

# Report on the Tenth ICFP Programming Contest

Eelco Dolstra\*    Jurriaan Hage<sup>†</sup>    Bastiaan Heeren<sup>‡</sup>    Stefan Holdermans<sup>†</sup>    Johan Jeuring<sup>†</sup>  
Andres Löh<sup>†</sup>    Clara Löh<sup>§</sup>    Arie Middelkoop<sup>†</sup>    Alexey Rodriguez<sup>†</sup>    John van Schie<sup>†</sup>

## 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 language. 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 traveling 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 a DNA string. Additionally, two images were handed out.

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

<sup>†</sup> Software Technology, ICS, Utrecht University, {jur, stefan, johanj, andres, ariem, alexey, jcschie}@cs.uu.nl

<sup>‡</sup> Open Universiteit Nederland, bastiaan.heeren@ou.nl

<sup>§</sup> WWU Münster, clara.loeh@uni-muenster.de



Figure 1: Arrival sequence

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 image



Figure 3: Target image

- 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.

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 show how the contest went, give statistics about the participants, and reveal the winners.

## 2. DNA and RNA

This section briefly summarises the contest task. The full description can be found in (Dolstra et al. 2007a).

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.

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

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

Figure 5: Template language

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.

## 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.

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  $(!_2)\underline{P}$ , i.e., start a group, skip two bases, end a group, match a P. The middle part is interpreted as the template  $\underline{P}I_0$ , i.e., insert PI, and insert what was matched against the first group. The match-replace command, which can be written as

$(!_2)\underline{P} \mapsto \underline{P}I_0$

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 ( $I_0$  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 4.4 and 4.5, 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 4.2.

## 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 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 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. How would we proceed to solve 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 with a little care, this amount can be drastically reduced. For example, 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.

#### 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 very similar to 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 also says that “something curious” happens if the prefix `IIPFFCPCICICICPICIPPPCICIC` 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 thou-



Figure 6: First field repair guide page

sands 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:

`IIPFFCPCICICICICPICIPPPCICIC`

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 – 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 task advises. 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 straightforward, 347-line Haskell reference implementation *using the*

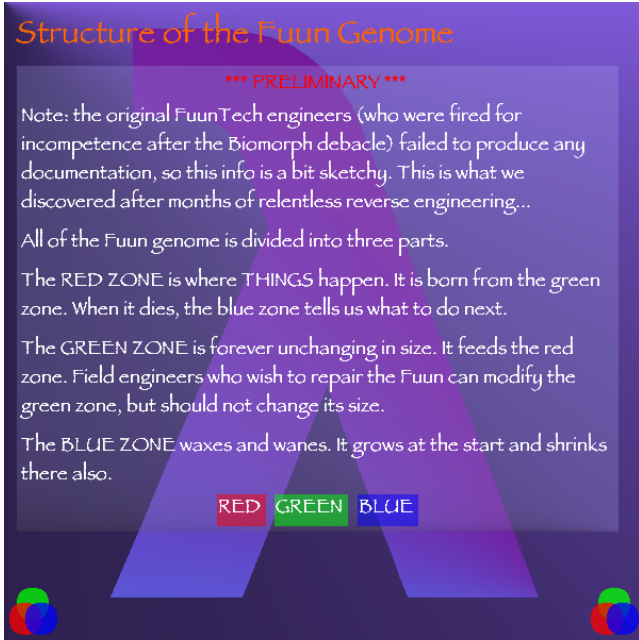


Figure 7: Repair guide page with strange terminology

right data structure takes about 50 seconds. Our optimised C++ implementation took about 5 seconds.

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:

$$(?_{IFPCFFP})\underline{I} \mapsto 0_0C$$

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:

$$(?_{IFPCFFP})\underline{II} \mapsto 0_0IC$$

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

$$(?_{IFPCFFP})\underline{IIIIIIIIII} \mapsto 0_0CIICCCIICIC$$

or, in concrete DNA,

IIPIFFCPICFPPICICICCCCCCCCCCIICIPPPFCFFCFCFIIC

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.

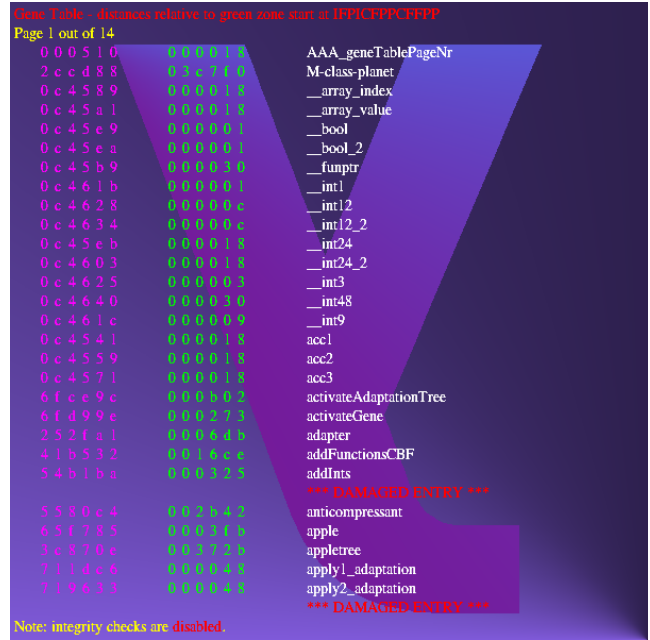


Figure 8: Gene list (or symbol table)

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 a prefix that searches for the special sequence IFPCFFPFFPP (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,

$$(?_{IFPCFFPFFPP!1283})!_4 \mapsto 0_0ICIC$$

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

IIPIFFCPICFPPICICFPPICICIPCCIIIIIIIC  
ICPIICIPICPIICIPPPFCFFCFCFIIC

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.

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 func-

tion *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:

```
(?IFPICFPPCFPPP!823763)!3 ↦ 0_0CIC
```

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. 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:

```
(?IFPICFPPCFPPP!5049987)!33 ↦ 0_0[!727 ↦]
```

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,

```
(?IFPICFPPCFPPP!5043058)!33 ↦ 0_0[!154 ↦]
```

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.

### 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 IICFPICFP or IICnnnnnn, 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 exact dynamic call graph within Endo’s DNA.

### 3.5 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).

### 3.6 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 excellent programming skills, or the advanced 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 chal-

lenging 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.

## 4. Creating the contest

We started thinking about ideas for the task in November 2005, but it wasn't until August 2006 that we began to put real effort into it. Various ideas were considered, but we settled on reverse engineering alien DNA. We intended this year's contest to be a reverse-engineering contest: the idea was to let contestants make debuggers, disassemblers, and whatever other means they could come up with to understand and repair Endo's DNA.

### 4.1 Requirements for an ICFP programming 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. Finally, the contest should be fun!

Recent contests had thousands of contestants, working from anywhere in the world, with very different backgrounds. The contestants may use any tools they like, and any information at their disposal. Furthermore, teams may be of arbitrary size. These 'contestant-friendly' rules have some important practical consequences for the organizers.

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. We have to think about how to deal with the fact that contestants are going to use many different programming languages and compilers.
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.

### 4.2 The evolution of DNA

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.

While programming in pure regular expressions is clearly an idea whose time has come, it 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.

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

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<sup>1</sup>.

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 would require a lot of arithmetic. 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 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,  $x - y = x + (\sim y) + 1$ , so all we need is a way to perform bitwise negation...)

### 4.3 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 had many 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.

<sup>1</sup> This is exactly the problem that many contestants encountered if they didn't use an efficient datatype for DNA.

#### 4.4 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 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 [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 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.

#### 4.5 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.

#### 4.6 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 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:

<i>image</i>	<i>before</i>	<i>after</i>	<i>RNA</i>	<i>DNA</i>
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 all their submits, and view the rendered result of their prefix with the best score (highest survival chance). The scoreboard, visible for everyone, shows the teams ranked by score, with the top 15 in random order and without a score.

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.

#### 4.7 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 actually 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 draw the target picture using brute force by drawing it with just RNA commands.

#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

## 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, 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.

### 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, 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.

#teams	Language(s)
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

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

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, and 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 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 color 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, thereby 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 improve-

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%	SwfPI
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

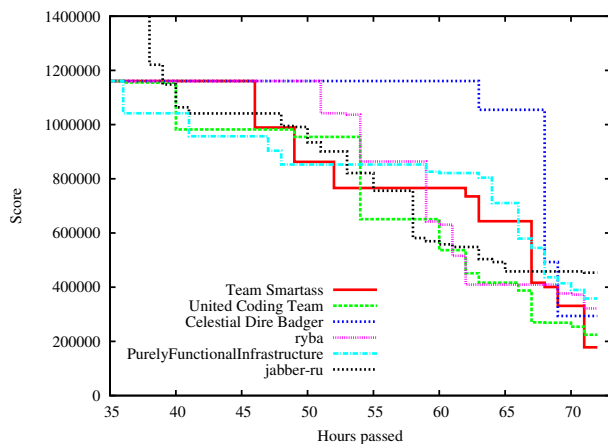


Figure 9: Scores of the Top 6 during the contest

ments 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.

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

**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 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.

### 5.4 Aftermath and reflection

In his posting of July 24, 2007, Marco Gallotta (<http://marco-za.blogspot.com/2007/07/icfp-how-we-reached-top-15.html>) provides a list of dozens of blogs related to the contest. This list includes 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. Furthermore, we observed that most of the secrets we included in Endo's DNA were discovered.

The blog messages often not only discuss technical issues, but also 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. We discuss some negative opinions found on the web:

- *Too many puzzles, too little programming:* programming was needed to find the secrets (the puzzles) and implement the solutions. Typically, programming was not needed to solve the puzzles.
- *The organizers show off themselves:* we take this comment to mean that we provided too many puzzles on widely different subjects. We indicated earlier in this report why:

