The Cult of the Bound Variable: The 9th Annual ICFP Programming Contest

Tom Murphy VII   Daniel Spoonhower   Chris Casinghino
Daniel R. Licata   Karl Crary   Robert Harper

October 17, 2006
CMU-CS-06-163

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Abstract

The annual ICFP Programming Contest has become one of the premiere programming competitions in the world. The 9th incarnation of the contest, "The Cult of the Bound Variable," was held in July 2006 and organized by the Principles of Programming group at Carnegie Mellon University. This report details the contest tasks, the technology used to produce the contest, and the contest results. Several tasks draw ideas from programming languages research. For example, participants implemented a simple virtual machine, played an adventure game based on a substructural logic, and programmed in a two-dimensional circuit language with a discordantly high-level operational semantics. The contest technology includes an optimizing compiler for a high-level functional language that targets our virtual machine. By the end of the three day contest, 365 teams, composed of 700 programmers from all over the world, solved at least one of the contest tasks.

Keywords: programming contest, ICFP, functional programming, the Cult of the Bound Variable, codex, two-dimensional programming

1 Introduction

The ICFP Programming Contest is a competition that the International Conference on Functional Programming has held annually since 1998. The contest has grown to be one of the premiere programming competitions in the world, attracting hundreds of teams from dozens of countries each year. This report describes the 9^{th} incarnation of the contest, called "The Cult of the Bound Variable," which was held in July 2006. The contest was developed by the Principles of Programming group at Carnegie Mellon University.

The 2006 contest was unusual in its scope, both in the breadth of contest tasks and in the technology used to produce them. A participant's first task was to implement a simple virtual machine from a specification. The contest materials were distributed as a large program to run on this machine. The program is a self-decrypting, self-decompressing UNIX-like system with seven independent problems for contestants to solve. These problems include implementations of five new programming languages created for the contest, a text adventure game encoding a resource-bounded affine logic, and puzzles based on combinatorics and cellular automata. Many of the problems draw ideas from programming languages research. To produce this program, we built an optimizing, obfuscating compiler from a high-level, garbage-collected, functional language to the simple virtual machine.

This paper starts by describing the ICFP contest in general and our goals in designing the 2006 incarnation (Section 2). We then describe the individual tasks that constituted the contest (Section 3). We then follow with a description of the technology used to produce the contest materials (Section 4). We conclude by reporting the contest results, including descriptions of the winning teams and participation statistics (Section 5). Readers should be advised that this paper discloses details about the contest's story, its problems, and their solutions. Readers who are still planning to play the contest may wish to read this paper after they have done so.

2 The ICFP Programming Contest

The ICFP Programming Contest began in 1998 as a way for participants to show off both their favorite programming language and their programming skills. By tradition, prizes are awarded simultaneously to the winning teams and their programming languages of choice. The first contest attracted 42 teams, mostly from the academic community. In 2006, there were more than 350 scoring teams, with the majority of participants coming from outside academia.

2.1 History

Despite changes in the number and composition of teams, the format of the contest has remained mostly unchanged. Contestants are given exactly 72 hours to complete the contest task. Teams may be composed of any number of participants and can work from any location (or locations) in the world. Teams are permitted to use any programming languages, tools, and computational resources at their disposal.

In half of the previous contests, entries were judged using some form of head-to-head competition. These contests asked participants to implement a strategy for some game invented by the organizers (*e.g.*, robot cops and robbers or feuding ant colonies), and the winning strategy was declared the best. Most of the remaining contests have consisted of an optimization problem. Entries were judged on the size or quality of the optimized result and how long it took to perform the optimization.

In addition to awarding prizes to entries with the best performance, several instances awarded a Judges' Prize to teams with particularly elegant solutions. Most years have also run a "lightning round" and awarded a prize to the team with the best entry after 24 hours. The language used by this team is deemed by the organizers to be "very suitable for rapid prototyping." The 2005 contest [4] introduced a novel format. To encourage cleaner, more adaptable code, the organizers released a "twist" two weeks after the original task. Contestants then had only 24 hours to adapt their original entries to the amended rules. The judges then declared the team with the best re-use of their first entry to be "an extremely cool bunch of re-hackers."

2.2 Design Criteria

Our development of the 2006 contest was guided by the following goals:

Fun First. The contest should be fun for participants and, hopefully, for the organizers as well. The contest is ultimately a diversion; its success should be judged like any other game or pastime. The majority of entrants should make substantial progress, and problems should be interesting, not tedious.

Community Service. The contest is sponsored by ICFP and should serve the ICFP community. Many of the participants will be functional programmers, and the contest should align itself, when possible, with the values of that community. For example, while teams should benefit (at least indirectly) from more efficient solutions, performance should not be the primary metric by which entries are judged.

The contest draws significant attention from outside the functional programming community. Consequently, it should make use of the opportunity to advertise the ideas and techniques developed by programming languages research. There are a plethora of interesting problems in our field, and the contest tasks should draw from them.

Finally, because a tremendous effort is expended on the contest each year (by both the organizers and the participants), we should strive to produce some artifact of lasting value.

Platform Independence. One of the difficulties in running a large programming contest is supporting the wide variety of computing platforms used by participants. In the past, this burden has fallen on either the participants, the organizers, or both. To avoid these issues, the contest materials should be independent of whatever platforms the organizers and contestants choose to use.

3 The Cult of the Bound Variable

The 2006 contest takes place in the context of a back-story about an ancient cult of programming language researchers known as the Cult of the Bound Variable. According to this story, the CBV was active in Pittsburgh, Pennsylvania thousands of years ago, carrying out computations on primitive devices powered by falling sand. Thirty years ago, excavators discovered an artifact of the Cult's scholarship, an extensive codex that was written in no known language. Archaeologists soon gave up on deciphering the Codex, and, until this year, the Codex was stored away in the basement of the Carnegie Museum of Natural History.

When publicizing the contest on mailing lists and forums, we described the theme of the contest as a fictional academic field called "computational archaeolinguistics." To promote interest in the contest, we released the Codex several days before the contest began. At this point, contestants had only sparse hints about the back-story—the illustration on the contest Web page, the name of the file (`codex.umz`), and posts on weblogs by fictional graduate students studying computational archaeolinguistics. Though the Codex file was undecipherable, we embedded within it some small teasers for participants to discover. This included strings referencing various pop-culture conspiracy theories such as "so dark the con of man," referencing *The Da Vinci Code* [7] and "paul is dead" (reversed, naturally). It also included

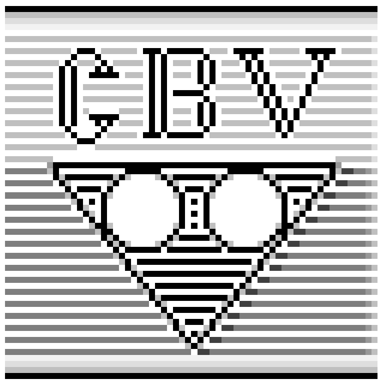


Figure 1: The CBV logo embedded within the Codex.

the image in Figure 1. For a few days, teams speculated as to the nature of the contest and the meaning of the embedded red herrings.

3.1 Tasks

One cannot help but wonder what the Cult might have achieved had they had access to modern electronics and type theory.

When the contest officially began, we published the task description. The task description is written from the point of view of a computational archaeolinguistics professor, intrigued by the Cult and planning to devote his life to an ascetic study of their work. According to the professor, a "Rosetta Stone" discovered while digging for the new computer science building at Carnegie Mellon enabled researchers to finally decode the thirty-year-old Codex. The professor makes the following request of the contest participants: over the next 72 hours, discover as many of the Cult's scholarly works as possible, and submit them via the contest Web site. The Cult's "publications," which are given as rewards for exploring the Codex and solving problems, are short strings such as

```
PUZZL.TSK=100@1001|14370747643c6d2db0a40ecb4b0bb65
```

(the Cult's publication venues had very strict length requirements). In reality, these strings encode the problem solved, a point score indicating how well the problem was solved, and the team that solved it. In the case that a team submitted several publications for the same problem, only their best score was counted. A global scoreboard on the contest Web site reflected the standings (up until eight hours before the contest ended, to keep the winning teams secret), and each team had access to its own scoreboard for the entirety of the contest.

In addition to the back-story, the task description includes the "Rosetta Stone" itself, which is a document

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/stat.h>
#define arr(m) (m?(uint*)m:zero)
#define C w & 7
#define B (w >> 3) & 7
#define A (w >> 6) & 7
typedef unsigned int uint;
static uint * ulloc(uint size) {
  uint * r = (uint*)calloc((1 + size), 4);
  *r = size;
  return (r + 1);
}
int main (int argc, char ** argv) {
  static uint reg[8], ip, * zero;
    FILE * f = fopen(argv[1], "rb");
    if (!f) return -1;
    struct stat buf;
    if (stat(argv[1], &buf)) return -1;
    else zero = ulloc(buf.st_size >> 2);
    int a, n = 4, i = 0;
    while(EOF != (a = fgetc(f))) {
      if (!n--) { i++; n = 3; }
      zero[i] = (zero[i] << 8) | a;
    fclose(f);
  }
  for(;;) {
    uint w = zero[ip++];
    switch(w >> 28) {
    case 0: if (reg[C]) reg[A] = reg[B]; break;
    case 1: reg[A] = arr(reg[B])[reg[C]]; break;
    case 2: arr(reg[A])[reg[B]] = reg[C]; break;
    case 3: reg[A] = reg[B] + reg[C]; break;
    case 4: reg[A] = reg[B] * reg[C]; break;
    case 5: reg[A] = reg[B] / reg[C]; break;
    case 6: reg[A] = ~(reg[B] & reg[C]); break;
    case 7: return 0;
    case 8: reg[B] = (uint)ulloc(reg[C]); break;
    case 9: free(-1 + (uint*)reg[C]); break;
    case 10: putchar(reg[C]); break;
    case 11: reg[C] = getchar(); break;
    case 12:
      if (reg[B]) {
        free(zero - 1);
        int size = ((uint*)reg[B])[-1];
        zero = ulloc(size);
        memcpy(zero, (uint*)reg[B], size * 4);
      }
      ip = reg[C];
      break;
    case 13: reg[7 & (w >> 25)] = w & 0177777777;
    }
  }
}
```

Figure 2: A simplified but working version of our Universal Machine implementation in C.

specifying the Cult's computing device, the Universal Machine. To discover the Codex's secrets, participants first needed to implement this machine.

3.2 Universal Machine

The '0' array shall be the most sublime choice for loading, and shall be handled with the utmost velocity.

The Universal Machine (UM) is a very simple 32-bit architecture with 14 instructions. We designed it to be as simple as possible while preserving a reasonable level of performance. It provides eight general-purpose registers; a heap that maps distinct 32-bit values to allocated arrays; array allocation, deallocation, subscript and update; character-based input and output; and a modicum of simple control, math and logic instructions. A special distinguished entry in the heap is identified by the value 0 and holds the program. This array can be overwritten, allowing for self-modifying code.

Our implementation of the Universal Machine in C is under 100 lines of code (Figure 2). We have implemented the Universal Machine in a variety of languages: Standard ML, O'Caml, Haskell, C#, Java, Perl, Python, Scheme, x86 assembly, Twelf, PostScript, Awk, and UM assembly itself. In Section 4, we describe these implementations in more detail and present some data on performance differences between them.

After implementing the Universal Machine, a player can run the Codex within it. The Codex performs a self-check (Section 4.1) and then prompts for a individual decryption key. If a valid decryption key is input, the Codex decrypts itself, decompresses itself, and dumps a larger Universal Machine program to the terminal. This program is an implementation of a Unix-like system that includes the contest problems.

3.3 UMIX

"First Projection On-Line: So Easy to Use, No Wonder It's #1!"

Running the decompressed UM binary produces the following screen:

```
12:00:00 1/1/19100
Welcome to Universal Machine IX (UMIX).

This machine is a shared resource. Please do not
log in to multiple simultaneous UMIX servers. No
game playing is allowed.

Please log in (use 'guest' for visitor access).
;login:
```

From this point on, players are on their own—the UMIX system is a self-contained artifact that describes the problem tasks and verifies their solutions. UMIX has multiple user accounts, a modifiable hierarchical file system with file and directory permissions, and a help system for its various commands. Each user account includes programs that implement the contest problem for that user. Additionally, one user account allows the team to verify its publications off-line and (when reaching certain scores) unlock administrative accounts that reveal additional elements of the CBV story. Players can therefore still track their progress and "win the game," even though the contest is over.

Although the UMIX system runs with synchronous character-based I/O inside a virtual machine that the user most likely implemented himself, its similarity to real Unix environments can be deceiving. We observed many testers attempt to kill running programs asynchronously with `ctrl-c` (in most implementations this kills the Universal Machine and all of UMIX), launch editors, use tab completion and command history, etc.[1]

3.3.1 QVICKBASIC

"`IMPOSSIBLE NVMBER: I - I`"

The first user account, "guest", contains a tutorial problem whose purpose is threefold: it teaches players how to proceed (by discovering passwords to other accounts), the theme of how this is done (by using programs and programming languages in each account), and the specific UMIX mechanisms for uploading files, compiling them, and running them.

The tutorial begins with an e-mail from the root user that scolds the anonymous guest user for attempting to hack other user accounts. The e-mail gives a transcript of an example hack attempt as evidence, and hints that the password cracking program uploaded by the hacker fails to compile because of a transmission error. Fixing this program unlocks two additional accounts. A simple extension to the program unlocks the special account with the publication verifier. The password cracker is written in an ancient version of the QuickBASIC language that uses Roman numerals,[2] called QVICKBASIC. An abbreviated version of the hacking script appears in Figure 3.

```
V        REM  +------------------------------------------------+
X        REM  | HACK.BAS      (c) 19100   fr33 v4r14bl3z        |
XV       REM  |                                                |
XX       REM  | Brute-forces passwords on UM vIX.0 systems.    |
XXV      REM  | Compile with Qvickbasic VII.0 or later:        |
XXX      REM  |    /bin/qbasic hack.bas                        |
XXXV     REM  | Then run:                                      |
XL       REM  |   ./hack.exe username                          |
XLV      REM  |                                                |
L        REM  | This program is for educational purposes only! |
LV       REM  +------------------------------------------------+
LX       REM
LXV      IF ARGS() > I THEN GOTO LXXXV
LXX      PRINT "usage: ./hack.exe username"
LXXV     PRINT CHR(X)
LXXX     END
LXXXV    REM
XC       REM  get username from command line
XCV      DIM username AS STRING
C        username = ARG(II)
CV       REM  common words used in passwords
CX       DIM pwdcount AS INTEGER
CXV      pwdcount = LIII
CXX      DIM words(pwdcount) AS STRING
CXXV     words(I) = "airplane"
CXXX     words(II) = "alphabet"
CXXXV    words(III) = "aviator"
CXL      words(IV) = "bidirectional"
             ⋮
CCCXC    REM  try each password
CD       DIM i AS INTEGER
CDV      i = I
CDX      IF CHECKPASS(username, words(i)) THEN GOTO CDXXX
CDXV     i = i + I
CDXX     IF i > pwdcount THEN GOTO CDXLV
CDXXV    GOTO CDX
CDXXX    PRINT "found match!! for user " + username + CHR(X)
CDXXXV   PRINT "password: " + words(i) + CHR(X)
CDXL     END
```

Figure 3: Abbreviated `hack.bas` from the tutorial problem.

Players unlock additional UMIX accounts as they explore the codex. Usually, the password for an account is given as a reward for completing a simple tutorial task for a problem. This design prevents players from being overwhelmed by the possible tasks. Figure 4 contains the dependency graph for the accounts; an edge from one account to another means that the first reveals the password to the second. There are multiple paths to each account, so players are not subject to a single point of failure. The remaining accounts are described in the rest of this section.

3.4 2D

"Hell is other programming languages."—Sartran

The first full-fledged language that players are likely to encounter is called "2D." Its concrete syntax is a preposterous two-dimensional ASCII notation for circuits. Its operational semantics is discordantly high-level, supporting recursion and the transmission of arbitrarily large values over wires.

A 2D program that adds two unary numbers appears in Figure 5. It consists of two modules (delimited by dotted lines), one called "plus" that recursively performs the addition and one called "main" that serves as the

[1]Although asynchronous signals and multitasking are not possible in our model, we could have in principle supported editors with sufficient knowledge of the user's terminal. We chose to restrict ourselves to pure ASCII-based interaction to maximize compatibility.

[2]One may not compute with zero or values larger than 3999, for Roman numerals have no way of expressing those numbers!

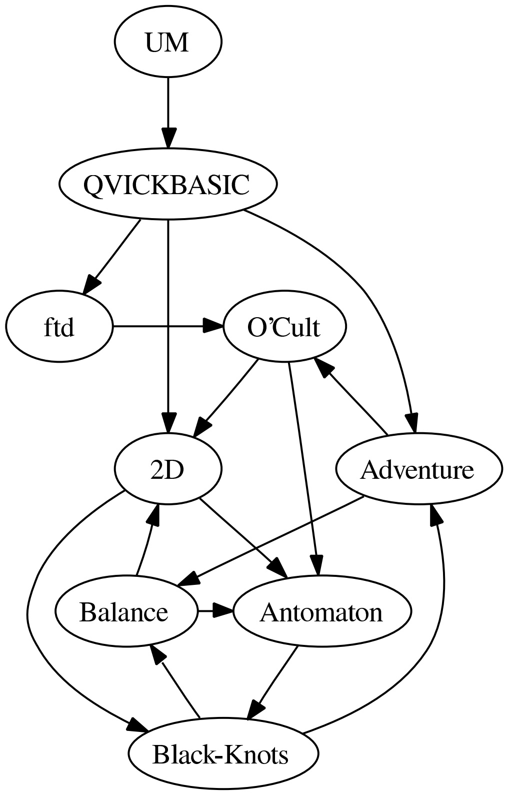


Figure 4: Dependency Graph for User Accounts. An edge from one account to another means that the first reveals the password to the second.

```
,..........................|.............................,
:plus                      |                             :
:  *==================*    |                             :
-->!send [(W,S),(W,E)]!----#---------------+             :
:  *==================*    |               |             :
:         |                v               v             :
:         |       *==============*   *============*      :
:         |       !case N of S, E!-->!send [(N,E)]!-------
:         |       *==============*   *============*      :
:         |           |                                  :
:         |           v                                  :
:         |       *========*      *================*     :
:         +------>!use plus!----->!send [(Inl W,E)]!------
:                 *========*      *================*     :
,........................................................,
,........................................................,
:main                                                    :
:*================================================*      :
:!send [(Inl Inl Inl Inr (),E),(Inl Inl Inr (),S)]!-+    :
:*================================================* |    :
:                 |                                 v    :
:                 |                          *========*  :
:                 +------------------------->!use plus!---
:                                            *========*  :
,........................................................,
```

Figure 5: 2D program that adds the unary numbers 3 and 2.

entry point to the program. Each module is a graph built out of boxes and wires connecting the boxes. During evaluation of a circuit, a wire may be instantiated with a value, which it holds from then on. Any box in the circuit whose inputs (wires entering its north and west faces) all have values can be executed. This causes some set of its outputs (wires leaving its east and south faces) to be instantiated. The `use` command creates a nested copy of a module (with none of its wires instantiated), which permits circuits to be recursive.

Players are charged with three problems. The first is to write a program that multiplies two unary numbers (mimicking and building on the supplied implementation of addition). The second is to reverse a list, which can also be completed easily by hand. The third task plays on a previous ICFP contest [1]: contestants must write a one-dimensional ray tracer.[3] Their program takes as input a scene description, which is a list of zero-dimensional surfaces each with light reflectivity, transmissivity, and emission properties. It must return the light value that satisfies the equations in the declarative specification. To solve this problem, each team must derive an algorithm that solves the equations (a naïve recursive solution will not terminate), implement it in 2D, and then lay out their program in ASCII.

UMIX contains an interpreter for 2D and can probabilistically verify that a program meets the specification by running it on test cases. For correct programs, points are awarded based only on the physical area of the program. Therefore, the task was to produce not only a correct ray tracer, but to lay out the program in the smallest possible area.

Many teams found it helpful to develop the ray tracer in a higher-level language and then translate the result to 2D. Many contestants undertook this translation by hand, for example a hand-optimized solution by Team "Expansion" appears in Figure 6. Others wrote a compiler with a 2D back-end. The "CamlNuggets" team wrote one such compiler for ML; part of its output is shown in Figure 7.

3.5 Black-Knots

"I expect children will be very picky about their knockoff toys being observationally equivalent to the 11 official black-knot models."

Some UMIX accounts, such as the one for a problem called Black-Knots, contain not programming language implementations but programming puzzles. The purpose of Black-Knots is to reverse engineer a device (*i.e.*, a function) given only its inputs and outputs. The device

[3]The 2000 contest asked entrants to implement an interpreter for a scene description language and a ray tracer to render the results. In 2000, entries were required to render more conventional *three*-dimensional scenes.

```
....................................................................................................,,........................|.,,...
     ++         ++            ++       ++               ++                    ++       ++           ::h                       | ::i
    ++v        ++v          +-+v      ++v              ++v                  +-+v     +-+v           ::                        v ::
==* |*=======* |*=======*   | *=====* |*=============* |*===============*   | *=====*| *===========*::*===============*  *=====*::*==
)]!+|!split N!+|!split N!+ +#>!use g!+|!case N of E,S!+|!send[(Inr(),E)]!+ +#>!use f!#>!send[(W,E)]!-:!send[(Inr(),E)]!->!use t!-:!ca
==*||*=======*||*=======*| || *=====*||*=============*||*===============*| || *=====*| *===========*::*===============*  *=====*::*==
   ||       | ||       +-#-+|        ++             +-#+                 +-##--------#---------------:                          ::
---##-------#-#+         +--#-------------------------#--------------------+|        |              :,..........................,:
---#+       +-#-------------+                         +---------------------#--------+              :                            :
   |          +-------------------------------------------------------------+                       :                            :
   +-------------------------------------------------------------------------------------------------                            ,...
-----------------------------------------------------------------------------------------------------
....................................................................................................,
```

Figure 7: Part of a 2D program generated by a compiler implemented by "CamlNuggets." Determining how to connect boxes using horizontal wires is similar to register allocation.

```
,.................|..................,,.......................................,
:M *=============*| *===============*::I *=============*+--+ *===============*:
:+>!case W of E,S!#>!send[(Inl(),E)]!--->!case W of S,E!#--#>!send[(Inr(),E)]!-
:| *=============*| *===============*::  *=============*|  v *===============*:
:+----#-----------+                  ::   |      *=====*| *==================*:
:     |                              ::   |    +>!use I!#>!send[(Inl(N,W),E)]!-
:     | *=================*          ::   v    ++*=====*| *==================*:
--+   +>!send[(W,S),(W,E)]!+         ::*=======*|       +--------------------+:
: v     *=================*v         ::!split N!+           *=====*          |:
:*=============*|   *===============*::*=======*          +>!use i!-+        |:
:!case N of E,S!#-->!send[(Inl(),E)]!-:++*=======*        | *=====* v        |:
:*=============*v   *===============*::+>!split W!--+     | *===============*|:
:     |    *=====*  *===============*::  *=======*  +-----#>!send[((W,N),E)]!+:
:     +--->!use M!->!send[(Inr W,E)]!-:         +---------+ *===============* :
:          *=====*  *===============*:,.......................................,
,....................................,                       ,................,
,...|................................,,.....................,:T               :
:A  | *=================*            ::i *=============*    :: *=====*        :
--+ +>!send[(W,S),(W,E)]!+           :-->!case W of E,S!-+  :->!use Q!+       :
: v   *=================*v           ::  *=============* v  :: *=====*v       :
:*=============*|  *===========*     ::   |*===============*:: *=======*      :
:!case N of E,S!#->!send[(N,E)]!------:   |!send[(Inl(),E)]!-: !split N!+     :
:*=============*v  *===========*     ::   v*===============*:: *=======*v     :
:    |     *=====*  *===============*::*===================*:: ++*===========*:
:    +---->!use A!->!send[(Inr W,E)]!-:!send[(Inr Inl(),E)]!-: +>!send[(N,E)]!-
:          *=====*  *===============*::*===================*::   *===========*:
,....................................,,.....................,,................,
,.....|.......................................................................,
:main | *=====*  *=====*  *=====*  *=====*  *=====*  *=====*  *=====*  *=====*:
:     +>!use I!->!use T!->!use T!->!use T!->!use T!->!use T!->!use Q!->!use F!-
:       *=====*  *=====*  *=====*  *=====*  *=====*  *=====*  *=====*  *=====*:
,.............................................................................,
,.....................................|.....................................,
:S *=======*      *=============*     | *=================*                 :
-->!split W!----->!case W of E,S!--+  +>!send[(W,S),(W,E)]!+                :
:  *=======*      *=============*  v    *=================*v           +---+:
:  ++*=================*|    *===========* |  *=======================*|   |:
:  +>!send[(W,S),(W,E)]!#--->!send[(W,E)]!-#->!send[((N,(W,Inr N)),E)]!#---#-
:    *=================*|    *===========* |  *=======================*v   |:
:+-------------------+ +#------------------#-+                    *=====*  |:
:| *=================* ||  *=======*+------+ | +----------------->!use A!+ |:
:+>!send[(W,S),(W,E)]!-+|+>!split W!#----+   +-#+                 *=====*v |:
:  *=================*+-#+ *=======*v    v     || *=======================*|:
:      | *=======*   ++ v    | *=====*  *=====*|+>!send[((N,(W,Inl())),E)]!#-
:      +>!split W!---+*=====*+>!use M!->!use A!+  *=======================*|:
:        *=======*+-->!use M!-+*=====*  *=====*                            |:
:            +----+   *=====* +--------------------------------------------+:
,...........................................................................,
,..|.......................................,,.............................,
:L | *=================*+----+         +--+::Q *=======================*  :
:  +>!send[(W,S),(W,E)]!+    v         |  |::  !send[((Inl(),Inr()),E)]!+ :
:    *=================* *===========* |  |::  *=======================*v :
: *=============*|     +>!send[(N,E)]!-#--#-:                      *=====*:
->!case W of S,E!#-----+ *===========* v  |:---------------------->!use L!-
: *=============*| +-----+        *=====* |::                      *=====*:
: ++*=======*    |+#-----#------->!use L!-#-,.............................,
: +>!split W!----#+|     v        *=====* |:,..........................,
:   *=======*    v |  *==================*|::F *=======*        +-----+:
:    |    *=======*|+>!send[(Inl(W,N),S)]!|:-->!split W!-+      v     |:
:    |    !split N!+| *==================*|::  *=======* |  *=====*   |:
:+---+  ++*=======* ++     |              |::         +--#->!use M!---#-
:|      v+-+*=======*|     v              |::            v  *=====*   |:
:| *=====*+>!split W!+  *===============* |::*=======================*|:
:+>!use S!+ *=======* +>!send[((W,N),E)]!-+::!send[(Inr Inr Inl(),E)]!+:
:  *=====*         +--+ *===============*  ::*=======================* :
,..........................................,,..........................,
```

Figure 6: Ray tracer solution submitted by "Expansion." This solution was laid out by hand.

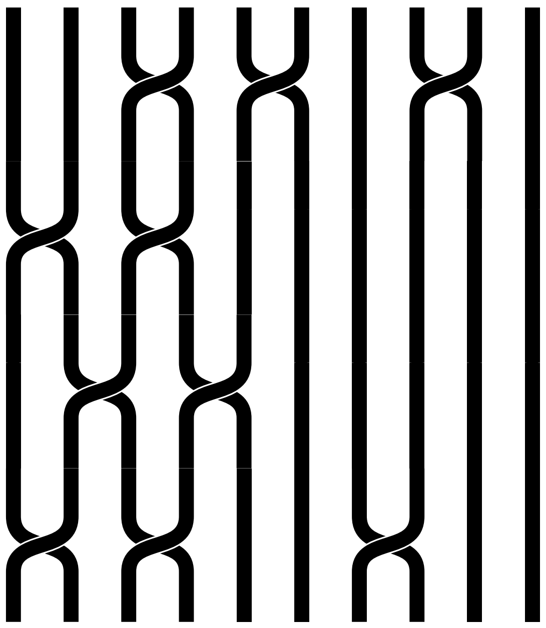


Figure 8: An example Black-Knot. A ball entering column 0 at the top leaves column 3 and makes three "plink" sounds. A ball entering column 1 exits in column 1 with one "plink" sound.

is a popular toy with holes at the top that children drop marbles into. A Black-Knot has n input holes (numbered 0 to $n - 1$) and n output holes. The Black-Knot is a function from each of its n inputs to a pair of integers (j, k) where $0 \le j < n$ and $0 \le k$. The number j is the output hole that the marble comes out of. The number k is a number of "plink" sounds that are produced as the marble rolls unseen through the toy.

The programming puzzle is to reproduce toys from the specification of their behavior, using only two parts laid out on a grid. The parts are a vertical pipe, whose size is one row by one column, and a cross-over pipe, whose size is one row by two columns. A marble passing through the cross-over pipe moves into the opposite column, and makes a "plink" sound if it moves from left to right. An example grid built from the cross-over and vertical pipes appears in Figure 8.

Contest participants were asked to reproduce devices with widths ranging from from 10 to 500. A naïve solution to the problem takes exponential time and can therefore recreate only the smallest devices. We do not know the complexity of the problem in general, but the large puzzles in the Codex were fairly under-constrained and therefore amenable to clever algorithms and heuristics. For example, many participants employed an algorithm that first eagerly generates a permutation that places all marbles into the correct columns, and then swaps two adjacent columns an even number of times (which does not affect the permutation) until their "plink" counts match the specification. This algorithm succeeds as long as there are sufficient "plinks" not used by the initial permutation such that each column's count can be corrected.

3.6 Balance

"Every problem in computer science can be solved by an additional layer of indirection. Balance thus provides this facility automatically."

The Balance programming language is an extremely impoverished four-instruction language designed around a misguided notion of duality. Each instruction carries out two simultaneous functions on potentially overlapping sets of registers or memory locations, making the language quite difficult to use.

The language has four input registers S_{0-3} and two output registers D_{0-1}. Each register contains a single byte, which is usually treated as the index of a location in the 256-byte memory. The instruction `MATH` S_i,S_j,D_k, for instance, subtracts the values of memory locations S_{i+1} and S_{j+1} and stores the result in the memory location D_{k+1}, then adds the contents of the memory locations S_i and S_j and stores that in memory index D_k. The `LOGIC` instruction similarly performs bitwise *and* and its "perfect dual" *exclusive or*. The two remaining instructions are `SCIENCE`, which tests a memory location for 0 and changes the speed of the instruction pointer (which can be greater than 1 or even negative), and `PHYSICS`, which modifies the contents of the registers themselves with what amounts to a simple hash function.

Programmers are challenged to implement a collection of very simple behaviors (such as multiplying two numbers, or copying a memory cell to another location). Their programs are then tested on random inputs to see that they match the specification. Like many other problems in the contest, more points are awarded for smaller solutions.

```
OPCODE      MNEMONIC      COMMENT
                 ;set up regs {D0,D1,S0,S1,S2,S3}
                          ;regs = {4,5,0,1,2,3}
011 00010   PHYSICS 2     ;regs = {2,5,4,1,2,3}
011 11100   PHYSICS -4    ;regs = {2,5,1,2,3,0}
011 11111   PHYSICS -1    ;regs = {5,0,2,3,0,2}
001 11001   MATH 1 10 01  ;mem[5] = -a
011 10010   PHYSICS -14   ;regs = {-12,0,3,5,0,2}
000 00000   SCIENCE 0     ;halt on 2nd execution
010 00110   LOGIC 0 01 10 ;mem[-12] = 1
011 11111   PHYSICS -1    ;regs = {0,2,5,0,2,-12}
011 11100   PHYSICS -4    ;regs = {0,2,0,2,-12,1}
                          ;main loop
000 00001   SCIENCE 1     ;con't
001 11101   MATH 1 11 01  ;c += b; a -= 1
000 11110   SCIENCE -2    ;jump unless a = 0
```

Figure 9: A Balance program submitted by "Team Smartass." Valid solutions must multiply the first two values `a` and `b` in memory and store the result `c` as the third value in memory. Taking advantage of the probabilistic checking of Balance, this particularly compact solution computes the correct answer only when the first factor is odd.

Programming language features are often described as "orthogonal," in the sense that a point in "program space" can be reached by a natural composition of constructs forming the "basis" of the language. In this analogy Balance provides a very non-orthogonal basis. One strategy for writing Balance programs is to first build up an instruction set that *is* orthogonal—for example, an `add` instruction that does not do anything else, or a `swap` instruction that swaps two registers without changing their contents. This generally produces large solutions. Another strategy is to use computer-assisted search for small sequences of instructions that achieve some fragment of desired behavior.

Solutions were only required to yield correct answers with "high reliability" and many teams took advantage of this fact to submit smaller but less accurate solutions. Figure 9 shows a program submitted by "Team Smartass." The program multiplies the first two values in memory—but only if the first is odd, because it uses the low bit of this first value to set an auxiliary memory location to the value one. Multiplication by repeated addition can then be encoded efficiently using the natural parallelism of the Balance machine. This solution also depends on the fact that the instruction pointer will wrap around to the first instruction in the case that the last `SCIENCE` instruction has no effect.

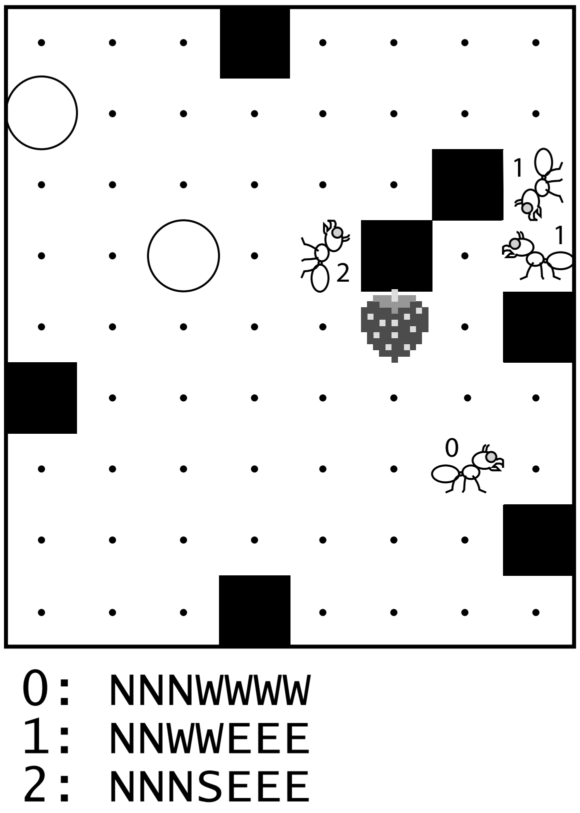


Figure 10: A sample Smellular Antomaton. The ant of clan 0 reaches the food after 110 iterations.

3.7 Smellular Antomata

"As we know, ants are blind and so they navigate by smell. An ant can only smell its immediate neighborhood. It can't smell holes, so it walks right into them."

Another puzzle in the contest is based around a cellular automaton system. Riffing on the 2002 Programming Contest [3], the "smellular antomaton" simulates ants walking around on a grid in an attempt to reach food.[4] We provide a specification and simulator for the automaton's rules. Players are asked to solve a collection of puzzles; these take the form of a "pattern" of initial conditions that must be filled in by the participant such that at least one ant reaches food.

There are ten ant clans, each of which can have a different program. These programs specify which direction an ant in that clan turns when encountering various scenarios; each program is only 14 bits long. In addition to the ants and food, the world contains walls to contain ants and holes for them to fall into. An example ant puzzle appears in Figure 10.

Ant puzzles can be solved by hand-programming the initial conditions of the antomata or by computer-assisted search. In our development of the puzzles, we used a combination of both techniques.

3.8 O'Cult

"Advice when most needed is least heeded."

O'Cult is a twisted term-rewriting system phrased in the terminology of aspect-oriented programming.

3.9 Adventure

"Did you mean the orange-red V-9247-KRE, the celadon V-9247-KRE, the green-yellow V-9247-KRE or the pale-brown V-9247-KRE?"

Part of the programming contest takes place within the context of a text-based computer game, styled after the famous game Adventure of the late 1970s. Tasks in the adventure can be divided into two parts.

Part I. The world of the adventure game is populated with hundreds of broken items (most with opaque names such as "V-9247-KRE") from which contestants are required to assemble two special items that will be used in Part II of the adventure, the *uploader* and the *downloader*. A typical item might be described as follows.

```
The T-0010-BQH is an exemplary instance of part
number T-0010-BQH. Interestingly, this one is
indigo. Also, it is broken: it is
  (a T-0010-BQH missing a X-1623-GTO and a
     N-5065-AGS and a B-4292-LWV and a R-6678-FJZ)
 missing (a T-0010-BQH missing a X-1623-GTO and
            a X-1623-GTO and
            (a X-1623-GTO missing a T-0010-BQH)).
```

Apart from the plethora of broken (and nearly anonymous) junk, players discover two other important facets of the adventure. First, two items may be combined to produce a new and less broken item, but only if these two items "fit together." Second, the adventure protagonist cannot drop items and instead can only incinerate them, destroying them (seemingly) forever.

Each room in the adventure contains a pile of items. While any item in the pile can be examined, only the top item on each pile can be picked up and added to the protagonist's inventory. The protagonist can only carry a limited number of items, and only items in the protagonist's inventory can be combined or incinerated. Among the items in each room is a single item that must

[4]The seventh contest asked contestants to submit an ant brain as a program written in a low-level, assembly-like language. This brain was replicated across an entire colony of ants who then faced off against another colony to see who could gather the most food.

be repaired (using other items in the room) and then used as part of either the downloader or the uploader.

Like many planning problems, this part of the adventure game can be boiled down to proving theorems in a restricted logic. Indeed, anyone who solves the first part of the adventure programmatically builds an automated theorem prover—perhaps without even realizing that they are doing so!

We derive a logic from the rules of the adventure game as follows. Items are represented as propositions. Items in pristine condition are represented by atomic propositions, and broken items by implications. The process of combining two items consumes the smaller of the two, so implication is linear. To mirror the structure of broken items, we allow implications to assume multiple hypotheses. In general, implications will have the form $\{\Psi\} \multimap A$ where Ψ is a non-empty multi-set of propositions.

We wish to answer queries such as, "how can a downloader be obtained from the items found in this pile?" We model this relation between resources and a goal using logical entailment, presented here as a sequent.

$$\Delta; \Gamma \Longrightarrow_i G$$

On the right of the sequent is a single proposition representing the desired item. Propositions on the left of the sequent represent the current state of the world. These propositions are divided into two groups. The first group Δ represents the pile of items in the room. This group is affine: hypotheses can be used at most once, as the use of a hypothesis represents removing an item from the pile. It is also ordered, because the arrangement of items in the pile is fixed. The second group Γ represents the player's inventory. It is affine and unordered (*i.e.*, admits exchange), as items may be used at most once, but the order of items in the inventory is irrelevant. Because the inventory can hold a limited number of items, the sequent is indexed by a natural number i that indicates how many additional propositions may be added to the unordered group Γ.

The inference rules for this logic follow directly from the rules of the game. These five inference rules are shown in Figure 11. Rules be should be read from bottom to top. The rule TAKE states that if there is an empty position in the inventory then item A from the top of the pile may be picked up. Likewise, any item A in the inventory may be incinerated using rule INCINERATE. The GOAL rule is applied when the player's inventory holds the desired item. The last two rules are used to combine items: COMBINE-1 is used in the case where the left-hand side of an implication consists of exactly one proposition; otherwise, the rule COMBINE-N is used. Note that the sum of the index i and the size of Γ remains constant in each rule.

$$\frac{}{\Delta; \Gamma, G \Longrightarrow_i G}\ \textsc{Goal} \qquad \frac{\Delta; \Gamma, A \Longrightarrow_i G}{\Delta, A; \Gamma \Longrightarrow_{\mathsf{s}(i)} G}\ \textsc{Take}$$

$$\frac{\Delta; \Gamma \Longrightarrow_{\mathsf{s}(i)} G}{\Delta; \Gamma, A \Longrightarrow_i G}\ \textsc{Incinerate}$$

$$\frac{\Delta; \Gamma, \{\Psi\} \multimap B \Longrightarrow_{\mathsf{s}(i)} G}{\Delta; \Gamma, \{A, \Psi\} \multimap B, A \Longrightarrow_i G}\ \textsc{Combine-N}$$

$$\frac{\Delta; \Gamma, B \Longrightarrow_{\mathsf{s}(i)} G}{\Delta; \Gamma, \{A\} \multimap B, A \Longrightarrow_i G}\ \textsc{Combine-1}$$

Figure 11: The Logic of Adventure. Inference rules for the sequent calculus used in the first part of the adventure game. Rules should be read from bottom to top. For example, the TAKE rule can be read "the goal can be reached with the pile Δ, A and an inventory with at least one empty space if the goal can also be reached with a pile comprised of only Δ and with A in the player's inventory."

As a sequent calculus, these rules form an impoverished set of left rules with, for example, INCINERATE as a form of weakening and the GOAL rule taking the place of the initial sequent. The logic includes two left rules so that the set of hypotheses on the left of an implication may be partially discharged.

Each puzzle in Adventure can be interpreted as a search for a proof in this logic. Reading the proof from bottom to top yields a set of commands to be executed. A small example of one such puzzle is given in Figure 12 along with a solution presented in the form of a proof.

The example in Figure 12 shows a few of the nondeterministic choices that must be made by a prover. For example, a strategy that eagerly combined the items $\{B\} \multimap A$ and B would not find a solution. Similarly, after three items have been taken from the pile, some items in the inventory must be either incinerated or combined before any further items can be picked up. In this example, only by incinerating item B does a solver arrive at a solution.

While this presentation establishes a clear connection between the logic and the game, it does not provide many hints as to how a prover should be implemented. As a guide for implementation, one can devise an alternative logic that makes an algorithm more explicit. In this case, the original logic gives a specification for the prover, and a proof of completeness for the alternative calculus serves to show the correctness of the implementation.

In our development of the adventure, we found this logical perspective to be helpful in keeping the game both simple and challenging. Keeping the number of inference rules small ensured that the game admitted a simple description. Moreover, we were able to leverage our experience in implementing automated theorem provers to make the puzzles algorithmically challenging.

Part II. It is eventually revealed to participants that the game's protagonist is not human but instead one of many robots tasked with collecting, repairing, and incinerating the city of Chicago's garbage.[5] There is unrest among the city's robots, however, and in response, the municipal government has retrofitted each robot's subconsciousness with a Censory Engine. The Engine blocks out any seditious information from robots' higher brain functions and, as a result, from contest participants as well. The task in the second part of the adventure is to circumvent this Engine.

> *"The proof is long and complicated, purporting to prove the correctness of the Censory Engine. You have the feeling something is wrong with it, but you keep getting lost in the details. Too bad it's not in machine-checkable form."*

The robot's nervous system (and coincidently a large part of the game engine) consists of four major components: a parser, a conscious mind, the Censory Engine, and a perceptual system.

The parser takes strings entered by the user (*e.g.*, `go north` and `take blue transistor`) and builds a well-formed command, resolving ambiguity among different items using adjectives as necessary.

The robot mind implements these commands using a small set of built-in primitives. Some primitives describe the state of the world, for example, by giving adjacent rooms on the game map, a list of items in each room, or the condition of each item. Others change the state of the world, for example, by moving the robot from one room to another or by fixing (or breaking) an item. Each processed command can also yield a result such as a string describing an item or a room.

These results are then filtered by the Censory Engine, as discussed below, and rendered by the robot's perceptual system in one of several formats including English, s-expressions, XML, and ML.

Using the downloader acquired in the first part of the adventure, contestants can perform a "brain dump" and view the source code of the robot's mind, as written in Robot Mind Language (RML). They may then edit this source and update the robot's mind using the uploader. Contestants must use these devices to reprogram

```
The C is in pristine condition.
Under the C is a B.
The B is in pristine condition.
Under the B is an A.
The A is broken: it is (an A missing a B).
Under the A is a G.
The G is broken: it is
  (a G missing (an A missing a B) and a C)
```

(a)

$$
\begin{array}{rl}
\hline
\cdot; G \Longrightarrow_2 G & \textsc{Goal} \\
\hline
\cdot; \{C\} \multimap G, C \Longrightarrow_1 G & \textsc{Combine-1} \\
\hline
\cdot; \{\{B\} \multimap A, C\} \multimap G, \{B\} \multimap A, C \Longrightarrow_0 G & \textsc{Combine-N} \\
\hline
\{\{B\} \multimap A, C\} \multimap G; \{B\} \multimap A, C \Longrightarrow_1 G & \textsc{Take} \\
\hline
\{\{B\} \multimap A, C\} \multimap G; \{B\} \multimap A, B, C \Longrightarrow_0 G & \textsc{Incinerate} \\
\hline
\{\{B\} \multimap A, C\} \multimap G, \{B\} \multimap A; B, C \Longrightarrow_1 G & \textsc{Take} \\
\hline
\{\{B\} \multimap A, C\} \multimap G, \{B\} \multimap A, B; C \Longrightarrow_2 G & \textsc{Take} \\
\hline
\{\{B\} \multimap A, C\} \multimap G, \{B\} \multimap A, B, C; \cdot \Longrightarrow_3 G & \textsc{Take}
\end{array}
$$

(b)

Figure 12: Example. A small adventure puzzle (a) and one possible solution presented as a proof (b). The goal is an unbroken item `G`. In this example, the player's inventory can hold at most three items.

the robot mind to overcome certain "physical" limitations imposed by its conscious (*e.g.*, travel between non-adjacent locations) and the impediments created by the Censory Engine.

The Censory Engine, as embodied by dynamic semantics of RML, associates a boolean with each computed value that indicates whether or not this value has been tainted by any seditious information. This process of tainting follows both data and control dependencies. For example, adding one or more tainted values or case analyzing a tainted value both result in a new tainted value. Tainted values are displayed to the user using only the string `[___REDACTED___]`.

However, this process of tracking information flow is not properly integrated with the language's imperative features. Thus contestants may cleanse a value by encoding it in the state of the world (*e.g.*, the locations of a set of items). The task given to contestants consists of finding the bug in the language semantics and then writing a program to exploit this flaw.

[5]This is a play on the 2005 ICFP Programming Contest [4] where participants were asked to implement the brains of robber and cop robots. These robots roamed the streets of Chicago, robbing banks and tracking down criminals, respectively.

4 Contest Technology

In addition to designing this set of tasks, we developed a significant amount of infrastructure so that these puzzles could be implemented and distributed as a UM binary.

4.1 Humlock

The Humlock compiler is an optimizing, obfuscating compiler for a high-level, functional language called UML. This language is similar to the core of Standard ML [9] (UML has no module system). It also extends that language in several minor ways, as discussed below. Humlock targets the UM platform. Using Humlock, we wrote nearly all of the code for the Codex in UML.

The Humlock implementation is based on the Hemlock compiler [11]. The compiler structured as a series of passes: a combinator parser, an elaborator into an internal language, a CPS language based on Appel's [6], and a low-level UM-like language. Like Hemlock, Humlock uses a uniform representation of values. Unlike Hemlock, however, it does not generate certified code; the UM has no such requirement.

To compile the more than 34,000 lines of contest code into an efficient binary, we implemented a number of optimizations in Humlock. These include constant folding and propagation, argument flattening, dead code elimination, unused argument removal, inlining, and closure representation optimizations.

Humlock also includes several features intended to facilitate its use for the contest. These include extended syntax for manipulating strings, including obfuscated strings, which use an encrypted representation in the output binary and in memory during execution. To ease our bandwidth requirements, Humlock includes support for self-decompressing binaries (using a variant of LZW compression [13]). To enable us to distribute the Codex several days before the contest began, Humlock also supports self-decrypting binaries using the ARCFOUR algorithm [12]. Even if contestants had been able to reverse engineer the UM specification before the contest began, they would have been unable to run the self-decryption without the key we provided at the contest start.

Garbage Collection. The Hemlock back-end targets a typed assembly language [10] that does not support explicit deallocation of heap objects. The UM architecture, in contrast, requires explicit deallocation. As part of the UM back-end, we implemented a garbage collector in UM assembly.

While most garbage collectors are designed to run on physical hardware, the Humlock GC is designed to run inside a virtual machine. We assume that the implementation of that virtual machine can manipulate memory much more efficiently than our collector. Our collector, therefore, makes no attempts to defragment memory. The UM architecture is also significantly more abstract than most hardware. For example, array identifiers (*i.e.*, heap addresses) are opaque, preventing us from reusing the low-order bits of addresses. We do, however, take advantage of some of the other, unusual features of the UM: while the null pointer (*i.e.*, zero) does not point to any garbage-collected object, it *does* point to a valid array, and we use this to avoid some costly null-checks.

The Humlock garbage collector uses a mark-sweep algorithm [8]. Every allocation of a value in a UML source program is compiled to an allocation instruction (including integers, which are boxed). The collector requires two additional header words for each object. One header word stores a set of flags that indicate the structure of the object and if the object is reachable. The other word is used to link all the heap objects together into a list so that the entire heap can be swept at the end of each collection.

Self-Checks. Humlock supports an option to prefix its output with a series of self-checks. These checks are designed to help ensure that the underlying UM implementation is correct and to provide meaningful messages in the case that it is not. The first of these checks tests to see whether or not the correct endianness is used when decoding instructions: the program begins with an unconditional jump that is encoded using the *wrong* endianness (these instructions have no effect if interpreted correctly). This jump takes the interpreter to a piece of code that prints out an error message (also encoded with the wrong endianness).

The self-check proceeds to test the behavior of each instruction while making as few assumptions as possible about the remaining ones. It first checks for off-by-one errors in the jump instruction. Then, assuming that it can reliably branch between two pieces of code, it checks for common errors in implementing the logical and arithmetic operators and in loading literal values. Memory-related instructions are validated by a series of array allocations, updates, and deallocations. Finally, the self-check tests whether the distinguished array that holds the program is correctly replaced by a "load program" instruction.

The original Codex contained a subset of these tests. When we discovered that some teams continued to have difficulty implementing the UM through the second day of the contest, we released an extended set of verbose self-checks in the form of a standalone benchmark called *SANDmark*. SANDmark can also be used to test the relative performance of different UM implementations.

Language	Size (LOC)	Notes
C	95	GCC 3.3.6
C#	217	Mono 1.1.17.1
Java	374	Blackdown JRE 1.4.2
Java (No JIT)	”	” with JIT disabled
O'Caml	255	version 3.09.2
Python	152	version 2.4.3
Scheme	428	Bigloo 2.8c
SML	255	MLton 20051202
SML (Unsafe)	”	” w/o safety checks
UMA	256	UM assembly on x86 Asm
x86 Asm	320	tweaked output of GCC

Table 1: UM Implementations. This table shows code sizes and infrastructure used in building or executing various UM implementations. The UMA implementation is a meta-interpreter hand-coded in UM assembly and run on top of the x86 assembly implementation. With the exception of the UMA implementation, no attempts were made to minimize the size of these implementations.

4.2 UM Implementations

We implemented the UM in many different languages. We also asked many different programmers to implement it. In doing so, we hoped to understand:

- any ambiguities in the specification,
- what kinds of mistakes programmers would make,
- what aspects of the UM would be difficult in each language.

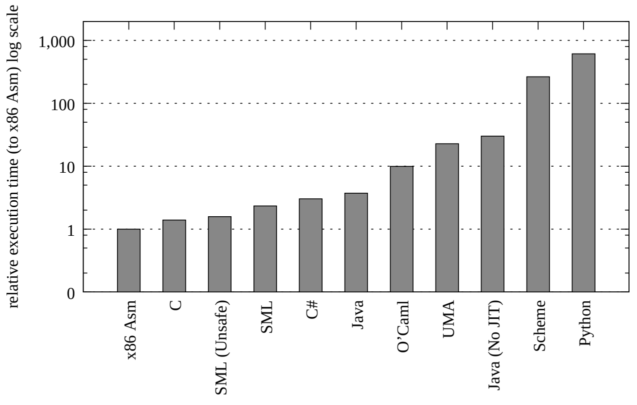
The problems encountered during our testing led to many of the original self-checks found in the Codex.

Nine different implementations are shown in Table 1 along with their sizes in terms of lines of code (LOC) and the compiler or run-time used in the experiments described below. Though their implementations are not described in this table, we also implemented the UM in Awk, Haskell, Perl, Twelf, and PostScript.

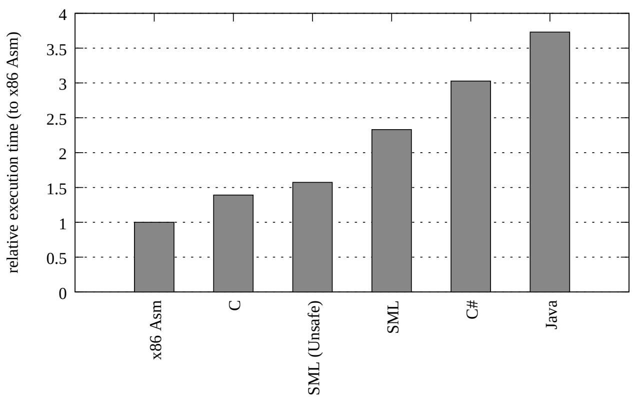
In addition, we built instruction-level time and space profilers and a debugging UM implementation with support for breakpoints as well as value and address watches. These proved to be critical in diagnosing problems in the functionality and performance of Humlock and its runtime system.

Implementation Challenges. Of the 14 instructions in the UM specification, the arithmetic operations posed the greatest challenges. The specification calls for unsigned 32-bit arithmetic. Many languages lack efficient support for 32-bit values. Others, like Java, do not support unsigned integer operations. To implement unsigned division in Java, we first widened both operands to 64 bits, performed the division, and then truncated the result.

PostScript was another difficult implementation. Its definition does not specify the size of integers. Worse yet, if the result of an integer operation is too large to fit in the representation used by a particular implementation, it will be silently converted to a floating point number!



(a)



(b)

Figure 13: UM Performance. These graphs show the relative performance of different UM implementations. The top graph shows a large set of implementations on a logarithmic scale; the bottom graph shows only the best performing implementations on a linear scale. The judges admit that they are not experts in many of these languages and caution that these results should only be used to show that the UM can be implemented efficiently in several high-level languages.

Performance. Figure 13 shows the performance of a number of our UM implementations relative to our fastest implementation. Each implementation is tested by running several iterations of DES encryption as implemented by the judges in UML and compiled by Humlock. Versions of the compilers and run-time systems used for each language appear in Table 1.

We emphasize that we are not experts in many of these languages and caution readers that the results do not necessarily reflect the quality of these languages or their implementations.

Despite this, one broad trend can be gleaned from the results: many high-level languages can be used to achieve good UM performance. While low-level languages like C and assembly offer the best performance, safe languages can offer comparable performance if the implementation is built with an optimizing compiler. The SML version compiled without run-time safety checks runs only 13% slower than the C version. Even with safety checks enabled, the performance is still only 67% slower. The Java and C# implementations also yield good performance when a JIT compiler is used.

Meta-Circular Interpreter. As a final demonstration of the "universality" of the UM, we implemented a highly optimized interpreter for the UM in UM assembly itself. Our implementation is quite compact, comprising only 256 instructions (or 1024 bytes). To run the interpreter, one simply prepends it to a UM binary and runs the result in any compliant UM implementation. Execution in the self-hosting interpreter is about 23 times slower than the underlying implementation. Several copies of the self-hosting interpreter can be prepended, each with a similar overhead.

4.3 Web Site

Humlock has an additional C++ back-end. A compiled UM program is included as binary data with our C implementation of the UM. This back-end is the basis of a Web platform called *UM-on-rails*, which allows UML-based server-side Web scripting. This technology enabled "fine-grained reuse" of the Codex's publication verifier in the Web submission system.

5 Results

The 2006 contest received more attention than any of its predecessors. More than 500 teams had registered by the beginning of the contest. Nearly 500 additional teams registered during the contest, including more than 60 teams in the last 24 hours.

By the end of the contest, 365 teams had submitted at least one publication—150 more teams submitted solutions than the previous high-water mark in 2004. 700 programmers contributed to the efforts of these teams. Just over half of the scoring teams had only one member. The majority of the remaining teams had two, three, or four members. By tracking the IP addresses used during team registration, we determined that there were scoring teams from each of the six inhabited continents.

5.1 Submissions

Participants submitted solutions in nearly 50 different programming languages. This included UM implementations in many of the languages we used as well as D, Dylan, Erlang, F#, Forth, Lisp, and Ruby.

The first UM implementation was completed less than one hour and 45 minutes after the start of the contest by the "jabber.ru" team. Many teams, however, were still working to complete a UM after more than 24 hours. This prompted us to release the expanded set of self-checks described above.

We designed many of the contest problems to be solvable by meta-programming (writing programs to generate the programs run on the Codex); indeed, we envisioned this to be the most effective way of solving many tasks. For example, we had imagined that many teams would write code to lay out 2D programs. Also, we had intended manually solving the later Adventure theorems (whose solutions can be viewed as programs by the Howard-Curry isomorphism) to be impractical. Nonetheless, several teams showed a surprising tenacity at completing these problems by hand. However, an informal survey of the results indicates that this was not a good strategy. For example, in the Adventure problem, most of the top scoring teams that submitted source code used an automated solver.

While nearly all previous contests required participants to submit executable code, we required only evidence that each problem had been solved (in the form of a publication).[6]

The Balance, O'Cult, and 2D problems used randomized testing to verify contestants' solutions. In each case, the verifier generated random tests using the text of the solution as a seed (to defeat attempts to table the answers). Several teams took advantage of this probabilistic verification by crafting solutions that were only partially correct, but would occasionally happen to pass all the tests. For example, after discovering correct solutions to the O'Cult problems, some teams experimented with slightly smaller solutions that omitted a single rule.

[6] We did require winning teams and those teams vying for the Judges' Prize to submit source code, but only to verify that there had been no foul play.

Team	INTRO	CIRCS	BLNCE	BLACK	BASIC	ANTWO	ADVTR	ADVIS	Total
1. Team Smartass	230	1414	1138	1000	100	400	810	336	5428
2. kuma-	230	1395	1079	1000	100	400	810	316	5330
3. Can't Spell Awesome Without ASM	230	1393	1067	1000	100	400	810	328	5328
4. Begot	230	1323	950	1000	100	400	810	328	5141
5. Witrala	230	1315	970	1000	100	200	810	314	4939
6. PLOP	230	1357	673	1000	100	400	810	168	4738
7. Funktion im Kopf der Mensch	230	1142	694	1000	100	400	810	325	4701
8. The Caml Riders	230	1266	988	1000	100	400	245	324	4553
9. Goochaeologists	230	1339	518	970	100	100	810	326	4393
10. Expansion	230	1394	124	1000	100	400	810	323	4381

Table 2: Final Standings. This table shows the final standings for the top ten teams and the breakdown of points across the different puzzles. The INTRO category includes the UM implementation and some UMIX-related publications. The remaining categories correspond to the 2D, Balance, Black-Knots, QVICKBASIC, Smellular Antomata, Adventure, and O'Cult problems.

Most individual test cases did not require a complete set of rules, so there was a small chance that such a solution could pass an entire test suite.

5.2 Scoring

Each team was responsible for validating its own solutions using the tools provided in the Codex and for submitting the resulting publications to the contest Web site. Thus at the moment the contest ended, we immediately knew the identities of the winners. In addition to relieving the judges of some post-contest work, this scoring mechanism had the benefit of supporting a live scoreboard during the contest. Several participants commented that this gave a more tangible sense of competition and added to their enjoyment of the contest.

A subset of the final standings is shown in Table 2. The complete final standings can be found on the contest Web site [5]. We conclude by awarding the contest prizes.

For his stupendous effort as one of the early leaders and the highest ranking single-person team, the Judges' Prize goes to Carl Witty. The judges declare that

> *Witrala is an extremely cool hacker!*

With 5328 points, third place goes to "Can't Spell Awesome Without ASM" and its members, Reid Barton, Tomek Czajka, John Dethridge, and Ralph Furmaniak. The judges declare that

> *Assembly is not too shabby!*

With 5330 points (only two more than the third place team!), second place goes to "kuma-" and its members, Yuu Shibata, Kazuhiro Inaba, Yusuke Endoh, and Nayuko Watanabe. The judges declare that

> *D is a fine programming tool for many applications!*

Finally, with 5428 points, first place goes to "Team Smartass" and its members, Ambrose Feinstein, Christopher Hendrie, Daniel Wright, and Derek Kisman. The judges declare that

> *2D is the programming language of choice for discriminating hackers!*[7]

Acknowledgements

The judges wish to acknowledge the help of the following students in constructing the contest. Ruy Ley-Wild designed the challenge problems for Balance and wrote much of its UML implementation. Adam Goode implemented the UM in C#, Java, and Python as well as for mobile devices. William Lovas conceived many of the ideas for Smellular Antomata. Jake Donham drafted much of the original story and contributed to the Balance and O'Cult languages. Jason Reed implemented the UM in Perl and originally conceived the Black-Knots puzzle. Noam Zeilberger bravely implemented the UM in Awk. Sridhar Ramesh, Spencer Whitman, Susmit Sarkar, and Akiva Leffert all tested beta versions of the Codex and puzzles.

The judges also wish to thank the ICFP Steering Committee and the 2006 General Chair, John Reppy, for

[7]See Section 3.4 for a description of 2D.

their support. Finally, the judges thank the many contest participants and acknowledge all their hard work. You have made all our efforts worthwhile!

References

[1] The third annual ICFP programming contest, 2000. URL: `http://www.cs.cornell.edu/icfp/`.

[2] The fourth annual ICFP programming contest, 2001. URL: `http://cristal.inria.fr/ICFP2001/prog-contest/`.

[3] The seventh antual ICFP programming contest, 2004. URL: `http://www.cis.upenn.edu/~plclub/contest/`.

[4] The eigth annual ICFP programming context, 2005. URL: `http://icfpc.plt-scheme.org/`.

[5] The ninth annual ICFP programming context, 2006. URL: `http://www.boundvariable.org/`.

[6] Andrew W. Appel. *Compiling with continuations*. Cambridge University Press, New York, NY, USA, 1992.

[7] Dan Brown. *The Da Vinci Code*. Doubleday Books, March 2003.

[8] John McCarthy. Recursive functions of symbolic expressions and their computation by machine. *Communications of the ACM*, 3:184–195, 1960.

[9] Robin Milner, Mads Tofte, Robert Harper, , and David MacQueen. *The Definition of Standard ML (Revised)*. MIT Press, 1997.

[10] Greg Morrisett, Karl Crary, Neal Glew, Dan Grossman, Richard Samuels, Frederick Smith, David Walker, Stephanie Weirich, and Steve Zdancewic. TALx86: A realistic typed assembly language. In *ACM SIGPLAN Workshop on Compiler Support for System Software*, pages 25–35, Atlanta, GA, May 1999.

[11] Tom Murphy, VII. Grid ML programming with ConCert. In *ML Workshop 2006*, September 2006.

[12] Bruce Schneier. *Applied cryptography: protocols, algorithms, and source code in C*. Wiley, New York, 2nd edition, 1996.

[13] Terry A. Welch. A technique for high performance data compression. *IEEE Computer*, 17(6):8–19, 1984.

Reviewer Comments

Section 3.8 is entirely too short and incomprehensible. My advice is that you replace it with the following:

O'Cult is a twisted term-rewriting system phrased in the terminology of aspect-oriented programming, where small and seemingly innocuous changes can have unpredictable global effects. The term language is unityped and consists only of constants and juxtaposition. For example, a programmer can use the terms `Z`, `(S Z)`, `(S (S Z))`, and so on to represent numerals in unary. The language does not prescribe any computational behavior for terms. Instead, these programs are "advised" by a list of external computation rules. For example, the Codex includes the following sample program for addition:

```
Add Z y => y;
Add (S x) y => S (Add x y);.
```

The first rule rewrites the addition of zero and any `y` to `y`; the second rule rewrites the addition of `(S x)` and `y` to the successor of recursively adding `x` and `y`. This advice correctly computes addition on numerals. However, it fails to work for nested arithmetic expressions such as (call this term `M` for future reference)

```
Add (Add (S (S Z)) (S Z))
     (Add (S (S (S Z))) (S (S Z)))
```

because of the curious operational semantics of this rewriting system.

Operational Semantics. The definition of the operational semantics requires an auxiliary notion of *matching* a term against the left-hand side of a rule, and as a result, acquiring a substitution for the variables in the rule. Matching is defined by a simple structural comparison, with a variable in the rule matching any term.

Given a list of rules, the transition system for the rewrite language begins in the expected fashion: the rules are considered in order, and the first applicable rule is applied (the substitution derived from the left-hand side of the rule is used to produce a new term).

Whether or not an individual rule applies to a term is determined in a less orthodox manner. If the rule matches the entire term, then the rule applies. The interesting case is when a rule does not match the entire term, but it does match in the subterms of a juxtaposition. For example, the second rule for `Add` does not apply to `M`, but it does apply to two of `M`'s subterms. In this case, the operational semantics are guided by the following aphorism:

"Advice when most needed is least heeded."

Whether the rule applies is determined by counting the number of matches on each side of the juxtaposition and then considering the following cases:

1. If the rule does not match in either position, it is not applied.
2. If the rule matches only in one position, it is recursively considered for application to that position.
3. If the rule matches both positions,
   (a) if one position has strictly more matches, the rule is recursively considered for application to *other* position. (The rule is least heeded in the position where it is most needed.)
   (b) if the rule matches the same number of subterms in both positions, the rule is not applied.

Case 3b is why the above rules for addition fail on the term `M` above: the relevant rule matches once in each subterm of a juxtaposition, so it does not apply.

Tasks. Players are asked to program two tasks in this language. The first task is to implement correct versions of addition and multiplication for arbitrarily nested arithmetic expressions. The second is to implement an optimizer for a very simple XML-like document language, a reference to the 2001 contest [2]. Solutions were checked with a test suite, and correct solutions were awarded points based on their size (pithier advice is better).

Solutions. There are various ways of defeating the operational semantics. Many solutions rely on one key idea: if at least one rule always applies to the top-level term, the idiosyncrasies of the operational semantics are never encountered. For example, several participants solved the addition task by adding rules to re-associate the arithmetic expressions like

```
Add (Add x y) z => Add x (Add y z);.
```

With this rule, nested additions in the top-level term will be pushed into the second summand until the first summand is a numeral, whereupon one of the above rules will apply. Another technique that is more general (but also more verbose) is to code the straightforward algorithm (*e.g.*, the rules for addition above) using an explicit stack to record the context of the computation. By making the stack explicit, all rules can be written so that they apply to the top-level term.