Department of Information and Computing Sciences
Utrecht University

Tenth Interstellar Contest on Fuun Programming

Morph Endo!

Task Description

1 Background

Endo is an alien life form, belonging to the species of the Fuun. Endo needs your help! Earth's environmental conditions can be harsh for a life form not properly adapted. Endo had the bad luck of being dropped on Earth by an Interstellar Garbage Collector. Both the life form and its faithful space ship Arrow were severely hurt in the crash, and even worse, after leaving the damaged space craft Endo was hit by a cargo container that was also dropped by the Garbage Collector.

Endo is now in serious trouble. It cannot survive on planet Earth in its present form, and Arrow is running low on power. According to Arrow, who was the one to have contacted us, the only hope for Endo is to change its DNA and thereby adapt it to the conditions of our planet. To this end, Arrow has been able to come up with a form in which Endo will survive. Unfortunately, given its current condition, Arrow lacks the resources for coming up with proper DNA modifications itself.

Your task is therefore to help us find such a DNA sequence within 72 hours. Shortly thereafter, Arrow's power will run out for good, and since only Arrow can perform the genetic modification on Endo, this would mean Endo's definite end.

Admittedly, we are partially responsible for the current state of urgency. It took us a long time before realizing the nature of Arrow's emergency message and decoding the information that was provided to us. We are sorry but nevertheless hope that you, the ICFP programming contest audience, have the proper expertise to solve this problem within the harsh time constraints.

Fortunately, there's also good news. When we weren't yet aware of the complete story of Endo and its struggle for life, we have been working on decoding the specification of Fuun DNA. We now know that Fuun DNA works significantly different from human DNA, and that the process of DNA resequencing can be properly described as an algorithm. The main purpose of this document is to describe this algorithm.

Figure 1: Running DNA

2 Fuun DNA

Fuun DNA can be represented as a finite sequence consisting of four bases, denoted by the four letters `I` (infinine), `C` (continuine), `F` (functorine), and `P` (polymorphine).

On Earth, DNA is a program that encodes a build plan. Execution of such a program produces RNA, which in turn directs building proteins. Proteins are the building blocks for all life forms.

For the Fuun, the situation is similar, but there are a few exceptions:

1. Besides producing RNA, executing Fuun DNA also manipulates the DNA itself. As in humans, Fuun RNA directs protein biosynthesis, resulting in a living being.

2. Fuun DNA can also transform one being into another. This is achieved by using a piece of DNA as a prefix to another piece. So prepending a piece of Fuun DNA to another piece of Fuun DNA may well lead to a living Fuun that looks rather different.

3. Fuun DNA has the special property that it not only affects itself, but also its immediate environment.

4. It is hypothesized that, contrary to life on Earth, the Fuun are a result of *intelligent* design. We believe this because Arrow hints at the fact that there may be "messages" from the creators in the DNA, and that there may be genes already present that could help with the plan to transform Endo.

We know how to simulate both the creation of the build plan and its execution. We lack, however, the ability to assemble Endo ourselves—only Arrow is capable of doing so. Fortunately, Arrow has explained to us a way to simulate the result of applying Fuun DNA visually, as a two-dimensional picture of the life form and its surroundings. We have Endo's DNA, and we have a picture that is the visual representation of Endo's current state.

We have also been provided with a "target" picture which Arrow thinks visualizes an optimal modification of Endo that would allow it to live. While it might not be necessary to match this target completely, we will at least have to come close if we want to improve its chances of survival.

DNA resequencing is a risky business, and the longer the prefix, the higher are the chances that the transformation will fail, or that Endo's mind will be irreparably damaged. We are therefore looking for a DNA prefix that not only matches the target as closely as possible, but also is as short as possible. DNA resequencing also consumes a lot of energy, which given the damaged state of Arrow is a scarce resource as well, and leads to further constraints.

Figure 1 shows the two-phased structure of the DNA transformation process. We describe the two phases as a pipeline of two programs.

The rest of this document provides information on how to

```
1  Base  ::= I | C | F | P
2  DNA   ::= Base*
3  RNA   ::= DNA*
```

Figure 2: DNA data type

4	X^*	type of sequences with elements of type X
5	ε	empty sequence
6	$x \lhd xs$	prepend an element x to a sequence xs
7	$xs \rhd x$	append an element x to a sequence xs
8	$xs \diamond ys$	concatenate two sequences xs and ys
9	$xs[m\,..\,n]$	subsequence of xs, from position m to **before** position n, never fails
10	$xs[m\,..]$	subsequence of xs, starting at m (abbreviation of $xs[m\,..\,\mathsf{len}\ xs]$)
11	$xs[n]$	the nth element (the single element in $xs[n\,..\,(n+1)]$) or ε
12	$\mathsf{len}\ xs$	length of sequence xs

Figure 3: Sequence type

- turn DNA into RNA (program execute, Section 3),
- simulate biosynthesis from RNA by generating an image (program build, Section 4),
- estimate the chances of Endo's survival given a particular DNA prefix, and submit such prefixes to us (Section 5),
- get started on the original DNA string, based on a tiny bit of experimentation we were able to perform ourselves while assembling this description (Section 6).

3 From DNA to RNA

If exposed to the right enzymes, Fuun DNA starts to transform itself and to produce RNA in the process. We call this process *executing* a DNA string. The chemical background is dealt with by Arrow, we focus here on the algorithmic aspects of running DNA. The following is based on descriptions that Arrow has given us. We have tried to understand, analyze, and rewrite the process in a style that should be understandable to programmers on Earth.

As mentioned before, a DNA string is a sequence of bases, defined in Figure 2. There are four possible bases: `I`, `C`, `F`, `P`. We write DNA strings (as opposed to a single base) surrounded by quotes, for instance '`ICFP`'. The resulting RNA is a sequence of DNA strings.

Sequences play a big role in executing DNA. A sequence consists of zero or more elements of a certain type. The notation we use as well as common operations on sequences are explained in Figure 3. Note that none of the operations ever fail and that indices are zero-based. The subsequence operation returns the empty sequence ε whenever the specified range has length 0 or less, or if the range is outside the boundaries of the sequence. If the specified range is only partially within the boundaries of the sequence, then that part

```
13 'ICFP'[0..2] returns 'IC'
14 'ICFP'[2..0] returns ε
15 'ICFP'[2..2] returns ε
16 'ICFP'[2..3] returns 'F'
17 'ICFP'[2]    returns F
18 'ICFP'[2..6] returns 'FP'
19 'ICFP'[2..]  returns 'FP'
20 'ICFP'[6]    returns ε
```

Figure 4: Examples of subsequence operations

```
21 global dna : DNA ← ε
22 global rna : RNA ← ε
23 proc execute () =
24   dna ← read
25   repeat
26     let p ← pattern ()
27     let t ← template ()
28     matchreplace (p, t)
29   end repeat
30 proc finish () =
31   write rna
32   exit
```

Figure 5: Executing DNA

of the sequence is returned. We also use ε as an exceptional return value in the case of the subscript operation. Figure 4 shows a few examples of substring operations on DNA strings.

Let us now look at the overall structure of executing DNA, which is given in Figure 5. 21 The algorithm works with a global DNA sequence (line 21). At the start, the initial DNA is 24 read as input (line 24). This initial DNA consists of a prefix (that is to be provided by you), followed by Endo's DNA that can be downloaded from the contest pages. The rest of the execution process is repeated until an exceptional condition arises (when the DNA string is fully consumed). In each iteration, there are three steps:

26 • decode a prefix of the current DNA into a *pattern* (line 26),

27 • decode a prefix of the remaining DNA into a *template* (line 27),

28 • match the pattern against the remaining DNA (line 28) and perform the match on it using the template, possibly generating new DNA.

In the following, we consider each of the steps in detail.

Sooner or later, during one of the iterations, the procedure finish is called. This proce- 31 dure outputs the final contents of *rna* (line 31) and terminates the program (line 32); the 32

```
33  Pattern ::= PItem*
34  PItem   ::= Base
35            | !ℕ
36            | ?DNA
37            | (
38            | )
```

Figure 6: Patterns

produced RNA string is then fed to the build program described in Section 4.

3.1 Decoding patterns

33 Figure 6 describes the syntax of patterns. A pattern is a sequence of pattern items (line 33). There are five different sorts of pattern items:

34 • an item for a single base, written I, C, F, or P (line 34),

35 • an item to skip n bases, written $!_n$ (line 35),

36 • an item to search for a DNA sequence s, written $?_s$ (line 36),

37 • two items to open and close a *group*, written **(** and **)**, respectively (lines 37 and 38).

38

Recognition of patterns is performed by function pattern, which is presented in Figure 7. The function pattern traverses the global *dna* and consumes part of it, building up a pat-
40 tern in the local variable p (line 40) along the way. The current grouping level (a natural
41 number) is stored in the local variable *lvl* (line 41). After initializing the variables, a loop is entered. In each iteration, we branch depending on the current prefix of *dna*. Only one branch is executed per iteration!

44–47 The first four cases produce constant items (lines 44–47).

48 The fifth case produces a 'skip' item (line 48). After consuming the initial 'IP', the function nat is called to decode a natural number n from the following piece of DNA. The function nat is described in Section 3.2. The natural number n becomes the parameter of the 'skip' item.

49 The sixth case produces a 'search' item (line 49), which is parameterized by a sequence of bases. Therefore, after consuming the initial 'IF' **and one additional base**, the function consts (described in Section 3.2) is called to scan for a sequence s of encoded bases in the DNA string, which then becomes the parameter of the generated item.

50 The seventh case produces an 'open' item (line 50) and at the same time increments the *lvl* variable.

51 The eighth case matches if the prefix of *dna* is 'IIC' or 'IIF' (line 51). In both cases, the three-letter prefix is removed. If the current *lvl* is 0, pattern recognition ends at this point
53 and the function returns the pattern accumulated thus far in p (line 53). If *lvl* is a positive
54 number, it is decremented and a 'close' item is produced (line 54).

```
39 function pattern () : Pattern =
40   let p   : Pattern ← ε;
41   let lvl : ℕ       ← 0;
42   repeat
43     case dna starts with
44       'C'              ⇒ dna ← dna[1..]; p ← p ▷ I
45       'F'              ⇒ dna ← dna[1..]; p ← p ▷ C
46       'P'              ⇒ dna ← dna[1..]; p ← p ▷ F
47       'IC'             ⇒ dna ← dna[2..]; p ← p ▷ P
48       'IP'             ⇒ dna ← dna[2..]; let n ← nat (); p ← p ▷ !n
49       'IF'             ⇒ dna ← dna[3..]; let s ← consts (); p ← p ▷ ?s      three bases consumed!
50       'IIP'            ⇒ dna ← dna[3..]; lvl ← lvl + 1; p ← p ▷ (
51       'IIC' or 'IIF'   ⇒ dna ← dna[3..]
52                           if   lvl = 0
53                           then return p
54                           else lvl ← lvl − 1; p ← p ▷ )
55                           end if
56       'III'            ⇒ rna ← rna ▷ dna[3..10]; dna ← dna[10..]
57       anything else    ⇒ finish ()
58     end case
59   end repeat
```

Figure 7: Pattern recognition

```
60 dna ← 'CIIC';                  pattern () returns 'I'
61 dna ← 'IIPIPICPIICICIIF'; pattern () returns '(!2)P'
```

Figure 8: Pattern recognition examples

If the first three bases in *dna* are 'III', then the next seven bases form an RNA command and are added to the output sequence *rna*. All ten bases (the three I's plus the RNA command) are consumed (line 56). The RNA commands are interpreted in the next phase of the DNA transformation process, described in Section 4.

If none of the given prefixes match, we are close to the end of the DNA string, and the final case is taken, which calls finish to print the RNA and terminate the program (line 57).

The specification of pattern recognition implies that the recognition process either ends successfully by encountering 'IIC' or 'IIF' at the beginning of an iteration when the current *lvl* is 0, or by termination of the program if none of the given prefixes match. The parentheses in patterns returned by the function pattern are always balanced.

Figure 8 presents two example calls of the pattern function. The second makes use of the natural number encoding that is described next.

```
62 function nat () : ℕ =
63   case dna starts with
64     'P'                          ⇒ dna ← dna[1..]; return 0
65     'I' or 'F'                   ⇒ dna ← dna[1..]; let n ← nat (); return 2 ∗ n
66     'C'                          ⇒ dna ← dna[1..]; let n ← nat (); return 2 ∗ n + 1
67     nothing (i.e., is empty)     ⇒ finish ()
68   end case
69 function consts () : DNA =
70   case dna starts with
71     'C'                          ⇒ dna ← dna[1..]; let s ← consts (); return I ◁ s
72     'F'                          ⇒ dna ← dna[1..]; let s ← consts (); return C ◁ s
73     'P'                          ⇒ dna ← dna[1..]; let s ← consts (); return F ◁ s
74     'IC'                         ⇒ dna ← dna[2..]; let s ← consts (); return P ◁ s
75     anything else                ⇒ return ε
76   end case
```

Figure 9: Helper functions for decoding DNA

```
77 Template ::= TItem*
78 TItem   ::= Base
79           | ℕ_ℕ
80           | |ℕ|
```

Figure 10: Templates

3.2 Helper functions

In this section, we describe the functions nat and consts which are used during pattern and template recognition. Both are specified in Figure 9 and are very similar in structure.

The function nat decodes a natural number from the global variable *dna*. Everything up
64 to the following P is read (line 64). The bases I and F are interpreted as 0 (line 65), a C as a 1
65 (line 66), with the most significant bit being last. If the end of the DNA string is encountered
66 before the next P, finish is called to print the RNA and exit the program (line 67).
67

The function consts decodes a sequence of bases (i.e., a DNA string). If *dna* starts with one of the four prefixes 'C', 'F', 'P' or 'IC', the prefix is consumed, a single base is produced, and
71–74 scanning continues via a recursive call to consts (lines 71–74). If none of the four prefixes
75 match, the function returns (lines 75).

3.3 Decoding templates

The phase that decodes a template from the DNA string proceeds in almost the same way as the pattern recognition phase. The syntax of templates is described in Figure 10. A template is a sequence of template items, and a template item can be one of three things:

78 • an item for a constant base (line 78), denoted like the corresponding pattern item as I, C, F, or P,

```
81  function template () : Template =
82    let t : Template ← ε;
83    repeat
84      case dna starts with
85        'C'              ⇒ dna ← dna[1..]; t ← t ▷ I
86        'F'              ⇒ dna ← dna[1..]; t ← t ▷ C
87        'P'              ⇒ dna ← dna[1..]; t ← t ▷ F
88        'IC'             ⇒ dna ← dna[2..]; t ← t ▷ P
89        'IF' or 'IP'     ⇒ dna ← dna[2..]; let l ← nat (); let n ← nat (); t ← t ▷ n_l
90        'IIC' or 'IIF'   ⇒ dna ← dna[3..]; return t
91        'IIP'            ⇒ dna ← dna[3..]; let n ← nat (); t ← t ▷ |n|
92        'III'            ⇒ rna ← rna ▷ dna[3..10]; dna ← dna[10..]
93        anything else ⇒ finish ()
94      end case
95    end repeat
```

Figure 11: Template recognition

79 • an item for reference number n with protection level l (line 79), written n_l,

80 • or an item encoding the length of reference n (line 80), written as $|n|$.

Template recognition is performed by function template in Figure 11. The structure of the function is similar to pattern, but a bit simpler, because there is only one local variable, the 82 template t accumulated so far (line 82). There is no grouping in templates, and therefore no need to keep track of a current grouping level.

After the initialization of t, the rest of template recognition is a loop. In each operation, the current contents of *dna* are analyzed. Depending on the prefix, one of the branches is 85–88 selected. The cases for the constant items (lines 85–88) as well as the cases for RNA generation (line 92) and the catch-all case (line 93) are completely analogous to the corresponding 92 93 cases in the function pattern (see Section 3.1).

On an 'IF' or an 'IP' prefix, a 'reference' item is generated: After consuming the prefix, two natural numbers l and n are decoded from the DNA string, using the function nat from 89 Section 3.2. The item generated is then n_l (line 89).

The prefixes 'IIC' or 'IIF' mark the end of a template. They are consumed, and the 90 current contents of t are returned (line 90).

The prefix 'IIP' causes the production of a 'length' item $|n|$, where n is a natural number 91 decoded from *dna* after removal of the three-base prefix (line 91).

3.4 Matching

The procedure matchreplace is shown in Figure 12. The procedure takes two arguments: the pattern *pat* and the template t that have been decoded (and removed) from the DNA string *dna*. In matchreplace, we traverse the pattern and the *dna* string, trying to match the contents of *dna* to the pattern items in the pattern. While each of the pattern items is 98 traversed in order, we keep the current position in the DNA string in a variable i (line 98).

```
96  Environment ::= DNA*
97  proc matchreplace (pat : Pattern, t : Template) =
98    let i : ℕ           ← 0
99    let e : Environment ← ε
100   let c : ℕ*          ← ε
101   foreach p ∈ pat
102     case p is of the form
103       b  ⇒ if    dna[i] = b
104            then i ← i + 1
105            else return
106            end if
107       !n ⇒ i ← i + n
108            if i > len (dna) then return end if
109       ?s ⇒ if    there is a smallest n : ℕ such that n ⩾ i and s is a postfix of dna[i..n]
110            then i ← n
111            else return
112            end if
113       (  ⇒ c ← i ◁ c
114       )  ⇒ e ← e ▷ dna[c[0]..i]; c ← c[1..]
115     end case
116   end foreach
117   dna ← dna[i..]
118   replace (t, e)
119   return
```

Figure 12: Pattern matching

99 During the matching process, we build up an environment e (line 99). An environment is a sequence of DNA strings, where each string represents the piece of DNA that has been matched against a group (everything between a corresponding 'open' and 'close' item). During the traversal of the pattern, we may be in positions where we have encountered 'open' items, but not yet the corresponding 'close' items. We use the variable c to store the
100 indices of the positions in DNA for each of the unmatched 'open' items (line 100).

After initializing the local variables, each of the pattern items p in pat is considered in
101 order (line 101). A pattern for a constant base b matches if there is a b at the current position
103 i in the DNA string (line 103). If yes, i is incremented. If not (and also if position i is beyond the end of the DNA string), the match fails, and we return from the procedure without modifying dna.

107 On a 'skip' item, the current position is adjusted accordingly (line 107). If the new posi-
108 tion is beyond the end of the DNA string, the match fails (line 108).

On a 'search' item, we look for the DNA string s in the DNA string dna. We only look in the part that starts at position i, and we look for the first occurrence. If the search succeeds
109–110 and n is the first position after the end of s, we set i to n (lines 109–110). If s is not contained
111 in $dna[i..]$, the match fails (line 111).

On an 'open' item, we prepend the current position i to the stored opening positions c
113 (line 113).

```
120 proc replace (tpl : Template, e : Environment) =
121   let r : DNA ← ε
122   foreach t ∈ tpl
123     case t is of the form
124       b   ⇒ r ← r ▷ b
125       n_l ⇒ r ← r ◇ protect (l, e[n])
126       |n| ⇒ r ← r ◇ asnat (len (e[n]))
127     end case
128   end foreach
129   dna ← r ◇ dna
130   return
```

Figure 13: Replacement

On a 'close' item, we read the first number in c to get the stored position of the corresponding 'open' item. The range between that position and the current position i is the part of the item that is matched to the current group, and we therefore append it to the environment. Finally, the first item of c is removed, because this group is now closed (line 114).

If we reach the end of the pattern, the match was successful. In this case, we remove everything up to the current position i from dna (line 117), and replace it with a DNA string generated from the template t and the accumulated environment e, using the procedure replace (line 118).

3.5 Replace

We now consider the replace operation as shown in Figure 13. It takes two arguments: a template tpl (as previously decoded from the DNA string), and an environment e (the result of the successful matching process). During the operation, we traverse the template item by item (line 122) and incrementally build a replacement piece r of DNA (line 121), which in the end is prepended to the global DNA string dna (line 129).

For a constant item $\underline{b}$, we add the appropriate base b to the replacement string (line 124).

For a 'reference' item n_l, we look up the nth element of the environment. Note that this lookup results in ε if n is greater than the length of e. We quote the resulting string l times using the function protect (see Section 3.6) before we append it to r (line 125).

For a 'length' item $|n|$, we also look up the nth element of the environment, and compute its length. Since the lookup returns ε if n is out of range, the length will be 0 for such values of n. We then encode the natural number as a DNA string via function asnat (see Section 3.7) and add it to the replacement (line 126).

3.6 Protection

The function protect takes two arguments, a level l and a piece of DNA d. It repeatedly applies quote to d, namely l times, and returns the resulting string.

The function quote replaces each `I` by a `C`, each `C` by an `F`, each `F` by a `P`, and each `P` by the sequence '`IC`'.

```
function protect (l : ℕ, d : DNA) : DNA =
  if    l = 0
  then return d
  else  return protect (l − 1, quote (d))
  end if
function quote (d : DNA) : DNA =
  case d starts with
    I              ⇒ return C    ◁ quote (d[1..])
    C              ⇒ return F    ◁ quote (d[1..])
    F              ⇒ return P    ◁ quote (d[1..])
    P              ⇒ return 'IC' ◇ quote (d[1..])
    anything else ⇒ return ε
  end case
```

Figure 14: Protection

```
function asnat (n : ℕ) : DNA =
  case n is
    0              ⇒ return 'P'
    positive even ⇒ return I ◁ asnat ⌊n/2⌋
    positive odd  ⇒ return C ◁ asnat ⌊n/2⌋
  end case
```

Figure 15: Encoding natural numbers

```
'IIPIPICPIICICIIFICCIFPPIICCFPC'      turns into 'PICFC'
'IIPIPICPIICICIIFICCIFCCCPPIICCFPC' turns into 'PIICCFCFFPC'
'IIPIPIICPIICIICCIICFCFC'             turns into 'I'
```

Figure 16: Full iteration examples

3.7 Encoding natural numbers

The function asnat is almost the inverse of function nat. It takes a number n and produces a DNA string encoding that number. In binary representation with the most significant bit last, each 0 is replaced by I, each 1 is replaced by C, and the end is marked with P.

3.8 Example

To conclude the description of execute, Figure 16 shows a few examples of how the *dna* variable changes during a single full iteration, i.e., the decoding of a pattern, a template, and the subsequent execution of matchreplace.

4 From RNA to a Fuun

As in humans, Fuun RNA directs protein biosynthesis, resulting in a living being. We lack the ability to assemble living beings ourselves – only Arrow, the spaceship, can do that – but fortunately, we can come up with a picture of the resulting life form and its surroundings. This section describes what we found out about how Fuun RNA can be displayed.

Recall from Figure 2 that Fuun RNA is a long list of commands, where each command is a seven-base long DNA string – except the very last RNA command, which might be shorter than seven bases. While building a Fuun from RNA, each RNA command is processed, possibly updating the internal state. The resulting image is part of that state.

The data types for the image are described in Figure 17. A coordinate (*Coord*) is a natural number in the range from 0 to 599. A pair of coordinates forms a position (*Pos*).

A *Component* is a natural number in the range from 0 to 255. A triple of components forms an RGB value (*RGB*). A *Transparency* (or alpha) value is a single component. A number of colors are predefined. In particular, black corresponds to $(0, 0, 0)$, white to $(255, 255, 255)$, the transparency value 0 means fully transparent, whereas 255 means fully opaque.

A *Bitmap* is an array indexed by positions, i.e., from $(0, 0)$ up to $(599, 599)$, where every element is a *Pixel*: a pair of an *RGB* and a *Transparency*. The position $(0, 0)$ denotes the upper-left, the position $(599, 599)$ the lower-right corner of the image. To refer to the pixel at position p in the bitmap b, we write $b@p$.

The Fuun assembly process will also make use of a *Bucket*, a sequence of *Color*s where each entry can either be an *RGB* value (denoted by the postfix **rgb**) or a transparency value (denoted by the postfix $\boldsymbol{\alpha}$).

Furthermore, we require a data type of directions (*Dir*), which comprises the four values north (**N**), east (**E**), south (**S**), and west (**W**).

4.1 State

While interpreting RNA, a significant amount of internal state is utilized. Each component of the state is kept in a global variable. The components and their initial values are shown in Figure 18.

There is a sequence of colors in the variable *bucket*, initially empty. Section 4.3 describes how the bucket works.

The variable *position* contains the position that RNA is currently working on. The initial position is the upper-left corner $(0, 0)$. There is also a saved position called *mark*, which is also initially $(0, 0)$.

There is a current direction *dir*, initially east (*E*), which indicates where the focus of the build process is likely to switch next.

The position, mark, and direction are used by the move and draw RNA commands, described in Sections 4.4 and 4.5, respectively.

Finally, there is a sequence of bitmaps in the variable *bitmaps*. Initially, it contains a single element, a fully transparent bitmap, i.e., $transparentBitmap@p = (black, transparent)$ for all valid positions p. Section 4.6 describes how the sequence of bitmaps is used. Note that *bitmaps* is never empty, and that after processing all the RNA commands, the first element of *bitmaps* contains the resulting image.

```
153 Coord       ::= a natural number n with 0 ⩽ n < 600
154 Pos         ::= Coord × Coord
155 Component   ::= a natural number n with 0 ⩽ n < 256
156 RGB         ::= Component × Component × Component
157 Transparency::= Component
158 Pixel       ::= RGB × Transparency
159 Bitmap      ::= an array from (0,0) to (599,599) with elements of type Pixel
160 Color       ::= RGB rgb
161               | Transparency α
162 Bucket      ::= Color*
163 Dir         ::= N | E | S | W
164 global black     : RGB ← (0   ,0   ,0   )
165 global red       : RGB ← (255,0   ,0   )
166 global green     : RGB ← (0   ,255,0   )
167 global yellow    : RGB ← (255,255,0   )
168 global blue      : RGB ← (0   ,0   ,255)
169 global magenta   : RGB ← (255,0   ,255)
170 global cyan      : RGB ← (0   ,255,255)
171 global white     : RGB ← (255,255,255)
172 global transparent : Transparency ← 0
173 global opaque      : Transparency ← 255
```

Figure 17: Coordinates, colors, bitmaps

```
174 global bucket   : Bucket   ← ε
175 global position : Pos      ← (0,0)
176 global mark     : Pos      ← (0,0)
177 global dir      : Dir      ← E
178 global bitmaps  : Bitmap*  ← transparentBitmap ◁ ε
```

Figure 18: Initial state for Fuun assembly

4.2 Processing RNA

The main function of the RNA processing phase is shown in Figure 19. The structure of build is simple. First, the input *rna* is read. The global variable *rna* then contains the RNA instructions that have been produced by the execute program described in the previous section (Section 3).

The sequence of RNA instructions is traversed from left to right. We look at each instruction *r* in turn. Arrow has provided us with information about 20 RNA codes that seem to be essential for Fuun assembly. If *r* is one of the 20 known RNA codes, we perform some action that modifies the state. If it is another, unknown, code, we ignore it and continue with the next.

In the very end, we look at the first bitmap in the sequence *bitmaps*, and draw it. The

```
179 proc build () =
180   rna ← read
181   foreach r ∈ rna
182     case r is of the form
183       'PIPIIIC'      ⇒ addColor (black rgb)
184       'PIPIIIP'      ⇒ addColor (red rgb)
185       'PIPIICC'      ⇒ addColor (green rgb)
186       'PIPIICF'      ⇒ addColor (yellow rgb)
187       'PIPIICP'      ⇒ addColor (blue rgb)
188       'PIPIIFC'      ⇒ addColor (magenta rgb)
189       'PIPIIFF'      ⇒ addColor (cyan rgb)
190       'PIPIIPC'      ⇒ addColor (white rgb)
191       'PIPIIPF'      ⇒ addColor (transparent α)
192       'PIPIIPP'      ⇒ addColor (opaque α)
193       'PIIPICP'      ⇒ bucket ← ε
194       'PIIIIIP'      ⇒ position ← move (position, dir)
195       'PCCCCCP'      ⇒ dir ← turnCounterClockwise (dir)
196       'PFFFFFP'      ⇒ dir ← turnClockwise (dir)
197       'PCCIFFP'      ⇒ mark ← position
198       'PFFICCP'      ⇒ line (position, mark)
199       'PIIPIIP'      ⇒ tryfill ()
200       'PCCPFFP'      ⇒ addBitmap (transparentBitmap)
201       'PFFPCCP'      ⇒ compose ()
202       'PFFICCF'      ⇒ clip ()
203       anything else ⇒ do nothing
204     end case
205   end foreach
206   draw bitmaps[0]                                  all alpha values are set to 255!
207   exit
```

Figure 19: Building a Fuun from RNA

resulting image is determined by the RGB values of $bitmaps[0]$. The transparency values of $bitmaps[0]$ are ignored and all set to opaque (255) for drawing.

The known RNA commands can be sorted into four groups: commands that affect the bucket, commands that change the focus, commands that draw, and commands that affect the sequence of bitmaps. Each of the command groups is discussed in detail below.

4.3 Bucket commands

For each of the eight predefined colors and the two predefined transparency values, there is an RNA command that prepends the *Color* to the bucket, using the procedure addColor in Figure 20. The instruction 'PIIPICP' empties the bucket.

The bucket encodes information about a pixel, i.e., a current color and transparency value. The function currentPixel, also in Figure 20, can be used to determine this value of type *Pixel*.

```
proc addColor (c : Color) =
  bucket ← c ◁ bucket
  return
function currentPixel () : Pixel =
  let r_c ← average ({r | (r,g,b) rgb ∈ bucket}, 0)
  let g_c ← average ({g | (r,g,b) rgb ∈ bucket}, 0)
  let b_c ← average ({b | (r,g,b) rgb ∈ bucket}, 0)
  let a_c ← average ({a | a α        ∈ bucket}, 255)
  return ((⌊r_c ∗ a_c/255⌋, ⌊g_c ∗ a_c/255⌋, ⌊b_c ∗ a_c/255⌋), a_c)
function average (values : Component*, default : Component) : Component =
  if values = ε then
    return default
  else
    return ⌊sum (values)/len (values)⌋
  end if
```

Figure 20: The color bucket

```
let b ← black rgb; let r ← red rgb; let m ← magenta rgb; let w ← white rgb
let y ← yellow rgb; let c ← cyan rgb; let t ← transparent α; let o ← opaque α
bucket ← t ◁ o ◁ o ◁ ε; currentPixel () returns ((0,0,0), 170)
bucket ← b ◁ y ◁ c ◁ ε; currentPixel () returns ((85,170,85), 255)
bucket ← y ◁ t ◁ o ◁ ε; currentPixel () returns ((127,127,0), 127)
bucket ← b ◁ b ◁ b ◁ b ◁ b ◁ b ◁ b ◁ b ◁ b ◁ b ◁ b ◁ b ◁ b ◁ b ◁ b ◁ b ◁ b ◁ b     18 blacks
      ◁ r ◁ r ◁ r ◁ r ◁ r ◁ r ◁ r                                                    7 reds
      ◁ m ◁ m ◁ m ◁ m ◁ m ◁ m ◁ m ◁ m ◁ m ◁ m ◁ m ◁ m ◁ m
      ◁ m ◁ m ◁ m ◁ m ◁ m ◁ m ◁ m ◁ m ◁ m ◁ m ◁ m ◁ m ◁ m
      ◁ m ◁ m ◁ m ◁ m ◁ m ◁ m ◁ m ◁ m ◁ m ◁ m ◁ m ◁ m ◁ m                         39 magentas
      ◁ w ◁ w ◁ w ◁ w ◁ w ◁ w ◁ w ◁ w ◁ w ◁ w                                       10 whites
      ◁ o ◁ o ◁ o                                                                  3 opaques
      ◁ t                                                                          1 transparent
      ◁ ε;          currentPixel () returns ((143,25,125), 191)
```

Figure 21: Examples for currentPixel

The current alpha a_c is the average of all transparency values in the bucket, assuming an opaque pixel (255) if there are no transparency values in the bucket. Each of the color components is the average of the values of that component currently in the bucket, normalized by the current transparency. Note that currentPixel will return $(black, opaque)$ for an empty bucket.

The function average is used by currentPixel to compute the average of a sequence of *Component*s. For an empty sequence, the given *default* is used. Otherwise, the average is the sum of the values divided by the number of components, rounded down.

Figure 21 shows four example calls to currentPixel, given different contents of the bucket.

```
function move ((x, y) : Pos, d : Dir) : Pos =
  case d is of the form
    N ⇒ return (x              , (y − 1) mod 600)
    E ⇒ return ((x + 1) mod 600, y               )
    S ⇒ return (x              , (y + 1) mod 600)
    W ⇒ return ((x − 1) mod 600, y               )
  end case
function turnCounterClockwise (d : Dir) : Dir =
  case d is of the form
    N ⇒ return W
    E ⇒ return N
    S ⇒ return E
    W ⇒ return S
  end case
function turnClockwise (d : Dir) : Dir =
  case d is of the form
    N ⇒ return E
    E ⇒ return S
    S ⇒ return W
    W ⇒ return N
  end case
```

Figure 22: Moving the focus

4.4 Move commands

The instruction 'PIIIIIP' changes *position* depending on the current value of *dir*. Given the current position and the direction, function move in Figure 22 returns the new position. For example, if the direction is east, and the current position is $(324, 210)$, the new position is $(325, 210)$. If the new position is invalid (beyond an edge of the grid), we take the position on the opposite edge. For example, moving from $(100, 0)$ in the direction north yields position $(100, 599)$.

The RNA code 'PCCCCCP' changes the direction counter-clockwise, using the function turnCounterClockwise. For example, if the current movement direction is south, the new direction is east.

The RNA code 'PFFFFFP' changes the direction clockwise, and the function turnClockwise is the inverse of turnCounterClockwise. For example, if the current movement direction is south, the new direction is west.

4.5 Draw commands

All drawing commands affect the first element of the *bitmaps* sequence, via the two operations getPixel and setPixel that are listed in Figure 23.

The function getPixel takes a position p and returns the *Pixel* (i.e., the color and transparency) of the first element of *bitmaps* at position p.

The procedure setPixel also takes a position p, and it modifies position p in the first el-

```
258 function getPixel (p : Pos) : Pixel =
259   return (bitmaps[0])@p
260 proc setPixel (p : Pos) =
261   (bitmaps[0])@p ← currentPixel ()
262   return
```

Figure 23: Pixel operations

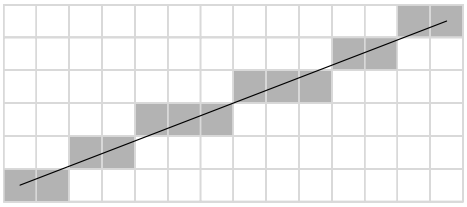

Figure 24: Example approximation of a line in a raster

ement of *bitmaps*. The new value is given by function currentPixel (see Figure 20), which means that the color and transparency of the pixel is determined by the current contents of the bucket.

The RNA command 'PCCIFFP' saves the current value of *position* in the variable *mark*. There is only a single marked position at a time, so the previous contents of *mark* are discarded. The contents of *mark* are used in the 'PFFICCP' command.

The command 'PFFICCP' draws a line (in the first bitmap in *bitmaps*) from the current position to the marked position. The line is drawn using the current color and transparency in the bucket.

The function line shown in Figure 25 specifies an algorithm that determines which points in a raster should be plotted in order to form a close approximation to a straight line between to given points. An example illustration of the algorithm is in Figure 24. The variables *deltax* and *deltay* are integer variables (i.e., they can be negative). The function abs returns the absolute value of an integer, and max computes the maximum value of two numbers. Note that setPixel only takes a *Pos* argument, because it uses the current color and transparency from the bucket.

The fill command 'PIIPIIP' calls the procedure tryfill, given in Figure 26. The procedure tries to fill a connected area surrounding the current position with the current color and transparency (as determined by getPixel), but only if the pixel at the current position does not already have this color and transparency.

A flood fill algorithm fill is used to determine the fill area and change the pixels. The algorithm is also specified in Figure 26. The procedure fill takes a position and a pixel *initial*. If the pixel at the current position is equal to *initial*, we change the color and transparency at this position to the value determined by the bucket (using setPixel) and recursively call fill on the horizontally and vertically adjacent positions.

If a fill command is encountered, the procedure fill is called at the current value of *position* with the color and transparency at that position (the result of getPixel).

```
263 proc line ((x0, y0) : Pos, (x1, y1) : Pos) =
264   let deltax : ℤ ← x1 − x0
265   let deltay : ℤ ← y1 − y0
266   let d : ℕ ← max (abs (deltax), abs (deltay))
267   if deltax ∗ deltay ⩽ 0 then let c ← 1 else let c ← 0 end if
268   let x ← x0 ∗ d + ⌊(d − c)/2⌋
269   let y ← y0 ∗ d + ⌊(d − c)/2⌋
270   repeat d times
271     setPixel (⌊x/d⌋, ⌊y/d⌋)
272     x ← x + deltax
273     y ← y + deltay
274   end repeat
275   setPixel (x1, y1)
276   return
```

Figure 25: Drawing a line

```
277 proc tryfill () =
278   let new ← currentPixel ()
279   let old ← getPixel (position)
280   if new ≠ old then fill (position, old) end if
281   return
282 proc fill ((x, y) : Pos, initial : Pixel) =
283   if getPixel (x, y) = initial then
284     setPixel (x, y)
285     if x > 0   then fill ((x − 1, y    ), initial) end if
286     if x < 599 then fill ((x + 1, y    ), initial) end if
287     if y > 0   then fill ((x    , y − 1), initial) end if
288     if y < 599 then fill ((x    , y + 1), initial) end if
289   end if
290   return
```

Figure 26: Filling an area

Unlike the move function, fill does not continue on the opposite side when it reaches the edge of the bitmap.

4.6 Bitmap commands

The RNA instruction 'PCCPFFP' creates a new bitmap in *bitmaps*. The new bitmap is always *transparentBitmap*, the bitmap that is (*black*, *transparent*) at every valid position. The bitmap is added to the front of the sequence using addBitmap in Figure 27. A new bitmap is only added if there are not already 10 bitmaps in the sequence. If there are 10 bitmaps already, the command is ignored.

The instruction 'PFFPCCP' composes the first two elements of *bitmaps*, using the proce-

```
proc addBitmap (b : Bitmap) =
  if len (bitmaps) < 10 then
    bitmaps ← b ◁ bitmaps
  end if
  return
proc compose () =
  if len (bitmaps) ⩾ 2 then
    foreach p ∈ Pos
      let ((r0, g0, b0), a0) ← (bitmaps[0])@p
      let ((r1, g1, b1), a1) ← (bitmaps[1])@p
      (bitmaps[1])@p ← ((r0 + ⌊r1 ∗ (255 − a0)/255⌋,
                         g0 + ⌊g1 ∗ (255 − a0)/255⌋,
                         b0 + ⌊b1 ∗ (255 − a0)/255⌋),
                         a0 + ⌊a1 ∗ (255 − a0)/255⌋)
    end foreach
    bitmaps ← bitmaps[1..]
  end if
  return
proc clip () =
  if len (bitmaps) ⩾ 2 then
    foreach p ∈ Pos
      let ((r0, g0, b0), a0) ← (bitmaps[0])@p
      let ((r1, g1, b1), a1) ← (bitmaps[1])@p
      (bitmaps[1])@p ← ((⌊r1 ∗ a0/255⌋, ⌊g1 ∗ a0/255⌋, ⌊b1 ∗ a0/255⌋), ⌊a1 ∗ a0/255⌋)
    end foreach
    bitmaps ← bitmaps[1..]
  end if
  return
```

Figure 27: Bitmap commands

dure compose in Figure 27, replacing the two images by one. If there are fewer than two bitmaps, the command is ignored. Otherwise, the first two bitmaps are composed pixel by pixel: at every position, the new pixel is determined by the pixels of the two original images at that position. Using this RNA instruction, Fuun images can be combined with background using alpha blending, a technique that humans only learned about 25 years ago. Note that if $bitmaps[0]$ is completely opaque (i.e., a_0 is 255 at every position), then the resulting image is equal to $bitmaps[0]$.

The instruction '`PFFICCF`' composes the first two elements of *bitmaps* in an alternative way, using procedure clip in Figure 27. Again, this replaces the first two elements by the resulting image, and again, the command is ignored if the length of *bitmaps* is less than 2. Of $bitmaps[0]$, only the transparency values matter for the resulting image. If $bitmaps[0]$ is completely opaque (i.e., a_0 is 255 at every position), then the resulting image is equal to $bitmaps[1]$. If $bitmaps[0]$ is completely transparent (i.e., a_0 is 0 at every position), then the resulting image is equal to *transparentBitmap*.

5 How to make Endo live

Your task is to construct a DNA prefix that makes Endo live. Endo's original DNA constructs the "source" image. Your prefix followed by the original DNA should match the "target" image as closely as possible. In addition, your prefix should not be too long, and the chemical process of modifying Endo with your prefix should not consume too much energy. In this section, we explain how to evaluate your solution.

5.1 Risk of a prefix

Each prefix can be assigned a *risk*. The *risk* is a natural number, and the lower the *risk*, the more chances of survival Endo has. The *risk* is defined to be

$$risk = 10 \cdot \text{number of incorrect pixels} + \text{length of the prefix}$$

Here, the number of incorrect pixels is the number of positions where the RGB value of the
206 picture printed in line 206 does not exactly match the RGB values of the provided target picture. Note again that the transparency values of $bitmaps[0]$ at the time of the **draw** command are set to 255, and that $(0, 0)$ is the upper-left and $(599, 599)$ is the lower-right corner. The length of the prefix is the number of bases that constitute the prefix.

5.2 Resource limitations

In addition to the *risk* value, we are subject to certain resource limitations. Arrow must perform the final process, and Arrow has very little energy left, so we must make sure that the submitted prefix does not cause Arrow to run out of energy before it can save Endo! There are two limitations:

- The length of the DNA string (len (dna)) must not exceed 25 million bases at any point of the execution process.
- Inspecting and quoting DNA costs energy; during the execution process, we therefore maintain a *cost* counter that must not exceed 3 billion ($3 \cdot 10^9$):
  - in pattern or template, each base consumed has a cost of 1;
  - during matchreplace,
    - a comparison of the current base with a constant base pattern has a cost of 1;
    - 109 a succesful search has a cost of $n - i$, evaluated after line 109;
    - 109 a failed search has a cost of len $(dna) - i$, evaluated after line 109;
    - however, **skips are cost-free**;
  - for a call to protect with a level of at least 1, the cost is equal to the length of the returned DNA string – however, a call to protect with level 0 is cost-free.

If execution and building of your prefix plus Endo's DNA stays within the above limits, Arrow will be able to apply your prefix. If not, your prefix may be rejected by Arrow because it is too energy-consuming. The prefix with the lowest risk received before the end of the contest and that is not rejected by Arrow will be used to hopefully save Endo. The team that submitted the prefix wins.

5.3 Submission procedure

That's it. We've come a long way figuring out how Fuun DNA works. Now it's up to you. However much we are interested in what clever methods you will develop in your attempts to find a life-saving DNA prefix, for the moment our only concern is Endo's survival. Therefore, please submit only *prefixes* to us, rather than complete programs et cetera.

We will automatically append Endo's DNA to the prefix and run it through Arrow. Prefixes can be uploaded by registered teams during the entire contest. Each team can submit many times. On submission, the risk score of your prefix will be determined and, after a short delay, made available. At the end of the contest, we will only take into a account, for each team, the DNA prefix with the lowest risk value.

During the contest, intermediate standings may be made available by us on a scoreboard. We will not pass on any further information about your submissions to other contestants. In particular, no other teams will see the pictures that result from your submitted prefixes.

Valid submissions are ASCII text files containing only the (uppercase) characters `I`, `C`, `F`, and `P`. If your submitted file contains any other characters or even whitespace, there is no guarantee that it will be accepted and correctly processed by Arrow. The file containing the prefix should be called `prefix.dna` and be submitted in a zip archive containing just this one file.

6 A piece of advice

Let us express our sincerest gratitude that you are trying to help Endo. During the time we have prepared this description and the infrastructure to organize this call for help, we unfortunately had only little time to analyze Endo's DNA ourselves, but we nevertheless have discovered a few things that we want to share with you.

First, be warned that the process of synthesizing Endo from the DNA string is complex and even the simulation of the process requires a somewhat efficient implementation. To give you an idea, calling execute on Endo's DNA performs 1891886 iterations, produces 302450 RNA commands, and has a cost of 192646205. It seems particularly important to ensure that performing skips and appending unquoted references perform better than in linear time.

Second, we noticed that something curious happens if the following prefix is used:

```
IIPIFFCPICICIICPIICIPPPICIIC
```

Good luck and happy morphing!

7 Changelog

2007-07-20 10:00 UTC. Version 1.0.

Contents