Report on the Tenth ICFP Programming Contest

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

Abstract

The ICFP programming contest is a 72-hour contest, which attracts thousands of contestants from all over the world. In this report we describe what it takes to organise this contest, the main ideas behind the contest we organised, the task, how to solve it, how we created it, and how well the contestants did.

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. 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++.

Categories and Subject Descriptors D.3.0 [*Programming Languages*]: General

General Terms Design, Languages

1. Introduction

The Tenth Annual ICFP Programming Contest was a 72-hour contest held 20–23 July 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 helps Endo, an alien life form stranded on earth, to survive. The DNA is a sequence of the letters `I`, `C`, `F`, `P` that encodes a program. In each reduction step, the beginning of the string is interpreted as a search-replace operation with a regular expression as pattern, and the operation is used to modify the rest of the string. In addition, specific sequences in the DNA generate commands to draw a picture. The contestants were given an operational semantics of DNA and an initial DNA string which produces an image of Endo and its immediate surroundings – the "source" image given in Figure 1. The "target" image in Figure 2 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;
- the energy consumption of the synthesis, i.e., the time and space complexity of performing the algorithm – in this case only a limit was given.

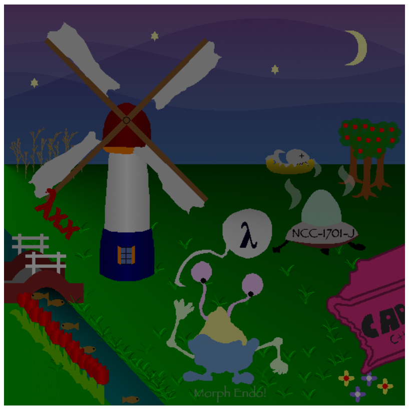


Figure 1: Source image

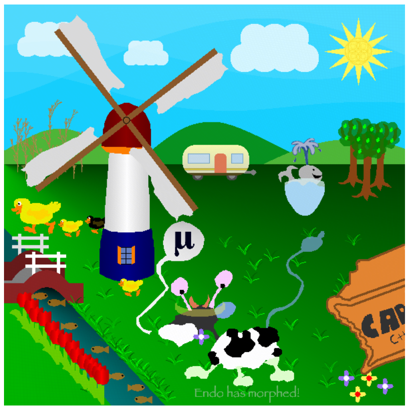


Figure 2: Target image

[*] Department of Software Technology, Delft University of Technology, e.dolstra@tudelft.nl

[†] Software Technology, ICS, Utrecht University, {jur, stefan, johanj, andres, ariem, alexey, jcschie}@cs.uu.nl

[‡] Open Universiteit Nederland, bastiaan.heeren@ou.nl

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

ICFP'08, September 22–24, 2008, Victoria, BC, Canada.
Copyright © 2008 ACM 978-1-59593-919-7/08/09...$5.00

In this report we describe what it takes to organise an ICFP contest, the main ideas behind the contest we organised, the task, how to solve it, how we created it, and how well the contestants did. Section 2 discusses the organization of the ICFP contest, including the main design choices we made for this edition of the contest. In Section 3 we give a summary of the story behind the task, and we have a closer look at the DNA language and the algorithm that turns DNA into a picture. Endo's specific DNA string has some peculiar properties, and in Section 4 we show how these properties can be used to obtain the target picture from the source picture. We also discuss some alternative approaches to producing the target picture. Section 5 discusses the design choices behind our DNA and RNA languages, and it shows some of the languages and tools we built to produce the task for the contest. In Section 6 we look back on the actual contest, give statistics about the participants, and announce the winners. The last section presents the aftermath and our reflections on organising the contest.

2. The ICFP contest

Organising an ICFP programming contest is a challenge. To allow contestants to demonstrate the superiority of their favourite programming languages and programming skills, the contest should at least test language capabilities and programming skills, and should require intelligent behaviour. Moreover, we wanted many teams to get somewhere into the contest, and at least a number of teams to get very far. On the other hand, getting very far should not be a direct consequence of being very knowledgeable about any niche domain within or outside computer science. Finally, and most importantly, the contest should be fun!

2.1 Requirements

Recent contests, including this one, had thousands of contestants, working from anywhere in the world, with very different backgrounds. The idea behind an ICFP contest is that contestants may use any tools they like, and any information at their disposal. Furthermore, teams may be of arbitrary size, although in some previous editions of the ICFP contest restrictions on team size were imposed. Organisers have to think about the following practical issues to cope with these "contestant-friendly" rules:

1. The programming task should not be easily solvable.
2. The programming task should not be solvable by using a lot of computing power.
3. A solution to the programming task should not be lying around (or for sale) somewhere.
4. The fact that contestants are going to use many different programming languages and compilers needs to be dealt with.
5. The solution space of the programming task should be large: we don't want many teams to submit the same, correct, or best solution.
6. There should be a clear way to determine the winner of the contest: the solution space of the task should have a total ordering, preferably without a top (or bottom).

Our problem satisfied all of these requirements, except for requirement 6: since the solution with the lowest score wins, there obviously is a bottom. However, this bottom is very hard to find, and the prefix that leads to the best possible score is not even known to be unique. Indeed, it may well be that the best score is obtained for a prefix that does not even generate a pixel-precise image. At the moment we know that the best score lies somewhere in between 1 and 3685, and we conjecture it to be closer to 3685 than to 1. The problem that contestants use many programming languages (requirement 4) is solved by designing our own low-level programming language, namely DNA. Some of the previous contests (2004, 2006) used a similar approach to this problem. We did not reuse an existing low-level language to meet the third requirement. Satisfying requirement 5 also helped us satisfy requirement 2: the solution space was so large that a brute-force approach of simply trying prefixes quickly becomes infeasible.

2.2 Design choices

We started thinking about ideas for the task in November 2005, but it wasn't until August 2006, after participating in the ICFP contest 2006, that we began to put real effort into it. We settled for the task of *reverse engineering* a large piece of complex low-level code.

We tried to devise a task in a way that contestants had to do the following:

- Understand how high-level languages are translated into low-level code, in order to get an idea of what the high-level language from which the code is generated looks like. Use this understanding to patch and/or generate low-level code.
- Write an efficient interpreter, using an advanced data structure, and test the interpreter on some small examples that would reveal sufficient information to start writing tools.
- Write debuggers, disassemblers, analysers, and whatever means to understand and repair the low-level code.

Contestants had to submit a piece of low-level code, which patches the original code we published. We scored patches on size and level of correctness.

We wanted the best tool writers with a good understanding of programming languages and compilers to win the contest. In Section 3.3, we explain in detail the measures we took to ensure this.

3. The task

In this section, we present the contest task. We briefly summarise the background story. Then, we explain the design of the DNA language. We also evaluate how our task relates to the design choices listed in Section 2.2. The full task description is available as a technical report (Dolstra et al. 2007a).

3.1 Background story

Endo, an alien life form belonging to the species of the Fuun, has crashed on Earth by accident. The pictures shown in Figure 3 show how. Endo has been severely hurt by the crash, and in addition, his physique is not adjusted to Earth's environmental conditions. Endo's intelligent spaceship called Arrow devised a plan to save Endo by mutating its DNA, but being damaged itself, it could not perform both the transformation itself and the search for a suitable DNA modification in the little time until Endo's imminent death. Arrow therefore contacted the ICFP contest community via us.

3.2 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.

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.

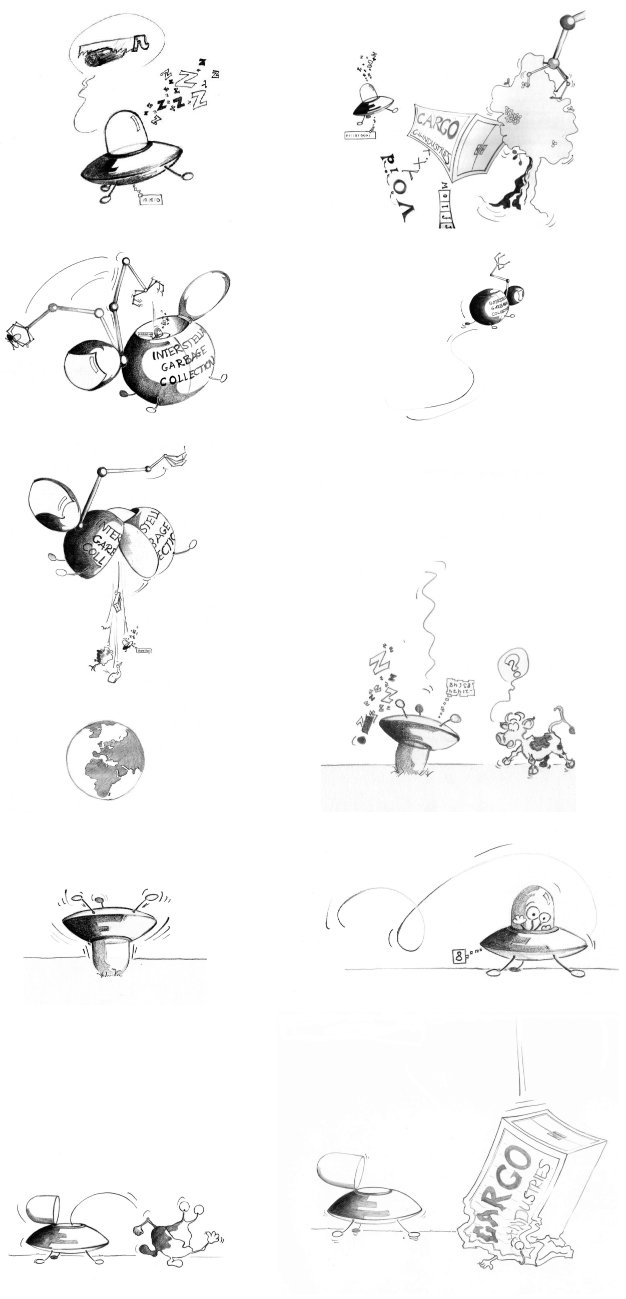


Figure 3: Arrival sequence

3.2.1 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 that an RNA command is to follow; the RNA command is then output immediately.

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

Figure 4: Pattern language

$\underline{\texttt{I}}, \underline{\texttt{C}}, \underline{\texttt{F}}, \underline{\texttt{P}}$	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

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 that matches 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` and `IIF` both denote the end of a group, 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 $(!_2)\underline{\texttt{P}}$, i.e., start a group, skip two bases, end a group, match a `P`. The middle part is interpreted as the template $\underline{\texttt{P}}\underline{\texttt{I}}0_0$, i.e., insert `PI`, and insert what was matched against the first group. The match-replace command, which can be written as

$$(!_2)\underline{\texttt{P}} \mapsto \underline{\texttt{P}}\underline{\texttt{I}}0_0$$

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 I and F both represent 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 5.3 and 5.4, 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 Section 4.1 and Section 5.1.

3.2.2 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 4.3).

The 20 commands form an extended 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 colour, 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 1.

Changing the colour with a limited set of commands while allowing a full range of 8-bit RGBA values is achieved by maintaining a colour *bucket*. There is one command to empty the bucket, and there are ten commands to add different base colours to the bucket. Each colour can be added to the bucket multiple times if desired. The currently active colour is then given by the average of all the colours 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 colour $(254, 255, 255)$ requires 255 RNA commands.

3.3 How our task satisfies the design choices

In Section 2.2 we described a number of design choices for our task. Here we describe how our task satisfies these design choices.

Reverse engineering In the contest, we wanted to encourage reverse engineering of the given DNA string. The DNA string is not a straightforward drawing of the source picture, but contains a significant amount of structured code, including functions that can be adapted in various ways, code that is unused in the original string, but might be helpful for the target picture, and documentation.

We included a hint in the task description that the DNA might contain messages and hints from the creators of Endo as well as genes that might help with the transformation.

Despite all this information, a brute-force approach remains attractive: if the target image can be generated from scratch, there is no need to look at the given code and to understand its structure. We did not want to disallow the brute-force approach, and we wanted participants to follow different routes – however, we decided to make this approach more difficult in order to encourage teams to use reverse engineering. We certainly did not want teams to feel that they had wasted time by looking at the internals of the given DNA.

Naive redrawing of the target by just RNA commands is prevented by choosing a very inefficient encoding of RNA: ten bases each for just 20 commands. We also designed the target picture such that there are no large areas of the same colour that can be flood-filled. Instead, we use gradients in many places. Furthermore, we placed a Moiré pattern on top of the image to make compression harder. We also checked that the PNG encoding of the target image has reasonable size (235 KB).

In the contest, this balancing seems to have worked quite well: brute-force approaches were tried and were even able to compete, but they were not superior. All teams that managed to hand in a good brute-force solution were required to also do some DNA programming – for instance, some participants managed to extract the Moiré pattern.

Writing an efficient interpreter We provided the possibility to see the best picture so far to registered teams. We decided against having a full web-based machine or freely providing reference code for a machine. This was not just for practical reasons, but also because teams without their own machines would have a hard time writing debugging tools, which profit from being integrated into the machine. Therefore, the choice of language for a running reference implementation would have put a strong bias on the contest.

Implementing the interpreter consisted of following a relatively long semi-formal description correctly, and choosing the right data structure to manage the DNA string. We warned in the task description that executing Endo's DNA performs 192646205 reduction steps in the machine.

Choosing the wrong data structure can easily lead to machine implementations that perform less than a hundred iterations per second. However, it certainly isn't necessary to engage in heavy bit-fiddling, hand optimisation, or assembler programming to get a reasonably fast machine: our straightforward, 347-line Haskell reference implementation using finger trees (Hinze and Paterson 2006) takes about 50 seconds for drawing the initial images. Our optimised C++ implementation using a variation of ropes (Boehm et al. 1995) takes about 5 seconds. Despite the hint about using a suitable data structure in the task description, writing an efficient machine turned out to be harder than we thought for many contestants.

Exploration In order to keep the already long task description manageable, we decided to include documentation on how Endo's DNA is structured not into the problem description, but into the DNA itself. We gave one prefix (a self-test) to try in the description, and made it easy to discover, with a bit of analysis, more prefixes from there that would lead to documentation pages. The documentation includes many hints on the machine model that most of the DNA code uses, the locations of specific functions, calling convention, etc. In addition, there were some puzzles for advanced parts of the target picture, for which some additional work or debugging would need to be performed. All this is explained in more detail in Section 4.

During the contest, it turned out that many participants had difficulties in interpreting the algorithm to generate prefixes for documentation pages, and considered this a stumbling block. It was possible to continue and start writing tools without this information, or to find out by simply analyzing the code, but many teams did not. At this point the structure of the task was thus probably too linear.

Tools To generate the target picture, it is helpful to analyse all aspects of the DNA machine, by tracing the execution, tracing parts of the changing DNA string, and observing the generated RNA instructions.

Once documentation is found, more possibilities become available: with knowledge of the machine model, it is possible to trace function calls, observe values of mutable variables, extract encoded strings, find hidden documentation pages and so on.

Unfortunately, since the task was to submit prefixes, and only relatively few teams submitted their code for the judges prize, we have no clear overview on how many teams wrote what tools. What we can say is that the top teams, whether using a brute-force or a reverse-engineering strategy, all wrote tools to help them. The fact that many secrets and some easter eggs in the DNA were found during or soon after the contest also indicates that a significant amount of analysis was performed on the DNA.

4. 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 2. How would we proceed to solve this task?

4.1 Getting started

The task description says that "something curious" happens if the prefix `IIPIFFCPICICIICPIICIPPPICIIC` is used. Obviously, we should try this first. *If* our DNA machine is correct, 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.

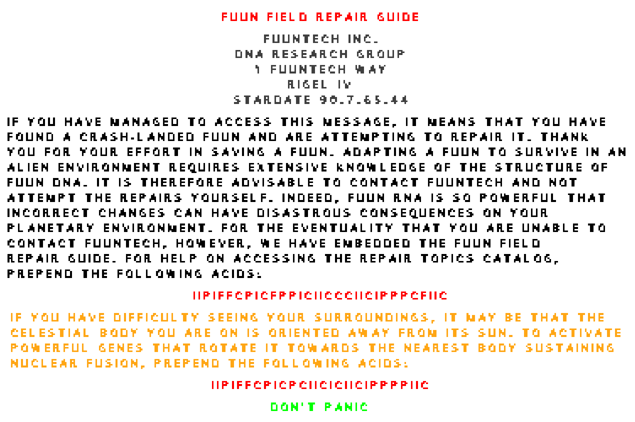
Of course it is good to know that the machine is at least partially correct, but it doesn't really help us get further (except that it is now clear that there are things hidden in Endo's DNA). So maybe we should look more closely at the DNA. It starts with `III` – an RNA command. In fact, there are thousands 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 the DNA machine allows us to step through commands interactively, like a debugger – an indispensable tool for reverse engineering. When we do, we see that a message is drawn before it is overwritten by a black flood fill:

```
IIPIFFCPICFPPICIICCIICIPPPFIIC
```

There are other ways to discover this prefix. In fact, it is even possible to see the hidden prefix by accident if, for instance, flood fills or bitmap operations do not work correctly yet; or if the machine is just very slow (which was the case for many contestants). 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 designers 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 6.3) 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 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



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

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 task advises. Thus, contestants would be stuck at this point unless they fixed the time complexity of their machine.

Given that we have a DNA machine by now, we proceed to disassemble a known repair guide prefix. 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` $\mapsto$ `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 single `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`, and the necessary prefix would be

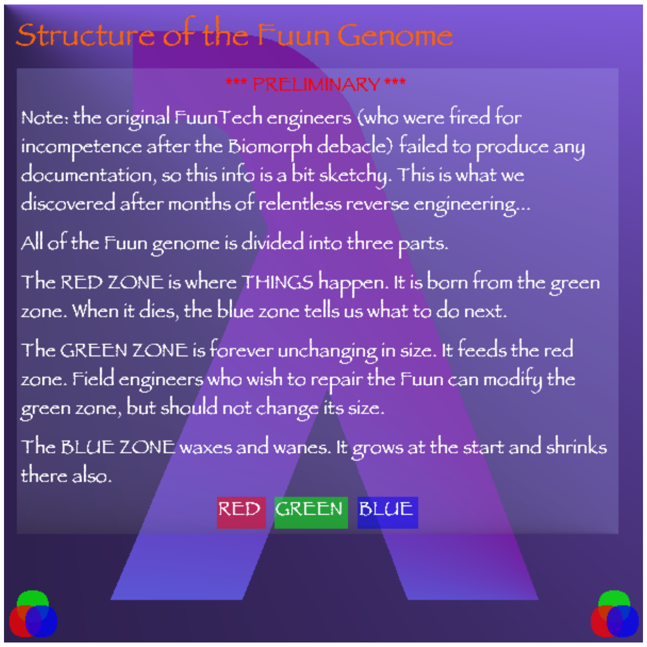
$$(?_{\texttt{IFPCFFP}})\underline{\texttt{IIIIIIIIIII}} \mapsto 0_0\underline{\texttt{CIICCCIICIC}}$$

or, in concrete DNA,

```
IIPIFFCPICFPPICIICCCCCCCCCCCIICIPPPFCCFFFCCFCFIIC
```

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. 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.

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. 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



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...

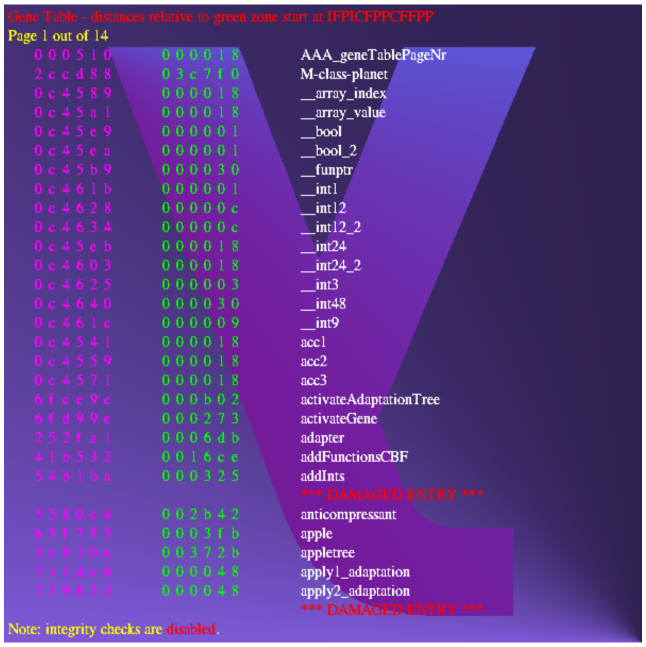
All of the Fuun genome is divided into three parts.

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: Repair guide page with strange terminology



Gene Table - distances relative to green zone start at IFPICFPPCFFPP

Page 1 out of 14

0 0 0 5 1 0	0 0 0 0 1 8	AAA_geneTablePageNr
2 c c d 8 8	0 3 c 7 f 0	M-class-planet
0 c 4 5 8 9	0 0 0 0 1 8	__array_index
0 c 4 5 a 1	0 0 0 0 1 8	__array_value
0 c 4 5 e 9	0 0 0 0 0 1	__bool
0 c 4 5 e a	0 0 0 0 0 1	__bool_2
0 c 4 5 b 9	0 0 0 0 3 0	__funptr
0 c 4 6 1 b	0 0 0 0 0 1	__int1
0 c 4 6 2 8	0 0 0 0 0 c	__int12
0 c 4 6 3 4	0 0 0 0 0 c	__int12_2
0 c 4 5 e b	0 0 0 0 1 8	__int24
0 c 4 6 0 3	0 0 0 0 1 8	__int24_2
0 c 4 6 2 5	0 0 0 0 0 3	__int3
0 c 4 6 4 0	0 0 0 0 3 0	__int48
0 c 4 6 1 c	0 0 0 0 0 9	__int9
0 c 4 5 4 1	0 0 0 0 1 8	acc1
0 c 4 5 5 9	0 0 0 0 1 8	acc2
0 c 4 5 7 1	0 0 0 0 1 8	acc3
6 f c e 9 c	0 0 0 b 0 2	activateAdaptationTree
6 f d 9 9 e	0 0 0 2 7 3	activateGene
2 5 2 f a 1	0 0 0 6 d b	adapter
4 1 b 5 3 2	0 0 1 6 c e	addFunctionsCBF
5 4 b 1 b a	0 0 0 3 2 5	addInts
		*** DAMAGED ENTRY ***
5 5 8 0 c 4	0 0 2 b 4 2	anticompressant
6 5 f 7 8 5	0 0 0 3 f b	apple
3 c 8 7 0 e	0 0 3 7 2 b	appletree
7 1 1 d c 6	0 0 0 0 4 8	apply1_adaptation
7 1 9 6 3 3	0 0 0 0 4 8	apply2_adaptation
		*** DAMAGED ENTRY ***

Note: integrity checks are disabled.

Figure 8: Gene list (or symbol table)

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,

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

where $!_n$ denotes a skip over n bases. In DNA, this would be:

```
IIPIFFCPICCFPICICFPPICICIPCCIIIIIC
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.

4.2 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.

For instance, there is a variable $polarAngleIncr$, which determines the rotation of the blades of the windmill. How could you know? 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 $windmill$ calls $drawPolylinePolar$. 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:

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

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).

Not every change can be made by changing a variable. Some changes 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:

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

where $[\mathsf{pattern} \mapsto \mathsf{template}]$ 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,

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

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.

4.3 Calling convention

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$nnnnnn$, where each n is one of I, P, or C. A bit of analysis (plus a 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 ns denote a unique function number in base-3 notation. Thus, these RNA sequences reveal the dynamic call graph within Endo's DNA.

4.4 Memory model

When we step through the DNA code, and from the help screens and the gene table, we should get a picture of the operation of the DNA, which is useful – we 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 is a useful 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 we know from standard Von Neumann machines. But there are many similarities, and it is only 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 cannot 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 is where the red zone comes in: it is 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 (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 we have to be very careful about modifying DNA. We cannot insert code into the green zone, since that would invalidate offsets. We have to be very careful to also discard the current red zone code when calling functions. And when we call a function from a prefix, we do not have a return address in the green zone, unless we patch the green zone first. Calling functions from a prefix is therefore 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 we do not have when calling from within a prefix).

4.5 Secrets

Endo's DNA contains many secrets that can help to produce a short prefix fixing as many pixels as possible. There are documentation pages for the imperative and functional languages used to build Endo's DNA, an audio prefix, L-systems, spirographs, a virus, easter eggs, and many more secrets hidden in the DNA. A brief description of the secrets can be found in a technical report (Dolstra et al. 2007b).

The secrets in the DNA turn the contest into an adventure style game, in which problems have to be solved at several levels in order to produce a good prefix. Finding secrets and using them to solve a problem requires advanced programming skills, more than required for the development of the DNA to RNA machine. Here participants can demonstrate their programming skills and the capabilities of their favourite programming language.

The secrets vary in difficulty: some are very easy to find and solve, others are much more difficult. Some of the easier secrets are included to give away some helpful information about the structure of the DNA. We expected the contestants to find these secrets early during the contest, and we hoped these would help the contestants to get started. An example of such an early secret is the sequence of RNA commands at the start of Endo's DNA, eventually leading to the daylight prefix and the catalog page. For this, the contestants had to go through a sequence of four steps.

Once the catalog page is found (and the gene table listed on this page), many more secret pages become accessible, including the more difficult ones. In most cases, the challenging secrets correspond to the more valuable information, such as cheap fixes for certain parts of the picture, or keys to unlock some other information. To avoid giving some contestants an advantage, we included secrets about a variety of topics, so that it is extremely unlikely that a contestant is an expert in all of these topics. Although we carefully designed the paths in which we expected the secrets to be explored, we anticipated that the information could also be found in unforeseen ways.

5. Creating the task

Section 2.1 explains that we used a low-level programming language because we wanted contestants to reverse engineer a large piece of complex low-level code, and because we had to deal with contestants using many different programming languages. This section discusses how we arrived at the design of our low-level code languages DNA and RNA. Since it is extremely hard to program in these languages, we developed higher-level languages and tools to produce this low-level code. This section discusses these languages and tools. Knowledge about these languages and tools makes it easier to reverse engineer the low-level code. Part of this information can be found in the low-level code we handed out, see for instance sections 4.3 and 4.4.

5.1 The DNA language

The initial DNA language looked a lot like regular expressions. We verified that all necessary programming features – variables, loops, conditionals, stacks, functions – could be implemented using regular expressions.

Programming in pure regular expressions turns out to have some challenging implementation issues. 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[1].

The solution is the skip operation, which allowed us to have 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 (Boehm et al. 1995), or even just a sufficiently short list of strings, programs can be executed efficiently.

Still, Endo's DNA requires a lot of arithmetic. Initially we used Peano arithmetic, which for some uses (such as a small loop

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

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 added 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 list of instructions that "grow" his DNA to 2^{24} bases.

But now subtraction was a bottleneck, so we used a final trick 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.)

5.2 Making pictures with RNA

The RNA language is inspired by turtle graphics (Abelson and diSessa 1981), although ultimately our "turtle" is primitive in some respects and advanced in others. 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.

5.3 The Imp language

In terms of ease of programming, Fuun DNA lies somewhere between assembler and Turing machines. Therefore, we designed 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 conventional C-like imperative language, except for some tricky details. For instance, you cannot 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 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      -- return type
    [intArg "x"]                 -- parameters
    []                           -- local variables
    [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 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.

5.4 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 nice 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.

5.5 Tools

We developed a lot of tools to generate, execute, and analyse DNA programs. For this, we used many 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 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, doing flood fills, and composing bitmaps were also implemented in C. This renderer also supported a debug option to see some intermediate bitmaps.

Being essential to our contest, the renderers and the DNA machines were implemented in different languages and by different people (N-version programming), to increase our confidence that (1) the machine specifications were correct and unambiguous, and (2) both machines could be implemented with comparable effort in a variety of languages. For example, if different people, using widely different languages, can implement a renderer from the same specification, and experiments show that these give exactly the same results, then this increases our confidence that (1) holds.

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 descriptions in this EDSL were translated to the Imp language and from there into DNA.

Font generators Endo's DNA sequence contains three embedded fonts. In fact, these fonts were embedded 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: colours are slightly changed to simpler colours (requiring fewer RNA commands) and larger areas of one colour 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 colour 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.

6. The contest

The contest took place from 12:00 (noon) CEST on 20 July 2007 till 12:00 on 23 July, 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, but also to make sure that teams did not use our DNA machine as an alternative to constructing their own. This was also the reason why we did not allow teams to register twice.

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, again 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.

#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

6.1 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, for those teams that gave this information. 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.

6.2 During the contest

As described before, teams could submit entries every ten minutes, but only the result of their best submission would be shown. This discouraged teams to submit anything but their current best prefix to us. Since we expected teams to implement their own DNA machine, our data on how the contest proceeded from the perspective of the participating teams is limited.

Teams started submitting shortly after the contest opened. Team escape started very early submitting seemingly random strings of relatively short length (mostly length 200 or 100) nearly every ten minutes – they submitted 273 times throughout the contest. Most of

#teams	Languages
81	C++
67	C
66	Haskell
64	Python
52	OCaml
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 mentioned by the teams

these pictures resulted in a completely black or a completely white square.

During the first hours we often saw the prefix given in the task description that produces the reference output for the self-check. This picture could actually be seen by the contestants as it results in a slightly better score than the beginning picture.

Teams were then obviously trying to come up with implementations of the DNA machine, since this was necessary to get to the first help screens and the "daylight" picture. The daylight picture produced the first noteworthy improvement in score, and the first team to submit this prefix of length 28 was camelimelo, seven hours into the contest. After twelve hours, eight teams had submitted daylight.

After 15 hours, team Smartass was the first to improve on the daylight score by finding a prefix of length 27 that results in the same picture, improving their score by 1. In the remaining time of the first day, we saw teams submitting some help screens, and teams trying brute-force approaches, but none of this was leading to any measurable success yet. After 24 hours, team PurelyFunctionalInfrastructure had also found this shorter prefix, and more than 20 teams had found the daylight picture, thus the first teams with a non-zero chance of Endo surviving became visible on the scoreboard (the top 20 were shown without score and in random order). Since Endo would never have survived after only 24 hours of work and we had essentially a tie at the time, we decided not to hand out the "lightning division" prize.

The first significant improvement over daylight was submitted by PurelyFunctionalInfrastructure 36 hours into the contest: they managed to change the colour of the cargo box correctly. Only six minutes later, United Coding Team changed the trees correctly and then ranked second. After four more hours, a few more teams were beginning to improve upon daylight. One of them was jabber-ru, one of the most successful brute-force teams. Team jabber-ru tried to approximate the target picture with hand-drawn polygons. Despite being a one-man team, he submitted very often, making small incremental improvements to his picture all the time, sleeping at most a couple of hours during the contest. Figure 9 shows the score development of the top teams from hour 35 on.

We kept the contest office manned by at least two persons throughout the contest at all times. We monitored the submissions, looked at discussions taking place on the internet, and answered questions on the official mailing lists. Some clarifications regarding the task description were asked for, but, fortunately, no bugs were found. The submission system was offline once for about fifteen minutes, but worked flawlessly for the rest of the contest. We realised that implementing the DNA machine was providing a bigger hurdle for the teams than anticipated. After multiple requests, we therefore decided to publish an execution trace of the first ten iterations of Endo's DNA online, in hour 37.

After 48 hours, a small number of teams was fiercely competing for the top spot. Teams pursuing brute-force approaches were competitive with those reusing the given DNA, but at most times, one of the reusing approaches was leading. It was not at all obvious who would win.

In hour 49, the first teams were beginning to discover the Biomorphological Unit (BMU) and playing with different shapes of Endo.

At the same time, slowly, more and more teams found daylight. However, even after 55 hours, there were still less than 20 teams with a score higher than daylight. We noticed that in certain discussion channels frustration rose because the scoreboard still did not show a single team that had achieved more than the daylight score. In order to give away at least this positive message, we decided to lower the amount of hidden scoreboard entries from 20 to 15.

This step also made the 27-length daylight prefix widely known for the first time. Until then, six teams had found this small variation. After the knowledge about the score became public, several teams managed to come up with this prefix as well – 14 of them within the next hour.

Shortly thereafter the first teams having significantly higher scores than daylight appeared in the public part of the scoreboard, sparking some optimism among the teams.

In hour 65, team cultboundvariable managed as the only team throughout the contest to submit a prefix that produced the target picture exactly. However, the prefix was more than seven million bases long, thus the score was not competitive.

The last few hours of the contest were by far the most exciting. Several teams managed to get significant improvements of their score shortly before the end of the contest. Most amazing probably was Celestial Dire Badger who made a few big leaps within the last ten hours. Several teams, including Team Smartass, discovered that simply eliminating incorrect elements of the source picture already improves the score significantly, which is part of the reason that the winning picture looks relatively empty.

6.3 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 $100\ e^{-1\ (0.000018\ \text{score})^2}$.

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.

Quite a number of teams were using various brute-force approaches to draw Endo. 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. This resulted in the

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

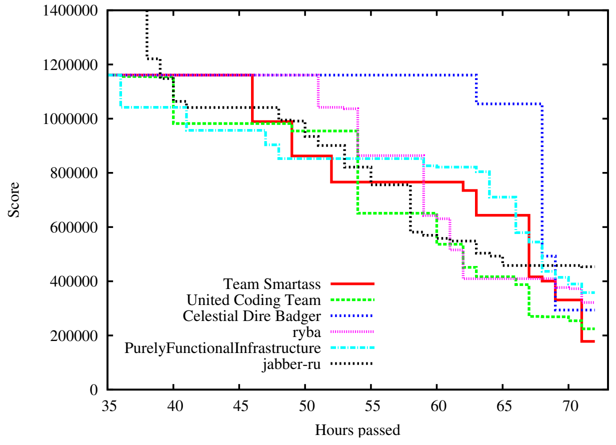


Figure 9: Scores of the Top 6 during the contest

third-best survival chance (75.59%). Therefore the jury is happy to declare that

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

Second prize The second-best survival chance of 84.92% was achieved by *United Coding Team* (Cape Town, South Africa). The team used various languages to implement tools to execute, reverse-engineer and debug Endo's DNA and to generate prefixes: Python, C++, Unix shell scripts, but primarily Perl. Therefore the jury is pleased to declare that United Coding Team 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 best survival chance during the contest, 90.22%, was accomplished by *Team Smartass* (Mountain View, California). They used C++ to implement the DNA/RNA simulator and for various reverse engineering tasks, as well as Python for reverse engineering and generating prefixes. The team identified C++ as the primary language used for the contest. The jury is thus honoured to declare that Team Smartass 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.

7. Conclusions

Many contestants posted a blog message about their experience with the contest. For example, Gallotta (Gallotta 2007) provides a list of dozens of blogs, including post-mortems of all teams that ended up in the top ten. Together, these blog messages give a good idea of the experience of participating in the contest. We observed that most of the secrets we included in Endo's DNA were discovered. The messages often not only discuss technical issues, but also give an evaluation of what the participants liked and disliked. Based on these messages, and other reactions we received after the contest, we found that reactions to the contest were mixed. The majority of the reactions was positive to very positive, in particular from the teams that did well in the contest. Still, participating in the contest was not a pleasant experience for everybody, with some blaming the contest, some blaming themselves. We had hoped for more teams to get further than they did. In retrospect, having a few more easier secrets to solve would probably have helped in this respect. All in all we are happy with how the contest went.

Since we did not put a limit on the size of teams, we were happy to see that the size of the teams did not seem to correlate with how they scored: the three prizes we handed out were for a single-person team, a team of moderate size, and a very large team.

One thing that struck us during and after the contest, reading IRC channels and blog postings, was that many programmers have little 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 here wasn't the language but algorithmic complexity: a straightforward Haskell implementation using the right data structure, such as *Data.Sequence* (Hinze and Paterson 2006), would be fast enough and outperform by several orders of magnitude an optimised C implementation using an unsuitable data structure. So programmers should worry less about languages and more about complexity.

Quite a few blogs showed that people continued to work on the contest after the deadline passed, as witnessed by for example the submission of Jochen Hoenicke of team SwtPl (see Appendix A). Furthermore, we have seen libraries appear that would have been useful in the contest: pattern matching in Haskell's bytestrings, and finger trees in OCaml.

As far as functional programming is concerned, functional languages didn't fare too well (although in the Top 15 there were five users of OCaml and three of Haskell).

So what happened to Endo? Thanks to the hard work of the contestants, Endo survived. 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 organising 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. Rinus Plasmeijer commented on an earlier version of the report. Jonathan Jeuring supported Endo to the end.

References

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

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

Eelco Dolstra, Jur Hage, Bastiaan Heeren, Stefan Holdermans, Johan Jeuring, Andres Löh, Arie Middelkoop, Alexey Rodriguez, John van Schie, and Clara Löh. Morph Endo! Task Description of the Tenth Interstellar Contest on Fuun Programming. Technical Report UU-CS-2007-027, Department of Information and Computing Sciences, Utrecht University, 2007a.

Eelco Dolstra, Jur Hage, Bastiaan Heeren, Stefan Holdermans, Johan Jeuring, Andres Löh, Arie Middelkoop, Alexey Rodriguez, John van Schie, and Clara Löh. Morph Endo! Report on the Tenth Interstellar Contest on Fuun Programming. Technical Report UU-CS-2007-029, Department of Information and Computing Sciences, Utrecht University, 2007b.

Marco Gallotta. ICFP: How we reached the top 15. Blog message on http://marco-za.blogspot.com/2007/07/icfp-how-we-reached-top-15.html, July, 24 2007.

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

A. Best known solution

The following 3685-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 99.9956%. The best prefix that the organisers made before the contest scored a meager 45.3%.

```
IIPIFICCFPIICIICIPPPIPPPIICIPCCCCICICPFCIIPIFICCFPIICIICIPPPIPPPIICPIIPIFICCFPIICIIPIFIIC
IICIIPIFIPIICIIPIFIPIICIICCCICCICFIFCPCCPCCFCCICCICIFPCPCICFIFCPICPCCFCICFIFCPCCPCCFIFCPPI
FCPPCCICFICCPICFICCPICICCCFIICFCPIPPIIPCCCPFCCCPFFICCCFCPCFCFCCFCPIICFCCFCPICCFCCFIICPPCPF
FFIICPPPIIPCPIFCCCPPFPCCCCCIIIIIIIIIIIIIIIIIPICCICIICPIICIICCIPCCIIICCIPCICCICIIPICCICICI
PICCIIICIPCICCICIIPICIIIIICPCIIIIICIPCICICICIPIICIIICIPCICCICIIPCIICIIICPCIICIIICPCCCCCCC
PIIIPICPCPCPPCPCFPIFIIIIIICCCCCCCCPCCCCPPFIFCICIIIIIICPIICCCPFFPPPICFIFCCIPFFCCCCPCFFCPII
PCFFICCPFFCICPIFFPCPCFFCIIIIIFFIPICPCFFCICCCPFFCCFFCPIIPFFICPCFFCCPICCCCPIFPFCICCIFPFIIIII
CFFIIPPIPIPFIFPPICCCPPICPIICFCCPFPCCIIIIIICCPIICIIIIIIIPCCPIIICPCPCFFCFFCPICPIIIFFIPPIIFFI
CPICFFCPPIIICCFFIICFFCCPCPCFFCPIICFFICCCCPCFFICCPFFCCPICPIPIIIIPFIFCFICPIIIFFPICPCCCCPICPI
IIIFPFFFFFFPPIFPFCCCCCPICPCCCCPIIIFIPFCICCICICCCICCPPPIIICFFIIICPCPPPPIIFFCCPPIPIPCCCCCCCP
FIPFFFCCCCCCFCFIIFPFCFFCFIIPPCPIICCCPCPFFFFFFIPPICCPCFPFCCCCCPPIICCPFIIFPCFCFFFPCCPFIFFFCC
PPCCCPFIIFFCCCFCCICCCPFIFCFFPICCPCPIPIICPFIIPFFPFCCCFIIFCPCPPPFPIIIFFPIFIPFPPIICFFICFPPICC
FFCPCFFFPCFFFIIPCPIIIIPCCPFFCPIPFPFFCCFCIIIPIFIFCFCIIIPPIIPIFFIFIFIIIPICPCPPCCFCPFFCFCFCFC
CPIIIFIIFFCFFCCFIPCCCPICFIIFFFCCCCFPFIFFFCCFIIFCCCFFFCPFIFCCCIIIPPPIFPFFFCFCIFFCIPIIFPFCCF
CFIICCPIIPFIPFCCCFCCCFFFFCPIICFIPFCFCFFFCCFFCPPPIFIIPFIICCCICICICCCICCIICICIICPFIIIFCCIICC
CCIIIIIICFIIPFICCCIIICCCICICCCCCIIICIIPCPIPCCCPFPFCFCFFCCCCPCFPFFCCFFPIPCPFIFIPPIPICCCPFII
FFFFCFFCCCCCPCFIFCFFPIIPIFPIFIIICCCICCICICIIFPFICIICIFPFPICICPFIFIICPCFCPFCCCIICICCFPFICCC
ICFIFCCIIPCCPCPFIIFCCFFFCCCCCCPFIFCFCPCCCPIPCCFFCCPCFPFCCFCFCFPFCFFCFCFPFFCCFFPCFFFCFPFCCC
FCCFPFFCFCCPFIFFCCCFPFFFCCCPCPIICPCFIIPFIIICIICIICCCCCCCIICCIICPICCFCPFICCCICICIPFIFCCCPIP
FPFCPIICPCPCCCPPIIPPFPFFFFFCIPIICFIIFCFCFFCCPPPICFPFPFFFFIFIFIIICCCCCPFIIFIPFFFFFPIPICPCPI
PICPFIIFFCCFFCCPCCCPIICCFPFFFFCFIIPICPFIFCFCPPPCPCFIFFCCCCPICPFPFCCFCFPPIIIPIPCPCPCFFIPPPC
CPPIICPCFPIIFICFCFCFCFCICCFFCFCFCICFCCCFFCFIPIFCFFFFCCICFCCCCCCFICFFFCCFFCPPFPFCCFFCIFIFFF
FPPPIIFIIPFCFFFFFCFFFCCFCFCFCFCCFIIICPIIPFCPFCCCIIICPIIIPPIIIPCFPPFFFCFFFFCFCCFCFCFFFFCCPI
PIIPFIFIICCFPFPIICIPIPPICCPIIFICPFPICCIICCIIICPIICIICCPPICCPFIIFCICCIICIIPIIFIFIICCPIIPIPC
CCCPCFFCPIIIFIIFFFFFCCCIIPCCCPIPFIFFFFCFIFFFFCCCPIFIPFFCFCFCCFFFFCCCCPFIFFFFCFFFCPPIFCPPFF
FCCCCCFCCFCFCCCCCFCCFFICFCFCFFCFCCCCCCFFICPCCCPCPIIIIIIFFPIIPICPPCFIFFFCIPIICPCFFFIIIIICFI
FCFFFIFFFCICPIPIFFFPIIICFPFCCCFCPIICCPIPFPFFCFCFPPICPCCFPFCCFCCIPIIFIPFFCFCCCFCFFCPPPCPIIF
IIFCCFFFFCPCCCPIIIICPIICFPFFCFCFIIIIIFPFFCCFCPIPIIIICFIIFCFCCFFCCPIIFCPFPFCCFCFCFIPPCPIFFF
FFCIIIIIPPIIFIFCCFIIIPIICPCFFIIPCCPPFFIIIPICCCPICPCFFCCPCPIPIFIIPIFCCCIIIIICICIICCCICIPIIP
IPIICICICICIIPIICIICIICIPPPIPPCPCPIPIPICPCCPCFPFCIICIPPIPPCFIIIFIICICIICICIIICCIPIICCCPPIF
IIFCCCCCCICCCPFIIFICCCIICCIIPCCICCICIICPFCIIPIFIFPCCFIICIICIPPPIPPPIICPIIPIFICCFPIICIIPIFI
ICIICIIPIFIPIICIICCCICCPCICCFCCFFCFCCFFFCFFCFFCCICICFFPFICFICFFPFCFCFCFFFPCPICICFCFFCFFCFF
FFFFFCCFCCICCICIFCPICPCICFFFFFFFFFFFICCCFCCFFFCFFCFPPFPCFFFPFICFFPFFCFFCFIPCPCPFICFPICFFPF
CFPPFPCFFFPIPICPPIPICPPFFCFPCFFICCFPCFFICPCFCFFFPCCICICCPICICCCFIICFIICCCCIICCIICCCICCCCIC
CPCPIPFIIIIPPFFICFIFIIICFCCPFPCFCPCPFIPIICFIIFICFICFPCFICFCCFCCFPCFIFICFICFICFFICFIFPICFII
IFICCFCPFCFIIFCFIPFICFCFIICFCFPCFFCCCFIIFIICCFCPFIICFIIFICFIIIFPCFPIFCFIIIFCFIIIFFPFPFIIPF
CPFPPFIIIFIPFCPFIIIFPPFPFPIFCFIPIFFPFCFCPFICFPIFCFIIFICCFIPFCCFCPFIPCFIIFPCCFIFICCFFPCFIIF
CCFPIFCFIIFIFIPIFIIFCPFCPFPIFIIFIIIFPIFIIIFICFPIFCFPFCCFCCPFFIFICFCPFCPCFPIIFCPICFPCPFIFCP
FPFPIFCPFPIFIPFIPFPIFPFPFPFPIFIFIFPFFIFCFIPFICFCFFIFIICFCFPPFPPFCIIIPIPIICICCICPIICIIPIPCI
CCPFIICICIICCCICCICCCFCCCFCFCFFCCCCFCCCCFFICCCICCICFICCICIPPCPICCCFCCFCICFICCCICCICCIPCPCP
ICCCICCICFICCCFCCFCCFIPCPPIPCPPCCICICCPICFICCPICCFICCPICFFICCCFIICICIICIICICCIICPIIII
```