Morph Endo!
Report on the Tenth Interstellar Contest on Fuun Programming

Eelco Dolstra

Jur Hage

Bastiaan Heeren

Stefan Holdermans

Johan Jeuring

Andres Löh

Arie Middelkoop

Alexey Rodriguez

John van Schie
Clara Löh

Department of Information and Computing Sciences, Utrecht University

Technical Report UU-CS-2007-029

www.cs.uu.nl
ISSN: 0924-3275

Morph Endo!

Report on the Tenth Interstellar Contest on Fuun Programming

Eelco Dolstra [*] Jurriaan Hage [*] Bastiaan Heeren [*] Stefan Holdermans [*] Johan Jeuring [*]
Andres Löh [*] Clara Löh [†] Arie Middelkoop [*] Alexey Rodriguez [*] John van Schie [*]

Abstract

The Tenth Annual ICFP Programming Contest was a 72-hour contest held July 20–23 2007 and organised in conjunction with the 12th ACM SIGPLAN International Conference on Functional Programming (ICFP 2007). As in the previous nine editions, the goal of the contest was to allow teams from all over the world to demonstrate the superiority of their favourite programming languag. This year's task was to reverse engineer the DNA of a stranded alien life form to enable it to survive on our planet. The alien's DNA had to be modified by means of a *prefix* that modified its meaning so that the alien's phenotype would approximate a given "ideal" outcome, increasing its probability of survival. In this report we describe the task, how to solve it, how we created it, and how the contestants fared. About 357 teams from 39 countries solved at least part of the contest. The language of choice for discriminating hackers turned out to be C++.

1. The story so far

On Saturday, April 21, 2007, Endo crashed on Earth. Endo is an alien life form, belonging to the species of the Fuun. While travelling through space to visit some relatives, Endo had fallen asleep. Endo's spaceship Arrow drifted into a heap of strange objects where it was picked up by an Interstellar Garbage Collector. Interstellar Garbage Collectors are run by another alien species, the Imps, who are known throughout the galaxy for dumping garbage on underdeveloped worlds for aeons. Hence, together with a lot of other stuff, Arrow and Endo were dropped on our planet.

Arrow is an intelligent spaceship, but was severely damaged by the crash. Therefore, it could not warn Endo that environmental conditions on Earth are not suitable for a Fuun. Endo left the ship, looked upwards, and got hit by a cargo container, also dropped by the Garbage Collector.

It took a while for Arrow to find out what had happened and to regain some of its reasoning power. When it finally knew what was going on, the situation looked grim: Endo was on the verge of death, and Arrow had consumed a lot of energy repairing itself to a certain extent, but no power supply seemed available anywhere close.

The environment seemed equally strange. Some animals were close by, but didn't react to communication attempts. Finally, Arrow seemed to have found a way to contact Utrecht University, via e-mail. The message was so strange that it not only crashed the mail server, but also caused a power outage at the Computing Sciences department. However, would it not have been for this crash and the investigation that followed, systems people might not have gained interest in the strange mail.

The huge message, containing several attachments and data in several unknown formats and languages, was shown around as a curiosity and finally found its way to the machine of Johan Jeuring, who together with some of his PhD students, started to look at it in the little time they had next to organizing the ICFP contest.

For a long time, little progress was made. The only success the group achieved, after quite a while, was the discovery of a sequence of pictures hidden and encoded within the message. The algorithm to decrypt the pictures was slow and running on just one machine, so it took about three months to decode the complete sequence of ten pictures (see Figure 1), which tell the story of Endo up to its arrival on Earth.

Arrow wondered why no answer was coming in, but already guessed that it might be difficult for the inhabitants of Earth to understand what it wanted, and focussed its efforts solely on prolonging Endo's life while preserving as much energy as possible.

On Friday, July 13, 2007, Alexey Rodriguez had an incredibly clever idea on how to decode large chunks of the message, and this turned out to be the breakthrough: large parts of the message were suddenly clear, and by Wednesday, July 18, contact with Arrow (via e-mail) was made.

Arrow's desperate plan was to save Endo by changing its DNA thereby adapting the Fuun to the conditions of Earth. Arrow usually could perform such an operation itself, but finding a suitable DNA modification is a tricky business. Arrow simply did not have enough energy to both come up with modified DNA and to administer the resequencing process.

Even then, considering the long delay after sending out the message, it turned out that Arrow's energy would run out sometime on Monday, July 23, implying certain death for Endo.

Johan Jeuring and his group managed to assemble a specification of how DNA resequencing works for Fuun, but it was clear that they would not have enough time to look for a suitable modification themselves. So as an emergency, they abandoned their original plan for the ICFP contest and instead asked the international hacker community to help to "Morph Endo!"

The Tenth Annual ICFP Programming Contest was a 72-hour contest held July 20–23 2007 and organised in conjunction with the 12th ACM SIGPLAN International Conference on Functional Programming (ICFP 2007). As in the previous nine editions, the goal of the contest was to allow teams from all over the world to demonstrate the superiority of their favourite programming languages.

This year's task was to construct a DNA prefix that makes Endo live. The contestants were given a way to simulate DNA synthesis in the form of an algorithm that generates a 2D-picture of Endo and its immediate surroundings from

[*] Software Technology, ICS, Utrecht University, {eelco, jur, bastiaan, stefan, johanj, andres, amiddelk, alexey, jcschie}@cs.uu.nl

[†] WWU Münster, clara.loeh@uni-muenster.de

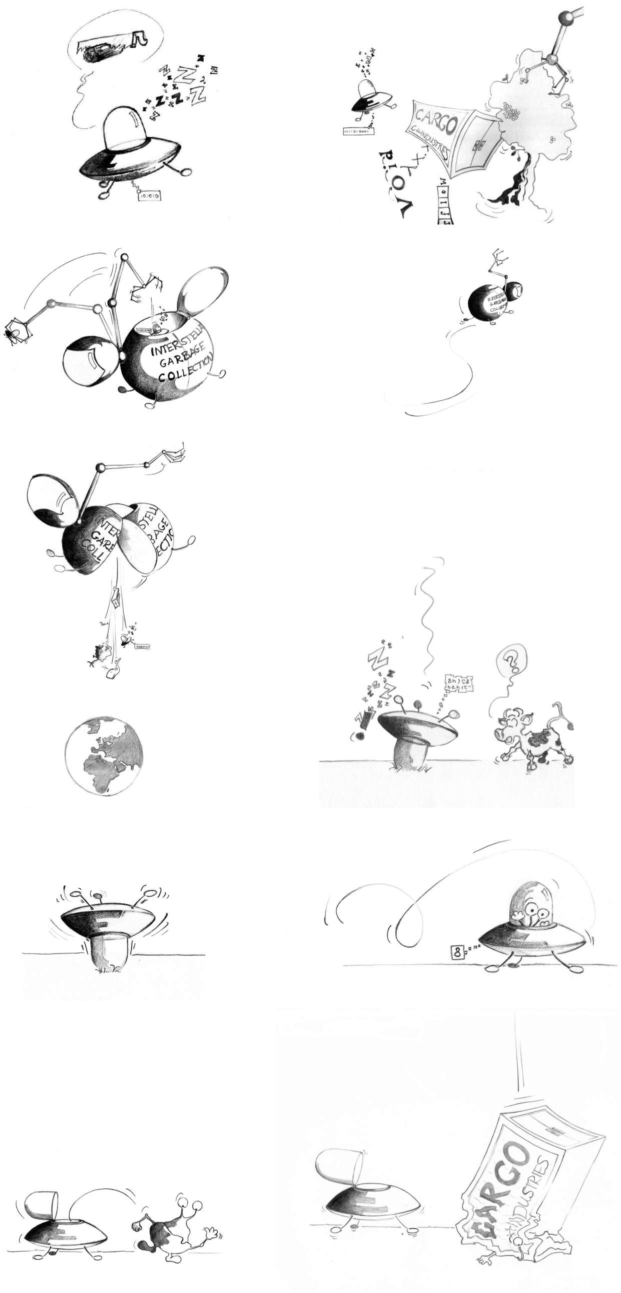


Figure 1: Arrival Sequence

a DNA string. Additionally, two images were handed out. The “source” image in Figure 2 is the one corresponding to Endo’s original DNA. The “target” image in Figure 3 is the goal: concatenating the prefix and the original DNA should result in an image that matches the target as closely as possible.

Prefixes were evaluated according to the following criteria:

- the number of incorrect pixels in the resulting picture compared to the target picture – the fewer, the better;
- the length of the prefix – the shorter, the better;

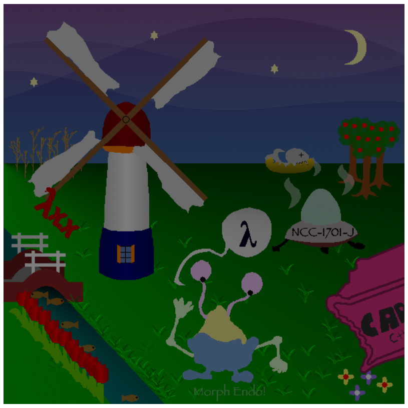


Figure 2: Source

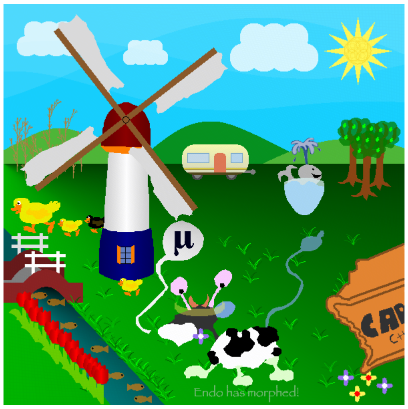


Figure 3: Target

- the energy consumption of the synthesis, i.e., the time and space complexity of performing the algorithm – just a limit was given.

In this report we describe the task, how to solve it, how we created it, and how well the contestants did. In Section 2, we have a closer look at Fuun DNA (and RNA) and the simulation algorithm that turns DNA into a picture. Endo’s DNA has some peculiar properties, and Section 3 shows how that structure can be used to obtain the target picture from the source picture. We also discuss some alternative approaches to producing the target picture. Section 4 shows the making of Endo: it discusses some of the tools we built to produce the task for the contest. In Section 5 we reveal the winners and give lots of statistics about the contestants.

2. DNA and RNA

Simulating the synthesis of Endo from Fuun DNA is a two-phase process. First, Endo’s DNA is converted into RNA using a process called *execution*. Then, the resulting RNA is used to *build* Endo and its immediate surroundings, or rather to produce a two-dimensional image of it.

Modifications of Endo such as requested by the task are performed by concatenating a *prefix* to Endo’s original DNA, and executing the resulting DNA.

$\underline{\mathtt{I}}, \underline{\mathtt{C}}, \underline{\mathtt{F}}, \underline{\mathtt{P}}$	match a literal `I`, `C`, `F`, or `P`, respectively
$!_n$	skip *n* bases
$?_{DNA}$	search for DNA string *DNA*
$\mathbf{(}\ p\ \mathbf{)}$	grouping: match pattern *p* and save the match

Figure 4: Pattern language

$\underline{\mathtt{I}}, \underline{\mathtt{C}}, \underline{\mathtt{F}}, \underline{\mathtt{P}}$	insert a literal `I`, `C`, `F`, or `P`, respectively
l_n	insert saved group *n* at quoting level *l*
$\lvert n \rvert$	insert the length of saved group *n*

Figure 5: Template language

DNA is a sequence where each element is one of four letters (`I`, `C`, `F`, or `P`), called *bases*. Endo's original DNA string is 7523060 bases long. RNA is a sequence of *commands*, each command being a DNA string consisting of seven bases.

2.1 Execution

Execution is a process that consists of one operation – a match-replace – on the DNA string, that is performed repeatedly. In each iteration, the DNA string is inspected from the beginning. The DNA is scanned until an initial segment is recognized as a *pattern*. The rest of the DNA is then scanned until an initial segment is recognized as a *template*. The rest at that point is *matched* against the pattern. The part that matches is removed and *replaced* by an instantiation of the template. If the match fails, nothing happens.

While scanning for the pattern and the template, a special three-base sequence `III` indicates a following RNA command which is then output immediately.

As long as the resulting DNA string of a match-replace operation can be interpreted again as another match-replace command, the process continues. If the end of the DNA string is reached while scanning for the pattern or template, the process stops. Executing Endo's original DNA performs 1891886 match-replace operations before it stops.

The pattern language comprises *constant* patterns (literal sequences of bases), *skipping* a non-negative number of bases, *searching* for a certain sequence of bases, and grouping (Figure 4). Templates contain literal sequences of bases, but can also refer to grouped parts of the DNA that the pattern has matched, and query the length of such parts (Figure 5).

Quoting Encoding constant bases in patterns and templates requires an escape mechanism: it is necessary to distinguish a literal `I` from the pattern matching a literal `I`. At the very least, we have to know where a pattern ends. Therefore, DNA uses a quoting mechanism in many places, which works as follows:

`I` becomes `C`
`C` becomes `F`
`F` becomes `P`
`P` becomes `IC`

As a result, we know that if we are looking for a quoted string, the sequences `II`, `IF`, and `IP` can never occur. Even if a string is quoted multiple times, this observation remains true. Sequences starting with one of those three pairs are therefore associated with special meaning. For instance, `IIC` or `IIF` denotes the end of a group or pattern or template, `III` indicates a subsequent RNA command, `IP` introduces a skip in a pattern, and `IF` introduces a reference in a template.

When strings are reinserted into the DNA via template references, one can choose an arbitrary quoting level at which they should be inserted.

Here is a full example of a single execution step – let us assume we have the following DNA:

```
IIPIPICPIICICIIF ICCIFPPIIC CFPC
```

For better readability, the above DNA is split into three parts. The first two parts are interpreted as a match-replace operation which is then applied to the rest – the third part.

The first part is interpreted as the pattern $\mathbf{(}!_2\mathbf{)}\underline{\mathtt{P}}$, i.e., start a group, skip two bases, end a group, match a `P`. The middle part is interpreted as the template $\underline{\mathtt{PI}}0_0$, i.e., insert `PI`, and insert what was matched against the first group. The replace command, which can be written as

$$\mathbf{(}!_2\mathbf{)}\underline{\mathtt{P}} \mapsto \underline{\mathtt{PI}}0_0$$

is now applied to `CFPC`, i.e., the remainder of the DNA. Two bases are skipped, thus the group is bound to `CF`, and the third base is a `P`, so the match is successful. The part that has been matched – the string `CFP` – is then removed, and instead, the template is instantiated (0_0 by `CF`) and inserted, so that the final result is `PICFC`.

Numbers Execution also makes use of encoded numbers in various places. A simple binary encoding is used, where an `I` or `F` represents 0, `C` represents 1 and `P` indicates the end.

Issues The DNA language as interpreted by the execution process is a Turing-complete language. We have encoded both an imperative and a functional (combinator-based) language in DNA, which are described in Sections 4.3 and 4.4, respectively.

Execution of Endo's DNA makes extensive use of skipping, matching and inserting large chunks of DNA, and given the total number of iterations, efficiency becomes very important. It turns out that the central issue is the choice of a suitable data structure. Some additional details are given in Sections 3.2 and 4.1.

2.2 Building

After executing DNA, we end up with a sequence of RNA commands. Although RNA commands are seven bases long, only 20 commands have an effect on the resulting picture. Other commands produced by Endo's DNA are ignored and have no documented effect (but see Section 3.4).

The 20 commands form a spiced-up turtle control language for generating a 600 by 600 bitmap picture. In addition to the basic commands for moving the turtle one step forward and turning it left or right by 90 degrees, there is support for changing the color, for saving a position, for drawing a line between the current and the saved position, for flood-filling an area of the picture, and for maintaining a stack of pictures where the top-two elements can be composed in different ways, allowing alpha-blending and clipping.

Endo's original DNA produces 302450 RNA commands, of which 237484 are among the 20 "legal" RNA commands. Performing these commands results in the source picture shown in Figure 2.

Changing the color with a limited set of commands while allowing a full range of 8-bit RGBA values is achieved by maintaining a color *bucket*. There is one command to empty the bucket, and there are ten commands to add different base colors to the bucket. Each color can be added to the bucket multiple times if desired. The currently active color is then given by the average of all the colors in the bucket. As a result of this approach, some RGBA values are very cheap to compute, while others are very costly. For instance, producing the opaque RGB color $(254, 255, 255)$ requires 255 RNA commands.

3. Solving the task

To reiterate, the task for the contestants is to adapt Endo's DNA to life on Earth. This adaptation is in the form of a *prefix*, a (hopefully) small piece of DNA that, when prepended to Endo's DNA and executed, produces the picture in Figure 3. So how would a contestant go about this task?

Of course, we could ignore Endo's original DNA entirely and try to generate DNA that draws the target picture. Done naively, this approach will take tens or hundreds of millions of bases, but by being clever, we can drastically reduce this amount. For instance, adjacent areas with the same colour can be drawn efficiently with a flood fill. Gradients in the picture are trickier, but with some cleverness these too can be drawn efficiently. Finally, we could write a DNA or RNA compressor *in DNA* – maybe even a PNG decompressor! But this approach has its limits: a PNG encoding of the target picture is about 235 KB, which would amount to almost a million bases, and then we would still have to include the PNG decompressor. So this isn't a winning approach (though you might get the Judges' Prize!).

3.1 Reverse engineering

Instead of trying to solve the task by means of the brute-force approach given above, it is better to *reuse* the existing DNA. After all, the target picture is quite similar to the original picture in many respects. We should try to figure out how Endo's DNA works, then write a prefix that "patches" the original DNA appropriately. For instance, the dome of the flying saucer in the original picture looks suspiciously like the "cup" in which the whale swims in the target picture. Maybe we can find the bases responsible for drawing the dome and prepend some rotation and move commands to transform it into the cup. In fact, the task description hints at this approach:

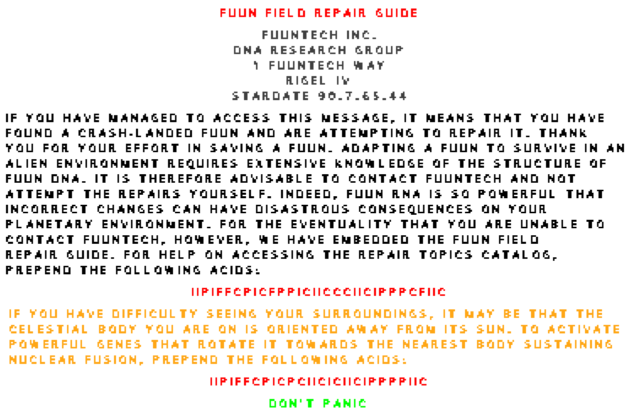
> It is hypothesized that, contrary to life on Earth, the Fuun are a result of *intelligent* design. We believe this because Arrow hints at the fact that there may be "messages" from the creators in the DNA, and that there may be genes already present that could help with the plan to transform Endo.

3.2 Getting started

The task description says that "something curious" happens if the prefix `IIPIFFCPICICIICPIICIPPPICIIC` is used. Obviously, we should try this first. And voila – *if* our DNA machine is correct, then we get a "self check" screen showing a number of tests, each followed by "OK". On the other hand, if some subtle aspect of the specification is implemented incorrectly, then some or all of the screen will be mangled, e.g., everything drawn after a certain test will be rotated by 90 degrees.

Well, it's good to know that our machine is correct, but it doesn't really help us get further (except that it's now clear that there are things hidden in Endo's DNA). So maybe we should look more closely at the DNA. It starts with `III` – that's an RNA command. In fact, there are thousands of RNA commands right at the start of Endo's DNA, before it goes off doing mysterious match-replace operations. What does the RNA do? Here it really helps if your DNA machine allows you to step through commands interactively, like a debugger – an indispensable tool for reverse engineering. When you do, you will see that a message is drawn before it is overwritten by a black flood fill:

```
IIPIFFCPICFPPICIICCIICIPPPFIIC
```



FUUN FIELD REPAIR GUIDE

FUUNTECH INC.
DNA RESEARCH GROUP
1 FUUNTECH WAY
RIGEL IV
STARDATE 90.7.65.44

IF YOU HAVE MANAGED TO ACCESS THIS MESSAGE, IT MEANS THAT YOU HAVE FOUND A CRASH-LANDED FUUN AND ARE ATTEMPTING TO REPAIR IT. THANK YOU FOR YOUR EFFORT IN SAVING A FUUN. ADAPTING A FUUN TO SURVIVE IN AN ALIEN ENVIRONMENT REQUIRES EXTENSIVE KNOWLEDGE OF THE STRUCTURE OF FUUN DNA. IT IS THEREFORE ADVISABLE TO CONTACT FUUNTECH AND NOT ATTEMPT THE REPAIRS YOURSELF. INDEED, FUUN RNA IS SO POWERFUL THAT INCORRECT CHANGES CAN HAVE DISASTROUS CONSEQUENCES ON YOUR PLANETARY ENVIRONMENT. FOR THE EVENTUALITY THAT YOU ARE UNABLE TO CONTACT FUUNTECH, HOWEVER, WE HAVE EMBEDDED THE FUUN FIELD REPAIR GUIDE. FOR HELP ON ACCESSING THE REPAIR TOPICS CATALOG, PREPEND THE FOLLOWING ACIDS:

IIPIFFCPICFPPICIICCCIICIPPPCFIIC

IF YOU HAVE DIFFICULTY SEEING YOUR SURROUNDINGS, IT MAY BE THAT THE CELESTIAL BODY YOU ARE ON IS ORIENTED AWAY FROM ITS SUN. TO ACTIVATE POWERFUL GENES THAT ROTATE IT TOWARDS THE NEAREST BODY SUSTAINING NUCLEAR FUSION, PREPEND THE FOLLOWING ACIDS:

IIPIFFCPICPCIICICIICIPPPPIIC

DON'T PANIC

Figure 6: First field repair guide page

There are other ways to discover this prefix. In fact, it's quite possible to see the hidden prefix by accident if, for instance, flood fills or bitmap operations don't work correctly yet; or if your machine is just very slow (which was the case for many contestants).

Hopefully this bit of DNA is another prefix, like the one for the self check. And indeed, when we prepend it to Endo's DNA and execute it, we make a remarkable discovery: the first page of the *Fuun Field Repair Guide* (Figure 6)!

Apparently Endo's intelligent designers – FuunTech Inc. on Rigel IV – helpfully created information on repairing broken Fuun in the field. The page shows two prefixes: a prefix that shows the next repair guide page, and one that rotates the planet, i.e., *turns the picture from night into day*. This prefix alone fixes a huge number of pixels (although the survival chance – see Section 5.3 – only increases to 1.27% to reflect the fact that this is far from enough to save Endo). About 160 teams managed to discover this prefix.

Actually, the other prefix sounds even more interesting – something about a catalog of repair topics – and indeed, when used, it shows a repair guide page that describes how integers are encoded in DNA, and suggests that one can access other pages by taking a known repair guide prefix and changing the embedded number to the number of the desired page. It also mentions that the catalog has page number 1337.

(This page presented a serious obstacle for many contestants: the first page renders quickly even on slow DNA machines, but the second one (like the actual picture) takes an excruciating amount of time if skips and template replacement aren't sublinear, as the spec advised. Thus, contestants would be stuck at this point unless they fixed the time complexity of their machine. It certainly isn't necessary to engage in heavy bit-fiddling, hand optimisation or assembler programming to get it fast enough: for instance, our straight-forward, 347-line Haskell reference implementation *using the right data structure* takes about 50 seconds. Our über-optimised C++ implementation took about 5 seconds.)

Onward, then! Given that we have a DNA machine by now, disassembling a known repair guide prefix shouldn't be hard. Let's take the prefix for Figure 6. It disassembles to the following DNA operation:

$$(?_{\texttt{IFPCFFP}})\underline{\texttt{I}} \mapsto 0_0\underline{\texttt{C}}$$

We use the notation pattern ↦ template to denote the DNA string encoding the corresponding match-replace operation. The above prefix searches for the base sequence `IFPCFFP` (and binds everything up to and including that sequence), then matches a lone `I`; it then rewrites the matched DNA string by putting back everything up to and including the `IFPCFFP`, and writing a single `C`. Thus, it replaces a certain `I` with a `C`. According to the description of the encoding of numbers on the second help page, that would be changing the number 0 to 1. To test this a bit further, we could look at the prefix for the second page:

$$(?_{\texttt{IFPCFFP}})\underline{\texttt{II}} \mapsto 0_0\underline{\texttt{IC}}$$

and indeed, this would appear to change 0 to 2.

According to the second page, we have to set the number to 1337 to get access to the catalog page. The encoding of 1337 is `CIICCCIICIC`. So the necessary prefix would be

$$(?_{\texttt{IFPCFFP}})\underline{\texttt{IIIIIIIIIII}} \mapsto 0_0\underline{\texttt{CIICCCIICIC}}$$

or, in concrete DNA,

```
IIPIFFCPICFPPICIICCCCCCCCCCCCIICIPPPFCCFFFCCFCFIIC
```

This prefix finally reveals the catalog page, which lists the numbers of many other repair guide pages. With the same technique as above we can now access all of them. Now we are really getting somewhere. There are a lot of interesting pages, although many are quite cryptic – a lot of talk about red zones and green zones and blue zones (e.g. Figure 7), and at least one page is "encrypted" according to the catalog.

But there is one page in particular that looks very interesting: page number 42 shows a "gene list" (Figure 8)! For each "gene", it shows the size and offset relative to a special base sequence. Finally – something that a compiler hacker can appreciate! Alas, this is only the first page. But there is a colossal hint in there: the gene named *AAA_geneTablePageNr*. So what if we constructed a prefix that searches for the special sequence `IFPICFPPCFFPP` (the marker to which the gene offset are relative according to the gene table), then skips 0x510 bases (minus the length of the special sequence), and writes a number? To write page number 10 would be, for instance,

$$(?_{\texttt{IFPICFPPCFFPP}}!_{1283})!_4 \mapsto 0_0\underline{\texttt{ICIC}}$$

where $!_n$ denotes a skip over n bases. In DNA, that's

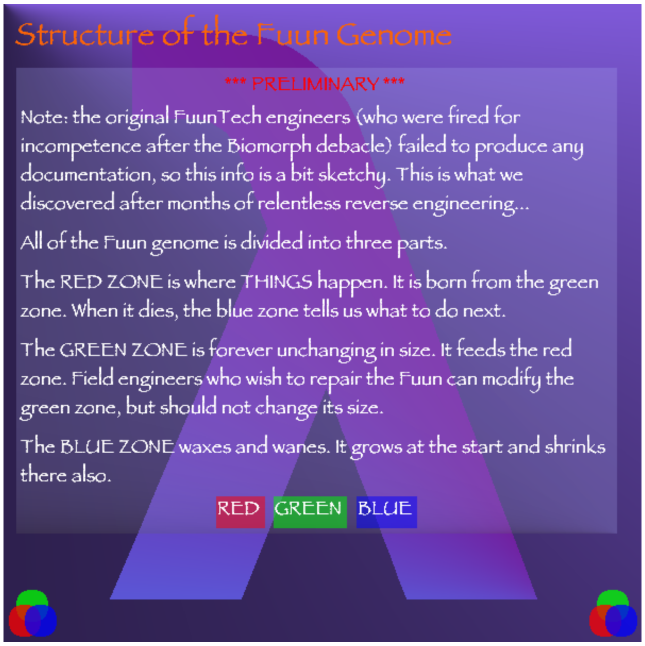
```
IIPIFFCPICCFPICICFPPICICIPCCIIIIIIC
ICPIICIPIICPIICIPPPCFCFIIC
```

Of course, this prefix has to be appended to the prefix that sets the help page number to 42.

From the gene list we learn that there are hundreds of these genes, although some entries in the gene table appear to be damaged.

3.3 Improving the picture

Now we have enough information to try to find ways to improve the picture. For instance, there are lots of apparent variables in the gene table (like *AAA_geneTablePageNr*). Perhaps tweaking them will have some effect on the picture. Of course, the more you know about the code (say, through tracing or disassembling), the easier this becomes.



Structure of the Fuun Genome

*** PRELIMINARY ***

Note: the original FuunTech engineers (who were fired for incompetence after the Biomorph debacle) failed to produce any documentation, so this info is a bit sketchy. This is what we discovered after months of relentless reverse engineering...

All of the Fuun genome is divided into three parts.

The RED ZONE is where THINGS happen. It is born from the green zone. When it dies, the blue zone tells us what to do next.

The GREEN ZONE is forever unchanging in size. It feeds the red zone. Field engineers who wish to repair the Fuun can modify the green zone, but should not change its size.

The BLUE ZONE waxes and wanes. It grows at the start and shrinks there also.

Figure 7: Alien software engineers use strange terminology

For instance, there is a variable *polarAngleIncr*, which, it turns out, determines the rotation of the blades of the windmill. How could you know? Well, the blades are rotated slightly in the target picture compared to the original picture, which makes one hopeful that the vertices of the blades are not positioned absolutely but are subject to some transformation. Plus, there are sine and cosine tables in the gene list. Finally, the call graph (see below) shows that the function *windmill* makes calls to *drawPolylinePolar*. It takes a bit of experimenting, but it turns out that setting it to 5 gives the rotation that matches with the target picture. The command to do so is

$$(?_{\texttt{IFPICFPPCFFPP}}!_{823763})!_3 \mapsto 0_0\underline{\texttt{CIC}}$$

Other interesting variables include *enableBioMorph* (which adapts Endo to the local ecosystem, though not necessarily in the desired way) and *weather* (which enables various weather patterns).

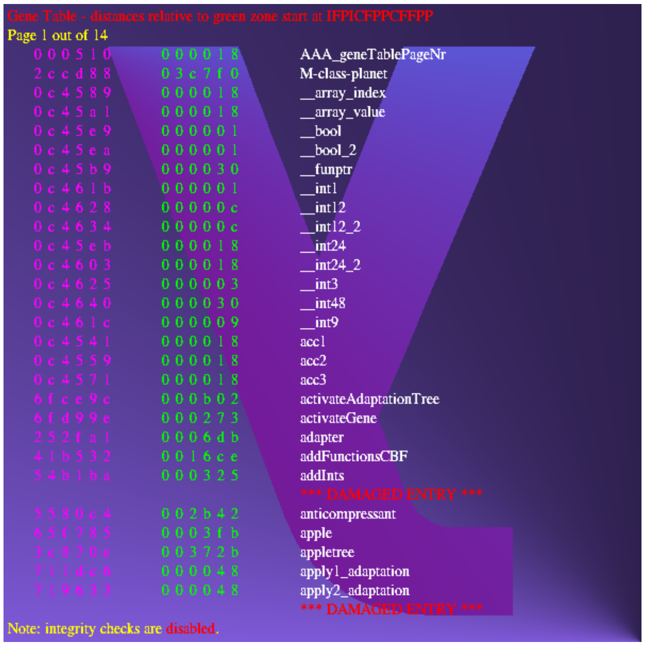
Far from all necessary changes are as simple as changing a variable. For instance, some involve modifying DNA code in some way, such as disabling certain bits of code or enabling dead code. An example is removing the $\lambda x.x$ stuck in the windmill:

$$(?_{\texttt{IFPICFPPCFFPP}}!_{5049987})!_{33} \mapsto 0_0\left[!_{727} \mapsto\right]$$

where [pattern ↦ template] denotes the encoding of a match-replace instruction in DNA. (The encoding of this instruction is 33 bases, hence the $!_{33}$.) In other words, this prefix places a skip of 727 bases at offset 5049987, which is the start of the code that draws the lambda. Similarly,

$$(?_{\texttt{IFPICFPPCFFPP}}!_{5043058})!_{33} \mapsto 0_0\left[!_{154} \mapsto\right]$$

causes the ducks to appear. (The ducks, it turns out, are drawn in a conditional: **if** *true* **then** *nop* **else** *drawSomeDucks*. The skip jumps over the conditional to the else-branch.)



```
Gene Table - distances relative to green zone start at IFPICFPPCFFPP
Page 1 out of 14
   000510   000018   AAA_geneTablePageNr
   2ccd88   03c7f0   M-class-planet
   0c4589   000018   __array_index
   0c45a1   000018   __array_value
   0c45e9   000001   __bool
   0c45ea   000001   __bool_2
   0c45b9   000030   __funptr
   0c461b   000001   __int1
   0c4628   00000c   __int12
   0c4634   00000c   __int12_2
   0c45eb   000018   __int24
   0c4603   000018   __int24_2
   0c4625   000003   __int3
   0c4640   000030   __int48
   0c461c   000009   __int9
   0c4541   000018   acc1
   0c4559   000018   acc2
   0c4571   000018   acc3
   6fce9c   000b02   activateAdaptationTree
   6fd99e   000273   activateGene
   252fa1   0006db   adapter
   41b532   0016ce   addFunctionsCBF
   54b1ba   000325   addInts
                     *** DAMAGED ENTRY ***
   5580c4   002b42   anticompressant
   65f785   0003fb   apple
   3c870e   00372b   appletree
   711dc6   000048   apply1_adaptation
   719633   000048   apply2_adaptation
                     *** DAMAGED ENTRY ***
Note: integrity checks are disabled.
```

Figure 8: The Holy Grail: the Gene Table

3.4 Reverse engineering the DNA

One important secret of Endo's DNA is the presence of certain undocumented RNA sequences. These RNA sequences are of the form `IIICFPICFP` or `IIIC`*nnnnnnn*, where each *n* is one of `I`, `P`, or `C`. A bit of analysis (plus a big hint in the help screen on "abnormal RNA") makes it clear that the former indicates a return from a function (a.k.a. "gene"), while the latter indicates the entry of a function, where the *n*s denote a unique function number in base-3 notation. Thus, these RNA sequences reveal the exact dynamic call graph within Endo's DNA.

3.5 Memory model

When you step through the DNA code, and from the help screens and the gene table, you should get a picture of the operation of the DNA, which is useful – you have to patch the code, after all. Endo's designers – the misnamed FuunTech – seem to have programmed Endo in an *imperative* language called *Imp*. It's a perfectly serviceable language though: it has functions, recursion, local and global variables, conditionals, loops, arrays, and even pointers. Due to the strange properties of Fuun DNA, the compilation scheme and memory model is not quite the same as what one would expect in Earth-bound Von Neumann machines, but if you squint just right, it almost looks like one. It's just a matter of understanding the FuunTech terminology.

The repair guide page in Figure 7 talks about several "zones" in the DNA: red, green and blue, which appear in the DNA string in that order. The blue zone (which "waxes and wanes") is just a stack: it contains return addresses, local variables and function arguments (and there is even a page on the precise layout of stack frames). The green zone contains code and global variables. But DNA does not have an instruction pointer – it can only execute instructions *at the front of the DNA string*. So we can't execute a function directly within the green zone, since then we would lose the function forever (plus all the functions and variables that precede it).

That's where the red zone comes in: it's simply a copy of (the remainder of) the current function from the green zone. A function is called by copying it to the front of the DNA, i.e. the red zone. The caller pushes the return address on the stack, then discards its remaining code and copies the callee to the front of the string.

A function returns by popping the return address from the stack, discarding its own remaining code, and copying the remaining code of the caller back to the front of the DNA string. Here it has to know *how much* of the remaining code of the caller to copy back. Therefore, an address consists not just of an offset (relative to the start of the green zone) but also a size in bases.

As each instruction is executed (i.e., appears at the front of the string), the offsets of all functions and variables that it references are statically known. This is because the compiler for the imperative language knows the size of the remaining "red zone" code and it knows the size of each object in the green zone. Similarly, it knows the offset of the start of the stack, and therefore of all variables in the current stack frame.

All of this means that you have to be very careful about modifying DNA. You cannot insert code into the green zone, since that would invalidate offsets. You have to be very careful when calling functions that you also discard the current red zone code. And when you call a function from a prefix, you don't have a return address in the green zone, unless you patch the green zone first. Calling functions from a prefix is therefore pretty tricky.

However, the Fuun engineers were aware of this difficulty and provided a "function call adapter" that makes it easier to call functions from prefixes. It is explained in detail in a repair guide page, but essentially it just saves the current red zone on a special stack (i.e., it saves the actual code, rather than a return address in the green zone, which you don't have when calling from within a prefix).

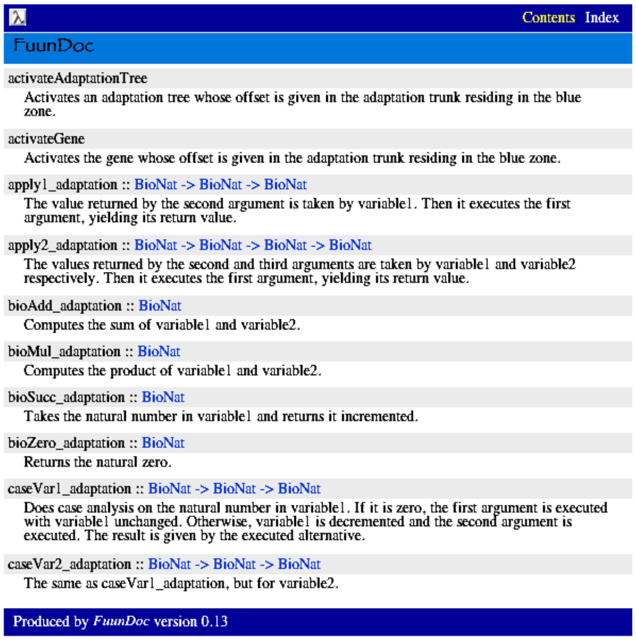
3.6 Secrets

Endo's DNA contains many secrets that can help you to produce a short prefix fixing as many pixels as possible. We won't reveal them in detail, but we will give some hints.

Intergalactic character set Strings in Endo's DNA are encoded in the Intergalactic character set. The character set help page does not appear very helpful as it's drawn in a "bullet" font (every character's glyph is an identical dot). However, overwriting the contents of the *fontTable_Dots* variable with that of *fontTable_Cyperus* (or any other font) will reveal the character set, which turns out to be EBCDIC shifted by 64 positions. Given this knowledge, it's easy to write the equivalent of the Unix *strings* program and discover all kinds of interesting information. However, as one help screen notes, a lot of strings are actually constants in imperative instructions (e.g. to push them on the stack) and are therefore quoted.

Gaps in the gene table The gene table has a number of "damaged" entries. By looking at the "holes" between the known genes, the locations and sizes of most missing genes are easily reconstructed. (Actually, the array of genes in the *printGeneTable* function isn't damaged at all, it just has a "damaged" boolean for every gene. So dumping that array will show all genes including their names.)

Damaged genes Some functions are "damaged". You can see which by calling the function *printGeneTable* with a *true* argument, which causes it to verify a checksum over each


Contents Index

FuunDoc

activateAdaptationTree
Activates an adaptation tree whose offset is given in the adaptation trunk residing in the blue zone.

activateGene
Activates the gene whose offset is given in the adaptation trunk residing in the blue zone.

apply1_adaptation :: BioNat -> BioNat -> BioNat
The value returned by the second argument is taken by variable1. Then it executes the first argument, yielding its return value.

apply2_adaptation :: BioNat -> BioNat -> BioNat -> BioNat
The values returned by the second and third arguments are taken by variable1 and variable2 respectively. Then it executes the first argument, yielding its return value.

bioAdd_adaptation :: BioNat
Computes the sum of variable1 and variable2.

bioMul_adaptation :: BioNat
Computes the product of variable1 and variable2.

bioSucc_adaptation :: BioNat
Takes the natural number in variable1 and returns it incremented.

bioZero_adaptation :: BioNat
Returns the natural zero.

caseVar1_adaptation :: BioNat -> BioNat -> BioNat
Does case analysis on the natural number in variable1. If it is zero, the first argument is executed with variable1 unchanged. Otherwise, variable1 is decremented and the second argument is executed. The result is given by the executed alternative.

caseVar2_adaptation :: BioNat -> BioNat -> BioNat
The same as caseVar1_adaptation, but for variable2.

Produced by *FuunDoc* version 0.13

Figure 9: FuunDoc documentation page

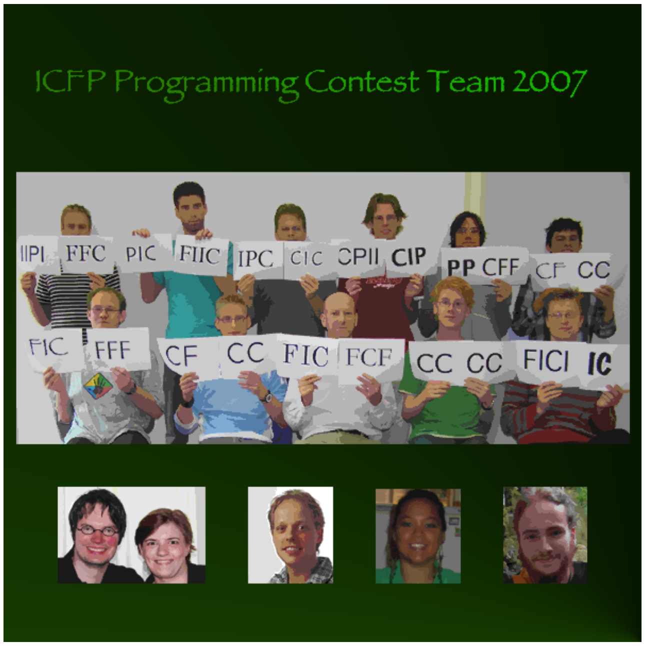


Figure 10: Alien life forms

gene. Damaged genes can be repaired in various ways: some genes are palindromes, and some have Hamming codes that allow them to be error-corrected.

ImpDoc and FuunDoc The gene table contains functions intriguingly named *impdoc1*, *impdoc2*, ... *impdoc10* and *fuundoc1* ... *fuundoc3*. These provide documentation for functions in the imperative and functional languages used to build Endo's DNA. (The functional language is discussed below.) They can be executed very easily using the function adapter. Figure 9 shows a FuunDoc documentation page, which looks suspiciously like Haddock [4].

Organisers' easter egg No piece of software would be complete without an easter egg commemorating the developers[1]. The function *alien-lifeforms* (hidden in a gap in the gene table) does just that (Figure 10). Presumably the letters do something useful.

Previous contests As this was the tenth ICFP Programming Contest, a homage to the previous nine contests was included. It can be shown by calling the functions *contest-1998* through *contest-2007*. There is a prefix hidden in there, and a (maybe not very useful) hint.

Audio prefix The voice of an alien reading a useful prefix is hidden somewhere in Endo's DNA. How do you find it? Open the DNA in an editor, set the width of the window to 16 characters, scroll down a bit and you might see a hint. The prefix provides a hint on how to fix the weeds; see below.

Biomorphological Unit Unless you want to draw Endo yourself (which would incur a heavy cost), you'll want to use Endo's *Biomorphological Unit* (BMU), which morphs Endo to imitate life forms in the surrounding ecosystem. However, as one repair guide page notes, the BMU is rather flakey, as evidenced by the fact that depending on environmental conditions Endo will morph into strange creatures such as OCamls and MLephants – and even ducks. The BMU also malfunctions quickly when it rains, but maybe this bug is really a feature?

Vehicular Morphological Unit Endo's flying saucer can morph into less conspicuous forms of transportation, but it does need a license key. Maybe one of the hidden prefixes discussed above will help.

Encryption Various genes are encrypted. The repair guide page "How to Activate Genes" is "encrypted" with a simple ROT-2 cipher (though attempting to crack it may still violate intergalactic copyright legislation). Various other genes are encrypted with a more advanced algorithm, discussed in detail in the "Fuun Security Features" page (the text of which is ROT-13 encrypted, just because the FuunTech engineers are unpleasant people).

Cargo box The colour of the cargo box changes from purple to orange (which colour would be more appropriate in a Dutch landscape?). By carefully inspecting the precise RGB values of the cargo box in both pictures, one can easily fix the box by changing just a few RNA color bucket operations. This gives a sizeable increase in Endo's survival chance.

L-systems There are some weeds in the background that look differently between the source and target pictures. They are generated using an L-system [5], and can be efficiently fixed by changing the pseudo-random number generator seed to 8128, the fourth perfect number, which is hinted at in the help screen activated by the audio prefix.

Spirograph In the target picture, the sun contains a subtle, strange pattern. This pattern can be drawn using a *spirograph*. Endo's DNA contains a function to draw spirographs, but the trick is to figure out the parameters that draw the desired spirograph.

[1] Rumour has it that there is another easter egg, and that it provides another hint – maybe regarding the hills.

There is a repair guide page on spirographs that shows various examples of spirographs, including the very one needed for the sun. However, this page is hard to find – it is not listed in the catalog, has a high page number, and is compiled directly into the main function so that it can only be found by disassembling main.

Whale The whale is drawn by a function that takes a boolean argument that determines whether the whale is happy or sad. The whale's blow is hidden in plain sight in the gene table function. Rotating the cup is a matter of injecting some RNA rotation and movement commands. Finally, the cup can be filled with rain water – maybe changing the weather would help?

Hills The hills are drawn using a bunch of higher-order functions in Endo's imperative language to combine sines and parabolas. A prefix that activates the hills (with a slightly off shape) is hidden somewhere.

Virus A malicious virus from outer space is residing in Endo's DNA and waiting to be activated. This virus will definitely mess things up. Ironically, the help page describing this virus appears to be one of the locations that is infected.

Steganography Steganographic content was hidden in one of the not so informative help pages, revealing a key for an encrypted part of the cow. The hidden message can be seen by only considering the least significant bits of the RGB color channels of that help page.

Fish Unlike other image elements, the fish are drawn by code written not in the Imp language but in the functional Fuun language (see section 4.4). To fix the direction some of the fish are facing, it is important to know that the layout of the fish is given in a tree-like structure in the gene *goldenFish_adaptation*. The tree describes how parts of a picture, represented as subtrees, are combined into bigger pictures using operators. These picture operators are documented in the FuunDoc pages. The encoding of values in the Fuun compilation scheme is described in the repair guide page on so-called *super adaptive genes*.

Grass The original picture contains 70 clumps of grass, whose locations are computed using a pseudo-random number generator. They must be relocated in the target picture by setting a new seed value in the variable *seed*. This value can be found by solving the puzzle given in the "Biomorphic initial conditions" repair guide page. The two functions in that page, *bioAdd* and *bioMul*, are compiled using the functional Fuun language using combinators documented in FuunDoc. If *enableBioMorph_adaptation* is set to true, the grass seed will be disturbed using *bioMul_adaptation*. However the definition for *bioMul* is wrong, therefore the wrong seed is computed. To solve this problem, you must come up with a correct implementation of Peano-style multiplication and encode it in *bioMul_adaptation*. For inspiration on how to encode it, you may want to have a look at *bioAdd* and its corresponding encoding *bioAdd_adaptation*.

Major Imp For people tired of hacking DNA, there was a diversion in the form of various episodes of the galaxy-famous adventure series "Major Imp's Next Assignment". Nobody discovered a crucial hint to solving the contest sneakily hidden in the Major Imp stories.

4. The making of Endo

We started thinking about ideas for the task way back in November 2005, but it wasn't until August 2006 that we began to put real effort into it. Some oddball ideas were considered – writing Quake bots, reverse engineering music, and counting rings in photographs – until we settled on hacking alien DNA. We intended this year's contest to be a reverse-engineering hackfest: the idea was to let contestants make debuggers, disassemblers, and whatever other means they could come up with to grok and repair Endo's DNA.

4.1 The evolution of DNA

The initial DNA language looked suspiciously like regular expressions, since we all felt that sed(1) is under-utilised as a programming language paradigm. We quickly verified that all necessary programming features – variables, loops, conditionals, stacks, functions – could be implemented using regular expressions.

While programming in pure regular expressions is clearly an idea whose time has come, it turns out to have some challenging implementation issues. (We intend to publish extensively at POPL and PLDI on these.) The most vexing problem is to make DNA evaluation efficient enough. Our original DNA language relied exclusively on pattern matching to locate variables and functions in memory. For instance, each variable would be preceded by a unique "marker", and could be updated by searching for its marker and updating the succeeding bases, e.g.

$$(?_{\texttt{ICFP}})\ldots \mapsto 0_0\underline{\texttt{IPI}}$$

to update the 3-base variable marked ICFP to IPI. However, this means that every DNA instruction takes $O(n)$ time in the length of the DNA, which is much too slow[2]. Thinking about how to cache the location of markers gave us a headache, so a different approach was necessary.

The solution is the skip operation, which allowed us to sneak in random-access memory while still retaining the flavour of regular expressions. After all, a skip operation of n bases is really just the regular expression . repeated n times. If DNA is stored in a data structure like a rope [2] or even just a sufficiently short list of strings, programs can be executed efficiently.

Still, Endo's DNA would require a lot of arithmetic (for a while we even wanted to draw the windmill with an embedded 3D engine, but we ran out of time). Initially we used Peano arithmetic, which for some uses (such as a small loop counter) is very efficient in DNA. But Peano doesn't scale very well, so we moved towards a binary encoding of numbers and wrote DNA functions to do addition, subtraction and multiplication. As these operated at the bit level, they were quite slow. We cheated once more and snuck addition into the DNA specification in the form of the "length of match" operation: to add natural numbers n and m, you skip n and m bases within a group, then in the replacement store the length of the group thus matched. However, this fails when $n + m$ is longer than the length of the DNA, which is why Endo's DNA starts with a mysterious bunch of instructions that "grow" his DNA to 2^{24} bases.

But now subtraction was a bottleneck, so we committed a final hack and changed the semantics of DNA quotation such that we could use it to do efficient subtraction. (Hint: in two's complement, $x - y = x + (\tilde{}y) + 1$, so all we need is a way to perform bitwise negation...)

[2] This is exactly the problem that many contestants encountered if they didn't use an efficient datatype for DNA.

4.2 Making pictures with RNA

The RNA language is inspired by turtle graphics [1], although ultimately our "turtle" is primitive in some respects and advanced in others. We had interminable debates about whether arbitrary directions should be permitted (as opposed to just top/bottom/left/right), and whether the specification should be pixel-precise (which would preclude the use of floating-point numbers in the specification). For a while we even considered specifying output in terms of Scalable Vector Graphics (SVG). We added an alpha channel and compositing operations to be able to draw nice-looking images (see for instance the transparent backgrounds in the repair guide pages). Floodfills were added to obviate the need for an explicit polygon drawing operation.

4.3 The Imp language

In terms of ease of programming, Fuun DNA lies somewhere between assembler and Turing machines. Therefore, we made a simple, high-level imperative language called Imp that compiles to DNA code. (Some code, such as the self-check, was more-or-less written by hand.) The Imp compiler is written in Haskell, and Imp programs are written as an embedded domain-specific language in Haskell. Imp is a pretty conventional C-like imperative language, except for some tricky details. For instance, you can't really pass a pointer to a stack variable to a function, because pushing things on the stack causes the addresses of stack variables to shift.

Here is an example of an Imp function that returns the length of a string. (Strings are sequences of 9-base integers, terminated by the value 0xff.)

```
stringLength =
  comment "Return the length of a string." $
  function "stringLength" intType     -- return type
    [stringArg "s"]                   -- parameters
    [intVar "i" 0]                    -- local variables
    [while ("s" !!! "i" ≠ byte 0xff)
      ["i" ⟵ "i" + 1]
    , ret "i"
    ]
```

Haskell functions such as *while*, *ret*, and *function* are combinators that build the abstract syntax tree that the compiler translates into DNA. We used some very nasty and ill-advised operator overloading to be able to write object-language expressions such as `"i"` $+ 1$.

Embedding the Imp language in Haskell obviates the need for a grammar and parser, but more importantly, it allows all kinds of meta-programming in Haskell. After all, the full expressive power of Haskell is available to transform abstract syntax trees at compile time. For instance, here is the definition of a function that performs a bitwise increment of an integer; note that the *foldr* essentially unrolls a loop that iterates over the bits.

```
incInt = comment "Increment an integer by one." $
  function "incInt" intType [intArg "x"] [ ]
    [foldr (λindex carryToNextBit →
      iff (base "x" index ≡ encodeOneBit)
            -- bit at index is 1, set it to 0 and go to the next bit
          [base "x" index ⟵ encodeZeroBit, carryToNextBit]
            -- bit at index is 0, set it to 1
          [base "x" index ⟵ encodeOneBit]
      )
      Nop    -- overflow; ignore
      [0 . . (defaultIntLength − 2)]
    , ret "x"
    ]
```

We wrote quite a bit of code in Imp, such as arithmetic operations, string operations, turtle graphics, RC4 encryption, Hamming error correction, functions for drawing L-systems and spirographs, and lots more. Also, the functions that draw the Endo scene and the repair guide screens were generated from a picture combinator language that translated into Imp code.

4.4 The Fuun language

The code that positions the fish is generated using a compiler for a functional language called Fuun. Like the Imp language, the Fuun language is a DSL embedded in Haskell. A neat detail of the embedding is that Fuun programs are statically typed by the Haskell compiler using phantom types. Fuun is a call-by-name functional language, so expressions are evaluated only if their values are demanded. It is not a call-by-need language, so expressions are evaluated repeatedly rather than that they reuse previous evaluations.

4.5 Tools

We developed a lot of tools to generate, execute and analyse DNA programs. For this, we used an amalgam of programming languages: Haskell, C++, C, Java, Perl, PHP, and Ruby.

DNA machines We wrote implementations of the `execute` function in Haskell (several implementations), C++ (several), Perl, and Java. In the end, we decided to use the C++ version for our submission system, but the Haskell implementations were pretty fast as well. Some of these implementations offered additional features to support some very basic debugging.

Renderers Renderers (the function `build`) were written in Haskell and Java. For the submission system, we used a Haskell/C implementation: we used Haskell's Foreign Function Interface to communicate with the *libpng* library to do some low-level bitmap operations. The algorithms for drawing lines, flood fill and bitmap composition were also implemented in C. This renderer also supported a nice debug option to see some intermediate bitmaps.

Picture combinators We designed an embedded domain-specific language to compose pictures from primitive elements such as texts and circles. This combinator library was written in Haskell, and was used to describe the source and the target picture, as well as all the help pages. It supported both relative and absolute positioning, allowed us to apply some gradients to elements, and to rotate and scale parts of a picture. Picture description in this EDSL were translated to the Imp language and from there into DNA.

Font generators Endo's DNA sequence contains 3 embedded fonts such as the well-known *Tempus*. In fact, these fonts were embedded several times in various sizes and in different styles (e.g., italic). We made a tool in Java to translate an existing font by turning its characters into RNA commands. Border pixels are drawn semi-transparently to achieve some anti-aliasing, which is essential for the readability of the smaller fonts. We also included the *Wingdings* font and used this for the help page on viruses.

Curve tool We wanted some of the picture elements to be cartoonish (such as the cow), and for this we implemented a simple tool for drawing curves interactively. This tool was written in Haskell using the wxHaskell GUI toolkit. The curves were approximated by quadratic Bézier curves, which were then rendered to a collection of points.

Image tools A couple of tools were developed (in Haskell) for embedding images. The first tool simplifies the images: colors are slightly changed to simpler colors (requiring fewer RNA commands) and larger areas of one color are created by merging areas that are sufficiently close to each other. The degree of simplification was determined for each of the images individually. The second tool converts the image to a sequence of RNA commands. Connected areas of one color are determined, the border is drawn (only where it is really necessary), and if needed, some flood fills are performed. The order in which the areas are dealt with highly influences the number of RNA commands. A few simple heuristics were used to determine the ordering. The last tool turns a list of RNA commands into DNA commands and performs some compression. All RNA commands are mapped to natural numbers: the more occurrences, the lower the natural number. A simple combinator, written in DNA commands, can turn the natural numbers back into RNA commands. The compression ratio is acceptable: although higher ratios could be obtained, a requirement was that decompression at execution-time should be reasonably efficient. The following table shows for two embedded images the sizes before and after simplification (number of bytes in PNG format), the number of RNA commands, and the number of DNA bases:

image	*before*	*after*	*RNA*	*DNA*
world map	95111	30687	111350	246500
contest team	284158	58634	226020	455133

Call graph A small Ruby script (88 loc) visualises the call graph by inspecting the undocumented RNA commands, linking them to the symbol table.

Strings tool We wrote a simple tool in Haskell that finds all (quoted) strings in a piece of DNA. Care was taken that not too much information could be found in this way.

Web submission system Teams used the web system to monitor their progress, and the progress of their competitors, thus stimulating competition. Each team could submit a prefix at most each ten minutes to receive a score, see an overview of the scores of their previous submits, and view the rendered result of their prefix with the lowest score (highest survival chance). The scoreboard, visible for everyone, shows the teams ranked by score, with the top 15 in random order.

The web system consists of two parts, both written in PHP; the front-end and the back-end. The front-end stores a submission in the database and the back-end retrieves it, executes the prefix, generates an image, stores it in the database, and scores the image. Because this back-end cycle takes approximately 20 seconds with our implementation and we received more than three submits per minute, we needed to run multiple instances of the back-end. We ran instances of the back-end on 18 machines, which resulted in an average waiting time of 5 seconds before the back-end picked up a new submit.

4.6 Encouraging reverse engineering

We realized that this programming contest could be solved in many different ways. Although we didn't want to limit the various approaches too much, we took some measures to make reverse engineering more attractive (compared to a brute-force approach). The target picture contained an "anti-compressant" (a Moiré pattern) which some people did manage to clone. The target picture contains several gradients that are difficult to draw.

Many people have wondered why RNA commands have such an inefficient encoding (ten bases for just 20 commands). This was to penalise attempts to simply "brute force" the target picture by drawing it with just RNA commands.

5. The contest

The contest took place from 12:00 (noon) CEST on July 20 2007 till 12:00 on July 23, giving contestants 72 hours to save Endo's life. Teams were not required to pre-register, but could do so. There was no limit on team sizes. Teams with members associated with the Information and Computing Sciences department of Utrecht University were allowed to participate, but were ineligible for any of the prizes.

Teams could submit DNA prefixes any number of times during the 72-hour period, with a ten minute waiting period between submissions to prevent overloading the server. Each submission was immediately evaluated by our submission system. The score of the submission was then reported back to the team. Also, a rendering of the *best* submission of the team so far was shown. We didn't show each team's *latest* submission to prevent teams from using our submission system as a substitute for writing a DNA machine. Teams were judged on the basis of their best submission over the course of the contest.

During the contest, a scoreboard showed the scores of each team, with the exception of the 15 highest-scoring teams which were shown unordered without a score.

5.1 The contestants

869 teams registered before and during the contest. Ultimately, 357 teams submitted at least one prefix. The average size of the submitting teams was 2.6 members.

Teams were asked to specify their physical locations. Some teams were distributed across countries or even continents. Table 1 lists how many teams had members in each country. There were 137 teams with members in North America, 1 in South America, 55 in Asia, 188 in Europe including Russia (with 136 in the European Union), 16 in Oceania and 2 in Africa. Incidentally, Africa was the continent with the highest percentage of winning teams.

Teams were not required to submit source code and other contest materials unless they wished to be eligible for a prize (in particular the Judges' prize). Thus, we cannot make certain pronouncements regarding the programming languages used by teams. However, teams were asked to specify the languages they used on their team information page, which they could change during and after the contest.

Table 2 shows how many times languages were mentioned by teams. Some entries may need to be taken with a grain of salt. Imperative languages continue to dominate the field. Haskell and OCaml are the most popular functional languages by some margin. Interpreted languages are also popular.

5.2 During the contest

The organisers spent their time watching movies, monitoring progress on the big scoreboard and answering questions from teams on the contest mailing list. Happily for us, most questions could be answered by stating that the spec was cor-

#teams	Countries
130	USA
35	Japan
33	Germany
30	France
29	Russia
15	UK
10	Australia, Ukraine
9	India, Sweden
7	Canada, Netherlands
6	Belgium, New Zealand
5	Austria, Belarus, Latvia, Switzerland
4	Finland, Italy
3	China, Ireland, Norway, Spain
2	Denmark, Greece, Hungary, Israel, Poland, Singapore, Slovakia, South Africa
1	Bulgaria, Colombia, Romania, South Korea, Taiwan, Thailand, Uzbekistan

Table 1. Countries of team members

#teams	Language(s)
81	C++
67	C
66	Haskell
64	Python
52	Objective Caml
48	Java
35	Perl
26	Ruby
22	Lisp
22	C#
17	Scheme
9	Unix shell (sh, bash)
8	D
5	PHP
4	Erlang, Delphi
3	ML
2	AWK, Fuun DNA, LOLCODE, Lua, Octave, Prolog, Refal, Scala
1	2D, Basic, Blub, Brainfuck, CWEB, Cobol, Dylan, Emacs Lisp, Excel, FP, F#, Grep, Hub, MUMPS, Nemerle, PL/I, Pascal, R, Sed, Silcc, Smalltalk, Unlambda

Table 2. Languages used by teams

rect and that this or that section should be read more carefully. Technically, the contest went by uneventfully — the submission system held up fine except for a 15 minute downtime near the start of the contest, when the Xen virtual machine hosting the PHP script began to run out of memory.

However, for a long time it seemed that poor Endo was not going to make it! Most teams seemed to have more trouble than we had hoped getting the DNA machine fast enough — a precondition to doing the actual Endo-saving work. Many teams managed to discover the prefix that turns on daylight, but unless they had fast DNA machines, they would get stuck at that point.

Fortunately, in the second half of the contest and the last day in particular, Endo's chance of survival went up by leaps and bounds (Figure 11 shows how the scores of the best six teams began to improve – i.e. decrease – around 40 hours into the contest). Most of this progress wasn't visible to the world, since the scores of the best 15 teams weren't shown on the scoreboard.

Place	Score	Survival chance	Team name
1	178246	90.22%	Team Smartass
2	224623	84.92%	United Coding Team
3	293898	75.59%	Celestial Dire Badger
4	321617	71.52%	ryba
5	358246	65.98%	PurelyFunctionalInfrastructure
6	453744	51.32%	jabber-ru
7	498781	44.66%	Begot
8	514121	42.47%	Basically Awesome
9	543163	38.45%	SwtPl
10	608964	30.07%	shinh
11	682894	22.07%	SzM
12	819614	11.34%	kuma–
13	862213	8.99%	Unknown?
14	865556	8.83%	voyo
15	872788	8.47%	kokorush

Table 3. Top 15 teams

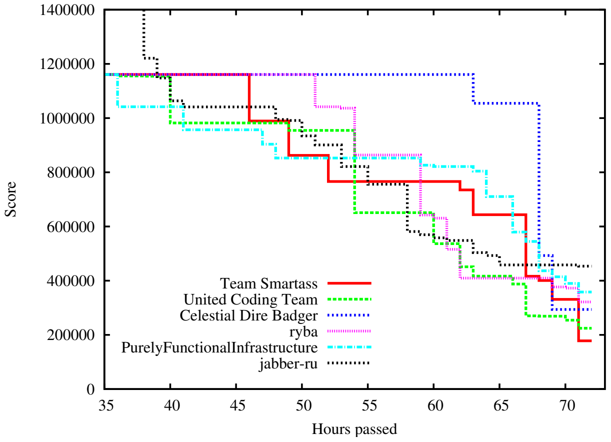


Figure 11: Scores of the Top 6 during the contest

5.3 The winners

Table 3 lists the scores and survival chances of the best 15 teams. A team's score is the length of its best prefix, plus the number of incorrect pixels in the generated image times 10. The survival chance is defined as $100\ e^{-1\ (0.000018\ \text{score})^2}$.

The jury decided not to award a winner for the "lightning division" prize (for the best submission within 24 hours) because there were no non-trivial submissions within 24 hours.

Judges' prize While the first and second prizes followed directly from the teams' scores, for the Judges' prize we looked at the materials that 31 teams submitted. We were looking in particular for clever techniques and tools that resulted in a good score.

Half-way through the contest the jury worried somewhat about how well some "brute force" approaches were doing, i.e. approaches that did not attempt to reverse engineer Endo's DNA but rather wrote DNA and RNA code that drew an approximation of the target image directly. In fact, this approach – though perfectly legitimate – had been a concern for us in advance since we intended this to be a reverse-engineering contest. We had even put in place several counter-measures: there is an almost invisible Moiré pattern super-imposed on the source and target images, and RNA commands are 10 bases long to penalise brute-force drawing code.

Nevertheless, a number of teams proceeded with the "re-engineering" approach undaunted and made good progress, even leading the scoreboard for a while. Of these, Celestial Dire Badger (using OCaml and C++) had the most elegant approach. He combined a more-or-less "brute force" approximation of parts of the target picture (with increasing resolution in the final hours of the contest) with a re-use not of Endo's DNA but its captured RNA output, as well as a compressor for the generated DNA. Therefore the jury is happy to declare that

> *Celestial Dire Badger (Jed Davis) is an extremely cool hacker.*

Second prize The jury is pleased to declare that *United Coding Team* (Cape Town, South Africa) has proven that

> *Perl is a fine tool for many applications.*

The members of this team were Richard Baxter, Marco Gallotta, James Gray, Carl Hultquist, Alexander Karpul, Julian Kenwood, Bertus Labuschagne, Hayley McIntosh, Bruce Merry, Max Rabkin, Ian Saunder and Harry Wiggins.

First prize The jury is honoured to declare that *Team Smartass* (Mountain View, California) has demonstrated beyond doubt that

> *C++ is the programming language of choice for discriminating hackers.*

This team consisted of Ambrose Feinstein, Christopher Hendrie, Derek Kisman and Daniel Wright. Team Smartass also won the 2006 ICFP Programming Contest. An impressive achievement!

5.4 Reflections

One thing that struck us during and after the contest, reading IRC channels and blog postings, was that many programmers don't have a lot of confidence in their favourite (functional) language: when they realised that their implementation of the DNA machine was too slow, their first instinct was often to switch to a "faster" language such as C. But the problem wasn't the language but algorithmic complexity: a straight-forward Haskell implementation using the right data structure (e.g. *Data.Sequence* [3]) would be fast enough and outperform by several orders of magnitude an optimised C implementation using a dumb data structure. So programmers should worry less about languages and more about good old complexity.

As far as functional programming is concerned, we must conclude that functional languages didn't fare too well this year (although in the Top 15 there were five users of OCaml and three of Haskell). Better luck next year!

5.5 So what happened to Endo?

Thanks to the hard work of the contestants, Endo survived. It spent some time as a cow in a meadow near Utrecht, during which time it revealed lots of advanced computer science techniques that we will be publishing about at some future ICFP. It joined us on our trip to Freiburg to say 'thank you' to the contestants present at the conference.

Acknowledgements. We would like to thank several people that helped in organizing this contest. Atze Dijkstra and Doaitse Swierstra participated in our early brainstorming sessions. Chris Eidhof, Maaike Gerritsen, Jeroen Leeuwestein, Eelco Lempsink, Martijn van Steenbergen and Mark Stobbe tested an early version of the problem, and helped extending it. Eric Bouwers, Thomas van Noort, Sander Mak and Michiel Overeem tested the second version of the problem. The systems people at the Information and Computing Sciences department of Utrecht University kindly provided us with an environment in which we could set up and run the contest. The Information and Computing Sciences department of Utrecht University provided us with the necessary facilities for the contest. Jonathan Jeuring supported Endo to the end.

References

[1] Harold Abelson and Andrea diSessa. *Turtle Geometry: The Computer as a Medium for Exploring Mathematics*. MIT Press, 1981.

[2] Hans-J. Boehm, Russ Atkinson, and Michael Plass. Ropes: an alternative to strings. *Software—Practice and Experience*, 25(12):1315–1330, December 1995.

[3] Ralf Hinze and Ross Paterson. Finger trees: A simple general-purpose data structure. *Journal of Functional Programming*, 16(2):197–217, 2006.

[4] Simon Marlow. Haddock. `http://www.haskell.org/haddock/`.

[5] Przemyslaw Prusinkiewicz and Aristid Lindenmayer. *The Algorithmic Beauty of Plants*. Springer, 1990.

A. Best solution

The following 3903-base prefix was supplied by Jochen Hoenicke of team SwtPl after the contest. It is a perfect solution: it produces the target image exactly. It has a survival chance of about 99.9951%. The best prefix that the organisers made before the contest scored a meager 45.3%.

```
IIPIPCCICCICCPIICIICIPPPIPPPIICIPCICIICICCPFCIIPIFCCCFPIICIICIPPPIPPPIICIIPIPCCICCICCPIICI
IPIFCICIICIIPIFCPIICIIPIFCPIICIICCCICCICIIPCCPCCFICCCFFFCFFCPFICICPCFFFPFFCFFCFIFCPCPFCFPI
FICPICPFFPFCPFICICCCFPIPPCCPFICCCFIFCPPIFCPPCCICFICCPICFICCPICICCCFIICCPCCCCCPIICCCPFCPIIF
ICICICIPICCICICIPCIIICCICPIICCCCIIPCIIIIIICPCCCIICCIPCCIICCIIICPPCPFFFIICPPPIIPCPIFCCCCPCF
PCCCCCIIIIIIIIIIIIIIIIIPICCICIICPIICIICCIPCCIIICCIPCICCICIIPICCICICIPICCIIICIPCICCICIIPIC
IIIIICPCIIIIICIPCICICICIPIICIIICIPCICCICIIPCIICIIICPCIICIIICPCCCCCCCCPCPIICPCPCPPCPFCPFIII
IIICCCCCCCPCCCCPPFIFCICIIIIIICPIICCCPFFPPPICFIFCCIPFFCCCCPCFFCPIIPCFFICCPFFCICPIFFPCPCFFC
IIIIIFFIPICPCFFCICCCPFFCCFFCPIIPFFICPCFFCCPICCCCPIFCCFCICCFCCFIIIICFFIIPPIPIPCFCFPPCCCPPIC
PIIFICCCFPCCIIIIIICCPIICIIIIIIIPCCPIIICPCPCFFCFFCPICPIIIFFIPPIIFFICPICFFCPPIIICCFFIICFFCCP
CPCFFCPIICFFICCCCPCFFICCPFFCCPICPIPIIIIPCFCFCFCPIIIFFPCCCCPPCPCPIIIIFPCFFCCCCCCFFCIPIIFPCF
CFCFFCFFFFIIIPICPCFICFFFFFFICPIFICFCCCCCICPPCCCCPIIIFPFCICCICFIFCCCIPCPPIIIFFIIICPCPPPPIIF
FCCPPIPIPCCCCCCCPCFPCFFFCCCCCCFCPCCFICFCFFCFIIPPCPIICCCPCPFFFFFFIIIPICCPCFFCFFCICPIICCPFPF
CFCFFFICCCPFIFFFCIPPCCCPFPFFCCCFCICCCPFIFCFFPIPICCPIPIICPFIIIFPPIPIPCFFCICPIICPFCCFPICCCFP
FCFCICFFIFCFPIICFFFPCFFPPCFFCPIIIPIIIIPCCPCCFFCPIPFCCFFCCFIIIPICFCFCFIIIPPIIPIFFICFCFIIPIC
PCPPCFICCFFCFCFCFCCPIIIFIIFFCFFCCFCCCCCCPIFIIIFFFCFFFFFCCCCCFFICFIIIFFFCFFFFFFCCCFCCIPIIPP
ICFCCFFFCFIFFCIPIIFCCFCCFCPIIIIIPIFPICFIICCCICICICCCICCIICICICPFIIIFCCIICCCCIIIIIICCFPICFI
CCCIIICCCICICCCCCIIICCPCPIPCCCPCFICFCFCFFCCCCPCFCCFFCCFPIPCPFIFIPPIPICCCPFIIFFFFCFFCCCCCPC
FIFCFFPIIPIFIPFIIICCCICCICICIICFCCFCIICIFCCFICICICFIFIICCFPIFCCCIICICCICCCICPFIFCCIPICCPCP
FPFCFFCCCPIIIFIIFCCFCCCFPCCCPIPCCFFCCPFICFCCFCFFICFCFFCFCFCCFFCCFPCFFFFICFCCCFCCCFCCFCFCCP
FCFFCFICFFFCCCIIPPCCPCFIIFFFFFFFFCPPIIFCCCCFFFCFCCCCFFFFFFFCPCFPIFCCCCFFFCCFCFFFPCCCCCCPPI
IPPCCFICFFFFFCPIIICFPFCFCFFCPPPIFICFPFFFFIFIFIIICCCCCPFIIFIPFFFFFPIPICPCPIPICPCCFPFFCCFFCP
CCCPIICFICFFFFCFIIPICPFIFCFCPPPCPCCFCFFCIIIPCPCFICFCCFCFPPIIIPIPCPCPCFFICPIPCCPPIICPFIIPFF
CCFFFCFICFCFCFFCFICCCFCFFCPICFIPFCFFCFICFFFFFFPIPIIIFPICCFFFFFCFCCFCFCFFFFCCCCICFCCCFCFCCF
FFCCCCCPIPIPCFCCCCFCCCICICCCPIICIICPCCPIIIPCFCCPCFFCFFFFFCFFFCCFCFCFCFCCFICCCFFFCFCCCFCFCC
PIPIIPCFCFIICFCCFPIICIICPPICCPIIFCCPFPICICIICCIIICICICPIICIICIIPIPPPCCCPCFFCPIIICCFPFFFFFC
CIIPCCCPIPFIFFFFCFIFFFFIIIIPCFIICFFCFCFCCFFFFCCCCPFIFFFFCFFFIPIPIFCPFCCIIIIICIICIIFIIFIIIC
IICIFIIFCICICCICFICFIIICCCCFIPFIICICCIIIIIIIIPFCFCIIIIIIFPFIIICICIIPIIFCFIIPIICPCCCPIIIIII
CCFFPCPIICPPCFIFFFCIPIICPCFFFCCCCPFCFCFFIFFFCICPIPIFFFPIIICFCCFCCCFIPICCPIPCFICFFCFCFPPICP
CCFCCFCCFCIPIICFPCFFCFCCCFCFFPPPCPIIFIIFCCFFFFCICPPIIICPPIIFPFFCCFFCPIIIFICFFCCFCFCFCFCPII
CPPIFFCIIPIIIIICFPFCCFFFCIIIIIFCCFFCCCIPIPCFIFCCFPIIPICPCCFIFCFFCPCCPPFIFFCFPCCPCCPICPCCFF
CCCCCPIPICFPPIFICCICCCIIIIICICIICCCICIPIIPIPICICIICICIIPIICIICIICIPPPIPPCPIIPIPICPCCPCFCCF
CIICIIPIPPFPIFIICICIICICIIICIPIICCCPPIFIIFCCCCCICCCPFIIFICCCIICCIIPICICCCCCICPFCIIPIPIICI
CCICPIICIICIPPPIPPPIICPIIPIFCCCFPIICIIPIFCICIICIIPIFCPIICIICCCICCICCFFCFICCCICCPFICCFCCFFC
CFCFCCFFFCFFCFCCICICFFPFCFFPPCFFFPCPFICFICFCFCFCFFFPCFFFFFFFCCFCCICCPFPCFCFFFPCCFCCICCICIF
CPICPCICFFFFFFFFFFICCCFCCFCPICICCCICFICCPICFICCCFCCICCICIIPICPCCICCPFFFPCICIPPCPCCFCCFCCF
FFCFFCFPPFPCFFFPFICPFPFFCFFCFCCICICCPICFICCCFCCFCICFFCFICCCFIPCPPIPCPPCCICFICCPICFICCPFICI
CCCFIICFCCCICCIICCIICCCICCCCICCPCCPCPFPIIPFCPPIFCFIICFCFPCFFCCCFIIFIICCFCPFIICFIIFICFIIIFP
CFPIFCFIIIFCFIIIFFPFPFIIPFCPFPPFIIIFIPFCPFIIIFPPFPFPIFCFIPIFFPFCFCPFICFPIFCFIIFICCFIPFCCFC
PFIPCFIIFPCCFIFICCFFPCFIIFCCFPIFCFIIFIFIPIFIIFCPFCPFPIFIIFIIIFPIFIIIFICFPIFCFPFCCFCCPFFIFI
CFCPFCPCFPIIFCPICFPCPFIFCPFPFPIFCPFPIFIPFIPFPIFPFPFPFPIFIFIFPFFIFCFIPFICFCFFIFIICFCFIIICFI
IICFFICFIFIIICFCCPFPCFCPCPFIPIICFIIFICFICFPCFICFCCFCCFPCFIFICFICFICFFICFIFPICFIIIFICCFCPFC
FIIFCFCCFCPFCIIIPIPIICICCICPIICIIPIPCICCPFIICICIICCCICCICCCFCCCFCFCFFCCCCFCCCCFFICCCICCICF
ICCICIPPCPICCCFCCFCICFICCCICCICCIPCPCPICCCICCICFICCCFCCFCCFIPCPPIPCPPCCICICCPICFICCPICCFIC
CPICFFICCCFIICICIICIICICCIICPIIII
```