double-stranded DNA molecule \( 3'-\alpha-5' \) without nicks).

For example, if \( \alpha = \text{ACATG} \), then \( \langle \uparrow \alpha \rangle \) denotes \( \langle \text{ACATG} \rangle \), \( \langle \downarrow \alpha \rangle \) denotes \( \langle \text{ACATG} \rangle \), and \( \langle \updownarrow \alpha \rangle \) denotes \( \langle \text{ACATG} \rangle \).

In the basic DNA expressions, the three operators have one argument, an \( \mathcal{N} \)-word \( \alpha \). In general, however, the operators \( \uparrow \) and \( \downarrow \) may have more than one argument. Moreover, the arguments of an operator do not have to be \( \mathcal{N} \)-words; they may also be DNA expressions. Then, starting from the simple, basic DNA expressions, one can build more and more complex DNA expressions. There are, however, some restrictions on the arguments, which we will describe now for each of the operators.

The operator \( \uparrow \) can have an arbitrary number \( n \geq 1 \) of arguments. Each argument \( \varepsilon_i \) (\( i = 1, 2, \ldots, n \)) must be either an \( \mathcal{N} \)-word \( \alpha \), or a DNA expression \( E \). The resulting DNA expression is \( \langle \uparrow \varepsilon_1 \varepsilon_2 \ldots \varepsilon_n \rangle \).

From the molecular point of view, the effect of the operator \( \uparrow \) is threefold: (1) it produces upper strands corresponding to arguments that are \( \mathcal{N} \)-words \( \alpha \) (as in the basic DNA
4.1 Operators and DNA expressions

\[
S\left(\langle \uparrow C \atop G \right) AT \left( \overrightarrow{GC} \atop CG \right)\right) = CATGCG \quad S\left(\langle \uparrow A \atop T \right) T \left( \overrightarrow{A} \atop A \right)\right) = AT_{TA} \quad (a)
\]

\[
S\left(\langle \downarrow T \atop G \right) CATGCG \left( \overrightarrow{AT} \atop TGA \right)\right) = CATGCGAT \quad (b)
\]

\[
S\left(\langle \uparrow \downarrow \atop T \right) CATGCG \left( \overrightarrow{AT} \atop TGA \right)\right) = ACATGCGAT \quad (c)
\]

**Figure 4.1:** Examples of the effects of the three operators\(^2\) (a) The effect of the operator \(\uparrow\). (b) The effect of the operator \(\downarrow\). (c) The effect of the operator \(\uparrow\).

expression \(\langle \uparrow \alpha \rangle\), (2) it repairs all nicks occurring in the upper strands of its arguments by establishing the missing phosphodiester bonds and (3) it fixes such connections between the upper strands of consecutive arguments. In short, \(\uparrow\) connects all pairs of adjacent nucleotides in the upper strands of its arguments.

The third type of effect imposes a (semantic) restriction on the arguments of \(\uparrow\): consecutive arguments must prefit each other by upper strands. Otherwise, there would be a gap in the upper strand ‘between’ two arguments, and we would not be able to connect the upper strands. Since we have defined ‘prefitting each other by upper strands’ only for formal DNA molecules and for DNA expressions, we consider an \(\mathcal{N}\)-word \(\alpha\) here as the DNA expression \(\langle \uparrow \alpha \rangle\), which represents the upper \(\mathcal{A}\)-word \(\langle \alpha \rangle\).

The three types of effect of \(\uparrow\) are illustrated by the first example in Figure 4.1(a).

Nicks that are present in the lower strands of the arguments are not repaired by the operator \(\uparrow\). As a matter of fact, \(\uparrow\) introduces nicks between the lower strands of consecutive arguments if these consecutive arguments also happen to prefit each other by lower strands, i.e., if they have a blunt edge at each other’s side. The second example in Figure 4.1(a) shows such a situation.

The operator \(\downarrow\) is the dual of \(\uparrow\). It can have an arbitrary number \(n \geq 1\) of arguments, with each argument \(\varepsilon_i\) \((i = 1, \ldots, n)\) being either an \(\mathcal{N}\)-word or a DNA expression. The resulting DNA expression is \(\langle \downarrow \varepsilon_1 \varepsilon_2 \ldots \varepsilon_n \rangle\).

The effect of this operator is similar to that of \(\uparrow\); the only difference is that the roles of the upper strands and the lower strands of the arguments are changed. Consequently, also the requirement on consecutive arguments is changed: for \(i = 1, 2, \ldots, n - 1\), \(\varepsilon_i\) must prefit \(\varepsilon_{i+1}\) by lower strands. Here, when an argument \(\varepsilon_i\) is an \(\mathcal{N}\)-word \(\alpha\), it is interpreted as the DNA expression \(\langle \downarrow \alpha \rangle\), which denotes the lower \(\mathcal{A}\)-word \(\langle \overline{\alpha} \rangle\). The effect of \(\downarrow\) is illustrated by Figure 4.1(b).

Unlike the other two operators, \(\uparrow\) can have only one argument \(\varepsilon_1\). It is either an \(\mathcal{N}\)-word or an (arbitrary) DNA expression. The resulting DNA expression is \(\langle \uparrow \varepsilon_1 \rangle\).

If \(\varepsilon_1\) is a DNA expression \(E\), then, intuitively, in the DNA molecule denoted by \(E\), the operator \(\uparrow\) provides a complementary nucleotide for every nucleotide which is not yet complemented. So it fills up every gap in the DNA molecule. Further, the operator establishes phosphodiester bonds between the nucleotides added and their respective neighbours in the strand. Hence, it does not introduce new nicks. On the other hand, if the DNA molecule denoted by \(E\) has nicks already, then these nicks are not repaired by \(\uparrow\). The

\(^2\)The reader should not be diverted by the informal presentation of the examples. Formally, the arguments of our operators are \(\mathcal{N}\)-words and/or DNA expressions, and not DNA molecules. And formally, the semantics of a DNA expression is not a DNA molecule, but a formal DNA molecule.
effect of this operator is illustrated in Figure 4.1(c).

The basic DNA expression $\langle \uparrow \alpha \rangle$ was the result of applying $\uparrow$ to an $N$-word $\alpha$. This result can also be explained in terms of complements, as follows: if the argument of $\uparrow$ is an $N$-word $\alpha$, the operator conceives it as the DNA expression $\langle \uparrow \alpha \rangle$ and then performs the same action as for ‘ordinary’ DNA expressions.

The notation $\uparrow$ may be a bit misleading. It may suggest to be a combination of the operators $\uparrow$ and $\downarrow$. It would, e.g., repair nicks in both upper strands and lower strands then, like the function $\nu$ does with formal DNA molecules. In fact, an operator with such effect might be more realistic than the separate operators $\uparrow$ and $\downarrow$ that we have, as this effect comes closer to the effect of the enzyme ligase than the separate effects of $\uparrow$ and $\downarrow$.

Indeed, we could have chosen to use other, completely different operators to construct DNA expressions. Our choice for the three operators $\uparrow$, $\downarrow$ and $\uparrow$ was based on two considerations: (1) the basic two components of a double-stranded DNA molecule are the two strands, and (2) the operators we consider should obey some notion of locality.

In the case of the operators $\uparrow$ and $\downarrow$, ‘locality’ means that they act on one of the strands – in particular, $\uparrow$ seals (repairs) the nicks only in the upper strand, while $\downarrow$ seals the nicks only in the lower strand. Note that applying both $\uparrow$ and $\downarrow$ (in any order) to one argument will seal any existing nick. In the case of the operator $\uparrow$, ‘locality’ means that the string of nucleotides filling in a gap gets also properly connected (bonded) to its neighbours, while the pre-existing nicks are not sealed.

Therefore, in this thesis, we will build a theory with the operators $\uparrow$, $\downarrow$ and $\uparrow$ as we have introduced them.

There is a nice pictorial interpretation of the operators’ effects. We can consider a nucleotide as a puppet, the phosphate group at the 5′-site and the hydroxyl group at the 3′-site being its arms. When there is a horizontal connection between two adjacent nucleotides, we can view that as if both puppets raised one arm and joined hands. A phosphate group or a hydroxyl group that is not used for a phosphodiester bond corresponds to an arm hanging down. So in case of a nick, the two nucleotides involved keep the arm on the other one’s side down.

Now when the operator $\uparrow$ is applied, the puppets in the upper strand raise their arms and, if there is an adjacent puppet, they connect. The effect of $\downarrow$ can be viewed similarly. Finally, when $\uparrow$ complements a nucleotide, it inserts a puppet with both arms raised. Either of these arms seizes the arm of a neighbour and makes a connection. This case is depicted in Figure 4.2.
We are ready now to give a formal definition of DNA expressions and their semantics.

**Definition 4.1** A DNA expression is a string in any of the following forms:

- $\langle \uparrow \epsilon_1 \epsilon_2 \ldots \epsilon_n \rangle$, where $n \geq 1$, for $i = 1, 2, \ldots, n$, $\epsilon_i$ is either an $\mathcal{N}$-word or a DNA expression, and for $i = 1, 2, \ldots, n - 1$, $S^+(\epsilon_i) \sqsupset S^+(\epsilon_{i+1})$, where the function $S^+$ is defined by

$$S^+(\epsilon) = \begin{cases} \alpha & \text{if } \epsilon \text{ is an } \mathcal{N}\text{-word} \\ S(\epsilon) & \text{if } \epsilon \text{ is a DNA expression} \end{cases}. \quad (4.1)$$

Further,

$$S(\langle \uparrow \epsilon_1 \epsilon_2 \ldots \epsilon_n \rangle) = \nu^+(S^+(\epsilon_1))y_1\nu^+(S^+(\epsilon_2))y_2\ldots y_{n-1}\nu^+(S^+(\epsilon_n)) \quad (4.2)$$

with

$$y_i = \begin{cases} \triangle & \text{if } S^+(\epsilon_i) \sqsupset S^+(\epsilon_{i+1}), \text{i.e., if both } R(S^+(\epsilon_i)) \in A_+ \\
\land & \text{and } L(S^+(\epsilon_{i+1})) \in A_+ \\
\lambda & \text{otherwise, i.e., if either } R(S^+(\epsilon_i)) \in A_+ \\
& \text{or } L(S^+(\epsilon_{i+1})) \in A_- \text{ (or both)} \end{cases} \quad (4.3)$$

- $\langle \downarrow \epsilon_1 \epsilon_2 \ldots \epsilon_n \rangle$, where $n \geq 1$, for $i = 1, 2, \ldots, n$, $\epsilon_i$ is either an $\mathcal{N}$-word or a DNA expression, and for $i = 1, 2, \ldots, n - 1$, $S^-(\epsilon_i) \sqsubset S^-(\epsilon_{i+1})$, where the function $S^-$ is defined by

$$S^-(\epsilon) = \begin{cases} \alpha & \text{if } \epsilon \text{ is an } \mathcal{N}\text{-word} \\ S(\epsilon) & \text{if } \epsilon \text{ is a DNA expression} \end{cases}. \quad (4.4)$$

Further,

$$S(\langle \downarrow \epsilon_1 \epsilon_2 \ldots \epsilon_n \rangle) = \nu^-(S^-(\epsilon_1))y_1\nu^-(S^-(\epsilon_2))y_2\ldots y_{n-1}\nu^-(S^-(\epsilon_n))$$

with

$$y_i = \begin{cases} \triangledown & \text{if } S^-(\epsilon_i) \sqsubset S^-(\epsilon_{i+1}), \text{i.e., if both } R(S^-(\epsilon_i)) \in A_+ \\
& \text{and } L(S^-(\epsilon_{i+1})) \in A_+ \\
\lambda & \text{otherwise, i.e., if either } R(S^-(\epsilon_i)) \in A_- \\
& \text{or } L(S^-(\epsilon_{i+1})) \in A_- \text{ (or both)} \end{cases} \quad (4.3)$$

- $\langle \Leftarrow \epsilon_1 \rangle$, where $\epsilon_1$ is either an $\mathcal{N}$-word or a DNA expression. Further,

$$S(\langle \Leftarrow \epsilon_1 \rangle) = \kappa(S^+(\epsilon_1)),$$

for the function $S^+$ defined above.
One can verify that indeed, for each DNA expression $E$ satisfying this definition, $S(E)$ is a formal DNA molecule. Now, the formal language $D$ is the set of all DNA expressions.

**Example 4.2** The DNA expression

$$E = \langle \downarrow T \langle \uparrow \downarrow C \rangle \downarrow \langle \downarrow \downarrow G \rangle \langle \downarrow \uparrow C \rangle \rangle \langle \uparrow \downarrow A \rangle \langle \uparrow \uparrow T \rangle \rangle,$$

uses all three operators. It is easily verified that $E$ denotes the DNA molecule from Figure 4.1(b).

We call a DNA expression of the form $\langle \uparrow \varepsilon_1 \ldots \varepsilon_n \rangle$ an $\uparrow$-expression, one of the form $\langle \downarrow \varepsilon_1 \ldots \varepsilon_n \rangle$ a $\downarrow$-expression, and one of the form $\langle \uparrow \varepsilon_1 \rangle$ an $\uparrow$-expression. Hence, the DNA expression in Example 4.2 is a $\downarrow$-expression.

In this thesis, we will often introduce a general $\uparrow$-expression as \(\langle \uparrow \varepsilon_1 \ldots \varepsilon_n \rangle\) for some $n \geq 1$ and $N$-words and DNA expressions $\varepsilon_1, \ldots, \varepsilon_n$. Here, the phrase ‘$N$-words and DNA expressions $\varepsilon_1, \ldots, \varepsilon_n$’ does not necessarily mean that there is at least one argument $\varepsilon_i$ that is an $N$-word and at least one argument $\varepsilon_i$ that is a DNA expression. It is just an easy way to express that for $i = 1, \ldots, n$, $\varepsilon_i$ is either an $N$-word or a DNA expression. It is in principle possible that each $\varepsilon_i$ is an $N$-word or that each $\varepsilon_i$ is a DNA expression. Of course, we use this type of formulation also for $\downarrow$-expressions.

The formal DNA molecule $S^+(\varepsilon)$, occurring in the definition of a DNA expression of the form $\langle \uparrow \varepsilon_1 \varepsilon_2 \ldots \varepsilon_n \rangle$, can be considered as a kind of ‘upper semantics’ of the argument $\varepsilon$. Similarly, the formal DNA molecule $S^-(\varepsilon)$, occurring in the definition of a DNA expression of the form $\langle \downarrow \varepsilon_1 \varepsilon_2 \ldots \varepsilon_n \rangle$, can be considered as a kind of ‘lower semantics’ of the argument $\varepsilon$.

When we define functions $\text{Exp}^+$ and $\text{Exp}^-$ by

$$\text{Exp}^+(\varepsilon) = \begin{cases} \langle \uparrow \alpha \rangle & \text{if } \varepsilon \text{ is an } N\text{-word } \alpha \\ \varepsilon & \text{if } \varepsilon \text{ is a DNA expression} \end{cases} \quad (4.5)$$

and

$$\text{Exp}^-(\varepsilon) = \begin{cases} \langle \downarrow \alpha \rangle & \text{if } \varepsilon \text{ is an } N\text{-word } \alpha \\ \varepsilon & \text{if } \varepsilon \text{ is a DNA expression} \end{cases}, \quad (4.6)$$

it is easy to see that for every $N$-word or DNA expression $\varepsilon$, $S^+(\varepsilon) = S(\text{Exp}^+(\varepsilon))$ and $S^-(\varepsilon) = S(\text{Exp}^-(\varepsilon))$. Consequently, for $N$-words or DNA expressions $\varepsilon_1$ and $\varepsilon_2$, we have $S^+(\varepsilon_1) \triangleright S^+(\varepsilon_2)$, if and only if $\text{Exp}^+(\varepsilon_1) \triangleright \text{Exp}^+(\varepsilon_2)$, where the relation $\triangleright$ is used in the context of formal DNA molecules first, and in the context of DNA expressions next. Analogously, $S^-(\varepsilon_1) \triangleright S^-(\varepsilon_2)$, if and only if $\text{Exp}^-(\varepsilon_1) \triangleright \text{Exp}^-(\varepsilon_2)$. The DNA expressions $\text{Exp}^+(\varepsilon)$ and $\text{Exp}^-(\varepsilon)$ can be considered as a kind of ‘upper DNA expression’ and ‘lower DNA expression’ corresponding to the argument $\varepsilon$, respectively.

Note that, indeed, the operator $\uparrow$ does not introduce new nicks in its argument, simply because the function $\kappa$ does not do so.

We need to mention that the interpretation of the arguments of $\uparrow$-expressions and $\downarrow$-expressions may be ambiguous. For example, consider DNA expression $E$ from Example 4.2. Unless we have additional information, we cannot tell whether the $N$-word AT is itself an argument of the first occurrence of $\uparrow$, or that it is the concatenation of two arguments A and T. Consequently, we cannot tell, either, how many arguments this occurrence of $\uparrow$ has. This ambiguity occurs whenever an operator $\uparrow$ or $\downarrow$ has consecutive arguments that are $N$-words, or has an argument that is an $N$-word $\alpha$ with $|\alpha| \geq 2$. 


4.1 Operators and DNA expressions

Fortunately, even though it may be unclear what exactly the arguments of operators $\uparrow$ and $\downarrow$ occurring in a DNA expression are, there can be no doubt about the (formal) DNA molecule denoted by the DNA expression. This is implied by the following result:

**Theorem 4.3** Let $1 \leq i_0 < j_0 \leq n$, and let $E = \langle \uparrow \varepsilon_1 \ldots \varepsilon_{i_0-1} \alpha_{i_0} \ldots \alpha_{j_0} \varepsilon_{j_0+1} \ldots \varepsilon_n \rangle$ be a DNA expression, for some $N$-words or DNA expressions $\varepsilon_1, \ldots, \varepsilon_{i_0-1}, \varepsilon_{j_0+1}, \ldots, \varepsilon_n$ and some $N$-words $\alpha_{i_0}, \ldots, \alpha_{j_0}$. Let $\alpha = \alpha_{i_0} \ldots \alpha_{j_0}$. Then $S(E)$ is the same, regardless of the interpretation of $\alpha$ as one argument or as a sequence of separate arguments $\alpha_{i_0}, \ldots, \alpha_{j_0}$.

Hence, any partitioning of an argument $\alpha$ of $\uparrow$ into a sequence of arguments $\alpha_{i_0}, \ldots, \alpha_{j_0}$ yields the same semantics. Of course, an analogous result holds for $\downarrow$-expressions.

**Proof:** When we interpret $\alpha$ as one argument, Equation (4.2) becomes

$$S(E) = \nu^+(S^+(\varepsilon_1))y_1 \ldots y_{i_0-2}\nu^+(S^+(\varepsilon_{i_0-1}))(\alpha) \cdot \nu^+(S^+(\varepsilon_{j_0+1}))y_{j_0+1} \ldots y_{n-1}\nu^+(S^+(\varepsilon_n)),$$

where the $y_i$’s are defined by (4.3). Note that $S^+(\alpha) = (\alpha)$ and also $\nu^+(S^+(\alpha)) = (\alpha)$. Indeed, because $L((\alpha)) = R((\alpha)) \in A_+$, both $y_{i_0-1}$ and $y_{j_0}$ equal $\lambda$.

On the other hand, when we interpret $\alpha$ as a sequence of separate arguments $\alpha_{i_0}, \ldots, \alpha_{j_0}$, we obtain

$$S(E) = \nu^+(S^+(\varepsilon_1))y_1 \ldots y_{i_0-2}\nu^+(S^+(\varepsilon_{i_0-1}))(\alpha_{i_0}) \cdot \ldots \cdot (\alpha_{j_0}) \cdot \nu^+(S^+(\varepsilon_{j_0+1}))y_{j_0+1} \ldots y_{n-1}\nu^+(S^+(\varepsilon_n)),$$

where the $y_i$’s are the same as in (4.7). Because $(\alpha_{i_0}) \ldots (\alpha_{j_0}) = (\alpha_{i_0} \ldots \alpha_{j_0}) = (\alpha)$, Equation (4.8) reduces to (4.7).

Note that the interpretation of $N$-words $\alpha$ of length $|\alpha| \geq 2$ as argument(s) of an operator is unambiguous for the operator $\uparrow$, because this operator can have only one argument.

**Example 4.4** Let $E = \langle \uparrow \text{ACATG} \rangle$. Then there are many possible interpretations of the arguments of the operator $\uparrow$. We might, e.g., interpret $E$ as $\langle \uparrow \alpha_1 \ldots \alpha_5 \rangle$, with five arguments $\alpha_1 = A, \alpha_2 = C, \alpha_3 = A, \alpha_4 = T$ and $\alpha_5 = G$. But we might as well interpret $E$ as $\langle \uparrow \alpha_1 \alpha_2 \rangle$ with two arguments $\alpha_1 = AC$ and $\alpha_2 = ATG$, as $\langle \uparrow \alpha_1 \alpha_2 \rangle$ with two arguments $\alpha_1 = \text{ACAT}$ and $\alpha_2 = G$, or as $\langle \uparrow \alpha_1 \rangle$ with only one argument $\alpha_1 = \text{ACATG}$. Whatever interpretation we choose, $S(E) = \langle \text{ACATG} \rangle$.

By the above, we are free to interpret consecutive $N$-words in a DNA expression as one $N$-word. This motivates the definition of a maximal $N$-word occurrence in a string $X$ (e.g., in a DNA expression $E$) as an occurrence $(X_1, X_2)$ of an $N$-word $\alpha$ in $X$ such that (1) if $X_1 \neq \lambda$ then $R(X_1) \notin N$ and (2) if $X_2 \neq \lambda$ then $L(X_2) \notin N$. Hence, the $N$-word $\alpha$ ‘cannot be extended either to the left or to the right’.

**Example 4.5** In the DNA expression

$$\langle \downarrow \text{T} \langle \uparrow \downarrow \text{C} \rangle \text{AT} \langle \uparrow \downarrow \text{GCAT} \rangle \rangle$$

\[\text{If } i_0 = 1 \text{ or } j_0 = n, \text{ then, of course, } y_{i_0-1} \text{ or } y_{j_0}, \text{ respectively, does not even exist.}\]
the first occurrence of C and the first occurrence of AT are maximal $N$-word occurrences. This is, however, not the case with the second occurrences of these $N$-words, as they can be extended to GCAT.

Although we may interpret consecutive $N$-words in a DNA expression as one $N$-word, we do not always do so in this thesis. In particular, we still allow occurrences of the operators $\uparrow$ and $\downarrow$ in a DNA expression to have consecutive arguments that are $N$-words.

**Additional terminology**

We say that an operator *governs* its argument(s) and everything inside its argument(s). In every DNA expression we can identify an outermost operator. This is the operator which has been performed last. It governs the entire DNA expression.

Because of the 1–1 correspondence between a DNA expression and its outermost operator, we will sometimes use one term while meaning the other. In particular, we may speak of the *arguments of a DNA expression*, while we actually mean the arguments of the outermost operator of a DNA expression. For example, the (three) arguments of the DNA expression from Example 4.2 are the $N$-word T, the $\uparrow$-expression $\langle \uparrow \downarrow \rangle \langle \uparrow \rangle C \langle \downarrow \rangle AT \langle \downarrow \downarrow \rangle \langle \uparrow \rangle G \langle \downarrow \rangle \langle \uparrow \rangle C \langle \downarrow \downarrow \rangle$ and the $\uparrow$-expression $\langle \uparrow \rangle A \langle \downarrow \rangle T \langle \downarrow \rangle$.

We call (an occurrence of) an operator in a DNA expression $E$ which is not the outermost operator, an *inner occurrence* of this operator in $E$.

An operator may occur more than once in a DNA expression. To denote a specific occurrence of an operator, we may provide the operator with a subscript. For example, we may have $\uparrow_0$ or $\downarrow_1$.

A DNA *subexpression* $E^*$ of a DNA expression $E$ is a substring of $E$ which is itself a DNA expression. If $E^* \neq E$, then we call $E^*$ a *proper DNA subexpression* of $E$. Clearly, the outermost operator of a proper DNA subexpression of $E$ is an inner occurrence of this operator in $E$.

We will use the term $\uparrow$-*subexpression* of $E$ to refer to a DNA subexpression of $E$ which is an $\uparrow$-expression. Analogously, we may have a $\downarrow$-subexpression and an $\uparrow$-subexpression of $E$.

For every $N$-word $\alpha$ occurring in a DNA expression $E$ and for every proper DNA subexpression $E^*$ of $E$ we define its *parent operator* to be the operator which has the $N$-word or DNA subexpression as an immediate argument. For example, in the DNA expression from Example 4.2, the parent operator of the $N$-word AT is the first occurrence of the operator $\uparrow$ in the DNA expression; for the second occurrence of the $N$-word C it is clearly the operator $\downarrow$ standing in front of it; and the parent operator of the DNA subexpression $\langle \downarrow \rangle G \langle \downarrow \rangle$ is the second occurrence of the operator $\downarrow$.

An occurrence of an operator is an *ancestor operator* of an $N$-word or a DNA subexpression $\varepsilon$ occurring in $E$, if $\varepsilon$ is contained in an argument of the operator. For example, the ancestor operators of the second occurrence of the $N$-word C in the DNA expression from Example 4.2 are: the first occurrence of $\downarrow$ (the outermost operator), the first occurrence of $\uparrow$, the second occurrence of $\downarrow$ and the third occurrence of $\downarrow$ (the parent operator of C).

If an argument of a certain (occurrence of an) operator is an $N$-word, then we may call it an $N$-*word-argument* of the operator. If, on the other hand, the argument is a DNA expression, then we may call it an *expression-argument* of the operator. In particular, if it is an $\uparrow$-expression, then we may call it an $\uparrow$-argument. In an analogous way, we define a $\downarrow$-argument and an $\uparrow$-argument of an operator. At some point in this thesis, it will be useful
to have a single term for arguments that are not $\uparrow$-expressions, i.e., for $N$-word-arguments, $\uparrow$-arguments and $\downarrow$-arguments. We call such arguments non-$\downarrow$-arguments.

Let us assume that the $N$-word-arguments of a certain $\uparrow$-expression or $\downarrow$-expression $E$ are maximal $N$-word occurrences. We say that $E$ is alternating, if its arguments are maximal $N$-word occurrences and DNA expressions, alternately. Because by definition, a maximal $N$-word occurrence cannot be preceded or succeeded by another $N$-word-argument, this is equivalent to saying that $E$ does not have consecutive expression-arguments. An occurrence of an operator $\uparrow$ or $\downarrow$ is alternating, if the corresponding DNA subexpression is alternating.

Example 4.6 Let

$$E_1 = \langle \uparrow \alpha_1 \rangle,$$
$$E_2 = \langle \uparrow \downarrow \alpha_1 \rangle,$$
$$E_3 = \langle \downarrow \uparrow \alpha_1 \downarrow \alpha_2 \rangle \alpha_3 \alpha_4 \langle \downarrow \downarrow \alpha_5 \rangle,$$
$$E_4 = \langle \downarrow \alpha_1 \downarrow \downarrow \alpha_2 \rangle \langle \uparrow \downarrow \alpha_3 \rangle \alpha_4 \rangle.$$

Both $E_1$ and $E_2$ have only one argument, and are by definition alternating. The $N$-word-arguments $\alpha_3$ and $\alpha_4$ of $E_4$ together form a maximal $N$-word occurrence. This makes $E_3$ alternating. Finally, $E_4$ is alternating, although its second argument $\langle \downarrow \downarrow \alpha_2 \rangle \langle \uparrow \downarrow \alpha_3 \rangle \alpha_4 \rangle$ is not alternating. The $\downarrow$-expression in Example 4.2 is not alternating, because both its second argument $\langle \uparrow \downarrow C \rangle \text{AT} \langle \downarrow \downarrow G \rangle \langle \downarrow C \rangle$ and its third argument $\langle \uparrow \downarrow A \rangle \langle \downarrow T \rangle$ are DNA expressions.

Let $E$ be a DNA expression, and let $\alpha_1, \ldots, \alpha_k$ for some $k \geq 1$ be the maximal $N$-word occurrences in $E$, in the order of their occurrence from left to right. Then we will sometimes write $E$ as a function of these maximal $N$-word occurrences, hence $E = E(\alpha_1, \ldots, \alpha_k)$. Clearly, $\alpha_1, \ldots, \alpha_k$ also show up in the corresponding formal DNA molecule $S(E)$, and they occur in $S(E)$ in the same order as in $E$.

Note, however, that different maximal $N$-word occurrences $\alpha_i$ in $E$ may end up in the same component of $S(E)$. Moreover, if the parent operator of a maximal $N$-word occurrence $\alpha_i$ is $\downarrow$ (which implies that a lower $A$-word $(\alpha_i)$ is introduced into the semantics), then this lower $A$-word may be complemented by an occurrence of $\uparrow$. This would result in a double $A$-word $(\alpha_i^c)$. Hence, the component of $S(E)$ in which a maximal $N$-word occurrence $\alpha_i$ of $E$ appears, is not necessarily an element of $\left\{ (\alpha_i), (\alpha_i^c), (\alpha_i^c) \right\}$. For example, if $E = E(\alpha_1, \alpha_2) = \langle \downarrow \alpha_1 \downarrow \alpha_2 \rangle$, then $S(E) = \langle (\alpha_1^c)\alpha_2 \rangle$.

4.2 Brackets, arguments and DNA subexpressions

The brackets in a DNA expression determine a structure with different levels. An opening bracket $\langle$ corresponds to an increase of the level by 1, a closing bracket $\rangle$ to a decrease of the level by 1. The resulting levels are called the nesting levels of the brackets.

Initially, before the first letter of a DNA expression, the nesting level is 0. Since every opening bracket precedes the corresponding closing bracket, the nesting level is non-negative at any position in a DNA expression. Further, because the number of opening brackets equals the number of closing brackets, the nesting level is back at 0 at the end of a DNA expression. In Figure 4.3, we show the nesting level as a function of the position in...
the DNA expression from Example 4.2. The maximal nesting level of a DNA expression is of particular interest. For example, the maximal nesting level of the DNA expression from Figure 4.3 is 4.

A DNA expression consists of an opening bracket, an operator, one or more arguments and a closing bracket. Hence, the nesting level structure of a DNA expression is determined by the nesting level structure of its arguments. In particular, the maximal nesting level of a DNA expression is determined by the maximal nesting levels of those arguments that are DNA expressions themselves:

**Lemma 4.7** Let $E$ be a DNA expression and let $E_1, \ldots, E_r$ for some $r \geq 0$ be the expression-arguments of $E$.

1. If $r = 0$ (i.e., if $E$ only has $N$-word-arguments), then the maximal nesting level of $E$ is 1.

2. If $r \geq 1$, then the maximal nesting level of $E$ is equal to

$$\max_{j=1}^r (\text{maximal nesting level of } E_j) + 1.$$ 

Of course, in the expression in Claim 2, the expression-arguments $E_j$ are viewed as independent DNA expressions, which start at level 0.

We can use the notion of the nesting level for identifying substrings of a DNA expression. We do this in the following two results.

**Lemma 4.8** Suppose that the opening bracket of a DNA subexpression $E^*$ of a DNA expression $E$ raises the nesting level of $E$ from $l - 1$ to $l$ for a certain positive integer $l$. Then the closing bracket of $E^*$ is the first closing bracket after this opening bracket to lower the nesting level from $l$ to $l - 1$. In particular, between the opening bracket and the closing bracket of $E^*$, the nesting level is at least $l$. 

**Figure 4.3:** Nesting level as a function of the position in the DNA expression from Example 4.2. Horizontal dotted lines connect changes of the nesting level due to pairs of corresponding brackets.
4.2 Brackets, arguments and DNA subexpressions

\[ E: \langle \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \rangle \]
\[ E_1^*: \langle \ldots \ldots \rangle \]
\[ E_2^*: \langle \ldots \ldots \ldots \ldots \rangle \]

Figure 4.4: Schematic representation of two (hypothetically) overlapping DNA subexpressions \( E_1^* \) and \( E_2^* \) of a DNA expression \( E \).

**Proof:** Straightforward by induction on the number of operators occurring in \( E^* \).

To illustrate this lemma, we have drawn dotted lines between corresponding increases and decreases of the nesting level in Figure 4.3. We can thus use the nesting levels of the brackets in a DNA expression \( E \), to reconstruct the DNA subexpressions that occurred in the recursive definition of \( E \). We proceed with arbitrary arguments of operators occurring in \( E \).

**Theorem 4.9** Let \( E \) be a DNA expression, and assume that each \( \mathcal{N} \)-word-argument of an operator occurring in \( E \) is a maximal \( \mathcal{N} \)-word occurrence. Let \( |_0 \) be an operator at nesting level \( l \) in \( E \). Then (an occurrence of) a substring between \( |_0 \) and the closing bracket of \( |_0 \) is an argument of \( |_0 \), if and only if

- either it is a maximal \( \mathcal{N} \)-word occurrence in \( E \) at nesting level \( l \),
- or it starts with an opening bracket raising the nesting level from \( l \) to \( l + 1 \) and ends with the corresponding closing bracket.

This result is important, because it enables us to determine the structure of a DNA expression, i.e., how the DNA expression has been built up, even though it is just a sequence of symbols.

Note that by Theorem 4.3, the assumption that each \( \mathcal{N} \)-word-argument of an operator is in fact a maximal \( \mathcal{N} \)-word occurrence, is not restrictive.

Clearly, as every DNA (sub-)expression is of the form \( \langle |_0 \varepsilon_1 \ldots \varepsilon_n \rangle \) for an operator \( |_0 \) and arguments \( \varepsilon_1, \ldots, \varepsilon_n \), the arguments are indeed substrings between \( |_0 \) and the closing bracket of \( |_0 \). Hence, this theorem covers all arguments of \( |_0 \).

We do not give a proof for Theorem 4.9. First, because the result is intuitively clear anyway, and second, because the inductive arguments that are used in the proof are a bit tedious, although not extremely complicated. We only mention that both in the proof from left to right and in the proof from right to left, there is a crucial role for Lemma 4.8.

Lemma 4.8 may also be used in a formal proof of the following result. Again, however, because this result is standard in the world of bracketed expressions, we omit the proof.

**Theorem 4.10** Two (occurrences of) DNA subexpressions in a DNA expression \( E \) cannot overlap. So either one is contained in the other, or they do not have a common (occurrence of a) substring at all.

Hence, a situation as depicted in Figure 4.4 is not possible.

**Corollary 4.11** If \( E^* \) is a proper DNA subexpression of a DNA expression \( E \), then \( E^* \) is contained in an argument of \( E \).
Proof: Because $E^*$ is a proper DNA subexpression of $E$, it is a substring of $\varepsilon_1 \ldots \varepsilon_n$, the concatenation of the arguments of $E$. Let $\varepsilon_i$ be the first argument that has a non-empty intersection with $E^*$. Then $\varepsilon_i$ contains the opening bracket of $E^*$, which implies that $\varepsilon_i$ is a DNA expression (and not an $N$-word).

If the opening bracket of $E^*$ is the opening bracket of $\varepsilon_i$, then also the closing brackets must match, so $E^*$ is equal to $\varepsilon_i$. In particular, $E^*$ is contained in $\varepsilon_i$. If the opening bracket of $E^*$ is not the opening bracket of $\varepsilon_i$, then $\varepsilon_i$ is clearly not contained in $E^*$. By Theorem 4.10, $E^*$ must be (properly) contained in $\varepsilon_i$.

We conclude this section with a simple, but useful result. It says that arguments of DNA expressions cannot just consist of brackets and operators:

**Lemma 4.12** Let $E \in \mathcal{D}$ be a DNA expression. Every argument of every operator in $E$ contains at least one $N$-word $\alpha$.

**Proof:** Straightforward by induction on the number of operators occurring in an argument.

### 4.3 Recognition of DNA expressions

As mentioned before, not every string over $\Sigma_D$, i.e., consisting of $N$-words $\alpha$, operators and brackets, is a DNA expression. Given an arbitrary string $E$ over this alphabet, we may want to verify whether or not it is a DNA expression. A natural way to do this, is simply to check all requirements from the (recursive) definition of a DNA expression, as given in Definition 4.11. One requirement is that the arguments of (occurrences of) operators $\uparrow$ and $\downarrow$ must fit together by upper strands or lower strands, respectively. In this section, we discuss how to check this without explicitly computing the semantics of the arguments.

Before we can examine the arguments of operators, we must look at the structure of the string $E$ we are given. In particular, we must verify (1) that there are as many opening brackets as closing brackets in the string, (2) that each opening brackets comes before the corresponding closing bracket, (3) that the first symbol of the string is an opening bracket and the last symbol is the corresponding closing bracket, (4) that each opening bracket is immediately succeeded by an operator, and (5) that there are no other occurrences of operators in the string.

Next, by using Theorem 4.9, we can determine the arguments $\varepsilon_i$ of the outermost operator $|_0$ of the string. If $|_0$ is $\uparrow$, then there has to be exactly one argument; if it is either $\uparrow$ or $\downarrow$, then there has to be at least one argument. In particular, we cannot have $E = \langle \uparrow \rangle$, $E = \langle \downarrow \rangle$ or $E = \langle \uparrow \rangle$. For those arguments that are no (maximal) $N$-word occurrences, we can check recursively whether they are DNA expressions.

If, up to here, all requirements are met and $|_0$ has only one argument, then the string is a DNA expression. If the number of arguments $n$ is greater than 1 (which implies that $|_0$ is $\uparrow$ or $\downarrow$), then we have to do some more work. We must verify the semantic restriction that the arguments $\varepsilon_1, \ldots, \varepsilon_n$ fit together by upper strands or lower strands (depending on the operator), see Definition 4.11. In fact, we may have had to perform such a check already for occurrences of $\uparrow$ and $\downarrow$ inside the arguments, when we checked that these arguments are really DNA expressions.
4.3 Recognition of DNA expressions

The requirement for $\uparrow$-expressions can be expressed formally in terms of $R(S^+(\varepsilon_i))$ and $L(S^+(\varepsilon_{i+1}))$ for $i = 1, \ldots, n - 1$. However, if we only want to check whether or not two arguments of an operator fit together by upper strands, then we are not interested in the complete semantics of these arguments. In fact, it could be very inefficient to compute them. For consecutive arguments $\varepsilon_i$ and $\varepsilon_{i+1}$, both $R(S^+(\varepsilon_i))$ and $L(S^+(\varepsilon_{i+1}))$ must be in $\mathcal{A}_+ \cup \mathcal{A}_\pm$.

Therefore, it would be desirable if we could compute $L(S^+(\varepsilon_i))$ and $R(S^+(\varepsilon_i))$ for an $\mathcal{N}$-word or DNA expression $\varepsilon_i$ without having to compute $S^+(\varepsilon_i)$ explicitly. Actually, we only need to know which of the subsets $\mathcal{A}_+$ and $\mathcal{A}_\pm$ the $\mathcal{A}$-letters $L(S^+(\varepsilon_i))$ and $R(S^+(\varepsilon_i))$ belong to. For consecutive arguments $\varepsilon_i$ and $\varepsilon_{i+1}$, both $R(S^+(\varepsilon_i))$ and $L(S^+(\varepsilon_{i+1}))$ must be in $\mathcal{A}_+ \cup \mathcal{A}_\pm$.

Of course, to check if the arguments $\varepsilon_1, \ldots, \varepsilon_n$ of an operator $\downarrow$ fit together by lower strands, we need to answer a similar question for $L(S^-(\varepsilon_i))$ and $R(S^-(\varepsilon_i))$. Note that if $\varepsilon_i$ is a DNA expression $E_i$, then $S^+(\varepsilon_i) = S^-(\varepsilon_i) = S(E_i)$. In that case, $L(S^+(\varepsilon_i)) = L(S^-(\varepsilon_i))$ and $R(S^+(\varepsilon_i)) = R(S^-(\varepsilon_i))$.

We can use the following result to recursively determine the subsets of $\mathcal{A}$ that $L(S^+(\varepsilon_i))$, $R(S^+(\varepsilon_i))$, $L(S^-(\varepsilon_i))$ and $R(S^-(\varepsilon_i))$ belong to:

**Lemma 4.13** Let $\varepsilon_i$ be an $\mathcal{N}$-word or a DNA expression.

1. If $\varepsilon_i$ is an $\mathcal{N}$-word $\alpha$, then
   
   \[
   L(S^+(\varepsilon_i)), R(S^+(\varepsilon_i)) \in \mathcal{A}_+,
   \]
   
   \[
   L(S^-(\varepsilon_i)), R(S^-(\varepsilon_i)) \in \mathcal{A}_-.
   \]

2. If $\varepsilon_i$ is an $\uparrow$-expression, then
   
   \[
   L(S^+(\varepsilon_i)) = L(S^-(\varepsilon_i)) = L(S(\varepsilon_i)) \in \mathcal{A}_+, \]
   
   \[
   R(S^+(\varepsilon_i)) = R(S^-(\varepsilon_i)) = R(S(\varepsilon_i)) \in \mathcal{A}_-.
   \]

3. If $\varepsilon_i$ is an $\uparrow$-expression $\langle \uparrow \varepsilon_{i,1} \ldots \varepsilon_{i,m} \rangle$ for some $m \geq 1$ and $\mathcal{N}$-words and DNA expressions $\varepsilon_{i,1}, \ldots, \varepsilon_{i,m}$, then
   
   \[
   L(S^+(\varepsilon_i)) = L(S^-(\varepsilon_i)) = L(S(\varepsilon_i)) = L(S^+(\varepsilon_{i,1})),
   \]
   
   \[
   R(S^+(\varepsilon_i)) = R(S^-(\varepsilon_i)) = R(S(\varepsilon_i)) = R(S^+(\varepsilon_{i,m})).
   \]

4. If $\varepsilon_i$ is a $\downarrow$-expression $\langle \downarrow \varepsilon_{i,1} \ldots \varepsilon_{i,m} \rangle$ for some $m \geq 1$ and $\mathcal{N}$-words and DNA expressions $\varepsilon_{i,1}, \ldots, \varepsilon_{i,m}$, then
   
   \[
   L(S^+(\varepsilon_i)) = L(S^-(\varepsilon_i)) = L(S(\varepsilon_i)) = L(S^+(\varepsilon_{i,1})),
   \]
   
   \[
   R(S^+(\varepsilon_i)) = R(S^-(\varepsilon_i)) = R(S(\varepsilon_i)) = R(S^-(\varepsilon_{i,m})).
   \]

**Proof:**

1. This claim follows immediately from the observation that for an $\mathcal{N}$-word $\alpha$, $S^+(\alpha) = \left( \frac{\alpha}{\alpha} \right)$ and $S^-(\alpha) = \left( \frac{-\alpha}{-\alpha} \right)$. 


2. Assume that \( \varepsilon_i = \langle \uparrow \varepsilon_{i,1} \rangle \) for an \( \mathcal{N} \)-word or a DNA expression \( \varepsilon_{i,1} \). By the definition of the semantics of an \( \uparrow \)-expression, \( S(\varepsilon_i) = \kappa(S^+(\varepsilon_{i,1})) \). Hence, \( L(S(\varepsilon_i)) = L(\kappa(S^+(\varepsilon_{i,1}))) \) and \( R(S(\varepsilon_i)) = R(\kappa(S^+(\varepsilon_{i,1}))) \). By Lemma 3.11 these are in \( \mathcal{A}_\pm \).

3. Assume that \( \varepsilon_i = \langle \downarrow \varepsilon_{i,1} \ldots \varepsilon_{i,m} \rangle \) for some \( m \geq 1 \) and \( \mathcal{N} \)-words and DNA expressions \( \varepsilon_{i,1}, \ldots, \varepsilon_{i,m} \). According to the definition of an \( \uparrow \)-expression and its semantics,

\[
S(\varepsilon_i) = \nu^+(S^+(\varepsilon_{i,1}))(y_1 \ldots y_{m-1} \nu^+(S^+(\varepsilon_{i,m}))
\]

for \( y_i \)’s as in (4.3). Consequently,

\[
L(S(\varepsilon_i)) = L(\nu^+(S^+(\varepsilon_{i,1}))) = L(S^+(\varepsilon_{i,1})).
\]

The second equality in this derivation follows from Lemma 3.11.

In a similar way, we find \( R(S(\varepsilon_i)) = R(S^+(\varepsilon_{i,m})) \).

4. The proof of this claim is analogous to that of the previous claim.

□

Once we know \( L(S^+(\varepsilon_i)) \) and \( R(S^+(\varepsilon_i)) \) (if \( |0 = \uparrow) \) or \( L(S^-(\varepsilon_i)) \) and \( R(S^-(\varepsilon_i)) \) (if \( |0 = \downarrow) \) for \( i = 1, \ldots, n \), it is easy to check whether or not the arguments fit together by upper strands or lower strands, respectively. If so, then the string \( E \) is a DNA expression; otherwise, it is not.

In Figure 4.5 we give a recursive function \texttt{CheckExpression}, which uses Lemma 4.13 to decide whether or not a string \( E \) over \( \Sigma_D \) is a DNA expression. Whenever the function is called (recursively) for a DNA expression \( E \), it returns the subsets of \( \mathcal{A} \) that \( L(S(E)) \) and \( R(S(E)) \) belong to. These subsets can be used higher up in the recursion to verify that consecutive arguments of operators \( \uparrow \) and \( \downarrow \) fit together. \texttt{CheckExpression} assumes that the brackets and the operators in \( E \) are positioned correctly. This implies in particular that it is possible to actually identify the arguments of \( E \), using Theorem 4.9.

It is not difficult to verify the assumption about the positioning of the brackets and the corresponding operators in \( E \). One can do this by simply traversing the string from left to right, counting opening brackets (followed by operators) and closing brackets. Then the entire algorithm for the recognition of a DNA expression takes time that is linear in the length of the string.

**Concatenation of DNA expressions**

By Lemma 3.10, the concatenation of two formal DNA molecules is not necessarily a formal DNA molecule itself. For DNA expressions, the situation is even worse. The mere concatenation of two DNA expressions \( E_1 \) and \( E_2 \) is never a DNA expression, not even if \( E_1 \) and \( E_2 \) fit together.

This conclusion follows immediately from an examination of the brackets. The first and the last symbol of a DNA expression have to be corresponding opening and closing brackets. However, although the first and the last symbol of the string \( E_1 E_2 \) are an opening and a closing bracket, respectively, they are not corresponding opening and closing brackets.
4.3 Recognition of DNA expressions

1. \text{bool CheckExpression} (E, L_0, R_0)
   \hspace{1em} // checks if the string E, whose brackets and operators
   \hspace{1em} // are positioned correctly, is a DNA expression;
   \hspace{1em} // if so, then also returns the subsets \( L_0 \) and \( R_0 \) of \( A \)
   \hspace{1em} // which \( L(S(E)) \) and \( R(S(E)) \) belong to
2. \{ \n3. \quad |_0 = \text{outermost operator of } E; \n4. \quad OK = \text{true}; \n5. \quad n = 0; // number of arguments \n6. \quad \text{while} (\text{OK and there are arguments of } E \text{ left}) \n7. \quad \hspace{1em} \text{do} \quad n++; \n8. \quad \hspace{1em} \varepsilon = \text{next argument of } E; \n9. \quad \hspace{1em} \text{if} (|_0 == \rhd) \n10. \quad \hspace{2em} \text{then if} (n == 1) \n11. \quad \hspace{3em} \text{then if} (\varepsilon \text{ is not an } \mathcal{N}\text{-word}) \n12. \quad \hspace{4em} \text{then} // it should be a DNA expression \n13. \quad \hspace{5em} OK = \text{CheckExpression} (\varepsilon, L_1, R_1); \n14. \quad \hspace{4em} \text{fi} \n15. \quad \hspace{3em} \text{if} (\text{OK}) // in particular, if \( \varepsilon \text{ is an } \mathcal{N}\text{-word} \n16. \quad \hspace{4em} \text{then} \quad \text{then} L_0 = \mathcal{A}_{\pm}; \n17. \quad \hspace{4em} \quad \text{then} \quad \text{then} R_0 = \mathcal{A}_{\pm}; \n18. \quad \hspace{4em} \text{fi} \n19. \quad \hspace{2em} \text{else} // n ≥ 2 \n20. \quad \hspace{3em} OK = \text{false}; // more than one argument for \( \rhd \) \n21. \quad \hspace{2em} \text{fi} \n22. \quad \text{else} // |_0 == \ldownarrow \text{ or } |_0 == \uparrow; \n23. \quad // \text{without loss of generality, assume } |_0 == \rhd \n24. \quad \text{if} (\varepsilon \text{ is an } \mathcal{N}\text{-word}) \n25. \quad \hspace{2em} \text{then} L_1 = \mathcal{A}_{\pm}; \n26. \quad \hspace{2em} \text{then} \quad \text{then} R_1 = \mathcal{A}_{\pm}; \n27. \quad \hspace{2em} \text{else} // \varepsilon \text{ should be a DNA expression} \n28. \quad \hspace{3em} \text{OK = CheckExpression} (\varepsilon, L_1, R_1); \n29. \quad \hspace{2em} \text{fi} \n30. \quad \text{if} (\text{OK}) \n31. \quad \hspace{2em} \text{then if} (n == 1) // first argument \n32. \quad \hspace{3em} \text{then} \quad \text{then} L_0 = L_1; \n33. \quad \hspace{3em} \quad \text{then} \quad \text{then} R_0 = R_1; \n34. \quad \hspace{3em} \text{else} // n ≥ 2 \n35. \quad \hspace{4em} \text{if} (R_0 \neq \mathcal{A}_{-} \text{ and } L_1 \neq \mathcal{A}_{-}) \n36. \quad \hspace{5em} // \text{last two arguments fit together} \n37. \quad \hspace{5em} \text{then} \quad \text{then} R_0 = R_1; \n38. \quad \hspace{4em} \text{else} OK = \text{false}; \n39. \quad \hspace{4em} \text{fi} \n40. \quad \hspace{4em} \text{fi} \n41. \quad \hspace{3em} \text{fi} \n42. \quad \text{fi} \n43. \quad \text{fi} \n44. \quad \text{od} \n45. \}

\textbf{Figure 4.5:} Pseudo-code of the recursive function \text{CheckExpression}.
1. ComputeSem \((E)\)
   // computes and returns the semantics of the DNA expression \(E\)
2. {
3.    if \((E\) is an \(\uparrow\)-expression \((\uparrow \varepsilon_1)\))
4.       then \(X = \left(\varepsilon_1\right)\);
5.    else  // \(\varepsilon_1\) is a DNA expression \(E_1\)
6.       \(X_1 = \text{ComputeSem} \((E_1)\);\)
7.       \(X = \kappa(X_1);\)
8.    fi
9. return \(X;\)
10. else  // \(E\) is an \(\uparrow\)-expression or a \(\downarrow\)-expression;
11.    fi
12. for \((i = 1\) to \(n)\)
13. do  // \(E\) is an \(\uparrow\)-expression or a \(\downarrow\)-expression;
14.    if \((\varepsilon_i\) is an \(N\)-word \(\alpha_i)\)
15.       then \(X_i = \left(\alpha_i\right);\)
16.    else  // \(\varepsilon_i\) is a DNA expression \(E_i\)
17.       \(X_i = \text{ComputeSem} \((E_i)\);\)
18.    fi
19. if \((i == 1)\)  // first argument
20. then \(X = \nu^+(X_i);\)  // semantics up to current argument
21. else  // \(i \geq 2\)
22.       if \((R(X) \in A_\pm \) and \(L(X_i) \in A_\pm)\)
23.          then \(X = X \cdot \Delta \cdot \nu^+(X_i);\)
24.       else \(X = X \cdot \nu^+(X_i);\)
25.    fi
26. fi
27. od
28. return \(X;\)
29. }

Figure 4.6: Pseudo-code of the recursive function ComputeSem.

Thus, \(E_1E_2\) is just a string consisting of two separate DNA expressions. This is in line with the (natural) interpretation of DNA expressions as DNA molecules. By putting two DNA molecules in each other’s vicinity, we do not automatically get a new DNA molecule. It requires a chemical reaction to achieve that. In the world of DNA expressions, the analogue of such a chemical reaction is an operator. In particular, the operators \(\uparrow\) and \(\downarrow\) that we have defined can be used to combine two or more DNA expressions into one new DNA expression.

4.4 Computing the semantics of a DNA expression

For a given DNA expression \(E\), we can compute the semantics \(S(E)\) directly from the definition, which is part of Definition 4.1. As this definition is recursive (the semantics of a DNA expression is built up of the semantics of the arguments of the DNA expression), it is natural to use a recursive function for this. In Figure 4.6, we give such a function, called ComputeSem, which closely follows the definition.
The computational complexity of \texttt{ComputeSem}, as it is described in Figure 4.6, is dominated by the calls of the function \(\kappa\) in line 8 and the function \(\nu^+\) in lines 19, 22 and 23. Parts of the semantics of \(E\) may be subject to these functions more than once, leading to at least a quadratic time complexity in the worst case. We consider two examples of this.

In line 8, the function \(\kappa\) complements its argument \(X_1\). In fact, it only complements the single-stranded components of \(X_1\); the other components are not affected by \(\kappa\) (see the definition in (3.4)). In Figure 4.6, we have not specified how to find the single-stranded components. The most natural way to do this, would be to examine all components of \(X_1\) to see if they are single-stranded.

**Example 4.14** Let \(\alpha\) be an arbitrary \(N\)-word, and let

\[
\begin{align*}
E_1 &= \langle \uparrow \alpha \alpha \rangle \\
E_{2p} &= \langle \uparrow E_{2p-1} \langle \uparrow \alpha \rangle \alpha \rangle \quad (p \geq 1) \\
E_{2p+1} &= \langle \uparrow E_{2p} \rangle \quad (p \geq 1).
\end{align*}
\]

Hence,

\[
\begin{align*}
E_1 &= \langle \uparrow \alpha \alpha \rangle \\
E_2 &= \langle \uparrow \langle \uparrow \alpha \rangle \langle \uparrow \alpha \rangle \alpha \rangle \\
E_3 &= \langle \uparrow \langle \uparrow \alpha \rangle \langle \uparrow \alpha \rangle \alpha \rangle \\
E_4 &= \langle \uparrow \langle \uparrow \langle \uparrow \alpha \rangle \langle \uparrow \alpha \rangle \alpha \rangle \langle \uparrow \alpha \rangle \alpha \rangle
\end{align*}
\]

\[
\ldots
\]

It is easy to prove by induction on \(p\), that for any \(p \geq 1\),

- both \(E_{2p}\) and \(E_{2p+1}\) are DNA expressions,

- \[
S(E_{2p}) = \left( \frac{\alpha \alpha}{c(\alpha \alpha)} \right) \triangle \ldots \left( \frac{\alpha \alpha}{c(\alpha \alpha)} \right) \triangle \left( \frac{\alpha}{c(\alpha)} \right) \left( \frac{\alpha}{c(\alpha)} \right) \quad (4.9)
\]

- \[
S(E_{2p+1}) = \left( \frac{\alpha \alpha}{c(\alpha \alpha)} \right) \triangle \ldots \left( \frac{\alpha \alpha}{c(\alpha \alpha)} \right) \triangle \left( \frac{\alpha \alpha}{c(\alpha \alpha)} \right) \quad \text{\(p\) times}
\]

- \(|E_{2p}| = 3 \cdot 3p + (2p + 2) \cdot |\alpha|\) and \(|E_{2p+1}| = 3 \cdot (3p + 1) + (2p + 2) \cdot |\alpha|\).

In particular, the lengths of \(E_{2p}\) and \(E_{2p+1}\) are linear in \(p\).

Now, let \(p \geq 1\) and let us apply the function \texttt{ComputeSem} to the \(\downarrow\)-expression \(E_{2p+1}\), with argument \(E_{2p}\). When we call the function recursively for \(E_{2p}\) (in line 7), it returns \(X_1 = S(E_{2p})\), as described in (4.9). This semantics consists of \(2p+2\) components. It takes time that is linear in \(p\) to examine them all to see if they are single-stranded. Only the last component actually is single-stranded, and thus is complemented by the function \(\kappa\) in line 8.

Likewise, at a higher level of the recursion, we have had to examine the \(2p, 2p - 2, 2p - 4, \ldots, 4\) components of \(S(E_{2(p-1)}), S(E_{2(p-2)}), S(E_{2(p-3)}), \ldots, S(E_2)\), respectively. Altogether, this takes time that is quadratic in \(p\), and thus in the length of \(E_{2p+1}\).
In lines 19, 22 and 23 of ComputeSem, the function $\nu^+$ is applied to the formal DNA molecule $X_i$. It removes the upper nick letters from this argument. The double components preceding and succeeding such an upper nick letter are merged. The other components of $X_i$ are not affected by $\nu^+$ (see the definition in (3.1)). In Figure 4.6, we have not specified how to find the upper nick letters. The most natural way to do this, would be to examine all components of $X_i$ to see if they are upper nick letters.

**Example 4.15** Let $\alpha$ be an arbitrary $N$-word, and let

$$E_1 = \langle \uparrow \alpha \alpha \rangle$$

$$E_{2p} = \langle \uparrow E_{2p-1} \alpha \downarrow \alpha \rangle \langle \uparrow \alpha \rangle$$

$$E_{2p+1} = \langle \downarrow E_{2p} \rangle$$

$(p \geq 1)$

Hence,

$$E_1 = \langle \uparrow \alpha \alpha \rangle$$

$$E_2 = \langle \uparrow \langle \uparrow \alpha \alpha \rangle \alpha \downarrow \alpha \rangle \langle \uparrow \alpha \rangle$$

$$E_3 = \langle \downarrow \langle \uparrow \alpha \alpha \rangle \alpha \downarrow \alpha \rangle \langle \uparrow \alpha \rangle$$

$$E_4 = \langle \uparrow \downarrow \langle \uparrow \alpha \alpha \rangle \alpha \downarrow \alpha \rangle \langle \uparrow \alpha \rangle \langle \uparrow \alpha \rangle \langle \uparrow \alpha \rangle$$

$$\ldots$$

It is easy to prove by induction on $p$, that for any $p \geq 1$,

- both $E_{2p}$ and $E_{2p+1}$ are DNA expressions,

$$S(E_{2p}) = \left( \frac{\alpha \alpha}{c(\alpha \alpha)} \right)^{\alpha} \cdots \left( \frac{\alpha \alpha}{c(\alpha \alpha)} \right)^{\alpha} \left( \frac{\alpha}{c(\alpha)} \right) \triangle \left( \frac{\alpha}{c(\alpha)} \right)$$

$(4.10)$

$$S(E_{2p+1}) = \left( \frac{\alpha \alpha}{c(\alpha \alpha)} \right)^{\alpha} \cdots \left( \frac{\alpha \alpha}{c(\alpha \alpha)} \right)^{\alpha} \left( \frac{\alpha}{c(\alpha)} \right)$$

$p$ times

- $|E_{2p}| = 3 \cdot 4 + (3p + 2) \cdot |\alpha|$ and $|E_{2p+1}| = 3 \cdot (4 + 1) + (3p + 2) \cdot |\alpha|$.

In particular, the lengths of $E_{2p}$ and $E_{2p+1}$ are linear in $p$.

Now, let $p \geq 1$ and let us apply the function ComputeSem to the $\downarrow$-expression $E_{2p+1}$, with argument $E_{2p}$. When we call the function recursively for $E_{2p}$ (in line 16), it returns $X_i = S(E_{2p})$, as described in (4.10). This semantics consists of $2p + 3$ components. It takes time that is linear in $p$ to examine them all to see if they are lower nick letters. Only the last but one component actually is a lower nick letter, and thus is removed by the function $\nu^-$ in line 19 (of the analogue for $\downarrow$-expressions $E$ of ComputeSem).

Likewise, at a higher level of the recursion, we have had to examine the $2p + 1, 2p - 1, 2p - 3, \ldots, 5$ components of $S(E_{2(p-1)})$, $S(E_{2(p-2)})$, $S(E_{2(p-3)})$, $\ldots$, $S(E_2)$, respectively. Altogether, this takes time that is quadratic in $p$, and thus in the length of $E_{2p+1}$.

The quadratic time complexity of ComputeSem can be brought back to a linear one, by means of a proper data structure to store the semantics. In particular, we could maintain lists of single-stranded components (to be utilized by $\kappa$) and lists of nick letters (to be
utilized by $\nu^+$ and $\nu^-$) occurring in the semantics. This data structure would be very similar to the data structure we propose in Section 4.3 to solve a similar problem.

Here, we choose a different approach to avoid the quadratic time complexity. In the description of this approach, we use $E^*_1$ to denote the DNA expression as a whole, to clearly denote the DNA expression from the parameter $E$ of (a recursive call of) $\text{ComputeSem}$ and the expression-arguments $E_i$. When we apply $\text{ComputeSem}$ to $E^*_1$, we recursively call the function for each DNA subexpression $E$ of $E^*_1$. We now give the function three additional, boolean parameters $\uparrow$-anc, $\downarrow$-anc and $\text{↓}$-anc, which indicate whether or not the parameter $E$ has ancestor operators $\uparrow$, $\downarrow$ and $\text{↓}$, respectively. We use these three parameters to adjust the semantics of $E$ to the presence of these operators, already while evaluating $E$. Obviously, the three parameters are false, when we call $\text{ComputeSem}$ for the first time, i.e., for $E^*_1$ itself.

For example, suppose that $E$ is a $\downarrow$-expression, which is a proper DNA subexpression of an $\uparrow$-expression ($\uparrow$-anc is true). Then it does not make sense (in the end) to introduce upper nick letters into $S(E)$, as they will be removed by the occurrence of $\uparrow$, anyway. Therefore, we simply omit these upper nick letters. This implies that we do not have to apply $\nu^+$ to the arguments of the operator $\uparrow$ any more.

The parameter $\downarrow$-anc is used in an analogous way, which takes away the need to apply $\nu^-$ to the arguments of the operator $\downarrow$.

Finally, we complement an $N$-word-argument of an $\uparrow$-expression or $\downarrow$-expression $E$, if $E$ is governed by an occurrence of $\text{↓}$ ($\uparrow$-anc is true). Consequently, we do not have to apply $\kappa$ to an expression-argument of an operator $\text{↓}$.

Note that this way, the function $\text{ComputeSem}$ no longer computes the semantics of its parameter $E$, but it computes the semantics corresponding to $E$, in the context of $E^*_1$.

**Example 4.16** Consider the $\uparrow$-expression

$$E^*_1 = \langle \uparrow \downarrow \alpha_1 \langle \text{↓} \alpha_2 \rangle \langle \text{↓} \alpha_3 \rangle \rangle \langle \uparrow \alpha_4 \langle \text{↓} \alpha_5 \rangle \rangle,$$

where $\alpha_1, \ldots, \alpha_5$ are arbitrary $N$-words. When we apply the modified version of $\text{ComputeSem}$ to $E^*_1$, the recursive call for its first argument $E_1 = \langle \downarrow \alpha_1 \langle \text{↓} \alpha_2 \rangle \langle \text{↓} \alpha_3 \rangle \rangle$ will produce \((-\alpha_1)(\alpha_2\alpha_3)\), while actually $S(E_1) = (-\alpha_1)(\alpha_2\alpha_3)\). The upper nick letter is omitted, because it would be removed by the outermost operator $\uparrow$ of $E^*_1$, anyway. Indeed,

$$S(E^*_1) = (-\alpha_1)(\alpha_2\alpha_3)\langle \alpha_4 \rangle \langle \alpha_5 \rangle$$

does not contain upper nick letters (and no lower nick letters, either).

When we complement $N$-word-arguments of operators $\uparrow$ and $\downarrow$ because of the presence of an ancestor operator $\text{↓}$, we must be careful not to introduce additional (and incorrect) nick letters.

**Example 4.17** Let us apply $\text{↓}$ to the DNA expression $E^*_1$ from Example 4.16 yielding:

$$E^*_0 = \langle \text{↓} \uparrow \downarrow \alpha_1 \langle \text{↓} \alpha_2 \rangle \langle \text{↓} \alpha_3 \rangle \rangle \langle \uparrow \alpha_4 \langle \text{↓} \alpha_5 \rangle \rangle.$$ 

The $\uparrow$-argument $E^*_1$ of $E^*_0$ has two expression-arguments $E_1 = \langle \downarrow \alpha_1 \langle \text{↓} \alpha_2 \rangle \langle \text{↓} \alpha_3 \rangle \rangle$ and $E_2 = \langle \uparrow \alpha_4 \langle \text{↓} \alpha_5 \rangle \rangle$. When we (recursively) call $\text{ComputeSem}$ for $E_1$ and $E_2$, the boolean parameters $\uparrow$-anc and $\text{↓}$-anc are true, because $E_1$ and $E_2$ have $\uparrow$ and $\text{↓}$ as ancestor operators. Consequently, the function yields $X_1 = (\alpha_1\alpha_2\alpha_3)\rangle \langle \alpha_4\alpha_5 \rangle$ for $E_1$ and $E_2$, respectively.
respectively. As \( E_1 \) and \( E_2 \) are consecutive arguments of an operator \( \uparrow \), and both \( R(X_1) \) and \( L(X_2) \) are in \( A_\pm \), we might be tempted to introduce a lower nick letter between \( X_1 \) and \( X_2 \). This would, however, be incorrect because \( S(E_2) = \binom{\alpha_4}{c(\alpha_3)} \) and \( L(S(E_2)) \in A_+ \). Indeed,

\[
S(E_0) = \binom{c(\alpha_1)\alpha_2\alpha_3\alpha_4\alpha_5}{\alpha_1c(\alpha_2\alpha_3\alpha_4\alpha_5)}
\]

is nick free.

We can avoid the incorrect nick letters by giving \texttt{ComputeSem} two more return values \( L_0 \) and \( R_0 \). For a DNA expression \( E \) (the first parameter of \texttt{ComputeSem} ), \( L_0 \) and \( R_0 \) indicate the subsets of \( \mathcal{A} \) (\( A_+ \), \( A_- \) or \( A_\pm \) ) that \( L(S(E)) \) and \( R(S(E)) \) belong to, respectively. These values are used (together with \( \uparrow \)-anc and \( \downarrow \)-anc) to decide whether or not to introduce a nick letter between two arguments of an operator \( \uparrow \) or \( \downarrow \).

In Figure 4.7, we give a pseudo-code implementation of \texttt{ComputeSem}, which includes the additional features. In this implementation, the values \( L_0 \) and \( R_0 \) are returned as parameters of the function.

It is important to realize that, when we make a recursive call to \texttt{ComputeSem} for an argument \( E_i \) of a DNA expression \( E \), we do not have to copy \( E_i \) as a sequence of individual characters into the actual parameter of the call. Instead, we can make a ‘call by reference’. For example, we may simply pass the starting position of the argument (the position of its opening bracket) in the overall DNA expression \( E^* \). When, in the course of \texttt{ComputeSem}, we just keep track of the current position in the DNA expression, it does not cost time to determine this starting position. This implies that the time needed to set the actual parameters of the function is constant for each call.

For the efficiency of \texttt{ComputeSem}, it is also important that the concatenation of pieces of the semantics (as in lines 30’ and 31’ of the function) can be done in constant time. It should not require time that is proportional to the length of the pieces involved. Otherwise, we could easily construct an example DNA expression for which the running time of \texttt{ComputeSem} is quadratic, due to these concatenations. Fortunately, it is not difficult to achieve constant time concatenation, e.g., by storing the semantics in linked lists of \( \mathcal{A} \)-letters and nick letters. With such a data structure, it is also possible to efficiently return \( X \) at the end of the function (lines 11’ and 36’).

We can now prove that the running time of \texttt{ComputeSem} is linear in the length of its argument \( E \).

\textbf{Theorem 4.18} \textit{Let} \( E \) \textit{be an arbitrary DNA expression. The time required by the function} \texttt{ComputeSem} \textit{for} \( E \) \textit{is linear in} \( |E| \).

\textbf{Proof:} \textit{Let us use} \( T_{CS}(E) \) \textit{to denote the time required by} \texttt{ComputeSem} \textit{for} \( E \).

When \texttt{ComputeSem} is applied to \( E \), the function recursively examines all arguments of \( E \). This way, in principle, every letter of \( E \) is considered. This implies that \( T_{CS}(E) \) is at least linear in \( |E| \).

We now derive an \textit{upper} bound on \( T_{CS}(E) \). For this, we define five constants, which are upper bounds on the time spent in specific parts of \texttt{ComputeSem}:

\( c_1 \) \textit{is the maximum time spent in} \texttt{ComputeSem} \textit{for an} \( \uparrow \)-expression \( E \), \textit{except the time spent in recursive calls of} \texttt{ComputeSem} \textit{and the time spent to obtain the double} \( \mathcal{A} \)-word \( \binom{\alpha_1}{c(\alpha_1)} \) \textit{for an} \( \mathcal{N} \)-word-argument \( \alpha_1 \).
4.4 Computing the semantics of a DNA expression

1'. ComputeSem \((E, \uparrow\text{-anc}, \downarrow\text{-anc}, \downarrow\text{-anc}, L_0, R_0)\)
   // computes and returns the semantics
   // corresponding to the DNA expression \(E\)

2'. 

3'. if \((E\) is an \(\downarrow\)-expression \((\downarrow \epsilon_1))\)

4'. then if \((\epsilon_1\) is an \(N\)-word \(\alpha_1)\)

5'. \( \text{then } X = (\alpha_1)_{\ell(\alpha_1)} \); 

6'. \( \text{else } // \epsilon_1\) is a DNA expression \(E_1\)

7'. \( X = \text{ComputeSem} (E_1, \uparrow\text{-anc}, \downarrow\text{-anc}, \text{true}, L_1, R_1) \); 

8'. \( \text{fi} \)

9'. \( L_0 = A_\pm; \)

10'. \( R_0 = A_\pm; \)

11'. \( \text{return } X; \)

12'. \( \text{else } // E\) is an \(\uparrow\)-expression or a \(\downarrow\)-expression;
   // without loss of generality, assume it is
   // an \(\uparrow\)-expression \((\uparrow \epsilon_1 \ldots \epsilon_n)\) for some \(n \geq 1\)
   // and \(N\)-words and DNA expressions \(\epsilon_1, \ldots, \epsilon_n\)

13'. \( \text{for } (i = 1 \text{ to } n) \)

14'. \( \text{do if } (\epsilon_i\) is an \(N\)-word \(\alpha_i)\)

15'. \( \text{then if } (\uparrow\text{-anc}) \)

16'. \( \text{then } X_i = (\alpha_i)_{\ell(\alpha_i)} \); 

17'. \( \text{else } X_i = (\alpha_i)_{\ell(-\alpha_i)} \); 

18'. \( \text{fi} \)

19'. \( L_i = A_\pm; \)

20'. \( R_i = A_\pm; \)

21'. \( \text{else } // \epsilon_i\) is a DNA expression \(E_i\)

22'. \( X_i = \text{ComputeSem} (E_i, \text{true}, \downarrow\text{-anc}, \downarrow\text{-anc}, L_i, R_i) \); 

23'. \( \text{fi} \)

24'. \( \text{if } (i == 1) // \text{first argument} \)

25'. \( \text{then } X = X_i; // \text{semantics up to current argument} \)

26'. \( L_0 = L_i; \)

27'. \( R_0 = R_i; \)

28'. \( \text{else } // i \geq 2 \)

29'. \( \text{if } (R_0 == A_\pm \text{ and } L_i == A_\pm \text{ and (not } \downarrow\text{-anc}) \)

30'. \( \text{then } X = X \cdot X_i; \)

31'. \( \text{else } X = X \cdot X_i; \)

32'. \( \text{fi} \)

33'. \( R_0 = R_i; \)

34'. \( \text{fi} \)

35'. \( \text{od} \)

36'. \( \text{return } X; \)

37'. \( \text{fi} \)

38'. \} 

Figure 4.7: Pseudo-code of the modified recursive function ComputeSem.
Hence, $c_1$ is the maximum time required for setting the actual parameters in line 1' and executing lines 3'–11' and 37', except line 5' and the recursive call in line 7'.

$c_2$ is the maximum time spent in ComputeSem per letter of an $\mathcal{N}$-word-argument $\alpha_1$ of an $\uparrow$-expression $E$.

Hence, $c_2$ is the maximum time required per letter of $\alpha_1$ for executing line 5'. We define $c_2$ as the time per letter, because it is natural to assume that the time required to obtain the double $\mathcal{A}$-word $\left(\frac{\alpha_1}{c(\alpha_1)}\right)$ from the $\mathcal{N}$-word $\alpha_1$ is (at most) proportional to the length of $\alpha_1$.

$c_3$ is the maximum time spent in ComputeSem for an $\uparrow$-expression $E$, except the time spent for each of its arguments $\varepsilon_1, \ldots, \varepsilon_n$.

Hence, $c_3$ is the maximum time required for setting the actual parameters in line 1' and executing lines 3', 12', 36'–37' and the initialization of the for-loop in line 13'.

$c_4$ is the maximum time spent in ComputeSem for an argument $\varepsilon_i$ of an $\uparrow$-expression $E$, except the time spent in recursive calls of ComputeSem and the time spent to obtain a double $\mathcal{A}$-word $\left(\frac{\alpha_i}{c(\alpha_i)}\right)$ or an upper $\mathcal{A}$-word $(\alpha_i)$ for an $\mathcal{N}$-word-argument $\alpha_i$.

Hence, $c_4$ is the time required for executing lines 14'–15', 18'–21', 23'–35' and the iteration in line 13'.

$c_5$ is the maximum time spent in ComputeSem per letter of an $\mathcal{N}$-word-argument $\alpha_i$ of an $\uparrow$-expression $E$.

Hence, $c_5$ is the time required per letter of $\alpha_i$ for executing lines 16' or 17'. We define $c_5$ as the time per letter, for the same reason as for $c_2$.

It follows from the observations made before this result (about passing parameters for the recursive calls of ComputeSem and about the data structure to store (pieces of) the semantics), that $c_1, \ldots, c_5$ are indeed constants.

Now, let

$$c^* = \max\left\{c_2, \frac{c_1 + c_4}{3}, c_4 + c_5, \frac{c_3 + c_4}{3}\right\}.$$  

We prove by induction on the number $p$ of operators occurring in $E$, that $T_{CS}(E) \leq c^* \cdot |E| - c_4$. We subtract $c_4$ here, to be prepared for the additional constant time required for every argument of an $\uparrow$-expression $E$. We come back to this later.

- If $p = 1$, then $E$ has only $\mathcal{N}$-word-arguments. When we apply ComputeSem to $E$, we do not have recursive calls of the function.

If $E$ is an $\uparrow$-expression, then $E = \langle \uparrow \alpha_1 \rangle$ for an $\mathcal{N}$-word $\alpha_1$. In this case, $|E| = 3 + |\alpha_1|$ and

$$T_{CS}(E) \leq c_1 + c_2 \cdot |\alpha_1| \leq 3c^* - c_4 + c_4|\alpha_1| = c^* \cdot |E| - c_4,$$

where the second inequality follows from $c^* \geq \frac{c_1 + c_4}{3}$ (which is equivalent to $c_1 \leq 3c^* - c_4$) and $c^* \geq c_2$. 

If $E$ is an $\uparrow$-expression, then $E = \langle \uparrow \alpha_1 \ldots \alpha_n \rangle$ for some $n \geq 1$ and $\mathcal{N}$-words $\alpha_1, \ldots, \alpha_n$. In this case, $|E| = 3 + |\alpha_1 \ldots \alpha_n|$ and

$$T_{CS}(E) \leq c_3 + (c_4 + c_5 \cdot |\alpha_1|) + \cdots + (c_4 + c_5 \cdot |\alpha_n|)$$

$$\leq 3c^* - c_4 + (c_4 + c_5) \cdot |\alpha_1| + \cdots + (c_4 + c_5) \cdot |\alpha_n|)$$

$$\leq 3c^* - c_4 + c^* \cdot |\alpha_1 \ldots \alpha_n| = c^* \cdot |E| - c_4,$$

where the second inequality follows from $c^* \geq \frac{c_4+c_5}{3}$ and $|\alpha_i| \geq 1$ for $i = 1, \ldots, n$, and the third inequality follows from $c^* \geq c_4 + c_5$.

If $E$ is a $\downarrow$-expression, then the proof is completely analogous.

- Let $p \geq 1$, and suppose that $T_{CS}(E) \leq c^* \cdot |E| - c_4$ for all DNA expressions containing at most $p$ operators (induction hypothesis). We now consider a DNA expression $E$ that contains $p + 1$ operators.

If $E$ is an $\uparrow$-expression, then $E = \langle \downarrow \, E_1 \rangle$ for a DNA expression $E_1$ with $p$ operators. By the induction hypothesis, $T_{CS}(E_1) \leq c^* \cdot |E_1| - c_4$. In this case, $|E| = 3 + |E_1|$ and

$$T_{CS}(E) \leq c_1 + T_{CS}(E_1) \leq 3c^* - c_4 + c^* \cdot |E_1| - c_4 = c^* \cdot |E| - 2c_4,$$

where the second inequality follows from $c^* \geq \frac{c_1+c_4}{3}$.

If $E$ is an $\downarrow$-expression, then $E = \langle \uparrow \, \varepsilon_1 \ldots \varepsilon_n \rangle$ for some $n \geq 1$ and $\mathcal{N}$-words and DNA expressions $\varepsilon_1, \ldots, \varepsilon_n$. In this case, $|E| = 3 + |\varepsilon_1 \ldots \varepsilon_n|$. We consider the time spent by $\text{ComputeSem}$ on a single argument $\varepsilon_i$ of $E$.

If $\varepsilon_i$ is an $\mathcal{N}$-word $\alpha_i$, then this time is bounded by $c_4 + c_5 \cdot |\alpha_i|$. As we have seen in the proof of the base case of the induction, this is at most $c^* \cdot |\alpha_i|$. If, on the other hand, $\varepsilon_i$ is a DNA expression $E_i$, then $E_i$ contains at most $p$ operators. Hence, we can apply the induction hypothesis to it: $T_{CS}(E_i) \leq c^* \cdot |E_i| - c_4$. Now the time spent by $\text{ComputeSem}$ on the argument $E_i$ of $E$ is bounded by $c_4 + T_{CS}(E_i) \leq c^* \cdot |E_i|$. Note that this is where we benefit from the term $- c_4$ in the induction hypothesis.

We conclude that both if $\varepsilon_i$ is an $\mathcal{N}$-word $\alpha_i$, and if it is a DNA expression $E_i$, $\text{ComputeSem}$ spends at most $c^* \cdot |\varepsilon_i|$ on this argument of $E$. This implies that

$$T_{CS}(E) \leq c_3 + c^* \cdot |\varepsilon_1| + \cdots + c^* \cdot |\varepsilon_n|$$

$$\leq 3c^* - c_4 + c^* \cdot |\varepsilon_1 \ldots \varepsilon_n| = c^* \cdot |E| - c_4,$$

where the second inequality follows from $c^* \geq \frac{c_4+c_5}{3}$.

If $E$ is a $\downarrow$-expression, then the proof is completely analogous.

We have thus proved that for each DNA expression $E$, $T_{CS}(E) \leq c^* \cdot |E| - c_4$. This implies that $T_{CS}(E)$ is at most linear in $|E|$.

We briefly discuss the space complexity of the function $\text{ComputeSem}$. For this, let us again use $E^*_i$ to denote the DNA expression that we apply the function to.

The assumption that we simply pass the starting position of an expression-argument $E_i$ to a recursive call of $\text{ComputeSem}$ (rather than copying $E_i$ itself) is also important for the space complexity. It implies that the space required to store the parameters of
a single call is constant. For each DNA subexpression of \( E_1^* \), the function \( \text{ComputeSem} \) is called exactly once. Since there is a 1–1 correspondence between DNA subexpressions and occurrences of operators in a DNA expression, the total space required to store the parameters for all calls is at most linear in \( |E_1^*| \).

For every \( N \)-word-argument \( \alpha_i \) of its parameter \( E \), \( \text{ComputeSem} \) produces an upper \( A \)-word, a lower \( A \)-word or a double \( A \)-word. For an expression-argument, it may produce a nick letter. These \( A \)-words and nick letters become parts of the overall semantics \( S(E_1^*) \). Because each part of the semantics is produced only once (and no copies are made) during the execution of \( \text{ComputeSem} \), the total space required to store (parts of) the semantics is linear in \( |S(E_1^*)| \). It is not too hard to prove that this is at most linear in \( |E_1^*| \).

It is instructive to consider examples of DNA expressions for which \( \text{ComputeSem} \) does not require linear space, or for which it really requires less than linear space.

**Example 4.19** Let \( \alpha \) be an arbitrary \( N \)-word, and let \( E_p \) be defined by

\[
E_p = \left( \uparrow \uparrow \cdots \uparrow \alpha \right) \quad (p \geq 1).
\]

It is easy to see that for any \( p \geq 1 \), \( E_p \) is a DNA expression, with \( |E_p| = 3p + |\alpha| \) and \( S(E_p) = \left( \alpha \right) \). When we call the recursive function \( \text{ComputeSem} \) for \( E_p \), the maximum nesting level of the recursion becomes \( p \). This implies that the space we need to store the parameters of subsequent recursive calls is linear in \( p \), and thus linear in \( |E_p| \).

**Example 4.20** Let \( \alpha \) be an arbitrary \( N \)-word, and let \( E_p \) be defined by

\[
E_p = \left( \uparrow \alpha \cdots \alpha \right) \quad (p \geq 1).
\]

It is easy to see that for any \( p \geq 1 \), \( E_p \) is a DNA expression, with \( |E_p| = 3 + p \cdot |\alpha| \) and \( S(E_p) = \left( \alpha \cdots \alpha \right) \). Hence, \( |S(E_p)| = p \cdot |\alpha| \), which implies that the space we need to store the result of \( \text{ComputeSem} \) is linear in \( p \), and thus linear in \( |E_p| \).

**Example 4.21** Let \( \alpha \) be an arbitrary \( N \)-word, and let \( E_p \) be defined by

\[
E_p = \left( \uparrow \left( \uparrow \cdots \uparrow \alpha \right) \cdots \uparrow \left( \uparrow \cdots \uparrow \alpha \right) \right) \quad (p \geq 1).
\]

It is easy to see that for any \( p \geq 1 \), \( E_p \) is a DNA expression, with \( |E_p| = 3 + p \cdot (3p + |\alpha|) = 3 + 3p^2 + p \cdot |\alpha| \) and \( S(E_p) = \left( \alpha \cdots \alpha \right) \). Because \( |E_p| \) is quadratic in \( p \), \( p \) is linear in \( \sqrt{|E_p|} \).

When we call the recursive function \( \text{ComputeSem} \) for \( E_p \), the maximum nesting level of the recursion becomes \( p + 1 \). This implies that the space we need to store the parameters of subsequent recursive calls is linear in \( p \), and thus linear in \( \sqrt{|E_p|} \).

Moreover, \( |S(E_p)| = p \cdot |\alpha| \), which implies that the space we need to store the result of \( \text{ComputeSem} \) is linear in \( p \), and thus linear in \( \sqrt{|E_p|} \).

We can therefore conclude:

**Theorem 4.22** Let \( E \) be an arbitrary DNA expression. In the worst case, the space required by the function \( \text{ComputeSem} \) for \( E \) is linear in \( |E| \).
4.5 A context-free grammar for $\mathcal{D}$

As we have established in Lemma 3.3, the language $\mathcal{F}$ of formal DNA molecules is regular. This is not the case with the language $\mathcal{D}$ of all DNA expressions. This is intuitively clear from the fact that every DNA expression contains matching brackets $\langle$ and $\rangle$, and that these brackets may be deeply nested. We use this intuition in a formal proof.

**Lemma 4.23** The language $\mathcal{D}$ of DNA expressions is not regular.

**Proof:** Let $\alpha$ be an arbitrary $N$-word. Then $E_1 = \langle \uparrow \alpha \rangle$ is a DNA expression, and $S(E_1) = \left( \overset{\alpha}{c(\alpha)} \right)$. By definition, also $E_2 = \langle \downarrow \langle \uparrow \alpha \rangle \rangle$ is a DNA expression, with the same semantics. It is easy to see that for arbitrary $p \geq 1$,

$$E_p = \langle \uparrow \langle \downarrow \ldots \langle \uparrow \alpha \rangle \ldots \rangle \rangle_{p \text{ times}}$$

is a DNA expression, with $S(E_p) = \left( \overset{\alpha}{c(\alpha)} \right)$. By the pumping lemma for regular languages (Proposition 2.7), a language requiring brackets to match and containing such DNA expressions is not regular. 

The language $\mathcal{D}$ is, however, context-free, because it can be generated by a context-free grammar. We will give such a grammar, here. It is a 4-tuple $G_1 = (V_1, \Sigma_1, P_1, S_1)$, which is based on three types of non-terminal symbols: $E$ (which denotes a DNA expression), $U$ (a sequence of one or more arguments of an $\uparrow$-expression) and $L$ (a sequence of one or more arguments of a $\downarrow$-expression).

The crucial issue in the construction of a context-free grammar generating $\mathcal{D}$, is that we must somehow incorporate the requirement that consecutive arguments of an operator $\uparrow$ or $\downarrow$ fit together by upper strands or lower strands, respectively. For this, the non-terminal symbols $E$, $U$ and $L$ have two subscripts. The first subscript denotes whether or not one of the strands of the (sub)molecule represented by the non-terminal has to cover the other strand to the left. If it is $+$, then the upper strand must cover the lower strand to the left; if it is $-$, then the lower strand must cover the upper strand to the left; if it is $\ast$, then it does not matter if either strand strictly covers the other strand to the left. The second subscript has an analogous meaning, with respect to covering to the right. For example, the symbol $U_{+, -}$ denotes a sequence of arguments of an $\uparrow$-expression and $L$ (a sequence of one or more arguments of a $\downarrow$-expression).

Before we present the productions in $G_1$ (i.e., the elements of $P_1$) we discuss why we have exactly those productions.

We first consider the productions for (rewriting) a non-terminal symbol $E_{x,y}$ with $x, y \in \{\ast, +, -\}$, which represents an arbitrary $N$-word. We thus have the following set of non-terminal symbols:

$$\{E_{x,y}, U_{x,y}, L_{x,y} \mid x, y \in \{\ast, +, -\}\} \cup \{\alpha\}.$$ 

The axiom is $S_1 = E_{x,*,}$, which denotes a DNA expression without restrictions on the two strands. The alphabet $\Sigma_1$ of terminal symbols is equal to $\Sigma_D$: $\Sigma_1 = \{A, C, G, T, \uparrow, \downarrow, \langle, \rangle\}$.

Before we present the productions in $G_1$ (i.e., the elements of $P_1$) we discuss why we have exactly those productions.

We first consider the productions for (rewriting) a non-terminal symbol $E_{x,y}$ with $x, y \in \{\ast, +, -\}$, which represents a DNA expression.

By Lemma 4.13(2), for any $\downarrow$-expression $E$, we have $L(S(E)), R(S(E)) \in A_\pm$. Hence, the upper strand of $E$ covers the lower strand to both the left and the right, and vice
versa. This implies that, regardless of the subscripts \(x\) and \(y\), we may rewrite \(E_{x,y}\) into any ↑-expression. Therefore, we have productions \(E_{x,y} \rightarrow \langle \downarrow \alpha \rangle\) and \(E_{x,y} \rightarrow \langle \uparrow E_{*,*} \rangle\). Indeed, the non-terminal \(\alpha\) occurring in the former production represents an arbitrary \(N\)-word, and the non-terminal \(E_{*,*}\) occurring as argument of \(\uparrow\) in the latter production represents an arbitrary DNA expression, without restrictions on the strands.

By Lemma 4.13(3), for an ↑-expression \(E\), the values of the functions \(L\) and \(R\) depend (solely) on the values for the first and the last argument of \(E\), respectively. Therefore, if we want to rewrite \(E_{x,y}\) into an ↑-expression, then the subscripts \(x\) and \(y\) simply carry over to the non-terminal \(U\) representing the arguments of the ↑-expression. We thus have a production \(E_{x,y} \rightarrow \langle \uparrow U_{x,y} \rangle\). Analogously, we have \(E_{x,y} \rightarrow \langle \downarrow L_{x,y} \rangle\).

Next, consider a non-terminal symbol \(U_{x,y}\) for some subscripts \(x, y \in \{*, +, -\}\). This non-terminal must be rewritten into a sequence of \(n \geq 1\) arguments for an occurrence of ↑. We do this in a right-linear, recursive way: we rewrite \(U_{x,y}\) into a non-terminal \(\alpha\) or \(E\) (with some subscripts) representing the first argument, possibly followed by another non-terminal \(U\) (with some subscripts), representing the second and later arguments.

If \(n \geq 2\), so that we indeed need a new non-terminal symbol \(U\) for the second and later arguments, then the subscripts in the right-hand side of the production reflect the requirement that the arguments of ↑ fit together by upper strands. In particular, if the first argument is a DNA expression, then the second subscript of the non-terminal symbol \(E\) representing it must be +. Further, the first subscript of the new non-terminal symbol \(U\) must be +.

**Example 4.24** The non-terminal symbol \(U_{*,+}\) represents a sequence of arguments of ↑ with no restrictions on the left-hand side of the first argument, but for which the upper strand of the last argument must cover the lower strand on the right. We have four productions for this symbol: \(U_{*,+} \rightarrow \alpha\) (indeed, the upper strand of \(S^+(\alpha) = \langle a \rangle\) covers the lower strand on the right), \(U_{*,+} \rightarrow E_{*,+}\), \(U_{*,+} \rightarrow \alpha U_{*,+}\) and \(U_{*,+} \rightarrow E_{*,+}U_{*,+}\) (see the productions in line 11 below).

**Example 4.25** The non-terminal symbol \(U_{-,*}\) represents a sequence of arguments of ↑ for which the lower strand of the first argument must cover the upper strand on the left, and for which there are no restrictions on the right-hand side of the last argument. Because the lower strand of \(S^+(\alpha) = \langle a \rangle\) does not cover the upper strand on the left, the first argument cannot be an \(N\)-word \(\alpha\). Hence, we have only two productions for this symbol: \(U_{-,*} \rightarrow E_{-,*}\) and \(U_{-,*} \rightarrow E_{-,*}U_{+,+}\) (see the productions in line 16 below).

There is, of course, an analogous explanation for the productions for a non-terminal \(L_{x,y}\) with \(x, y \in \{*, +, -\}\).

The grammatical structure of an \(N\)-word (represented by the non-terminal symbol \(\alpha\)) is similar to that of the sequence of arguments of ↑ or ↓. An \(N\)-word is an arbitrary sequence of \(r \geq 1\) \(N\)-letters. We obtain this sequence from the non-terminal symbol \(\alpha\) by recursively rewriting this symbol into an \(N\)-letter, possibly followed by another non-terminal \(\alpha\).

Thus, the set \(P_1\) consists of the following productions:

1. \(E_{*,+} \rightarrow \langle \downarrow \alpha \rangle\) | \(\langle \uparrow E_{*,*} \rangle\) | \(\langle \uparrow U_{*,*} \rangle\) | \(\langle \downarrow L_{*,*} \rangle\)
2. \(E_{*,+} \rightarrow \langle \downarrow \alpha \rangle\) | \(\langle \uparrow E_{*,*} \rangle\) | \(\langle \uparrow U_{*,+} \rangle\) | \(\langle \downarrow L_{*,+} \rangle\)
3. \(E_{*,-} \rightarrow \langle \downarrow \alpha \rangle\) | \(\langle \uparrow E_{*,*} \rangle\) | \(\langle \uparrow U_{*,-} \rangle\) | \(\langle \downarrow L_{*,-} \rangle\)
Note that the first nine lines of the above list can be summarized by

\[
E_{x,y} \rightarrow \langle \downarrow \alpha \rangle \mid \langle \downarrow E_{x,y} \rangle \mid \langle \uparrow U_{x,y} \rangle \mid \langle \downarrow L_{x,y} \rangle \quad (x, y \in \{*, +, -\}).
\]

Similarly, sets of two, three or four other lines of productions can be summarized by single lines. However, the description by separate lines for each of the non-terminal symbols makes it easier to refer to a particular production, as we do in the example below.

Note also that there is a subtle relation between grammar G1 and the recursive functions `CheckExpression` and `ComputeSem` from Figure 4.5 and Figure 4.7. Both functions return in their parameters \(L_0\) and \(R_0\) the subsets of \(A\) (\(A_+\), \(A_-\) or \(A_\pm\)) that the left-hand side and the right-hand side of \(S(E)\) belong to. That is, these values are passed upwards in the tree of recursive calls of the functions.

In the grammar, the subscripts \(+\), \(-\), \(*\) of the non-terminal symbols \(E\), \(U\) and \(L\) indicate the subsets \((A_+ \cup A_\pm, A_- \cup A_\pm\) or the entire set \(A\)) that the left-hand side and the right-hand side of the (sub)molecule represented should belong to. That is, this information is passed downwards in the derivation tree of a DNA expression.
Figure 4.8: Two derivation trees in $G_1$ for the DNA expression $\langle \uparrow \text{AT} \rangle$ from Example 4.26. (a) The derivation tree corresponding to Derivation (4.11). (b) The derivation tree corresponding to Derivation (4.12).

The context-free grammar $G_1$ is not entirely unambiguous, as can be seen from the following, simple example:

Example 4.26 Consider the $\uparrow$-expression $E = \langle \uparrow \text{AT} \rangle$. There exist two leftmost derivations of $E$ in $G_1$:

\[
\begin{align*}
E_{s,*} & \xrightarrow{11} \langle U_{s,*} \rangle \\
& \xrightarrow{11} \langle \uparrow \alpha \rangle \\
& \xRightarrow{28} \langle \uparrow \alpha \rangle \\
& \xRightarrow{28} \langle \uparrow \text{AT} \rangle \\
\end{align*}
\]  

and

\[
\begin{align*}
E_{s,*} & \xrightarrow{11} \langle U_{s,*} \rangle \\
& \xrightarrow{11} \langle \uparrow \alpha \rangle \\
& \xRightarrow{28} \langle \uparrow \alpha \rangle \\
& \xRightarrow{28} \langle \uparrow \text{AT} \rangle \\
\end{align*}
\]  

Here, numbers $i, j$ above an arrow $\Rightarrow$ indicate that we have used production $(i, j)$ for the corresponding derivation step. In the former derivation, the $\mathcal{N}$-word AT is derived from a single non-terminal symbol $\alpha$. In the latter derivation, the two $\mathcal{N}$-letters A and T are derived from two independent non-terminal symbols $\alpha$. The corresponding derivation trees are depicted in Figure 4.8. Indeed, for both trees, the yield is equal to $\langle \uparrow \text{AT} \rangle$.

Hence, let $\alpha_i$ be a maximal $\mathcal{N}$-word occurrence of length $|\alpha_i| \geq 2$, with parent operator $\uparrow$ or $\downarrow$. Then $\alpha_i$ can be derived in $G_1$ from $r$ independent, consecutive symbols $\alpha$, where
4.5 A context-free grammar for $\mathcal{D}$

$r$ can take any value satisfying $1 \leq r \leq |\alpha_i|$. Moreover, if $1 < r < |\alpha_i|$, then there are multiple ways to partition $\alpha_i$ over the $r$ symbols $\alpha$. As follows from a careful inspection of the productions in the grammar, this is the only type of ambiguity occurring in $G_1$.

**Example 4.27** The DNA expression from Example 4.2 is the result of many different derivations in $G_1$. A leftmost derivation is

\[
\begin{align*}
E_{*,*} & \xrightarrow{11} \langle \downarrow L_{*,*} \rangle \\
& \xrightarrow{12} \langle \downarrow \alpha L_{*,*} \rangle \\
& \xrightarrow{28} \langle \downarrow TL_{*,*} \rangle \\
& \xrightarrow{25} \langle \downarrow TE_{*,*} \rangle \\
& \xrightarrow{9} \langle \downarrow T \langle \uparrow U_{*,*} \rangle \rangle L_{*,*} \rangle \\
& \xrightarrow{18} \langle \downarrow T \langle \uparrow E_{+,*} \rangle \rangle \rangle L_{*,*} \rangle \\
& \xrightarrow{8} \langle \downarrow T \langle \uparrow \langle \alpha \rangle U_{+,*} \rangle \rangle L_{*,*} \rangle \\
& \xrightarrow{28} \langle \downarrow T \langle \uparrow \langle \alpha \rangle U_{+,*} \rangle \rangle L_{*,*} \rangle \\
& \xrightarrow{15} \langle \downarrow T \langle \uparrow \langle \alpha \rangle U_{+,*} \rangle \rangle L_{*,*} \rangle \\
& \xrightarrow{28} \langle \downarrow T \langle \uparrow \langle \alpha \rangle AT U_{+,*} \rangle \rangle L_{*,*} \rangle \\
& \xrightarrow{15} \langle \downarrow T \langle \uparrow \langle \alpha \rangle AT E_{+,*} \rangle \rangle L_{*,*} \rangle \\
& \xrightarrow{10} \langle \downarrow T \langle \uparrow \langle \alpha \rangle \rangle AT \langle \downarrow U_{+,*} \rangle \rangle L_{*,*} \rangle \\
& \xrightarrow{27} \langle \downarrow T \langle \uparrow \langle \alpha \rangle \rangle AT \langle \downarrow E_{+,*} \rangle \rangle L_{*,*} \rangle \\
& \xrightarrow{15} \langle \downarrow T \langle \uparrow \langle \alpha \rangle \rangle AT \langle \downarrow E_{+,*} \rangle \rangle L_{*,*} \rangle \\
& \xrightarrow{28} \langle \downarrow T \langle \uparrow \langle \alpha \rangle \rangle AT \langle \downarrow E_{+,*} \rangle \rangle L_{*,*} \rangle \\
& \xrightarrow{28} \langle \downarrow T \langle \uparrow \langle \alpha \rangle \rangle AT \langle \downarrow E_{+,*} \rangle \rangle L_{*,*} \rangle \\
& \xrightarrow{10} \langle \downarrow T \langle \uparrow \langle \alpha \rangle \rangle AT \langle \downarrow E_{+,*} \rangle \rangle L_{*,*} \rangle \\
& \xrightarrow{27} \langle \downarrow T \langle \uparrow \langle \alpha \rangle \rangle AT \langle \downarrow E_{+,*} \rangle \rangle L_{*,*} \rangle \\
& \xrightarrow{15} \langle \downarrow T \langle \uparrow \langle \alpha \rangle \rangle AT \langle \downarrow E_{+,*} \rangle \rangle L_{*,*} \rangle \\
& \xrightarrow{28} \langle \downarrow T \langle \uparrow \langle \alpha \rangle \rangle AT \langle \downarrow E_{+,*} \rangle \rangle L_{*,*} \rangle \\
& \xrightarrow{28} \langle \downarrow T \langle \uparrow \langle \alpha \rangle \rangle AT \langle \downarrow E_{+,*} \rangle \rangle L_{*,*} \rangle \\
& \xrightarrow{10} \langle \downarrow T \langle \uparrow \langle \alpha \rangle \rangle AT \langle \downarrow E_{+,*} \rangle \rangle L_{*,*} \rangle \\
& \xrightarrow{27} \langle \downarrow T \langle \uparrow \langle \alpha \rangle \rangle AT \langle \downarrow E_{+,*} \rangle \rangle L_{*,*} \rangle
\end{align*}
\]
Figure 4.9: The derivation tree corresponding to the derivation in $G_1$ from Example 4.27.

$$\langle \downarrow T \langle \uparrow \langle \downarrow G \langle \uparrow \langle \downarrow C \langle \uparrow \rangle \rangle \rangle \rangle \rangle$$.

Figure 4.9 contains the corresponding derivation tree. Indeed, the yield of the tree is equal to the DNA expression from Example 4.2.

Because the definition of $G_1$ closely follows the definition of DNA expressions, we have

**Theorem 4.28** $\mathcal{L}(G_1) = \mathcal{L}_{G_1}(E_{*,*})$ is the language $\mathcal{D}$ of all DNA expressions.
Corollary 4.29  The language $D$ of DNA expressions is context-free.

Theorem 4.28 may be viewed as an alternative definition of DNA expressions: a string is a DNA expression, if and only if it is an element of $L(G_1)$. Unlike Definition 4.1, this alternative definition does not explicitly refer to the semantics. Therefore, it may be considered a ‘cleaner’ definition. Note, however, that the semantics is implicitly present in $G_1$, in the subscripts of the non-terminal symbols $E$, $U$ and $L$.

4.6 The structure tree of a DNA expression

In Section 4.5, we have seen that we can represent a DNA expression by the corresponding derivation tree in a context-free grammar for $D$. However, already for the small example DNA expression from Example 4.2, the resulting tree was large (see Figure 4.9).

We now introduce a more concise tree notation for DNA expressions, which, moreover, does not depend on a particular context-free grammar. The resulting trees are again ordered, rooted and node-labelled. After a description of the new tree notation, we will discuss its relation with the derivation trees from Section 4.5.

Let $E$ be an arbitrary DNA expression. For an unambiguous definition of the tree corresponding to $E$, it is important to know what exactly are the arguments, and in particular the $N$-word-arguments, of operators ↑ and ↓ occurring in $E$. In line with Theorem 4.9, we assume that each $N$-word-argument of each operator occurring in $E$ is a maximal $N$-word occurrence. This is not really essential for the trees we want to describe, but it makes the description ‘cleaner’.

We define the structure tree of $E$ as follows. For each maximal $N$-word occurrence $\alpha$ and for each operator occurring in $E$, we have a node, labelled by this $N$-word or operator. Recall that there is a 1–1 correspondence between (occurrences of) DNA subexpressions and operators in $E$. Therefore, every node labelled by an operator corresponds to a DNA subexpression of $E$.

Of course, a node is something different than the label of that node. Much in the same way as that the occurrence of an operator in a DNA expression is something different than that operator itself. However, to keep the text more readable, we will sometimes say ‘$N$-word’ or ‘operator’ (‘DNA subexpression’) when we actually mean the corresponding node in the tree. This meaning will be clear from the context then.

In the structure tree we draw edges from operators to their arguments. By definition, these arguments are $N$-words and DNA subexpressions of $E$, and by assumption, the $N$-word-arguments are maximal $N$-word occurrences in $E$. Indeed, for every such argument, there is a corresponding node. Hence, the edges are well defined.

Because every maximal $N$-word occurrence and every proper DNA subexpression of $E$ has exactly one parent operator, we indeed obtain a tree. The node labelled by the outermost operator of $E$ is the root of the tree. It corresponds to the entire DNA expression.

Clearly, the node labelled by an operator is the parent of (the nodes corresponding to) its arguments. If an operator has two or more arguments, then its children in the structure tree are arranged from left to right in the same order as the corresponding arguments in the DNA expression.

The leaves of the tree are labelled by the maximal $N$-word occurrences $\alpha$ of $E$, and the internal nodes by the operators. As an example, in Figure 4.10 we have drawn the structure tree of the DNA expression from Example 4.2.
We just recalled the correspondence between DNA subexpressions and operators. In fact, in the structure tree of a DNA expression $E$, a DNA subexpression of $E$ is stored in the subtree rooted in its outermost operator.

There is a very close relation between the maximal nesting level of a DNA expression and the height of the corresponding structure tree:

**Lemma 4.30** Let $E$ be a DNA expression, let $l$ be the maximal nesting level of $E$, and let $t$ be the structure tree of $E$. Then the height of $t$ is $l + 1$.

As we observed in Section 4.2, the maximal nesting level of the DNA expression from Example 4.2 is 4 (see Figure 4.3). Indeed, the height of the corresponding structure tree in Figure 4.10 is $4 + 1 = 5$.

**Proof:** Straightforward by induction on the number of operators occurring in $E$. In the induction step, we can apply Lemma 2.1(2) and Lemma 4.7(2).

The transformation from DNA expressions to structure trees is injective. This means that when we are given a ‘syntactically correct’ ordered, rooted, node-labelled tree, we can perform the inverse transformation. The syntactic demands we impose on the trees are similar to those for a string over $\Sigma_D$ to be a DNA expression (see Section 4.3):

- internal nodes are labelled by operators and leaves by $N$-words
- a node labelled by $\uparrow$ has only one child
- if a node labelled by $\uparrow$ or $\downarrow$ has two or more children, then the DNA subexpressions corresponding to these children fit together by upper strands or lower strands, respectively.

This final requirement is in fact a recursive one, as it presupposes that the subtrees rooted in the children can be interpreted as DNA expressions.

If this assumption is valid, then the prefitting requirement can be checked in a way similar to that for DNA expressions. Suppose that the $i^{th}$ child of a node labelled by $\uparrow$ corresponds to an $N$-word or a DNA expression $\varepsilon_i$, which has to prefilt the $(i + 1)^{st}$ child by upper strands. Then, e.g., $R(S^+(\varepsilon_i))$ must be an element of $A_\pm \cup A_+$.

We can check this condition by walking the rightmost path in the subtree rooted in $\varepsilon_i$. This path ends in a certain $N$-word $\alpha$. If the parent of $\alpha$ is a node labelled by $\downarrow$ or $\uparrow$, then $R(S^+(\varepsilon_i))$ certainly belongs to $A_\pm \cup A_+$. If not (hence, if the parent of $\alpha$ is labelled...
by $\downarrow$), then the composite symbol $R(S^+(\varepsilon_i))$ cannot be an element of $\mathcal{A}_+$, as the lower part of the symbol is not equal to $\downarrow$. In order for $R(S^+(\varepsilon_i))$ to be in $\mathcal{A}_{\pm}$, there has to be a node labelled by $\downarrow$ on the path from $\varepsilon_i$ down to $\alpha$ (including $\varepsilon_i$), and this is easy to verify.

If (and only if) a tree satisfies the three requirements mentioned, it is a structure tree and thus represents a DNA expression: the DNA expression of the tree.

In order to obtain this DNA expression, we have to perform a depth first search walk through the structure tree. In this walk, when entering an internal node $X$ for the first time, we collect the opening bracket $\langle$ and the operator of the node. Next, we collect the argument(s) of the operator by recursively visiting the child(ren) of the node, and finally, when returning to $X$, we obtain the closing bracket $\rangle$. Apart from this closing bracket, the walk can be considered as a preorder walk.

Although the structure tree of a DNA expression is very different from (in particular, much smaller than) the derivation tree in the context-free grammar from Section 4.5, there is a natural relation between them. In four steps, removing more and more redundant nodes, we can transform the derivation tree of a DNA expression into the structure tree:

1. An internal node of a tree can have an arbitrary number $n \geq 1$ of children. Hence, all arguments of a DNA (sub-)expression can be connected directly to that DNA (sub-)expression. We do not need to recursively break down a sequence of arguments into one argument, possibly followed by another sequence of arguments. This way, we get rid of all nodes labelled by $U$ or $L$ with some combination of subscripts.

2. The label of a leaf in the tree may be an $\mathcal{N}$-word of arbitrary length $r \geq 1$. We do not need to recursively break down an $\mathcal{N}$-word-argument into one $\mathcal{N}$-letter, possibly followed by another $\mathcal{N}$-word. In particular, we get rid of all nodes labelled by $\alpha$.

3. Occurrences of operators in a DNA expression correspond 1–1 to occurrences of DNA (sub-)expressions. We do not need two nodes for them: a node $X$ for the DNA (sub-)expression and a node for the operator, which always is the second child of $X$. We can as well remove the second child of $X$ and label $X$ by the operator. This way, we get rid of all labels $E$ with some combination of subscripts.

4. The scope of an operator is formed by the labels of its descendants in the tree. We do not need brackets to delimit the scope explicitly, and can therefore remove the nodes labelled by brackets.

The result of this four-step procedure is exactly the structure tree of a DNA expression as defined earlier in this section. As an illustration of the procedure, Figure 4.11 shows the intermediate trees for the DNA expression from Example 4.2.

4.7 Equivalent DNA expressions

Different DNA expressions may correspond to the same DNA molecule. For example, both $\langle \uparrow \alpha \rangle$ and $\langle \uparrow (\uparrow \alpha) \rangle$ denote the formal DNA molecule $\left(\begin{array}{c} \alpha \\ \end{array}\right)$. It is also possible that different DNA expressions denote ‘almost the same’ DNA molecule for a certain interpretation of ‘almost the same’. To express these things formally, we define four binary relations on $\mathcal{D}$. 
Figure 4.11: The derivation tree of the DNA expression from Example 4.2 is transformed into the structure tree. The original derivation tree is shown in Figure 4.9.

1. All arguments of a DNA (sub-)expression are connected directly to that DNA (sub-)expression.
2. The $N$-letters constituting an $N$-word-argument are substituted by the $N$-word.
3. The operators move up the tree. They take over the places (the nodes) of the DNA (sub-)expressions that they govern.
4. Brackets are removed. The resulting structure tree is the one from Figure 4.10.
Definition 4.31 Two DNA expressions $E_1$ and $E_2$ are strictly equivalent, or equivalent for short, if $S(E_1) = S(E_2)$. We write $E_1 \equiv E_2$ then.

Hence two DNA expressions are equivalent if they denote exactly the same DNA molecule.

A somewhat weaker version of this relation is

Definition 4.32 Two DNA expressions $E_1$ and $E_2$ are equivalent modulo nicks, if $\nu(S(E_1)) = \nu(S(E_2))$. We write $E_1 \equiv_n E_2$ then.

Intuitively, $E_1$ and $E_2$ are equivalent modulo nicks, if they denote DNA molecules with the same nucleotides at the same positions; the DNA molecules may, however, have nicks at different positions. $E_1$ may have nicks not occurring in $E_2$ and/or the other way round. For example, if $S(E_1) = \left( \begin{array}{c} A \\ T \end{array} \right) \triangledown \left( \begin{array}{c} C \\ G \end{array} \right) \triangledown \left( \begin{array}{c} A \\ T \end{array} \right) \triangledown \left( \begin{array}{c} G \\ C \end{array} \right)$, and $S(E_2) = \left( \begin{array}{c} G \\ C \end{array} \right) \triangledown \left( \begin{array}{c} A \\ T \end{array} \right) \triangledown \left( \begin{array}{c} A \\ T \end{array} \right) \triangledown \left( \begin{array}{c} G \\ C \end{array} \right)$ then $E_1 \equiv_n E_2$.\footnote{Actually, this example is not really appropriate. As we will see in Section 5.1, a DNA expression with the semantics attributed to $E_1$ does not exist. At the level of formal DNA molecules, however, this example is a good illustration of the notion of equivalence modulo nicks.}

We further define a variant of this last relation.

Definition 4.33 A DNA expression $E_1$ is equivalent to a DNA expression $E_2$ pre-modulo nicks, if there are strings $X_1, \ldots, X_r$ with $r \geq 1$ over $A_{\triangledown,\triangle}$ and symbols $c_1, \ldots, c_{r-1} \in \{\triangledown, \triangle\}$ such that $S(E_1) = X_1c_1 \ldots c_{r-1}X_r$ and $S(E_2) = X_1 \ldots X_r$. We write $E_1 \equiv_{\triangledown} E_2$ then.

Hence, if $E_1 \equiv_{\triangledown} E_2$, then $E_1 \equiv_n E_2$ with the restriction that the DNA molecule denoted by $E_2$ does not contain nicks not occurring in the DNA molecule denoted by $E_1$. For example, if $S(E_1) = \left( \begin{array}{c} A \\ T \end{array} \right) \triangledown \left( \begin{array}{c} C \\ G \end{array} \right) \triangledown \left( \begin{array}{c} A \\ T \end{array} \right) \triangledown \left( \begin{array}{c} G \\ C \end{array} \right)$, and $S(E_2) = \left( \begin{array}{c} A \\ T \end{array} \right) \triangledown \left( \begin{array}{c} G \\ C \end{array} \right) \triangledown \left( \begin{array}{c} A \\ T \end{array} \right) \triangledown \left( \begin{array}{c} C \\ G \end{array} \right)$ then $E_1 \equiv_{\triangledown} E_2$. On the other hand, if $S(E_1)$ is as before and $S(E_2) = \left( \begin{array}{c} A \\ T \end{array} \right) \triangledown \left( \begin{array}{c} C \\ G \end{array} \right) \triangledown \left( \begin{array}{c} A \\ T \end{array} \right) \triangledown \left( \begin{array}{c} G \\ C \end{array} \right)$, then $E_1 \equiv_{\triangledown} E_2$.

If $E_1 \equiv_{\triangledown} E_2$, we may also write $E_2 \equiv_{\triangledown} E_1$ and say that $E_2$ is equivalent post-modulo nicks to $E_1$. Thus, the relations $\equiv_{\triangledown}$ and $\equiv_{\triangledown}$ are each other’s inverses: $\triangledown \equiv \equiv_{\triangledown}^{-1}$.

It is easy to verify that each of the binary relations $\equiv$, $\equiv_n$, $\equiv_{\triangledown}$ and $\equiv_{\triangledown}$ is reflexive and transitive. Further, $\equiv$ and $\equiv_n$ are symmetric, so these relations are (indeed) equivalence relations.

The relations $\triangledown \equiv$ and $\equiv_{\triangledown}$ are not symmetric. Hence, in spite of their names, they are no equivalence relations. At first glance, one might think that $\triangledown \equiv$ and $\equiv_{\triangledown}$ are antisymmetric: ‘if a formal DNA molecule $S(E_1)$ has more nicks than another formal DNA molecule $S(E_2)$, then certainly $S(E_2)$ does not have more nicks than $S(E_1)$’. However, if $E_1 \equiv E_2$, then both $E_1 \equiv_{\triangledown} E_2$ and $E_2 \equiv_{\triangledown} E_1$ (and analogously for $\equiv_{\triangledown}$). Since equivalent DNA expressions $E_1$ and $E_2$ are not necessarily the same, $\equiv_{\triangledown}$ and $\equiv_{\triangledown}$ are not antisymmetric. Consequently, they are no partial orders, either. We might, however, say that they are antisymmetric (and thus partial orders) up to equivalence.

It follows immediately from the definition that $E_1 \triangledown \equiv E_2$ implies $E_1 \equiv_n E_2$ and that $E_1 \equiv_{\triangledown} E_2$ implies $E_1 \equiv_{\triangledown} E_2$, so $\equiv_{\triangledown}$ and $\equiv_{\triangledown}$ are refinements of $\equiv_n$. On the other hand, the equivalence relation $\equiv$ is a refinement both of $\triangledown \equiv$ and of $\equiv_{\triangledown}$ (and thus certainly of $\equiv_n$).

We can combine the notions of transitivity and refinement. For example, if $E_1 \equiv E_2$ and $E_2 \equiv E_3$, then $E_1 \equiv E_3$. The following is also clear: if $E_1 \triangledown \equiv E_2$ and $E_1 \equiv_{\triangledown} E_2$ then $E_1 \equiv E_2$. Thus, the equivalence relation $\equiv$ is the intersection of the relations $\triangledown \equiv$ and $\equiv_{\triangledown}$. In other words:
equivalence relation:
reflexive, symmetric, transitive
≡
reflexive, not symmetric,
transitive
▽

Lemma 4.34  The relation ≡ is the largest equivalence relation contained in both \( \equiv \) and \( \equiv \).

On the other hand, we have

Lemma 4.35  The relation \( \equiv \) is the smallest equivalence relation containing both \( \equiv \) and \( \equiv \).

Note that there is indeed a unique smallest equivalence relation containing both \( \equiv \) and \( \equiv \), namely the intersection of all equivalence relations containing \( \equiv \) and \( \equiv \).

Proof: Let \( R_0 \) be the smallest equivalence relation containing both \( \equiv \) and \( \equiv \). We just observed that \( \equiv \) is an equivalence relation containing \( \equiv \) and \( \equiv \). Hence, \( R_0 \) must be a subset of \( \equiv \).

Consider two arbitrary DNA expressions \( E_1 \) and \( E_2 \) such that \( E_1 \equiv E_2 \). By definition, \( \nu(S(E_1)) = \nu(S(E_2)) \), or, in words, \( E_1 \) and \( E_2 \) denote DNA molecules that have the same nucleotides at the same positions, but may have different nicks.

If we let \( E_3 = \langle \downarrow \langle \uparrow E_1 \rangle \rangle \), then \( E_3 \in D \) and \( S(E_3) = \nu^{-}(\nu^{+}(S(E_1))) = \nu(S(E_1)) \) by \((3.7)\). Thus, \( S(E_3) = \nu(S(E_1)) = \nu(S(E_2)) \), or, in words, \( E_3 \) denotes a DNA molecule with the same nucleotides at the same positions as \( E_1 \) and \( E_2 \), but without nicks.

We have \( E_1 \equiv E_3 \) and \( E_3 \equiv E_2 \), implying \( E_1 R_0 E_3 \) and \( E_3 R_0 E_2 \). The transitivity of \( R_0 \) yields that also \( E_1 R_0 E_2 \). Because \( E_1 \) and \( E_2 \) were arbitrary DNA expressions with \( E_1 \equiv E_2 \), the equivalence relation \( \equiv \) must be a subset of \( R_0 \).

Thus, we can conclude that \( R_0 \) is equal to \( \equiv \). \( \square \)

The results of this section are summarized in Figure 4.12.

Note that, because \( \equiv \) and \( \equiv \) are each other’s inverses and symmetry is an inherent property of equivalence relations, every equivalence relation contained in \( \equiv \equiv \) is also contained in \( \equiv \equiv \), and vice versa. Similarly, every equivalence relation containing \( \equiv \equiv \) also contains \( \equiv \equiv \), and vice versa. Therefore, we may rephrase the two lemmas above as follows:

Lemma 4.34  The relation \( \equiv \) is the largest equivalence relation contained in \( \equiv \equiv \).

Lemma 4.35  The relation \( \equiv \) is the smallest equivalence relation containing \( \equiv \equiv \).

Of course, in the rephrased statements, we may as well replace the relation \( \equiv \equiv \) by \( \equiv \equiv \).