



STEPS Toward **Expressive Programming Systems** 2011 Progress Report Submitted to the National Science Foundation (NSF) October 2011

(In alphabetical order) Dan Amelang, Bert Freudenberg,
Ted Kaehler, Alan Kay, Stephen Murrell, Yoshiki Ohshima,
Ian Piumarta, Kim Rose, Scott Wallace, Alessandro Warth,
Takashi Yamamiya

This material is based upon work supported in part
by the National Science Foundation under
Grant No. 0639876. Any opinions, findings, and
conclusions or recommendations expressed in this
material are those of the author(s) and do not
necessarily reflect the views of the National
Science Foundation.

Viewpoints Research Institute, 1209 Grand Central Avenue, Glendale, CA 91201 t: (818) 332-3001 f: (818) 244-9761

VPRI Technical Report TR-2011-004

NSF Award: 0639876
Year 5 Annual Report: October 2011
STEPS Toward Expressive Programming Systems
Viewpoints Research Institute, Glendale CA

Important Note For Viewing The PDF Of This Report

We have noticed that Adobe Reader and Acrobat do not do the best rendering of text or scaled bitmap pictures. Try different magnifications (e.g. 118%) to find the best scale. Apple Preview does a better job.

The STEPS Project For The General Public

If computing is important—for daily life, learning, business, national defense, jobs, and more—then *qualitatively advancing computing* is extremely important. For example, many software systems today are made from millions to hundreds of millions of lines of program code that is too large, complex and fragile to be improved, fixed, or integrated. (One hundred million lines of code at 50 lines per page is 5000 books of 400 pages each! This is beyond human scale.)

What if this could be made literally 1000 times smaller—or more? And made more powerful, clear, simple, and robust? This would bring one of the most important technologies of our time from a state that is almost out of human reach—and dangerously close to being out of control—back into human scale.

An analogy from daily life is to compare the great pyramid of Giza, which is mostly solid bricks piled on top of each other with very little usable space inside, to a structure of similar size made from the same materials, but using the later invention of the arch. The result would be mostly usable space and requiring roughly 1/1000 the number of bricks. In other words, *as size and complexity increase, architectural design dominates materials.*

The “STEPS Toward Expressive Programming Systems” project is taking the familiar world of personal computing used by more than a billion people every day—currently requiring hundreds of millions of lines of code to make and sustain—and substantially recreating it using new programming techniques and “architectures” in less than 1/1000 the amount of program code. This is made possible by new advances in design, programming, programming languages, and systems organization whose improvement advances computing itself.

2011 One Page Summary

STEPS is now to the point where it is more fun to use and demonstrate than to talk and write about. Quite a bit of the original proposal is now working well enough to give all our presentations and write this entire report for the NSF and Viewpoints Research Institute websites. In the STEPS system, this document is *live*: the examples are actually runnable demos; the code shown is changeable and testable, etc.

A large part of the work over the last year has gone into making the test versions more real and usable.

Another addition is the presentation of code as “active essays” that explain the code by showing how to build it using the live code in STEPS as just another media type (see Appendix I for an example).

The balance has been directed towards creating additional visible features to round out STEPS’ comprehensive graphics abilities, and to make a number of invisible “chains of meaning” needed for our continuing experiments “at the bottom” to determine the best ways to connect the high-level problem-oriented languages in which STEPS is programmed to the “bare metal” CPUs of today. The latter issues are partly for pragmatic reasons (it is all too easy to use up thousands of lines of code at the bottom), and for esthetic reasons (the code at the bottom is often *ad hoc* and messy, especially when considerable optimizations need to be done).

One departure from the original plan is that our code-count was to be just of “runnable meaning” and not of the optimizations needed for real-time operations of the system. The idea was that one could reasonably prototype and debug the meanings on a supercomputer, and then add optimizations from the side that would not pollute the meanings (and where the meanings could be the ultimate tests of the optimizations). However, our graphics system ran its meanings faster than we had anticipated, and we found that doing just a little optimization was sufficient to make it pragmatically useful. This turned the project to make the first round of implementation usefully runnable. We have so far been counting all the code, including the optimizations even at “the bottom”. It is called “Frank” because it is a bit of a Frankenstein’s monster, and yet cute enough to be fun and very worthwhile to use for many real tasks.

Previous STEPS Reports

This summary and report are presented as incremental results to previous years. The direct line of reports starting with the original proposal can be found at [STEPS Reports]. However, for the convenience of present readers, we have gisted and recapped some of the previous work.

Yet To Be Done

Besides more rounding off of what has been made so far, we still have several more engine-room tasks to do, including: making the next version of our compact TCP/IP (that was used to make an experimental website) and installing it in a more comprehensive fashion, making the migratory caching architecture and mechanisms for moving computations around the Internet while eliminating most of traditional OS code, adding more processors to our suite of heterogeneous hosts, and more.

And we need to take another pass at the “connective tissue” of Frank and move the system from what is a well functioning patchwork to a more beautiful scaffolding. Frank has been invaluable in providing a working model of personal computing that is very close to the goals of STEPS without being so pristine as to cripple our degrees of freedom in inventing and designing.

Next STEPS for STEPS

STEPS is about modeling the main parts of personal computing “from the user down to the metal”, and its goals do not include improving personal computing. However, the best way to do the modeling in many cases has been to come up with much better designs. We have put many things on our list for “later” that we would like to do in STEPS that are outside the current STEPS goals. For example, the project has to provide the standard productivity tools, but we have been able to do this without making a big breakthrough in how applications are made. Yet the larger field desperately needs better ways to make applications for both professionals and end-users, and we intend to pursue this after next year.

Introduction and Context

STEPS is a 6 Year Project

We were originally funded for 5 years by NSF with matching funding from a private donor. The natural pace of the project required less funds per year than planned, and has resulted in enough funding for one more year. Though it is difficult at the beginning of an exploratory research project to accurately estimate person-years needed, it looks as though our estimate will be pretty accurate, but just spread over six years instead of five. This report is therefore a “progress so far report”, and next year will be the final one.

The Origin of the STEPS Project

The STEPS research project arose from asking embarrassing questions about many systems (including our own) such as: “Does this system have much too much code and is it messier than our intuition whispers?” Almost always the answer was “yes!” We wanted to find ways to write much smaller code, have it be more understandable and readable, and if possible, to have it be “pretty”, even “beautiful”.

STEPS Aims At “Personal Computing”

STEPS takes as its prime focus the modeling of “personal computing” as most people think of it, limiting itself to the kinds of user interactions and general applications that are considered “standard”. So: a GUI of “2.5D” views of graphical objects, with abilities to make and script and read and send and receive typical documents, emails and web pages made from text, pictures, graphical objects, spreadsheet cells, sounds, etc., plus all the development systems and underlying machinery:

- Programs and Applications – word processor, spreadsheet, Internet browser, other productivity SW
- User Interface and Command Listeners – windows, menus, alerts, scroll bars and other controls, etc.
- Graphics & Sound Engine – physical display, sprites, fonts, compositing, rendering, sampling, playing
- Systems Services – development system, database query languages, etc.
- Systems Utilities – file copy, desk accessories, control panels, etc.
- Logical Level of OS – e.g. file management, Internet and networking facilities, etc.
- Hardware Level of OS – e.g. memory manager, processes manager, device drivers, etc.

Our aim was not primarily to improve existing designs either for the end-user or at the architectural level, but quite a bit of this has needed to be done to achieve better results with our goals of “smaller, more understandable, and pretty”. This creates a bit of an “apples and oranges” problem comparing what we’ve done with the already existing systems we used as design targets. In some cases we stick with the familiar – for example, in text editing abilities and conventions. In other areas we completely redesign – for example, there is little of merit in the architecture of the web and its browsers – here we want vastly more functionality, convenience and simplicity.

Previous STEPS Results

The first three years were spent in general design and specific trial implementations of various needed functionalities for STEPS. These included high-quality anti-aliased 2.5D graphics that covers the same range of features as used in commercial personal computing, a viewing architecture, a document architecture, connection to the Internet, etc. Each of these was designed and implemented in a “runnable math” custom programming language that is highly expressive for its problem domain.

The fourth year was spent making a “cute Frankenstein monster” – called “Frank” – from these separate pieces. This has provided a workable “criticizable model” to help design the nicer, more integrated model that is the goal of STEPS.

Assessing STEPS

We set a limit of 20,000 lines of code to express all of the “runnable meaning” of personal computing (“from the end-user down to the metal”) where “runnable meaning” means that the system will run with just this code (but could have added optimizations to make it run faster). One measure will be what did get accomplished by the end of the project with the 20,000 lines budget. Another measure will be typical lines of code ratios compared to existing systems. We aim for large factors of 100, 1000, and more. How understandable is it? Are the designs and their code clear as well as small? Can the system be used as a live example of how to do this art? Is it clear enough to evoke other, better, approaches?

A number of the parts of STEPS now have a complete “chain of meaning” down through the various language systems to directly use the multicore CPUs we employ. We are still under our code limit of 20,000 lines, but we are not completely free of the scaffolding that has been supporting the Frank system. Until this happens, only relative judgments can be drawn.

Reflections On The Role Of Design

The STEPS Project was originally funded by NSF from a pool allocated to investigations in “The Science of Design”¹. The best results so far in STEPS have been design intensive—the parts that turned out to be the most understandable wound up using the kinds of design processes associated with mathematics, and the successful parts of the rest have much in common with structural engineering design.

The most surprising reflection at this point is just how much ground was covered with the latter. The large amount of covered ground was good—in that there is a lot that needed to be done compactly, especially in the “productivity applications” area of the project—but was also a bit underwhelming because just doing a lot of work on “parts and wholes”, partitioning modules and module boundaries, and other classic well known and recommended practices, resulted in a small high functioning system.

One way to think about design in this context is that a lot can be done before “something algebraic” can be pulled out of the needed organizations. And it is tempting to stop there—because one part of the engineering criteria has been achieved—“it works pretty darn well!

But the lack of deeper mathematical integrity will show up when the scales and ranges are increased, and it will not be clear where or how to tack on the new code.

This is where our field could use a real “Science of Design”, but we don’t really even have a real “Design Engineering” subfield. We do think that a “Science of Art” and an “Art of Design” do exist.

Science of Art

Part of our aim is to practice a “science of the artificial”, paralleling how natural science seeks to understand complex phenomena through careful observations leading to theories in the form of “machinery” (models) – classically using mathematics – that provide understanding by recreating the phenomena and having the machinery be as simple, powerful and clear as possible. We do the same, but draw our phenomena from *artifacts*, such as human-made computer systems.

We use many *existing and invented forms of mathematics* to capture the relationships and make “runnable maths” (forms of the maths which can run on a computer) to dynamically recreate the phenomena.

Art of Design

We *balance science with design* because the phenomena we generate only have to *be like* what we study; we are not reverse engineering. So the “math part” of science is used here to make *ideal designs* that can be much simpler than the actual underlying machinery we study while *not diluting the quality* of the phenomena. We have been struck by how powerfully the careful re-organization of long existing “bricks” can produce orders of magnitude improvements.

¹ Originally a concept developed at CMU by Newell, Simon, Freeman and others.

The STEPS Project in 2011

The overall goal of STEPS is to make a working model of as much personal computing phenomena and user experience as possible in a very small number of lines of code (and using only our code). Our total lines of code-count target for the entire system “from end-user down to the metal” is 20,000, which—if we can do a lot within this limit—we think will be a very useful model and substantiate one part of our thesis: that systems which use millions to hundreds of millions of lines of code to do comparable things are much larger than they need to be.

We think this is a better way to state our goal, because it is difficult to impossible—essentially apples vs. oranges—to come up with good code count comparisons between similar features in different systems. So when we say we can produce virtually all of 2.5D anti-aliased alphaed personal computer graphics in 457 lines of program, we can let that stand on its own as being “very small”—at least 100 times smaller than other extant systems.

Another “orange”: because there are several abstractly similar large scale design centers for personal computing (Apple, Microsoft, Linux/Open Office, etc.) we can devise a design center of our own that will be convenient to implement, and deliver “comparable” experiences—meaning that an end-user would feel they are experiencing “personal computing” but that it might be difficult to crisply quantify the comparative amounts of code in both systems. On the other hand, we are shooting for factors of 100 and even 1000, so there should be fairly easy *qualitative* comparisons—for example, being able to do quite a bit of the personal computing experience in a very small code count (such as 20,000 lines for everything).

Along the same lines, we can separate the “web experience” from how it is actually implemented. In this case—because the web is a very poor design—we have implemented an “alternative web” which delivers the same experience to the end-user but with a completely different and much more compact approach to design. We carry out similar finessing in other situations—for example, by eliminating the need for an “operating system”.

The main goal is to model the phenomena of personal computing, not *per se* to improve its design—and we have kept to that. Still, we wound up making many consolidations and improvements in order to allow our models to be really “small and nice”.

Our general approach is to pick an area whose parts seem to be related—for example: 2.5D anti-aliased alphaed computer graphics—try to find the mathematical relations that cover the desired phenomena, design a “problem oriented language” (POL) that is a “runnable math” version of the mathematics, implement that language, then use it to write the program code for the target area.

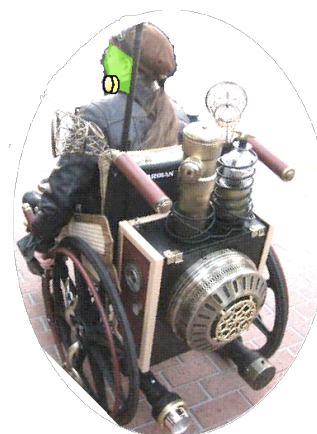
We have stitched together a working system from these parts—a kind of “Frankenstein’s monster”, called “Frank”—to better understand how the POLs can work together, and to provide integrated functionality that can match up with a user-interface design. For more history and context, see [STEPS Reports].



"Clear!"

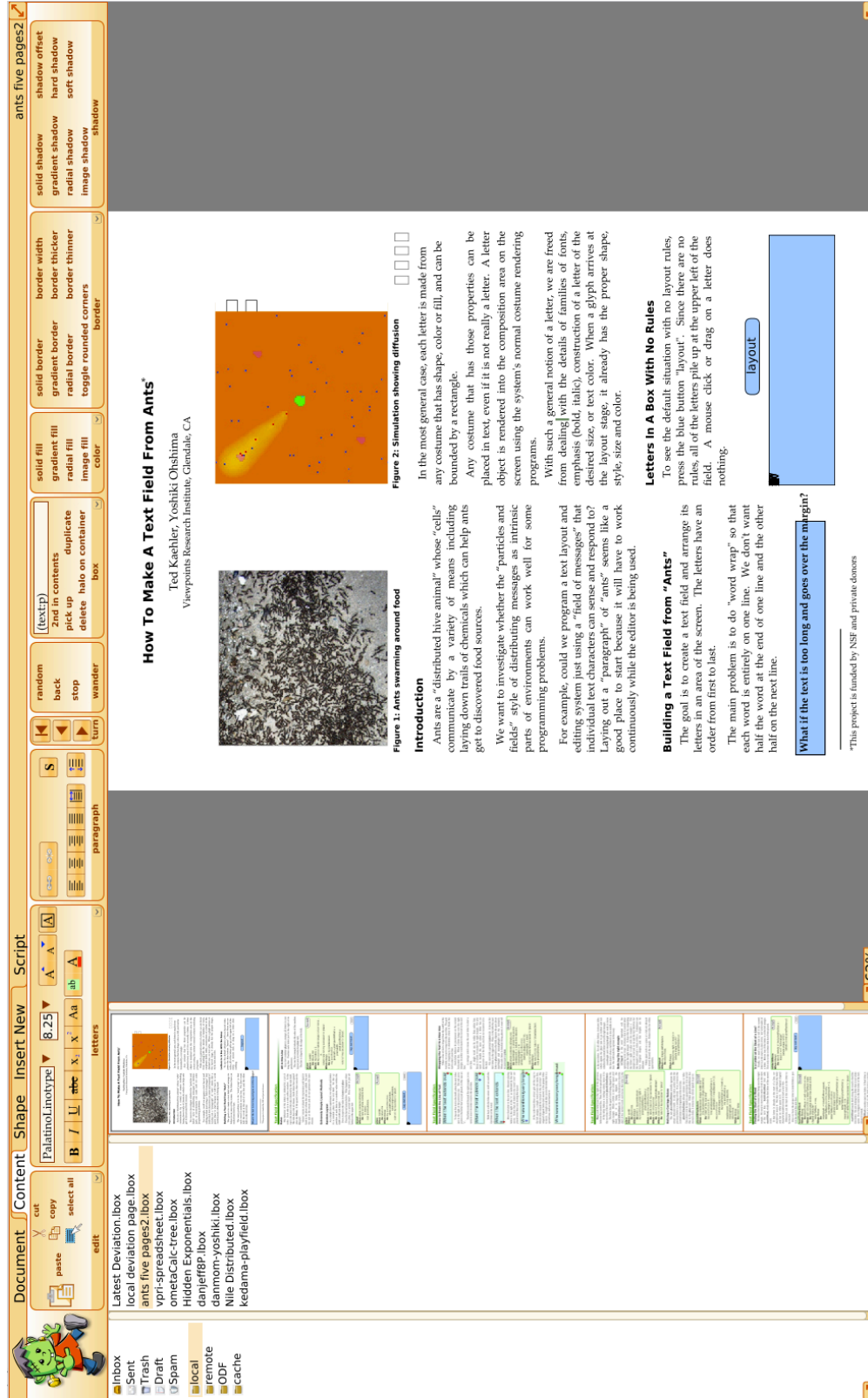
Frank in 2010 - Just coming alive

VPRI Technical Report TR-2011-004



Frank in 2011 - Supported by a steampunk wheelchair but heading out into the world

"Ribbon-style" GUI with "bubbles" and "spill areas"



Overview of the Frank System being used to make Appendix I of this report

VPRI Technical Report TR-2011-004

Slide out panel for holding scripts, etc

Area for making all constructions

Collapsible panel for showing thumbnails of all kinds of documents


Collapsible Panels for all external resources such as files, email, Internet, etc

Summary of how we have approached the following key parts of the STEPS project:


Graphics	Email	GUI	Data Bases	No Operating System
Sound & Movies	Internet "Browsing"	Desktop Publishing	Scripting	Simulation of "Time"
Graphical Objects & Views	"Universal Documents"	Presentations	Explanations	Dealing with CPUs
"Files"	Text Objects	Spreadsheets	Making Languages	Importing & Exporting

Graphics


A new set of mathematical relations was derived for the involvement of an arbitrary polygon and a pixel. This gave rise to a rendering scheme that directly produces excellent antialiasing. "Runnable math" – called "Gezira" – and a POL for it – called "Nile" – were devised and implemented. The several dozen standard compositing rules, shading, stroking, gradients, sampling, shadowing, etc. – 457 lines in total – were written in Nile and debugged to make a working graphics system, strong enough to be used for Frank, and to do all the graphics required for personal computing (and hence this report).



Dan Amelang

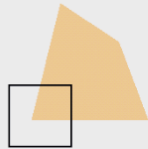


Mom



**Math Wins!
(especially MetaMath)**

Rendering



Nile: "Ken Iverson meets Christopher Strachey"

Do It
Reset

```

type Point = (x, y : Real)
type Bezier = (A, B, C : Point)
type EdgeSpan = (x, y, c, l : Real)
type EdgeSample = (x, y, a, h : Real)

| (a : Real) | : Real
{ -a if a < 0, a }

(a : Real) < (b : Real) : Real
{ a if a < b, b }

(a : Real) ~ (b : Real) : Real
(a + b) / 2

DecomposeBeziers : Bezier >> EdgeSample
  V (A, B, C)
  inside = ([ A ] = [ C ] v [ A ] = [ C ])
  if inside.x A inside.y
    P = [ A ] < [ C ]
    w = P.x + 1 - (C.x ~ A.x)
    h = C.y - A.y
    >> (P.x + 1/2, P.y + 1/2, w x h, h)
  else
    ABBC = (A ~ B) ~ (B ~ C)
    min = [ ABBC ]
    max = [ ABBC ]
    nearmin = | ABBC - min | < 0.1
    nearmax = | ABBC - max | < 0.1
    M = {min if nearmin, max if nearmax, ABBC}
    << (M, B ~ C, C) << (A, A ~ B, M)

CombineEdgeSamples : EdgeSample >> EdgeSpan
(x, y, A, H) = 0
V (x', y', a, h)
  if y' = y
    if x' = x
      A' = A + a
      H' = H + h
    else
      l = {x' - x - 1 if |H| > 0.5, 0}
      >> (x, y, |A| < 1, l)
      A' = H + a
      H' = H + h
    else
      >> (x, y, |A| < 1, 0)
      A' = a
      H' = h
  >> (x, y, |A| < 1, 0)

Rasterize : Bezier >> EdgeSpan
= DecomposeBeziers - SortBy (@x)
- SortBy (@y) - CombineEdgeSamples

```

$$\sigma(P, Q) = (Q_y - P_y)(x + 1 - \frac{Q_x + P_x}{2})$$

$$\gamma(P) = \min(x + 1, \max(x, P_x)), \min(y + 1, \max(y, P_y))$$

$$\omega(P) = \frac{1}{m}(\gamma(P)_y - P_y) + P_x, m(\gamma(P)_x - P_x) + P_y$$

$$coverage(\overline{AB}) = \sigma(\gamma(A), \gamma(\omega(A))) + \sigma(\gamma(\omega(A)), \gamma(\omega(B))) + \sigma(\gamma(\omega(B)), \gamma(B))$$

$$\min(|\sum coverage(\overline{AB}_i)|, 1)$$

"The Formula" in Nile (~ 45 LOC)

(Above) The basic rendering formula for the Gezira graphics system in the "runable math" computer language "Nile" was previously about 80 lines of code. It has now been reformulated for clarity and simplicity, and currently only requires 45 lines of Nile code.

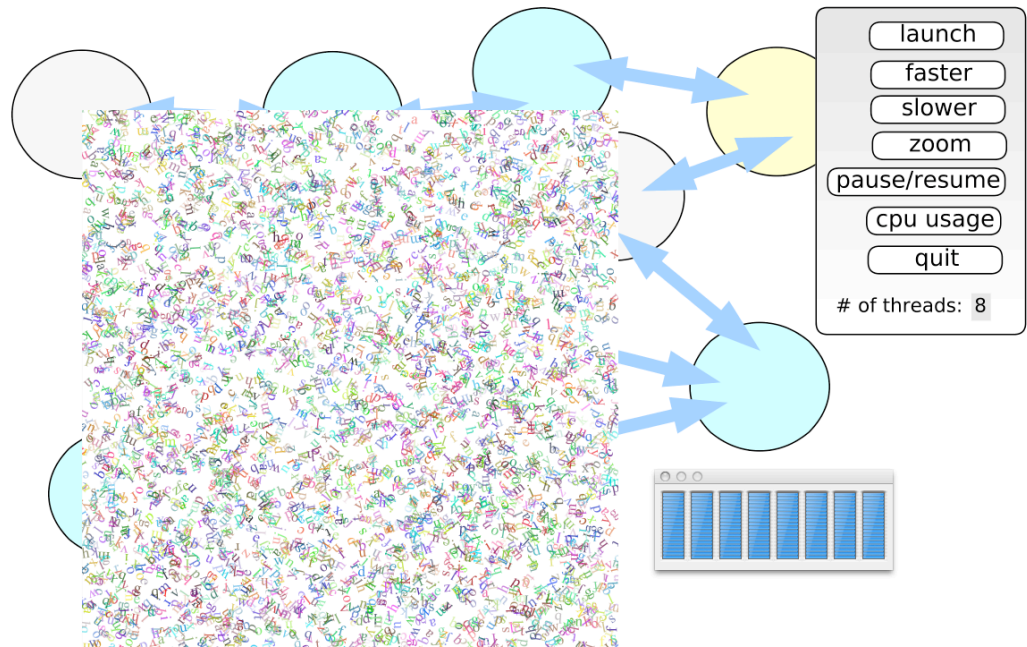
The Nile language could be described as a data flow stream language with functions at the nodes whose expression form reminds one a bit of APL—both large-scale structures (pixel maps) and small-scale features (partial word arithmetic for color and alpha components) are automatically comprehended.

The most significant development of the past year for the Nile subproject has been the creation of a multithreaded runtime for executing Nile programs. Part of the standard demo is to provide Nile with access to the 4 cores (and 8 threads) of a MacBook Pro and have it automatically map computational processing to computations.

Several new developments in the Nile system include adding a few more lines of Nile to include more graphical features (more control over gradients, Gaussian shadows, etc.) There are now 457 lines of Nile code used to produce all of the STEPS graphics.

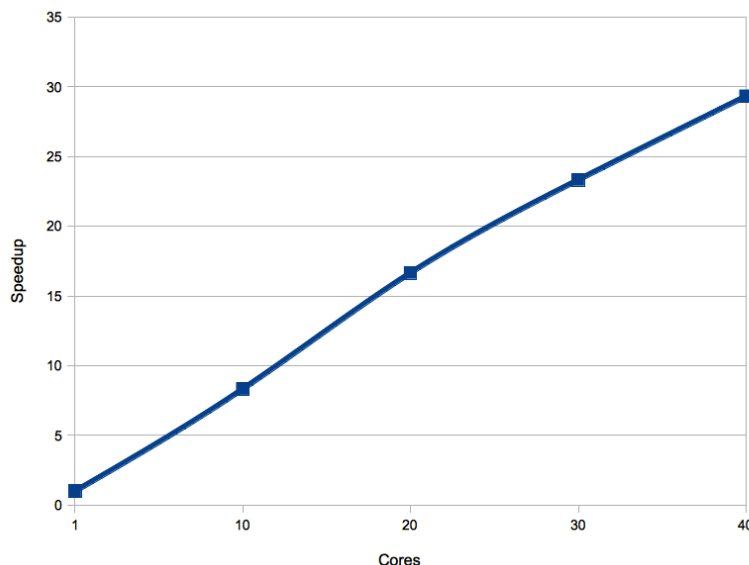
The screenshot shows 5000 anti-aliased translucent rotating “falling” characters with all 4 cores and 8 threads fully engaged in the computation.

The code examples here are all live and can be edited to see their effect.

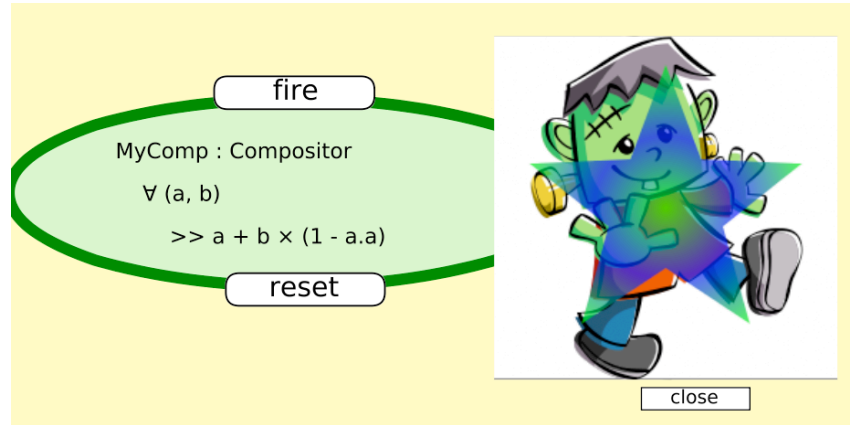


The runtime provides a substantial performance boost on multicore hardware, with an almost linear speedup per core. We see our approach as a successful, though partial solution to the parallel programming problem, because Nile programs, being very-high level, are implicitly parallel. This means that programmers are not required to concern themselves at all with the details of parallel execution.

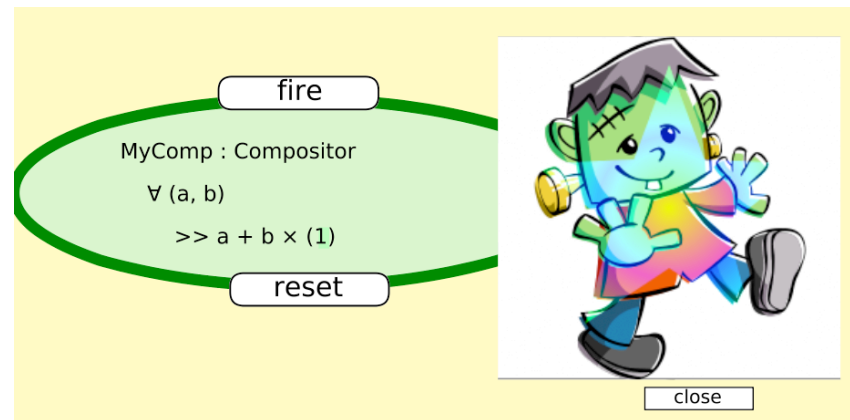
The multithreaded runtime is used by our 2D vector graphics renderer to support other STEPS subprojects while maintaining good performance. To gauge performance potential farther into the future, we have also run one of our rendering benchmarks on a 40-core machine, resulting in a nearly 30 times speedup (see figure 2). We believe that this demonstrates that implicit parallelism provides an avenue for high performance that doesn't sacrifice the minimalism and clarity that are important goals of the STEPS project.



Example of live editing of deep graphics code.



Here we are removing the subtraction of the translucency (alpha) component from the compositing rule.



Recently, the syntax of Nile has evolved to more closely match the notation of mathematical statements. This is to improve the clarity of programs that borrow heavily from traditional mathematical concepts. This is a continuation upon the existing mathematics support in Nile, which already included the definition of new types (e.g., vectors, bezier curves, colors) and operators on these types (e.g., dot product, interpolation, normalization). Like their mathematical equivalents, operators in Nile can be defined and applied using infix, outfix, prefix or suffix notation. Nile also permits the use of Unicode characters for operators and variable names. Here we give some examples of the mathematical syntax of Nile.

An expression for linear interpolation using a Unicode character for a variable and juxtaposition for multiplication:

$$\alpha x - (1 - \alpha)y$$

Definition of the absolute value operator, which makes use of the conditional expression syntax modeled after piecewise function notation:

$$\begin{aligned} &| (a:\text{Real}) | : \text{Real} \\ &\{ -a, \text{ if } a < 0 \\ &\quad a, \text{ otherwise } \} \end{aligned}$$

Definition of a 3D vector type and dot product operator:

$$\begin{aligned} &\text{type Vector} = (a:\text{Real}, b:\text{Real}, c:\text{Real}) \\ &(v1:\text{Vector}) \cdot (v2:\text{Vector}) : \text{Real} \\ &((a1, a2, a3), (b1, b2, b3)) = (v1, v2) \\ &\quad a1b1 + a2b2 + a3b3 \end{aligned}$$

Complex number type and multiplication operator:

$$\text{type Complex} = (a:\text{Real}, b:\text{Real})$$

```
(z1:Complex) (z2:Complex) : Complex
((a, b), (c, d)) = (z1, z2)
(ac - bd, bc + ad)
```

Affine transformation matrix type, point type, and multiplication operator:

```
type Point = (x:Real, y:Real)
type Matrix = (a:Real, b:Real, c:Real, d:Real, e:Real, f:Real)
(M:Matrix) (P:Point) : Point
(a, b, c, d, e, f) = M
(x, y) = P
(ax + cy + e, bx + dy + f)
```

Definition of a Nile process for transforming a stream of beziers by a matrix:

```
type Bezier = (A:Point, B:Point, C:Point)
TransformBeziers (M:Matrix) : Bezier >> Bezier
  ∀ (A, B, C)
    >> (MA, MB, MC)
```

Sound & Movies

Much of Nile is about ways to do digital signal processing efficiently and powerfully, so it can be used to generate and sample sounds as well as images. Similarly, Nile runs fast enough to be able to render picture frames fast enough to show movies.

Graphical Objects and Views

There is really just one kind of graphical object in STEPS—we can think of it as an entity that “can view”, has “shape” and “can carry”. They are recursively embeddable, so all composition and 2.5D layerings are built up from this basic mechanism. All search is done at the “organizational level”, etc.

“Files”

In STEPS, the equivalent of files are storage representations for STEPS objects (see page 7 for how these are unified in the UI.) These are thought of as being part of the “mobile network of virtual machines” which “float over the Internet” (see “NoOS” section) that make up STEPS computations.

Email

As mentioned, email is done with standard STEPS documents, and is presented to the user as part of the generalized way to look at external resources (see page 7). One view of this is to think of “gatherings” of objects (a generalization of folders as dynamic retrievers of object types—aka “spreadsheet cells”). One gathering is called “Inbox”, etc.

Internet “Browsing”

The equivalent of the “web experience” is provided in a very similar way to email. “Web pages” are just standard STEPS universal documents that contain hyperlinks, and can be browsed using the universal “In/Out” interface (see page 7). It is important to note that the main tool for most “web” perusals—Google—only exists on one’s personal computer as a very simple client interface. All the search machinery is external. We use this also. The experience of using the Web is very easy to provide using STEPS documents. And, it is much much easier to make “web” content using the WYSIWYG authoring provided by the current Frank system. (The Web should have used HyperCard as its model, and the web designers made a terrible mistake by not doing so.)

How To Make A Text Field From Ants*

Ted Kaehler, Yoshiki Ohshima
Viewpoints Research Institute, Glendale, CA

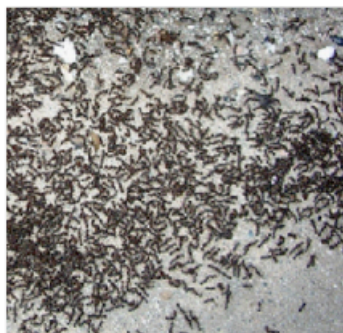


Figure 1: Ants swarming around food

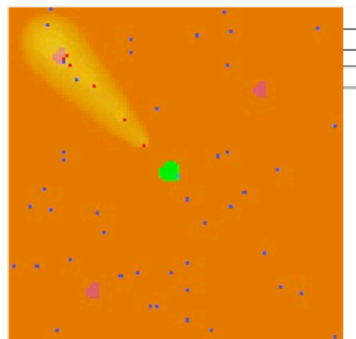


Figure 2: Simulation showing diffusion

Introduction

Ants are a “distributed hive animal” whose “cells” communicate by a variety of means including laying down trails of chemicals which can help ants get to discovered food sources.

We want to investigate whether the “particles and fields” style of distributing messages as intrinsic parts of environments can work well for some programming problems.

For example, could we program a text layout and editing system just using a “field of messages” that individual text characters can sense and respond to? Laying out a “paragraph” of “ants” seems like a good place to start because it will have to work continuously while the editor is being used.

Building a Text Field from “Ants”

The goal is to create a text field and arrange its letters in an area of the screen. The letters have an order from first to last.

The main problem is to do “word wrap” so that each word is entirely on one line. We don’t want half the word at the end of one line and the other half on the next line.

What if the text is too long and goes over the margin?

In the most general case, each “letter” is made from any costume that has a shape, color or fill, and can be bounded by a rectangle.

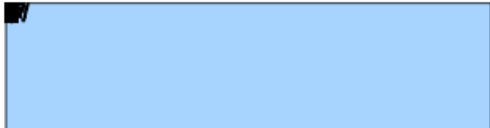
Any costume that has those properties can be placed in text, even if it is not really a letter. A letter object is rendered into the composition area on the screen using the system’s normal costume rendering programs.

With such a general notion of a letter, we are freed from dealing with the details of families of fonts, emphasis (bold, italic), construction of a letter of the desired size, or text color. When a glyph arrives at the layout stage, it already has the proper shape, style, size and color.

Letters In A Box With No Rules

To see the default situation with no layout rules, press the blue button “layout”. Since there are no rules, all of the letters pile up at the upper left of the field. A mouse click or drag on a letter does nothing.

layout



*This project is funded by NSF and private donors

Example of a “universal document” being used for “desktop publishing”.

Text Objects

Text is organized as “paragraph objects”. This paragraph is written in/using a paragraph object.

Nile can render from standard font formats, so the “text problem” is to do the different visual layouts that are needed for personal computing, and to allow WYSIWYG editing in ways that are completely standard. There are many scales of layout in personal computing, and we did not want to write special-purpose code for them. Instead, we thought it would be useful, compact and fun to adapt a “particles and fields” (or “ants”) metaphor to give the individual components rules to follow and let them sort out and solve the layout constraints.

We made a POL—called “Ants”—for this. The total number of rules needed for a WYSIWYG editor and layout paragraph is 35. An “active essay” explanation of how the layout is carried out in Ants is found in Appendix I. The code examples are live and can be run, edited and experimented with by the reader (but not in the PDF version of this report).

Example of making text paragraphs using distributed objects and “particles and fields”.

We have “released” the text characters in one of the paragraphs, and they are roaming freely.

When they are told to “feel” the fields around them, they will start to line up and follow the leader to format themselves.

How To Make A Text Field From Ants*

Ted Kaehler, Yoshiki Ohshima
Viewpoints Research Institute, Glendale, CA




Figure 1: Ants swarming around food

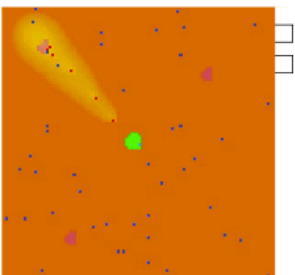
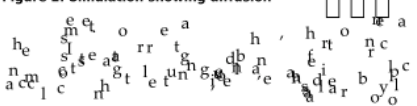


Figure 2: Simulation showing diffusion

Introduction

Ants are a “distributed hive animal” whose “cells” communicate by a variety of means including laying down trails of chemicals which can help ants get to discovered food sources.

We want to investigate whether the “particles and fields” style of distributing messages as intrinsic parts of environments can work well for some ~~programming problems~~



Any costume that has those properties can be placed in text, even if it is not really a letter. A letter object is rendered into the composition area on the screen using the system’s normal costume rendering programs.

GUI

Graphical user interfaces are hard to design, and we are fortunate that the STEPS goals are to mimic personal computing phenomena (so we don’t have to invent a new GUI from scratch).

Our approach—to use universal documents rather than separate applications—results in great simplicity, but requires a user interface that can exhibit and handle many properties.

For STEPS, we have adopted a “ribbon UI” style (see page 7 of this report). The ribbon style of interaction is essentially a Mac-style tool-bar, but instead of pull-down menus from the tool-bar, the next level items are distributed sideways in a “ribbon” of “bubbles” each of which shows the most likely properties and commands. Each bubble has an initially closed “spill area” for commands and properties that don’t fit in the bubble.

The UI tries to show as much as possible. Very careful choices of subdued coloration and visible features allow quite a bit to be shown without “a visual slap in the face”.

In the STEPS interface, many similar operations that are different in standard personal computing have been consolidated. For example, dealing with “files”, “web”, and “email” are all abstractly similar and have to do with “bringing things in, and sending things out”. There is just one interface for all, and one way to browse and search for resources.

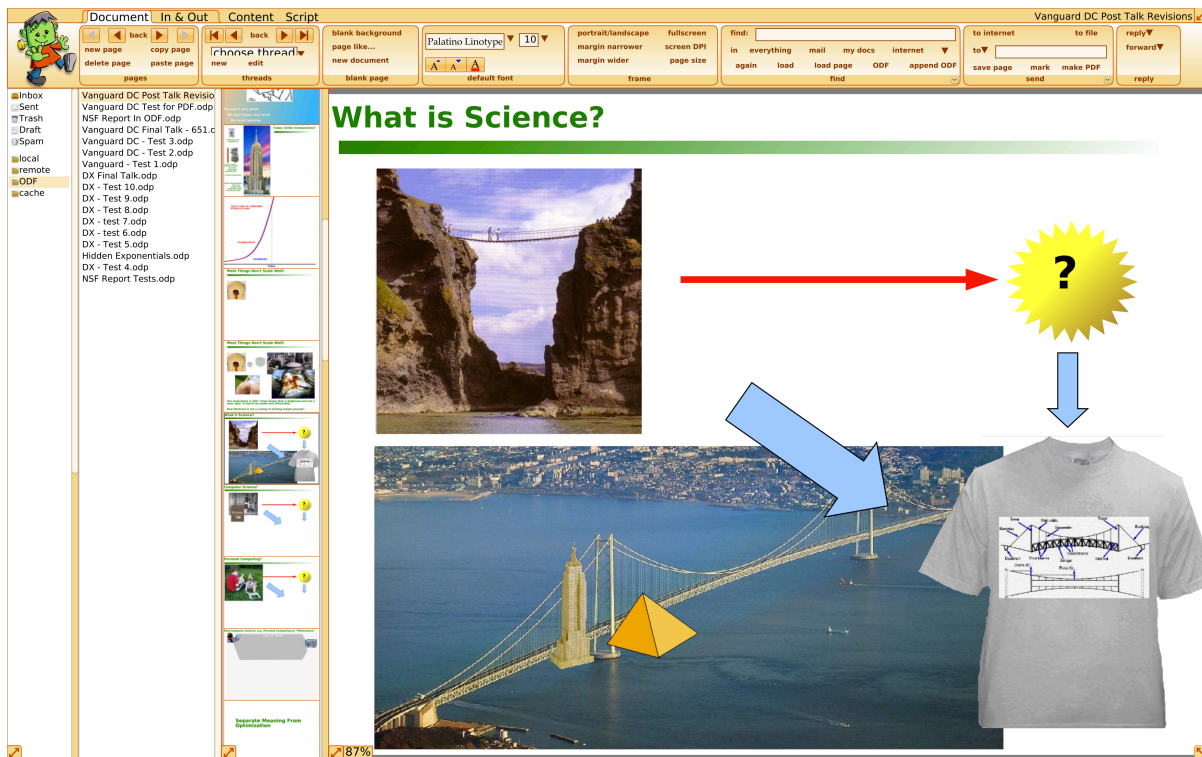
Desktop Publishing

In the original invention of today's style of personal computing at Xerox PARC, the basic idea was to "do desktop publishing right" and everything else could then be made from this. Visually, this certainly makes sense (desktop publishing has to be able to portray everything that can be thought of and displayed). The PARC scheme used "dynamic language scripting" to give more functionality to the graphical objects to allow "applications" (which were really what would be called "mashups" today). This approach was later taken in HyperCard, and we still think it is much better than the standard practice of hermetically sealing in rigid features using statically dead languages.

We take the dynamic approach here, and in total. So, for example, the STEPS GUI itself is actually a "DTP document" and can be edited in just the same ways as other more familiar document types. What are normally "separate applications" in a "productivity suite" are just standard universal documents used in different ways. Some of these are detailed here, but we emphasize that these are just different uses and perspectives on the same basic framework (see page 7 for the generalized authoring interface).

Presentations

Presentations are standard documents, often in landscape aspect (but not necessarily so) whose sequencing of builds and "slide flipping" are done by the STEPS general scripting facilities. In one sense, no extra code needed to be written to get "presentations" from "desktop publishing"; in STEPS they are one thing. Sequencing of "build effects" and "slides" is done by the general scripting system. All of the illustrations in this report are actually "live slides" from presentations, and this report itself is also a "presentation".



Here is a "slide" (just a regular universal document page) being made for an elaborate talk with live demos that was given in early October.

Spreadsheets and Cells

Spreadsheets in STEPS are just another “graphical object with dynamic properties”—a sibling to text paragraphs (which they contain among other things). We like spreadsheets, but think of individual cells as the important idea, and so our spreadsheets are actually documents of cells that can be organized in a variety of ways.

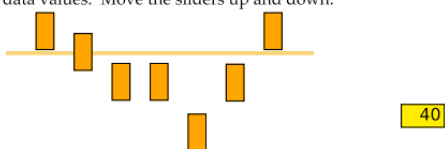
	A	B	C	D	E	F	G	H	I	J	K	L	M	N
1	Amalgamated Inc.													
2														
3		July '11	Aug '11	Sept '11	Oct '11	Nov '11	Dec '11	Jan '12	Feb '12	Mar '12	Apr '12	May '12	June '12	Total
4	Operations Expenses													
5	Office Space	1000	1000	1000	1000	1000	1000	1000	1000	1000	1000	1000	1000	12000
6	Shipping/Fed Ex	100	100	100	100	100	100	100	100	100	100	100	100	1200
7	Computer Equipment/C	5000	2000	5000	2000	2000	5000	2000	2000	2000	5000	2000	2000	36000
8	Storage	120	120	120	120	120	120	120	120	120	120	120	120	1440
9	Legal	250	250	250	1000	250	250	250	1000	250	250	250	1000	5250
10	HR Services	2000	2000	2000	2000	2000	2000	2000	2000	2000	2000	2000	2000	24000
11	Financial Services	10000	5000	5000	5000	5000	10000	5000	5000	5000	5000	5000	10000	75000
12	Consulting Services	15000	15000	15000	15000	15000	15000	15000	15000	15000	15000	15000	15000	180000
13	Biz Licenses/Insurance/	6000	1000	250	250	250	250	250	250	250	250	250	1000	10250
14	T&E	5000	5000	10000	5000	5000	5000	10000	5000	5000	10000	5000	10000	80000
15	Petty Cash	100	100	100	100	100	100	100	100	100	100	100	100	1200
16														
17	Subtotal	44570	31570	38820	31570	30820	38820	35820	31570	30820	38820	30820	42320	426340
17		=B5 + B6 + B7 + B8 + B9 + B10 + B11 + B12 + B13 + B14 + B15												

For example, a cell can be embedded in any text. They can be embedded in “tables” (to make what appears to be an ordinary spreadsheet). They can be organized as lists, etc. Below are examples from a one page “active essay” explaining the concept of Standard Deviation. Active cells are used freely. The yellow one shows the average – and drives the height of the horizontal bar. The gray cells are embedded in the text and show values from the sliders in the graph and results of calculations.

The Standard Deviation is a widely used measurement of variability or diversity of a set of measurements. Volatility in the stock market is the Standard Deviation of the value of a stock. Standard Deviation shows how much variation or “dispersion” there is from the “average” (the mean). A low standard deviation indicates that the data points tend to be very close to the mean, whereas high standard deviation indicates that the data is spread out over a large range of values.

A useful property of Standard Deviation is that, unlike the variance, it is expressed in the same units as the data.

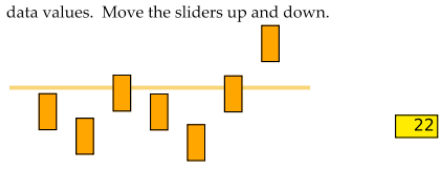
Let’s use the vertical position of the sliders below as the data values. Move the sliders up and down.



The first step is to find the average of the data samples. The average is the sum of the values divided by the number of values. The sum of the values is 71 + 55 + 31 + 31 + -9 + 31 + 71 which is 281. There are 7 data values. The average is 40.

Now that we know the average, we can find out how much each sample differs from the average. Subtract the average from each sample. We need the square of each difference. Add up the squares 952 + 221 + 84 + 84 + 2415 + 84 + 952 and take the average of those to get 684. This called the variance. Taking the square root gives a Standard Deviation of 26. It is in the same units as the data values. For a normal distribution (or bell curve) of data, 68.2% of the values will be within one standard deviation of the average.

data values. Move the sliders up and down.



The first step is to find the average of the data samples. The average is the sum of the values divided by the number of values. The sum of the values is 16 + -3 + 31 + 16 + -9 + 31 + 71 which is 153. There are 7 data values. The average is 22.

Now that we know the average, we can find out how much each sample differs from the average. Subtract the average from each sample. We need the square of each difference. Add up the squares 34 + 618 + 84 + 34 + 952 + 84 + 2415 and take the average of those to get 603. This called the variance. Taking the square root gives a Standard Deviation of 25. It is in the

Our cells are also more general than those of a standard spreadsheet. They can be thought of as “dynamic retriever-calculators” at the bottom, and as “generalized viewers” at their top visible manifestation. Many interesting and useful applications can be built quickly and easily with spreadsheet cells (we show an example later in this report).

Data Bases and Spreadsheets

HyperCard automatically indexed everything for search. The Mac does this for most files that have searchable content. Google does this for most web pages. STEPS does this for its “pages”. We can see that today, a personal computing “data base” is a way to make documents that allow various kinds of search engines to extract, index, and invert content for fast retrieval. A particularly useful object for retrieval is our generalized spreadsheet cells, which are constantly “looking” for objects that satisfy their formula, and which gather these objects into a “comprehension” that can be a further object of filtering.

An example of a “data base application” in STEPS is the “Condor PDA” application we did with the cooperation of SAP, using some of their design ideas and salepeople data.

The “app” is just a standard document with text and embedded cells.

The cells reach out to a global corporate database, which is also portrayed as a spreadsheet.

Many “end-user” apps can be made directly from the STEPS scripting facilities.

Scripting

In a sense, the entire Frank system is built from dynamic scripting. Most of the scripts for the general end-user are seen and created by using a tile-based scripting system (similar to Etoys and Scratch) but able to reach much deeper to get operations and resources. Any button or property can be dragged from the Frank ribbon UI and dropped on a page to create a script line. Similarly, any script or value can be dragged into the ribbon UI to make a button or property.



	A	B	C	D	E	F	G
1	Account name	Month	Sales Rep	gross	adjusted	invoiced	paid
2							
3	Shop Mart	December 2010	John Jones	12910	1634	0	0
4	Local Machinery	December 2010	John Jones	10000	1000	9000	9000
5	High Clothes	December 2010	John Jones	6230	580	5650	5650
6	Shop Mart	November 2010	John Jones	3457	169	3288	0
7	Local Machinery	November 2010	John Jones	8934	2362	6572	6572
8	High Clothes	November 2010	John Jones	8323	1759	6564	0
9	Shop Mart	October 2010	John Jones	9467	362	9105	9105
10	Local Machinery	October 2010	John Jones	25854	1743	24111	22000
11	High Clothes	October 2010	John Jones	6584	3274	3310	3310
12							
13	J Dance Studio	December 2010	Sam Smith	1681	634	0	0
14	ABC Warehouse	December 2010	Sam Smith	2614	846	1768	1768
15	Zees Nursery	December 2010	Sam Smith	6223	280	5943	5943
16	J Dance Studio	November 2010	Sam Smith	5197	1153	4044	0
17	ABC Warehouse	November 2010	Sam Smith	5733	2112	3621	3621
18	Zees Nursery	November 2010	Sam Smith	4568	1949	2619	0
19	J Dance Studio	October 2010	Sam Smith	3400	392	3008	3008
20	ABC Warehouse	October 2010	Sam Smith	5170	1823	3347	3347
21	Zees Nursery	October 2010	Sam Smith	9514	3194	6320	6320
A3	=						

Explanations

One of the goals expressed in the original proposal that NSF requested is that the system be presented as “A Compact And Practical Model of Personal Computing As A Self-Exploratorium”. Doing the “Exploratorium” part well requires that all parts of the system can not only be navigated to, but that safe “hands-on” experiments be possible by interested perusers. One way to do this is to have the code in the system presented in the form of “active essays” which “explain by coaching” the end-user how to build the code and why it works. An example of this in STEPS—which explains and builds a text layout facility using a rule-based POL—is included in Appendix I.

We are still learning how to do this to allow clarity and deep but safe experiments. Clarity is more difficult than “safe”, because there are a variety of ways to isolate experiments so that even extreme measures can be tried and still allow a safe “UNDO” to be carried out. One of these is the “Worlds” mechanism of STEPS (discussed in the next section).

For the purpose of the present project, clarity in STEPS is achieved primarily by careful writing, and through having the appearance and organization of our POLs be as important as their functionality. One could imagine a future system in which the user interface was a pro-active agent that could actively help perusers learn the system via answering questions and guidance.

An interesting trade-off is between the conventional route of using a single language that does not fit any problem area well, and this plus other practices leads to immense code-counts, vs. our tack of making a variety of POLs, each of which does its job with tiny code-counts. We think (and assert) that in the end, human brains are overwhelmed by size, and that the additional overhead of learning several languages more than pays for itself in producing compact “mind-sized” code. There are no substantiations of this at present, but it would be an interesting experiment to try as a follow-on to this project.

How To Make A Text Field From Ants*

Ted Kaehler, Yoshiki Ohshima
Viewpoints Research Institute, Glendale, CA




Figure 1: Ants swarming around food

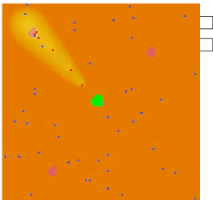


Figure 2: Simulation showing diffusion

Introduction

Ants are a “distributed hive animal” whose “cells” communicate by a variety of means including laying down trails of chemicals which can help ants get to discovered food sources.

We want to investigate whether the “particles and fields” style of distributing messages as intrinsic parts of environments can work well for some programming problems.

For example, could we program a text layout and editing system just using a “field of messages” that individual text characters can sense and respond to? Laying out a “paragraph” of “ants” seems like a good place to start because it will have to work continuously while the editor is being used.

Building a Text Field from “Ants”

The goal is to create a text field and arrange its letters in an area of the screen. The letters have an order from first to last.

The main problem is to do “word wrap” so that each word is entirely on one line. We don’t want half the word at the end of one line and the other half on the next line.

What if the text is too long and goes over the margin?

*This project is funded by NSF and private donors

Text Field Specification

Rules

The behavior of the letters is defined by a set of rules. Each rule is in a rule editor window. At the top is the name of the rule, followed by the rule itself. The rule has a list of clauses. Each clause has a guard after the **When**. If the guard is true, then execute the **Do** part.

“return” means evaluate the expression and hand it back to the place where the rule was called. We exit the rule at the return and do not perform the later clauses.

All in One Line

Now let’s redefine **place** to arrange all letters in one long line. Position each letter just to the right of its predecessor.

The one line example inherits the rules of the random layout and overrides the **place** rule.

The line is clipped by the edge of the field.

```
place
When I amNil
Do return me
When my index = 1
Do my position (whole shape leftATY 0)+4 ,
4.
return rule tell my successor to 'place'.
When Always
Do pred := my predecessor.
my pivotPosition
pred left + pred pivotOffset x +
pred width ,
pred pivotPosition y.
rule tell my successor to 'place'.
```

Extremely Simple Layout Methods

Random Layout

As a simple first experiment, we will put each letter in a random place in the text field. Set the x,y position of each letter to a random value within the field’s width and height.

Press Accept in the rules for **layout** and **place**.

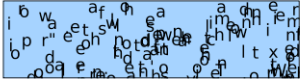
Press the layout button below. What happens when you press it a second time? Press reset to put all letters at the upper left.

```
layout
When whole contents is Empty not
Do rule tell whole contents first
to 'place'.
rule processActions.
```

```
place
When I amNil
Do return me
When Always
Do my positionBecomes
whole width atRandom.
whole height atRandom.
rule tell my successor to 'place'.
```

lay out text reset

The main problem is to do “word wrap”



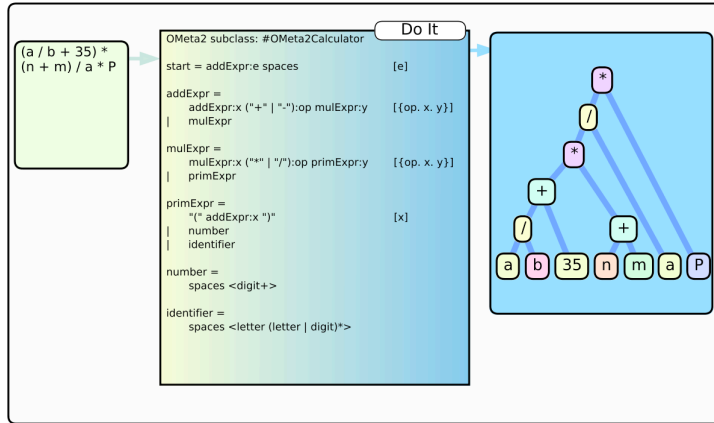
lay out text reset

Making Languages

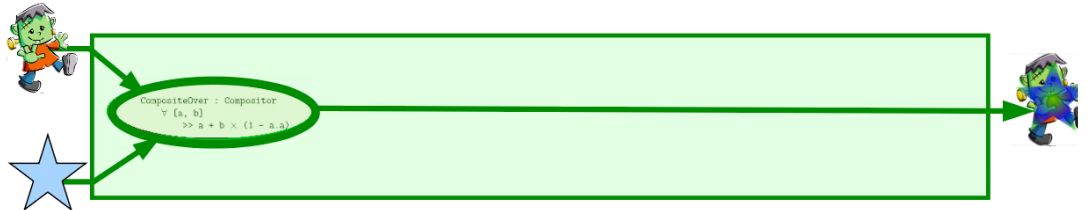
Our approach to making languages uses higher-level PEG parsers (with backup) that can match and transform general objects rather than just strings. This allows these transformation engines to be used for various kinds of pipeline optimizations as well as for parsing text according to a syntax. This follows a long tradition of such systems reaching back into the 60s. What's new here is the convenience, simplicity and range of the systems we've built. The PEG parsing languages—the main one we use is OMeta—can easily make themselves. The semantics of the transformations are drawn from the host languages that the PEG parsers are made in, and these run the gamut from very high level systems (Smalltalk or Lisp-like) to systems that are close to the CPU (e. g. the Nothing language we've made—a BCPL-like "lower-level-high-level language").

Live demonstration of using our OMeta system to translate an arithmetic expression into an "Abstract Syntax Tree". This is a standard form in which all the operator and operand relationships are made into a uniform structure for further processing.

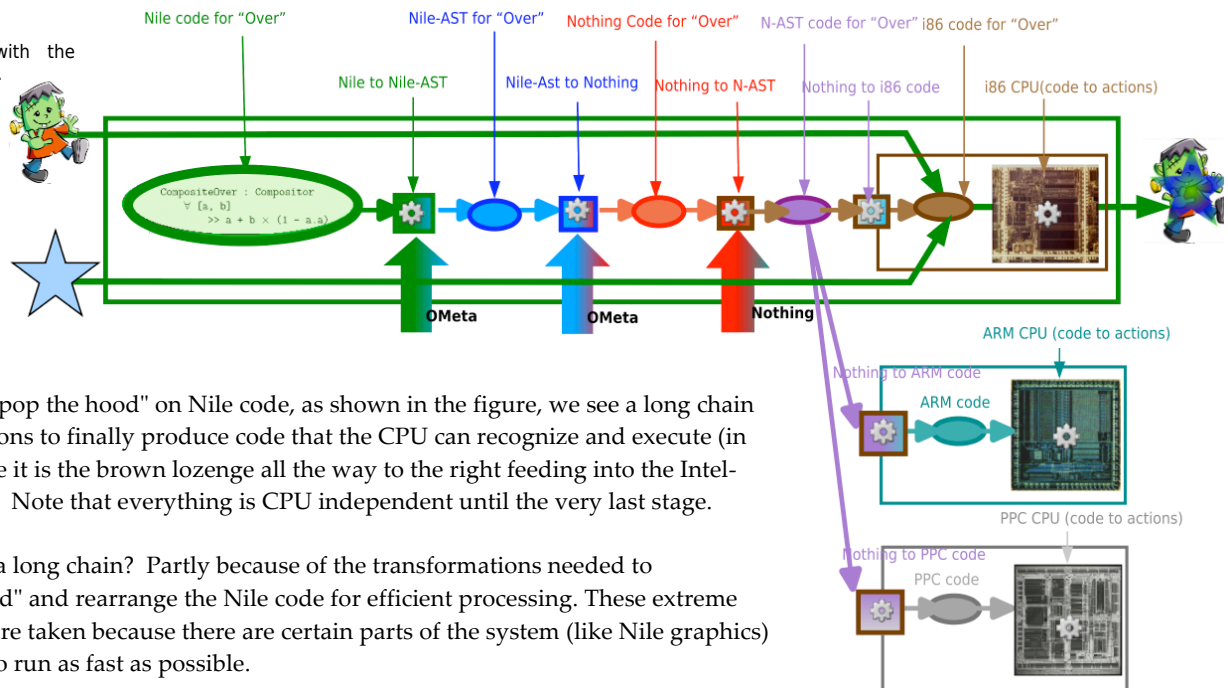
OMeta follows a long tradition of pattern directed translation. Its power lies in its convenience, range and flexibility, e.g. it can also transform the Abstract Syntax structures.



An example of Nile code with "the hood" on. It seems as though we merely write a few lines of higher level code and it somehow is able to execute to produce a composited result.



Nile code with the hood popped.



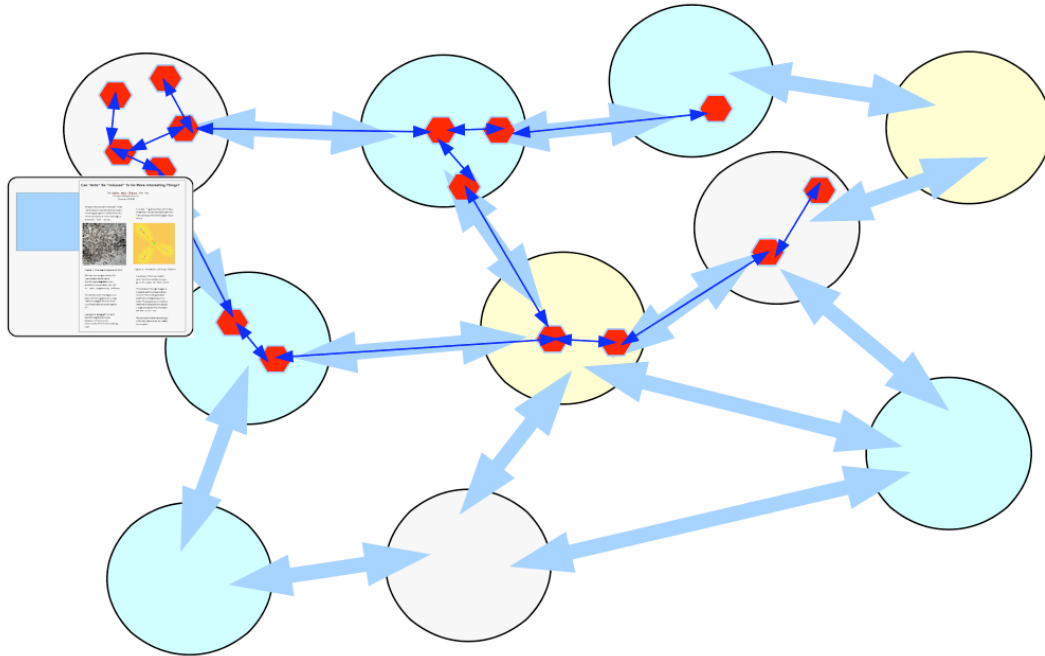
When we "pop the hood" on Nile code, as shown in the figure, we see a long chain of translations to finally produce code that the CPU can recognize and execute (in the top case it is the brown lozenge all the way to the right feeding into the Intel-type CPU). Note that everything is CPU independent until the very last stage.

Why such a long chain? Partly because of the transformations needed to "understand" and rearrange the Nile code for efficient processing. These extreme measures are taken because there are certain parts of the system (like Nile graphics) that have to run as fast as possible.

“NoOS = No Operating System”

In the conventional sense, STEPS does not really have an “operating system”, nor does it use one. Instead the STEPS system is thought of as existing on an Internet of linked physical computers as a virtual network of communicating virtual machines (“real objects”).

Today’s physical computers themselves have virtualized versions of themselves manifested as “processes” which have their own confined address spaces. The STEPS strategy is to map its networks of communicating real objects over the physical-and-virtual mechanisms provided by the physical hardware. In short, to STEPS, the world of physical computing resources is viewed as caches for its mobile computations.



The “NoOS” model for STEPS. The circles are physical computers, the light blue thick arrows are physical network connections. The red hexagons are “object/processes”; the dark blue arrows are messages. The user of a distributed mobile computation will have objects close by to create a UI, views, and controls. The code needed on each physical computer for resource allocation is tiny.

Much of this is future work for the STEPS project. One of our influences has been David Reed’s 1978 PhD thesis [NETOS], and we have previously implemented several versions of this [Croquet]. Another influence was the [Locus] migrating process operating system of heterogeneous hardware done by Gerry Popek and colleagues in the 80s.

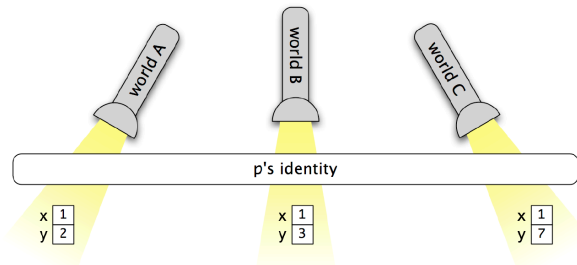
In order for a computer to act as a cache for STEPS, something does have to be written, and one could call it an “operating system”. In fact, it would be very much like the parsimonious kernels for OSs that have been done neatly and nicely over the years by individuals (for example the Linux kernel in about 1000 lines of code by Linus Torvalds—and this followed the original notion of Unix that almost all of it be vanilla code; only the bare essentials should be special). The bloat that followed (Linux is really large these days!) is mostly a result of various kinds of dependencies, poor design, and perhaps even the weak paradigms of “economies of scale” and “central shared facilities”.

By restricting the local code on a physical computer to simple caching and safe access to resources, we think that STEPS can emulate the inspiring small TCP/IP kernels which have allowed the Internet to scale by many orders of magnitude without themselves getting polluted by dependencies.

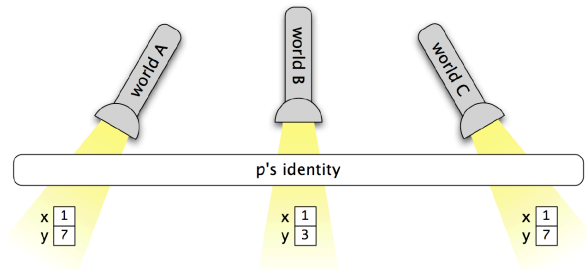
Simulation of "Time"

Traditional system designs allow the CPU(s) to determine "time" and then try to guard against what this implies in various ways. An equally old—but not mainstream—approach is to "simulate time" in various ways and run computations that are governed by "designed time". For example, for robot programming in the 60s, John McCarthy wanted to use logical reasoning *and* to advance time. He accomplished this by labeling "facts" with the "timeframe" in which they were true. Logical reasoning with no races or contradictions could be carried out in each timeframe, and new facts (the robot changing location, etc.) would be asserted in a new timeframe. One of the ways STEPS simulates time is via *worlds*.

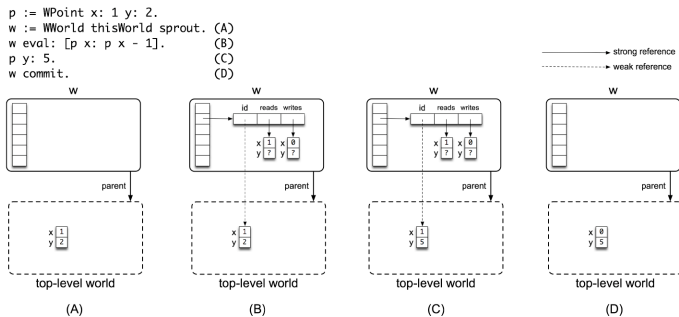
A *world* is a new language construct that reifies the notion of program state. All computation takes place inside a world, which captures all of the side effects—changes to global and local variables, arrays, objects' instance variables, etc.—that happen inside it. Worlds are first-class values: they can be stored in variables, passed as arguments to functions, etc. They can even be garbage-collected just like any other object. A new world can be "sprouted" from an existing world at will. The state of a child world is derived from the state of its parent, but the side effects that happen inside the child do not affect the parent. (This is analogous to the semantics of delegation in prototype-based languages with copy-on-write slots.) At any time, the side effects captured in the child world can be propagated to its parent via a commit operation. [Worlds]



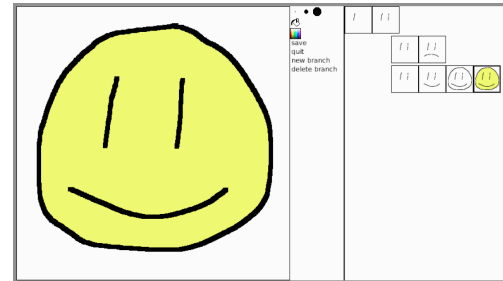
Projections/views of the same object in three different worlds



The state of the universe after world C has done a commit



Example of successful commit

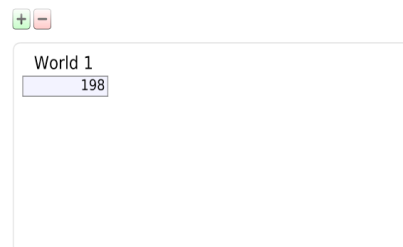


Example of a painting system using worlds at the individual pixel level

Having implemented worlds-enabled versions of Smalltalk's principal collection classes, and ensured that the design gave suitable performance on various basic use cases, we set up a simple demonstration of how one could interact with Frank in multiple independent worlds. In the figure below—"Setup 1"—the user has selected the airfare cell to make it appear in the world palette on the right-hand side.

Setup 1

	A	B	C
1	date	item	amount
2	April 23, 2010	air fare, southwest	198
3	April 23, 2010	taxi, BUR to AMI	28
4	April 24, 2010	parking, SJC	30
5	-----	TOTAL	257
6			
7			



Setup 2

	A	B	C
1	date	item	amount
2	April 23, 2010	air fare, southwest	198
3	April 23, 2010	taxi, BUR to AMI	28
4	April 24, 2010	parking, SJC	30
5	-----	TOTAL	257
6			
7			

World 1	
198	
257	

In "Setup 2" two further cells have been added: the total cell, and the thumb up/down indication calculated from that total.

Usage 1

	A	B	C
1	date	item	amount
2	April 23, 2010	air fare, southwest	298
3	April 23, 2010	taxi, BUR to AMI	28
4	April 24, 2010	parking, SJC	30
5	-----	TOTAL	356
6			
7			

World 1	World 2	World 3
198	248	298
256	306	356

In "Usage 1" the user has sprouted two further sibling worlds, and has entered a different air fare for each; the worlds' independent calculations reveal that two of these cases get a thumbs-down.

Usage 1

	A	B	C
1	date	item	amount
2	April 23, 2010	air fare, southwest	198
3	April 23, 2010	taxi, BUR to AMI	0
4	April 24, 2010	parking, SJC	30
5	-----	TOTAL	228
6			
7			

World 1	World 2	World 3
198	248	298
228	278	328

"Usage 2" shows the result of the user manipulating the taxi-fare value, which is still shared between all worlds. Setting it to zero changes the total in every world, bringing the amount in World 2 back under the threshold for a thumb-up.

TCP/IP

TCP/IP is a marvel of an idea and design, done by very good designers. A typical version of TCP/IP done in C is about 20,000 lines of code (which makes it a comparatively small program). There have been much smaller versions of TCP/IP in several thousand lines of C.

Most interesting ideas have more than one fruitful way to view them, and it occurred to us that, abstractly, one could think of TCP/IP as a kind of "non-deterministic parser with balancing heuristics", in that it takes in a stream of things, does various kinds of pattern-matching on them, deals with errors by backing up and taking other paths, and produces a transformation of the input in a specified form as a result.

Since the language transformation techniques we use operate on arbitrary objects, not just strings (see above), and include some abilities of both standard and logic programming, it seemed that this could be used to make a very compact TCP/IP. Our first attempt was about 160 lines of code that was robust enough to run a website. We think this can be done even more compactly and clearly, and we plan to take another pass at this next year.

Dealing with CPUs — High Level Language Solutions For Low Level Domains

We place this topic last—but not least—in this report both because of the size of the discussion and because we have been pursuing several parallel strategies in this crucial—and “code-eating”—domain.

Our goal is to find high-level abstract descriptions of systems, but realistic performance is also a necessity. A pattern that we see often is that the DSLs built for high-level descriptions of implementation and optimization themselves become targets for high-level descriptions and in particular optimizations, just to make the system run fast enough. An awkward and complicated 'optimization tower of babel' is avoided by giving our DSLs the ability to act on their own implementation and optimizations, flattening the 'tower' into a collection of reflexive and metacircular mechanisms. The line between strategy and implementation, between coordination and computation, is eliminated -- or at least guaranteed to be eliminable whenever necessary.

The same pattern occurs in end-user applications written in general-purpose languages. Optimization and implementation mechanisms that will be useful in specific situations cannot be predicted in advance. Eliminating all barriers to potential future needs, and dealing gracefully with unanticipated requirements, means our system description language has to be able to rewrite its own implementation -- including primitives and even execution model. To have a useful influence on the implementation of a system, a high-level language for strategy and coordination is destined either to be paired with other DSLs working at much lower levels, or to evolve into something more general that is designed to support its own necessary vertical diversity.

For a variety of reasons, we want to count every line of “meaning-code” that has to be written to do STEPS. This despite that the lines of code needed to make C, C++, Java, etc., and their IDEs are *not* counted in the standard systems whose phenomena we are modeling. Thus we need to be able to do (a) our own “bottom” and (b) the IDEs for all our languages within our target code-count. This means that we have to invent systems that can move from the very high level POLs that efficiently handle the various domains of STEPS all the way down to generating the binary code that a typical CPU of today can execute.

This is not so difficult *per se*, but an additional constraint is that we would really like to run “what needs to be run quickly”—such as the Nile graphics code—fast enough to allow the whole “chain of meaning” to be efficient enough for real-time use. This is *not* part of the original STEPS goals—in the sense that we only have to count “meaning code that can run” and not the optimizations—the original idea was that it would be sufficient if the “meaning-code” could be run fast enough on a supercomputer.

However, this new goal arose from esthetic considerations as Nile and other parts of the system started working well. In other words, it would be *really nice*TM if we could pull off a system in which the “meaning-code” not only ran successfully, but where it would run fast enough to be useful on a laptop.

This is the current case for STEPS with C modules at the bottom. The optimization processes of most C compilers successfully produce code that combined with modern CPUs runs everything in real-time.

But (a) C systems are quite large themselves, (b) we don't think C pays its way very well at the low level, and (c) C is far from being pretty, and thus doesn't fit in with the rest of the POLs we've made.

For the last year, we have been pursuing two paths—**Nothing**, and Maru —leading to making a system that can join the high level to the low level, and especially run the computationally intense parts of STEPS fast enough. These two systems have somewhat extreme and different points of view.

Nothing is based on the predecessor to C—BCPL—which is a very low level language couched in an Algol-like form. [Nothing]

Maru is based on a self-compiling Lisp-like language that takes a fairly high level semantics and moves directly to machine code. [Maru] This builds on the “COLA”'s work of the last 4 years.

The initial benchmarks for both **Nothing** and Maru are to be able to:

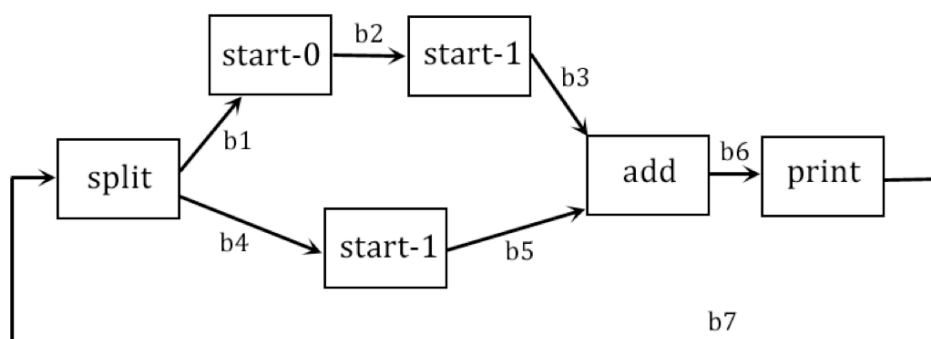
- self-compile themselves
- write their own garbage collectors and other storage management in themselves
- run the complete Nile language chain down to the CPU without using C anywhere.

Nothing

C is still used to make **Nothing** and make its garbage collector. The GC will soon be rendered in **Nothing**.

But **Nothing** does run the entire Nile language chain including all the Nile run-time without any C code, and can produce identical results to the Nile system that is currently used by STEPS (which does use some C).

The way the Nile runtime works was generalized. Instead of expecting each kernel to run only when all of its input is ready, then run to completion and die, the **Nothing** version keeps running any kernel that has anything to do until they have all stopped. **Nothing** also allows kernels to have multiple input and output streams. The kernels are parallel processes. The first test of the runtime was with the circuit that produces fibonacci numbers.



start-N first transmits N on its output stream, then just copies input to output. **split** copies its input to both of its outputs. **add** takes a number from each of its inputs and transmits their sum. **print** just copies inputs to outputs but prints everything it sees.

Maru

When Maru compiles itself to machine code, it does not have to include or link with any hand-written C at all. It still uses a few standard library routines provided as additional services by the host operating system (for buffered input/output operations to the terminal and to the file system) but these could easily be replaced with the underlying operating system calls, since they are just wrappers around the latter.

Maru has its own GCs written in Maru (one non-moving and the other generation-scavenging, both of them precise).

Maru does not yet run the complete Nile language chain. It is estimated that the Nile source (the 457 lines of code that describe STEPS graphics) will run in about a month. It will take another month or so to write and get running the lower-level language in which to express the Nile runtime.

Maru is a vertically-extended description language that bridges the gap between desirability of high-level representations and easy manipulation of low-level details. It is simultaneously an extensible, high-level language for strategy and coordination, and an intermediate representation for programs and semantics from a wide range of high- and low-level languages. Many projections are possible from the Maru IR: forward onto different architectures and execution mechanisms, and backward onto different views or syntactic presentations of its meaning. Most critical is its metacircularity: it provides low-level mechanisms that can implement new high-level abstractions for describing new high- and low-level mechanisms.

High-level features include higher-order functions, extensible composition mechanisms and extensible data types. These are used throughout Maru's description of itself and to build mechanisms for compiling that description to an implementation and runtime support system. Major components designed to be reusable include a PEG-based parser generator, an AST abstraction (supporting objects and primitive types, and operations on both), an evaluator for compile-time execution of ASTs (to analyze and manipulate ASTs or other data structures), and extensible mechanisms for code generation (including instruction selection and register allocation). The latter phases provide several points of entry for programming language implementations to target. The compile-time AST evaluator provides hosted languages with a low-cost mechanism to provide their own metalinguistic and introspective facilities during compilation, aided again by the system's metacircularity: analyses and optimizations are described using the forms and processes that they analyze and optimize.

The first serious language project in Maru was Maru itself. It is now stable enough to begin non-trivial experiments with other programming languages and paradigms. A Smalltalk-like language has been built as a thin veneer over Maru, using Maru's own PEG-based parsing and built-in evaluation mechanisms, which can run substantial pieces of code. The second serious language project began recently: an implementation of the Nile stream-processing language.

Nile-in-Maru's front end is a page-long parsing expression grammar that Maru converts into a parser producing abstract syntax trees specialized for Nile semantics. Type inference and other analyses are performed on these trees to prepare them for code generation, with possible targets including other compiled languages (such as C), machine code for various architectures, 'rich' binary formats, and virtual machine code.

Most significant in the embedding of Nile in Maru is that the system is self-contained, independent of external tools, and smaller than previous prototypes. In addition to running the system as an 'offline' Nile compiler, we are also able to enter Nile programs interactively from the terminal or from text supplied by a graphical programming environment, for example.

Initially the target for code generation is an equivalent C program, to establish a performance baseline and for comparison with previous work on generating C from Nile programs. Direct generation of machine code (via the reusable back-end components in Maru) will follow soon, as will the design and implementation of a simple (compared to C) low-level language in which to express Nile's runtime support. The generation of efficient machine code for Nile is a significant goal for the extensible framework in the Maru code generator, as is the provision of information specialized to whatever extent is necessary for building rich environments for measurement and debugging of Nile programs. Advances made in these areas for Nile will be immediately useful for all languages hosted by Maru, including itself.

Importing and Exporting

Quite a bit of work was done to make it easier to interface Frank with the rest of the world, including being able to import OpenOffice file formats (virtually all of an OpenOffice Impress file can be imported). This has allowed us to experiment with real world presentations using very elaborate desktop media that include many of the features found in standard productivity tools. A special ability is that the slides in an Impress file can have a reference to Frank media pages and these will be inserted during the importation process. The result is a Frank document using Frank scripting and Frank demos that can be used as a "super-and-live PowerPoint" for giving dynamic presentations.

Another major accomplishment has been the ability to export standard PDF files (all of the Frank constructions can be exported as a PDF file, including this report).

Experiments

We built a "MicroSqueak" graphics system as a minimal-system experiment. From a tiny 250kB image we can call on Gezira to render Bezier paths that interact with user events.

We defined a variant syntax for OMeta that can be read from a single text file rather than as individual methods. We also developed a simple module system that allows this OMeta variant to be loaded into a MicroSqueak-based image as a module. Using this syntax and module system, a new compiler can be defined and executed from a command-line interface.

A Squeak bytecode compiler was written in a parser generator based on the "leg" system. From the code of the regular Compiler written in Squeak, it can generate a module for the above-mentioned module system. Thus we can bootstrap Squeak from the state where there is no Squeak compiler. Along the way we added memoization to the parser generator, greatly improving its performance.

We've been experimenting with a new object system that explores an architecture in which objects are more loosely coupled than at present. It employs a tuplespace-based event notification mechanism, and elevates the role of our spreadsheet-cell-like abstraction.

We ported the Squeak VM to the Google Native Client. We also experimented with dynamic code generation within Native Client. We were able to copy code for a function and execute it. This is a proof of concept that we can run Nothing, or Maru with dynamic code-generation, in the Chrome browser.

Training And Development

Daniel Amelang continues as a graduate student at UCSD while working on his PhD thesis on the Gezira/Nile streaming graphics system at Viewpoints Research.

Outreach Activities

Alan Kay and others on the team presented this research at a variety of conferences and meetings throughout this most recent reporting period resulting in sharing the work to small and large groups of people in related academic, research and business communities around the world. Following is a list highlighting those talks, demos and presentations.

November 2010

Yoshiki Ohshima and Ian Piumarta spoke at Kyoto University, Kyoto, Japan. Ohshima: Modeling a computer system: the STEPS project, and Piumarta: A high-level model of programming language implementation Complex

<http://www.media.kyoto-u.ac.jp/ja/services/accms/whatsnew/event/detail/01628.html>

December 2010

Alan Kay presented at I/ITSEC 2010 (Immersive Infantry Training for US Marine Corps, Modeling Simulation and Training Conference) on Adaptability and Complex Decision-Making, in Orlando, Florida.

<http://www.iitsec.org/about/Pages/HighlightsFromLastIITSEC.aspx>

January 2011

Alan had meetings with staff from Department of Education to share and discuss ideas on computer-based learning using our research as basis for discussion.

March 2011

Alan was invited to participate in President Obama's educational event at TechBoston Academy - A Boston Public School for Technology & College Preparation. The President was joined by Secretary of Education Arne Duncan and Melinda Gates. The event was intended to highlight the importance of providing America's students with a high quality education so that they can be successful in the 21st century economy.

Alan gave a keynote address at the SRII (Service Research & Innovation Institute) Global Conference in San Jose, California. The annual SRII Global Conference is an annual conference focused on connecting Services to Science and Technology and is a unique opportunity to build liaisons with senior leaders from the industry, research organizations, academia as well as the government organizations from all around the world to address the "mega challenges" of Service industries in a holistic manner.

<http://thesrii.org/index.php/keynote-speakers>

Ian Piumarta was invited to speak on open systems at the ReGIS-Forum (Research Environment for Global Information Society) in Tokyo, Japan. The forum was attended by high level government /ministry officials, academics, and senior level business people from Japan.

<http://www.re-gis.com/gis/74-e.html>

April 2011

Dan Amelang presented "The Nile Programming Language: Declarative Stream Processing for Media Applications" at The SoCal Programming Languages and Systems Workshop, Harvey Mudd College, Claremont, CA.

Alan visited Carnegie-Mellon University and shared this work with Prof. Ken Koedinger and others from the computer science department.

May 2011

Alan presented to a group at Northrop Grumman comprised of lead technologists and CTOs from within the Northrop businesses in Los Angeles, CA.

Ian Piumarta presented our research at Kyoto University to students from the school of Social Informatics and Engineering.

June 2011

Alan Kay gave a talk to undergraduate and graduate students in Computer Science and Engineering at UC San Diego.

July 2011

Alan Kay, Yoshiki Ohshima, and Bert Freudenberg presented this research at a day-long colloquium to a group of researchers and academics at the Hasso-Plattner Institut in Potsdam, Germany.

<http://www.tele-task.de/archive/series/overview/844/#lecture5819>

Alan gave the banquet keynote talk at ECOOP 2011 (European Conference on Object Oriented Programming) in Lancaster, UK.

<http://ecoop11.comp.lancs.ac.uk/?q=content/keynote-speakers>

Ian Piumarta presented his work at FREECO'11, an ECOOP 2011 workshop in Lancaster, UK. Papers presented are part of the publications section of this report.

<http://trese.ewi.utwente.nl/workshops/FREECO/FREECO11/proceedings.html>

Yoshiki Ohshima and Alex Warth presented their "Worlds" work - Controlling the Scope of Side Effects - at ECOOP 2011 in Lancaster, UK. Paper presented is part of the publications section of this report.

<http://ecoop11.comp.lancs.ac.uk/?q=program/papers>

Yoshiki Ohshima presented "The Universal Document Application of the STEPS Project" at the demo session of ECOOP 2011 in Lancaster, UK.

August 2011

Alan presented work at a gathering of the Intel PC Client Group in Santa Clara, CA.

Alan presented to Prof. Ken Perlin and his students at New York University (NYU).

September 2011

Alan participated in a Georgia Tech sponsored event - the Georgia Tech Center for 21st Century Universities with Rich DeMillo, Roger Schank and others.

October 2011

Alan presented at TTI Vanguard's "Taming Complexity Conference" in Washington DC.

Alan presented at PwC's DiamondExchange program - The new science of management in a rapidly changing world, in Arizona.

Alan spoke to Professor Nancy Hechinger's New York University, Interactive Telecommunications Program (ITP) Communications class.

Throughout the year Alan has had continual contact with SAP worldwide and SAP Labs to share, disseminate and see how technology ideas and this work might be transferred into their workplace as a test for further use and outreach.

References

[Croquet] – David Smith, et al, “Croquet, A Collaboration Systems Architecture, Proceedings IEEE C5 Conference, 2003, Kyoto, Japan.

[Locus] - Gerry Popek, et al, “The LOCUS distributed systems architecture”, MIT Press, 1986.

[Maru] – Ian Piumarta, "Open, extensible composition models". Free Composition (FREECO'11), July 26, 2011, Lancaster, UK, <http://piumarta.com/freeco11/freeco11-piumarta-oecm.pdf>

[NETOS] – David Patrick Reed, “Naming and Synchronization in a Distributed Computer System, MIT Phd Thesis, 1978.

[Nothing] – STEPS 2010 4th Year Progress Report (see below).

[STEPS Reports]

[NSF Proposal For STEPS] - [STEPS Proposal to NSF - 2006](#)

[2007 1st Year Progress Report] - [STEPS 2007 Progress Report](#)

[2008 2nd Year Progress Report] - [STEPS 2008 Progress Report](#)

[2009 3rd Year Progress Report] - [STEPS 2009 Progress Report](#)

[2010 4th Year Progress Report] - [STEPS 2010 Progress Report](#)

[Worlds] Alessandro Warth, Yoshiki Ohshima, Ted Kaehler, and Alan Kay, *Worlds: Controlling the Scope of Side Effects*, ECOOP 2011, Lancaster England (and at http://www.vpri.org/pdf/tr2011001_final_worlds.pdf)

How To Make A Text Field From Ants*

Ted Kaehler, Yoshiki Ohshima
Viewpoints Research Institute, Glendale, CA



Figure 1: Ants swarming around food

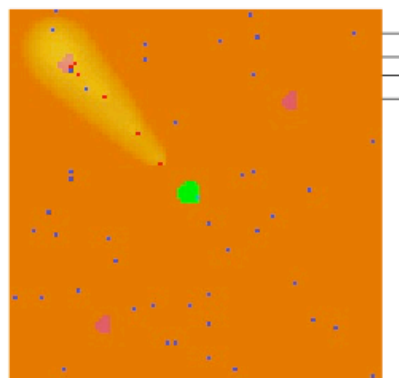


Figure 2: Simulation showing diffusion

Introduction

Ants are a “distributed hive animal” whose “cells” communicate by a variety of means including laying down trails of chemicals which can help ants get to discovered food sources.

We want to investigate whether the “particles and fields” style of distributing messages as intrinsic parts of environments can work well for some programming problems.

For example, could we program a text layout and editing system just using a “field of messages” that individual text characters can sense and respond to? Laying out a “paragraph” of “ants” seems like a good place to start because it will have to work continuously while the editor is being used.

Building a Text Field from “Ants”

The goal is to create a text field and arrange its letters in an area of the screen. The letters have an order from first to last.

The main problem is to do “word wrap” so that each word is entirely on one line. We don’t want half the word at the end of one line and the other half on the next line.

What if the text is too long and goes over the margin?

In the most general case, each letter is made from any costume that has shape, color or fill, and can be bounded by a rectangle.

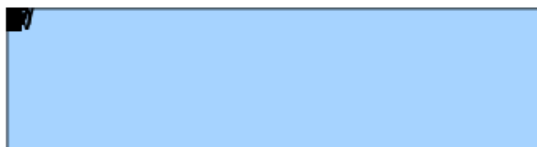
Any costume that has those properties can be placed in text, even if it is not really a letter. A letter object is rendered into the composition area on the screen using the system’s normal costume rendering programs.

With such a general notion of a letter, we are freed from dealing with the details of families of fonts, emphasis (bold, italic), construction of a letter of the desired size, or text color. When a glyph arrives at the layout stage, it already has the proper shape, style, size and color.

Letters In A Box With No Rules

To see the default situation with no layout rules, press the blue button “layout”. Since there are no rules, all of the letters pile up at the upper left of the field. A mouse click or drag on a letter does nothing.

layout



*This project is funded by NSF and private donors

Text Field Specification

Rules

The behavior of the letters is defined by a set of rules. Each rule is in a rule editor window. At the top is the name of the rule, followed by the rule itself. The rule has a list of clauses. Each clause has a guard after the **When**. If the guard is true, then execute the **Do** part.

"return" means evaluate the expression and hand it back to the place where the rule was called. We exit the rule at the return and do not perform the later clauses.

Extremely Simple Layout Methods

Random Layout

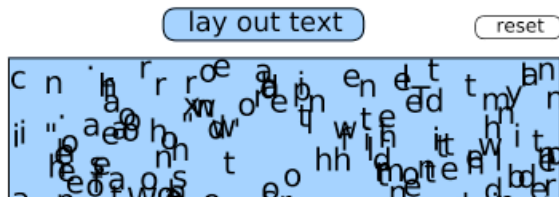
As a simple first experiment, we will put each letter in a random place in the text field. Set the x,y position of each letter to a random value within the field's width and height.

Press Accept in the rules for **layout** and **place**.

Press the layout button below. What happens when you press it a second time? Press reset to put all letters at the upper left.

```
layout
When whole contents isEmpty not
Do rule tell whole contents first
to 'place'.
rule processActions.
```

```
place
When I amNil
Do return me
When Always
Do my positionBecomes
whole width atRandom,
whole height atRandom.
rule tell my successor to 'place'.
```



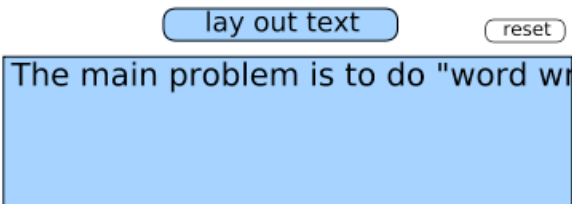
All in One Line

Now let's redefine **place** to arrange all letters in one long line. Position each letter just to the right of its predecessor.

The one line example inherits the rules of the random layout and overrides the **place** rule.

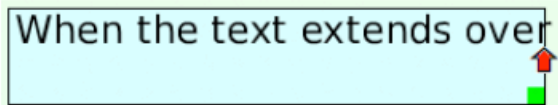
The line is clipped by the edge of the field.

```
place
When I amNil
Do return me
When my index = 1
Do my position (whole shape leftAtY 0)+4 ,
4.
return rule tell my successor to 'place'.
When Always
Do pred := my predecessor.
my pivotPosition
pred left + pred pivotOffset x +
pred width ,
pred pivotPosition y.
rule tell my successor to 'place'.
```



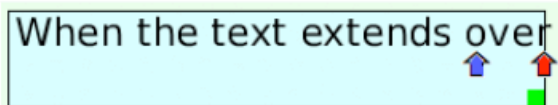
Text Field Specification

How to Break the Line of Text

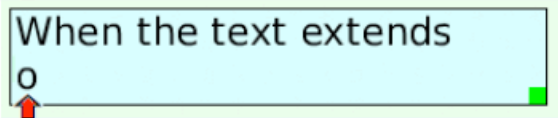


To layout the text in the field, place the next letter just to the right of the previous one. The red arrow points to the letter that was just placed, in this case an "r".

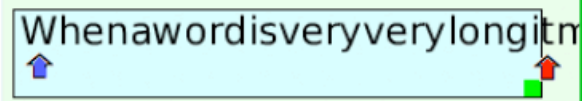
That letter extends beyond the right margin, so we need to move its entire word to the next line.



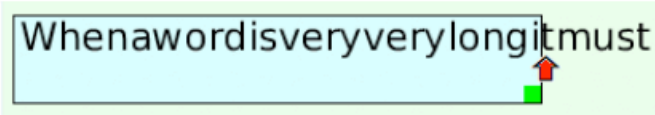
Walk back with the blue arrow until we reach the first letter of the word.



Move that letter to the next line. Resume placing letters to the right of the "o", as indicated by the red arrow. Notice that all letters in the word "over" are placed more than once during the layout.



If the walk back (blue arrow) gets all the way to the left margin, a single word covers the entire line. Which letter should be moved to the next line? The original clipped letter (red arrow) is the proper letter to move. A single word that covers the entire line is a special case, and we must test for it.



Wrapping the Text to a New Line

When a line of text is longer than the width of the text field, we want to wrap it to the next line. The goal of text wrapping is to determine where to break the text to start a new line.

Each letter follows its predecessor on the current horizontal line. When a letter hangs over the right margin, its entire word needs to be moved to the next line.

A carriage return causes the next letter to start a new line.

A single word can be wider than entire line. Break it where it touches the right margin. We also need to handle the cases when a letter has no predecessor (it is the first), and has no successor (it is the last).

We start with a general **layout** rule. It moves the first letter to the upper left. Then, it tells the next letter to **place** itself in the field. (Ignore the part about `maxHeight` and `missingHeight` for the moment.) When each letter is finished being placed, it must tell its successor to **place**.

```
layout Accept  
When whole contents isEmpty not  
  Do maxHeight := whole first ascent.  
      maxDescent := whole first descent.  
      missingHeight := 0.  
      whole first pivotBecomes  
          ((whole shape leftAtY 0) ,  
           maxHeight) + inset.  
      rule tell whole first successor to 'place'.  
When Always  
  Do rule tellLater rule to 'showSelection'.  
      rule processActions.
```


Text Field Specification

Place each letter just to the right of the previous letter on the current line. Then, look for special cases.

If the letter follows a carriage return, move it to the next line. (This is done inside **placeIfAfterReturn**).

The letter has the goal of not being clipped by the right margin. When a letter is not white space and finds that it is being clipped, run the **backToWordStart** rule. It looks backwards to find the start of the current word, and moves that letter to the

to the next letter. For all other letters, it returns false, which signals the place rule to go further and test whether the current letter is over the right margin.

maxHeight and **missingHeight** are used to move the line down when a tall letter is in the middle of the line. We will hook this up later in the essay.

Noticing the right margin

isClipped is the most important rule for specifying word wrap. It returns true if current letter overlaps the right margin. It does this by comparing the letter's right x-value with the margin's x. The margin can be curved, so we ask the text field box for the margin's actual x value at this y. Containers can have irregular shapes, and line lengths can be different.

White space such as a space or a tab are allowed to extend beyond the margin. Return false for white space letters.

```
place (Accept)  
When I amNil  
  Do return me.  
When Always  
  Do pred := my predecessor.  
  my pivotPosition pred right +  
    pred pivotOffset x ,  
    pred pivotPosition y.  
When rule placeIfAfterReturn me  
  Do return rule tell my successor to 'place'.  
When rule isClipped me  
  Do rule tell me to 'backToWordStart'.  
When (rule isClipped me) not  
  Do rule tell my successor to 'place'.
```

Noticing a Carriage Return

The rule **placeIfAfterReturn** actually ignores the return character itself. It only takes action when the previous letter is a carriage return. If so, it moves the current letter to the beginning of the next line.

placeIfAfterReturn always returns true or false. This allows it to be used in a guard clause. You can see this in the **place** rule. When **placeIfAfterReturn** has moved a letter, it returns true, which signals to go on

```
isClipped (Accept)  
When my shape isWhiteSpace  
  Do return false.  
  
When Always  
  Do return my right + inset x >  
    (whole shape rightAtY  
    my pivotPosition y)
```

```
placeIfAfterReturn (Accept)  
When my predecessor shape notNil and  
  [my predecessor shape isNewline]  
  Do "start of the next line"  
  my pivotPosition 0 ,  
    (my predecessor pivotPosition y +  
    maxHeight + maxDescent + 2).  
  my pivotPosition  
    ((whole shape leftAtY  
    my pivotPosition y) + inset x) ,  
    my pivotPosition y.  
  maxDescent := my descent.  
  missingHeight := 0.  
  return true.  
When Always  
  Do return false.
```

Text Field Specification

Finding the Start of a Word

We know that the current letter hangs over the right margin. We need to move the entire word to the start of the next line. **backToWorldStart** first calls **startOfWord**, which finds the first letter of the current word.

If that letter is already at the start of a line, we should not move it. The line is wider than the field and has no white space in it. The original clipped character should be forced to start a new line.

Otherwise, use the start of the word as the letter to be moved.

Once we have the proper letter in **letterToMove**, put it at the start of the next line.

```
backToWorldStart Accept  
When Always  
  Do letterToMove := self startOfWord me.  
When self isStartOfLine letterToMove index  
  Do "Word takes entire line, break at the  
    clipped character"  
    letterToMove := me.  
When Always  
  Do  
    maxHeight := letterToMove ascent.  
    letterToMove pivotYIncreaseBy  
      maxHeight + maxDescent + 2.  
    letterToMove pivotPosition  
      ((whole shape leftAtY  
        letterToMove pivotPosition y) +  
        inset x) ,  
        letterToMove pivotPosition y.  
    missingHeight := 0.  
    rule tell letterToMove successor to 'place'.
```

startOfWord travels back along the word to find the first letter. We are looking for a letter that is not white space. If we happen come to the first letter of the text, return it instead.

startOfWord considers just one letter. If that letter is not the start of a word, it calls itself again to consider the preceding letter.

```
startOfWord Accept  
When my predecessor isNil  
  Do return me.  
  
When my predecessor shape isWhiteSpace  
  Do return me.  
  
When Always  
  Do return rule startOfWord my predecessor.
```

Is a Letter at the Start of a Line?

Finally, we need a little test to tell if the current letter is at the start of a line of text. 'me' is the index of a letter. Return true if the letter is at the left margin. This only works on letters that have been placed.

```
isStartOfLine Accept  
When Always  
  Do return (whole at me) pivotPosition x -  
    inset x <=? "left margin"  
    (whole shape leftAtY  
      (whole at me) pivotPosition y)
```

lay out text

reset

The main problem is to do "word wrap" so that each word is entirely on one line. We don't want half the word at the end of one line and the other half on the next line.

Appendix II – The Maru code to make a Smalltalk-like language, write an FFT in it, and test it

1. Maru definitions for syntax and semantics of a Smalltalk-like language

The following Maru program implements a Smalltalk-like language. The semantics and runtime support are defined first, followed the syntax (which in this example is applied immediately to the rest of the input).

The Smalltalk section of the input defines a small class hierarchy and several kernel methods and primitives (the latter written as ASTs in Maru's s-expression language, embedded in the Smalltalk code between curly braces '{ ... }'). This implementation of Smalltalk is 333 lines of code (with blank lines and comments removed). The ASTs generated by the parser are interpreted in this example, but could as easily be passed to Maru's compiler.

The final part of the Smalltalk section defines methods to perform a fast Fourier transform on arrays of numbers and to plot a graph of the contents of an array. These methods are exercised with a 64-sample time-domain signal containing two sine waves of different frequency and amplitude that are converted to the frequency domain and then plotted.

```
;; Some utility functions for dealing with class and selector names

(define-function type-name (x)
  (concat-symbol '< (concat-symbol x '>)))

(define-function concat-symbols args
  (let ((ans (car args)))
    (while (pair? (set args (cdr args)))
      (set ans (concat-symbol ans (car args))))
    ans))

;; A Maru structure for representing Smalltalk block closures

(define-structure <block> (arity expr))

(define-function new-block (arity expr)
  (let ((self (new <block>)))
    (set (<block>-arity self) arity)
    (set (<block>-expr self) expr)
    self))

(define-method do-print <block> () (print "[:" self.arity "]"))

(define-form block-arity (b n)
  `(or (= (<block>-arity ,b) ,n)
    (error "this block expects ",n" argument(s))))

;; Mechanisms for managing the class hierarchy and for defining methods

(define %smalltalk-classes (array))
(define %smalltalk-topclasses)
(define %smalltalk-subclasses (array))

(define-function make-message-accessors (name fields i)
  (and (pair? fields)
    (cons `((, (car fields) () ((name) (list 'oop-at 'self ,i)))
      (make-message-accessors name (cdr fields) (+ i 1)))))

(define-form with-message-accessors (type . body)
  `(with-forms ,(make-message-accessors type (array-at %structure-fields (eval type)) 0)
    (let () ,@body)))

(define %smalltalk-methods)

(define-form define-message (src type selector args . body)
  (set type (type-name type))
  (set selector (concat-symbol '# selector))
  (or (defined? selector) (eval (list 'define-selector selector)))
  (or (assq selector %smalltalk-methods) (push %smalltalk-methods (cons selector (eval selector)))))
  `(set (<expr>-name (<selector>-add-method ,selector ,type
    (lambda ,(cons 'self args)
      (with-message-accessors ,type ,@body)))) ,src))

(define-form send (selector receiver . args)
  `((, (concat-symbol '# selector) ,receiver ,@args))
```

```

(define-form define-class (name basis fields)
  (let ((base (eval basis)))
    (set fields (concat-list (array-at %structure-fields base) fields))
    (sanity-check-structure-fields name fields)
    (let ((type (%allocate-type name))
          (size (list-length fields)))
      (set-array-at %structure-sizes type size)
      (set-array-at %structure-fields type fields)
      (set-array-at %structure-bases type base)
      (let ((derived (or (array-at %structure-derivatives base)
                        (set-array-at %structure-derivatives base (array))))))
        (array-append derived type))
      `(let ()
         (define ,name ,type)
         ,@(%make-accessors name fields)
         ,type))))

(define-function define-topclass (name fields) ;; the root of a hierarchy
  (println "topclass "name" "fields)
  (let ((type (type-name name)))
    (eval `(define-structure ,type ,fields))
    (eval `(define ,name (new ,type)))
    (eval `(push %smalltalk-topclasses ,name))
    (eval `(set-array-at %smalltalk-subclasses ,type (array)))
    (eval `(set-array-at %smalltalk-classes ,type ,name))))

(define-function define-subclass (name base fields) ;; a subclass in a hierarchy
  (println "subclass "name" "base" "fields)
  (let ((type (type-name name))
        (super (type-name base)))
    (eval `(define-class ,type ,super ,fields))
    (eval `(define ,name (new ,type)))
    (eval `(push (array-at %smalltalk-subclasses ,super) ,name))
    (eval `(set-array-at %smalltalk-classes ,type ,name))))

(define-function make-inits (args index)
  (and (pair? args)
       (cons `(set-ooop-at self ,index ,(car args))
             (make-inits (cdr args) (+ index 1)))))

(define-function define-sysclass (field name base) ;; a subclass based on a Maru structure
  (println "subclass "name" "base" ("field"))
  (let ((type (type-name name))
        (super (type-name base)))
    (eval `(define ,type ,field))
    (eval `(set-array-at %structure-bases ,field ,super))
    (eval `(set-array-at %type-names ,field ',type))
    (eval `(define ,name (new ,type)))
    (eval `(push (array-at %smalltalk-subclasses ,super) ,name))
    (eval `(set-array-at %smalltalk-classes ,type ,name))))

;;; Define the syntax of Smalltalk programs

{
  expected      = ..what -> (error what " expected near: "(parser-stream-context self.source)) ;

  pos           =      -> (<parser-stream>-position self.source) ;
  src           = ..s -> (group->string (group-from-to s (<parser-stream>-position self.source))) ;

# ----- the syntax of embedded s-expressions (for primitives)

higit         = [0-9A-Fa-f] ;
char          = "\\\" ( "t"                -> 9
                    | "n"                  -> 10
                    | "r"                  -> 13
                    | "x" (higit higit) @ $#16
                    | "u" (higit higit higit higit) @ $#16
                    | .
                    )

| . ;
sstring       = "\\\" (!"\" char)* $:s "\\\" -> s ;
scomment     = ";" (!eol .)* ;
sspace       = (blank | eol | scomment)* ;
symchar      = [-!#$%&*+./:<=>@A-Z^_a-z|~] ;
symrest      = symchar | [0-9] ;
ssymbol      = (symchar symrest*) @$ $ ;

```

```

sexpr      = ssymbol
           | number
           | sstring
           | "?" .
           | "\" " (!\" \" char)* $:e "\" "      -> e
           | "(" (sspace sexpr)*:e sspace ")"   -> e
           | "'" sexpr:e                        -> (list 'quote e)
           | "`" sexpr:e                        -> (list 'quasiquote e)
           | "," sexpr:e                        -> (list 'unquote-splicing e)
           | "," sexpr:e                        -> (list 'unquote e)
           | "[" _ expression:e "]"            -> e
           | ";" (![\n\r] .)*
           ;

sexpression = sexpr:s _                        -> s ;

# ----- the syntax of Smalltalk programs

blank      = [\t ] ;
eol        = "\n" "\r"* | "\r" "\n"* ;
comment    = "\" (&!\\" \".)*" "\"? ;
_          = (blank | eol | comment)* ;

digit      = [0123456789] ;
letter     = [ABCDEFGH IJKLMNOPQRSTUVWXYZ_abcdefghijklmnopqrstuvwxyz] ;
binchar    = [-!%&*+,/<=>?@\\^`|~] ;

uinteger   = digit+ $#:x _                    -> x ;
integer    = "-" uinteger:x                   -> (- x)
           |
           |
           |
           |
           |
           |
           |
           |
           |
           |
           |
           ;

ufloat     = (digit+ "." digit+ ("e" digit+)?):s _ -> (string->double s) ;
float      = "-" ufloat:x                     -> (- x)
           |
           |
           |
           ;

number     = float | integer ;

string     = "'" ("'" ->? ' | !'" char)* $:x "'" _ -> x ;

symbol     = "#"(idpart | binchar | ":")+ @ $$:x _ -> `',x ;

idpart     = (letter (letter | digit)* ) @ $$ ;
identifier = idpart:x !":" _                  -> x ;

unypart    = (letter (letter | digit)* ) @ $$ ;
unysel     = unypart:x !":" _                -> x ;

binpart    = binchar+ @ $$ ;
binsel     = binpart:x _                    -> x ;

keypart    = (unypart:":") @ $$ ;
keysel     = keypart:x _                    -> x ;

blockargs  = (":" identifier)+:a "\\|"_ -> a
           |
           |
           |
           |
           ;

block      = "[" _ blockargs:a statements:s ("_"*) _ "]" _
           -> `(new-block ,(list-length a) (lambda ,a ,@s)) ;

primary    = number | string | identifier | block | symbol
           | "[" _ expression:e "]" _        -> e
           | "$":e _                          -> e
           | "{" _ sexpression:e "}" _        -> e
           ;

unysend    = primary:r (unysel:s -> `(send ,s ,r):r)* -> r ;

binsend    = unysend:r (binsel:s unysel:a -> `(send ,s ,r ,a):r)* -> r ;

keysend    = binsend:r ( (keysel: s binsend:a -> `(s ,a))+:k
                        -> `(send ,(apply concat-symbols (map car k)) ,r ,@(map cadr k))
                        |
                        -> r
                        ) ;

```

```

assignment = identifier:i " := " _ expression:e          -> `(set ,i ,e) ;
expression = assignment | keysend ;
statements  = "| "_ identifier*:i "|" _ statements:s     -> `((let ,i ,@s))
            | expression?:e ("."_)* expression)*:f     -> `( ,@e ,@f) ;
methodbody = "[" _ statements:s ("."_)* "]" _          -> s
            | "{" _ sexpression*:s "}" _               -> s
            ;

typename   = identifier
            | "<" identifier:i ">" _ -> (concat-symbols '< i '>)
            ;

pattern    = unysel:s                                  -> (list s)
            | binsel:s identifier:i                    -> (list s i)
            | (keysel:s identifier:i -> (list s i))+:k  -> (cons (apply concat-symbols (map car k)) (map cadr k))
            ;

definition = identifier:i " := " _
            ( expression:e "."_
              | { expected "initialiser expression" }
            )
            | identifier:i " : " _
            ( "(" _ identifier*:f ")" _ -> (define-topclass i f)
              | identifier:b "(" _ identifier*:f ")" _ -> (define-subclass i b f)
              | identifier:b "(" _ typename:t " " _ -> (define-sysclass t i b )
              | { expected "class description" }
            )
            | pos:s typename:t pattern:p methodbody:b {src s}:s
              -> (eval `(define-message ,s ,t ,(car p) ,(cdr p) ,@b))
            | expression:e "."_ -> (eval e)
            ;

program    = _ definition* (!. | {expected "definition or expression"}) ;

# ending a grammar with an expression matches the rest of the input against that expression

program
}

" The rest of the file is read as a Smalltalk program (see last line of grammar). "

" Enough of a class hierarchy to support BlockClosure, SmallInteger, Float, Array and String. "

Object : ()
UndefinedObject : Object (<undefined>)
BlockClosure : Object (<block>)
CompiledMethod : Object (<expr>)
Symbol : Object (<symbol>)
MessageSelector : Object (<selector>)
Magnitude : Object ()
Number : Magnitude ()
Float : Number (<double>)
Integer : Number ()
SmallInteger : Integer (<long>)
Collection : Object ()
SequenceableCollection : Collection ()
IndexableCollection : SequenceableCollection ()
ArrayedCollection : IndexableCollection ()
String : ArrayedCollection (<string>)
Array : ArrayedCollection (<array>)

" Kernel methods for logic and evaluation of blocks "

Smalltalk : Object ()

Smalltalk error: message { (error message) }

Object yourself [ self ]
UndefinedObject yourself [ {} ]

nil := UndefinedObject yourself.
false := nil.
true := #true.

```

```

Object new                { (new (type-of self)) }

Object print
Smalltalk newline        { (print self) }
                          [ '\n' print ]
Object println           [ self print. Smalltalk newline. self ]

Object = other           { (= self other) }
Object ~= other          { (not (= self other)) }
Object not               [ false ]
UndefinedObject not     [ true ]

Object subclassResponsibility [ Smalltalk error: 'a subclass should have overridden this message' ]

BlockClosure value      { (block-arity self 0) ((<block>-expr self)) }
BlockClosure value: a   { (block-arity self 1) ((<block>-expr self) a) }
BlockClosure value: a value: b { (block-arity self 2) ((<block>-expr self) a b) }
BlockClosure value: a value: b value: c { (block-arity self 3) ((<block>-expr self) a b c) }

BlockClosure valueWithArguments: a { (block-arity self (array-length a)
                                     (apply (<block>-expr self) (array->list a)) ) }

Object or: aBlock        [ self ]
UndefinedObject or: aBlock [ aBlock value ]

Object and: aBlock       [ aBlock value ]
UndefinedObject and: aBlock [ self ]

BlockClosure whileTrue: b { (while [self value] [b value]) }

Object ifTrue: aBlock    [ aBlock value ]
UndefinedObject ifTrue: aBlock [ self ]

Object ifFalse: aBlock   [ self ]
UndefinedObject ifFalse: aBlock [ aBlock value ]

Object ifTrue: aBlock ifFalse: bBlock [ aBlock value ]
UndefinedObject ifTrue: aBlock ifFalse: bBlock [ bBlock value ]

Object ifFalse: aBlock ifTrue: bBlock [ bBlock value ]
UndefinedObject ifFalse: aBlock ifTrue: bBlock [ aBlock value ]

" Kernel methods for numbers "

Magnitude < other      [ self subclassResponsibility ]
Magnitude = other      [ self subclassResponsibility ]
Magnitude <= other     [ (other < self ) not ]
Magnitude ~= other     [ (self = other) not ]
Magnitude > other     [ (other < self ) ]
Magnitude >= other    [ (self < other) not ]

SmallInteger + aNumber { (+ self aNumber) }
SmallInteger - aNumber { (- self aNumber) }
SmallInteger * aNumber { (* self aNumber) }
SmallInteger // aNumber { (/ self aNumber) }
SmallInteger \% aNumber { (% self aNumber) }

SmallInteger << aNumber { (<< self aNumber) }
SmallInteger >> aNumber { (>> self aNumber) }

SmallInteger bitAnd: aNumber { (& self aNumber) }
SmallInteger bitOr: aNumber { (| self aNumber) }

SmallInteger < aNumber { (< self aNumber) }
SmallInteger = aNumber { (= self aNumber) }

SmallInteger asFloat { (long->double self) }

Integer negated [ 0 - self ]

Float asFloat [ self ]

Number pi [ 3.14159265358979323846264338327950288419716939937510820974944592 ]

Number squared [ self * self ]

Number sin [ self asFloat sin ]

```

```

Number cos          [ self asFloat cos ]
Number log          [ self asFloat log ]

Float sin           { (sin self) }
Float cos           { (cos self) }
Float log           { (log self) }

Float + aNumber     { (+ self aNumber) }
Float - aNumber     { (- self aNumber) }
Float * aNumber     { (* self aNumber) }
Float / aNumber     { (/ self aNumber) }
Float \ aNumber     { (% self aNumber) }

Float < aNumber     { (< self aNumber) }
Float = aNumber     { (= self aNumber) }

Float negated       [ 0.0 - self ]

Float truncated     { (double->long self) }
Float rounded       [ (self + 0.5) truncated ]

Number between: x and: y [ x <= self and: [self <= y] ]

Number timesRepeat: aBlock
[
  [self > 0]
  whileTrue:
    [aBlock value.
     self := self - 1]
]

Number to: a do: b   [ | i | i := self. [i <= a] whileTrue: [b value: i. i := i + 1] ]
Number by: d to: a do: b [ | i | i := self. [i <= a] whileTrue: [b value: i. i := i + d] ]
Number downTo: a do: b [ | i | i := self. [i >= a] whileTrue: [b value: i. i := i - 1] ]
Number by: d downTo: a do: b [ | i | i := self. [i >= a] whileTrue: [b value: i. i := i - d] ]

" Kernel methods for collections "

String size         { (string-length self) }
String new: n       { (string n) }
String at: n        { (string-at self n) }
String at: n put: c { (set-string-at self n c) }

Collection append: anObject [ self subclassResponsibility ]

IndexableCollection atAllPut: element
[
  0 to: self size - 1 do: [:i | self at: i put: element]
]

IndexableCollection new: n withAll: element
[
  self := self new: n.
  self atAllPut: element.
  self
]

IndexableCollection from: start to: stop do: aBlock
[
  start to: stop do: [:i | aBlock value: (self at: i)].
]

IndexableCollection do: aBlock
[
  self from: 0 to: self size - 1 do: aBlock
]

IndexableCollection do: aBlock separatedBy: bBlock
[
  self size > 0
  ifTrue:
    [aBlock value: (self at: 0).
     self from: 1 to: self size - 1 do: [:elt | bBlock value. aBlock value: elt]].
]

IndexableCollection select: aBlock
[

```



```

    | answer |
    answer := self new: 0.
    self do: [:e | (aBlock value: e) ifTrue: [answer append: e]].
    answer
]

IndexableCollection collect: aBlock
[
    | answer |
    answer := self new: self size.
    0 to: self size - 1 do: [:i | answer at: i put: (aBlock value: (self at: i))].
    answer
]

IndexableCollection with: other collect: aBlock
[
    | answer |
    answer := self new: self size.
    0 to: self size - 1 do: [:i |
        answer at: i put: (aBlock value: (self at: i) value: (other at: i))].
    answer
]

String toUpperCase          [ self collect: [:c | c toUpperCase] ]
String toLowerCase         [ self collect: [:c | c toLowerCase] ]

Array new: n                { (array n) }
Array size                 { (array-length self) }
Array at: n                { (array-at self n) }
Array at: n put: e         { (set-array-at self n e) }

Array print
[
    #'(' print.
    self do: [:elt | elt print] separatedBy: [' ' print].
    #')' print.
]

Array append: e            [ self at: self size put: e ]

ArrayedCollection copyFrom: start to: stop
[
    | end new newSize |
    end := (stop < 0) ifTrue: [self size + stop] ifFalse: [stop].
    newSize := end - start + 1.
    new := self new: newSize.
    start to: end do: [:i | new at: i - start put: (self at: i)].
    new
]

Symbol asString            { (symbol->string self) }
String asSymbol           { (string->symbol self) }

" A non-trivial demonstration program that creates an Array of floating-point samples of a signal
  containing mixed sine waves, runs a Fourier transform on the signal to extract the sine and cosine
  components at discrete frequencies, then prints a graph of the signal power at each frequency.
"
Array fftForwardReal
[
    | n nm1 nd2 imag pi m j |
    n := self size.
    (n bitAnd: n - 1) = 0 ifFalse: [Smalltalk error: 'FFT size is not a power of 2'].
    imag := Array new: n withAll: 0.0.
    nm1 := n - 1.
    nd2 := n // 2.
    j := nd2.
    " reorder input samples for an in-place FFT "
    1 to: nm1 - 1 do: [:i |
        | k |
        i < j ifTrue: [
            | tr "ti" |
            tr := self at: j.
            self at: j put: (self at: i).
            self at: i put: tr.
            "the imaginary parts are all zero: ignore them"
            "ti := imag at: j."
            "imag at: j put: (imag at: i)."
            "imag at: i put: ti."
        ].
        k := nd2.

```

```

    [k <= j] whileTrue: [
        j := j - k.
        k := k // 2.
    ].
    j := j + k.
].
" recombine N 1-point spectra into a single N-point spectrum "
pi := Float pi.
m := (n asFloat log / 2.0 log) rounded.
1 to: m do: [ :l | "for each power-of-two recombination stage"
    | le le2 ur ui sr si |
    le := 1 << l.
    le2 := le // 2.
    ur := 1.0.
    ui := 0.0.
    sr := (pi / le2 asFloat) cos.
    si := (pi / le2 asFloat) sin negated.
    1 to: le2 do: [ :j | "for each sub-DFT in the stage"
        | jml tr |
        jml := j - 1.
        jml by: le to: nml do: [ :i | "for each recombined pair"
            | ip tr ti |
            ip := i + le2.
            tr := ((self at: ip) * ur) - ((imag at: ip) * ui).
            ti := ((self at: ip) * ui) + ((imag at: ip) * ur).
            self at: ip put: (self at: i) - tr.
            imag at: ip put: (imag at: i) - ti.
            self at: i put: (self at: i) + tr.
            imag at: i put: (imag at: i) + ti.
        ].
        tr := ur.
        ur := (tr * sr) - (ui * si).
        ui := (tr * si) + (ui * sr).
    ].
].
" receiver contains the cosine correlations; answer the sine correlations "
imag
]

Array fftForwardRealPowerNormalised: n
[
    | imag |
    imag := self fftForwardReal.
    0 to: self size - 1 do: [ :k |
        | r i |
        r := self at: k.
        i := imag at: k.
        self at: k put: n * (r squared + i squared). "linear power = magnitude squared"
    ]
]

Array fftForwardRealPower
[
    self fftForwardRealPowerNormalised: (2.0 / self size asFloat) squared
]

" Plot the contents of the receiver between start and stop, with vertical scale between lo and hi.
For each value run aBlock with three arguments: the value, and min and max limits of the current
vertical bin in the plot. A point is plotted in each bin for which aBlock answers true.
"
Array from: start to: stop graphFrom: lo to: hi by: aBlock labeled: label
[
    | dy dyd2 |
    lo := lo asFloat.
    hi := hi asFloat.
    dy := hi - lo / 16.0.
    dyd2 := dy / 2.0.
    hi by: dy downTo: lo do: [:y |
        | z c |
        ' ' print. y < 0 ifFalse: [' ' print]. y print. ' |' print.
        z := 0.0 between: y - dyd2 and: y + dyd2.
        c := z ifTrue: ['-'] ifFalse: [' '].
        self from: start to: stop do: [:v |
            ((aBlock value: v value: y - dyd2 value: y + dyd2) ifTrue: ['*'] ifFalse: [c]) print].
        z ifTrue: [' ' print. stop print. label print].
        '' println.
    ]
]

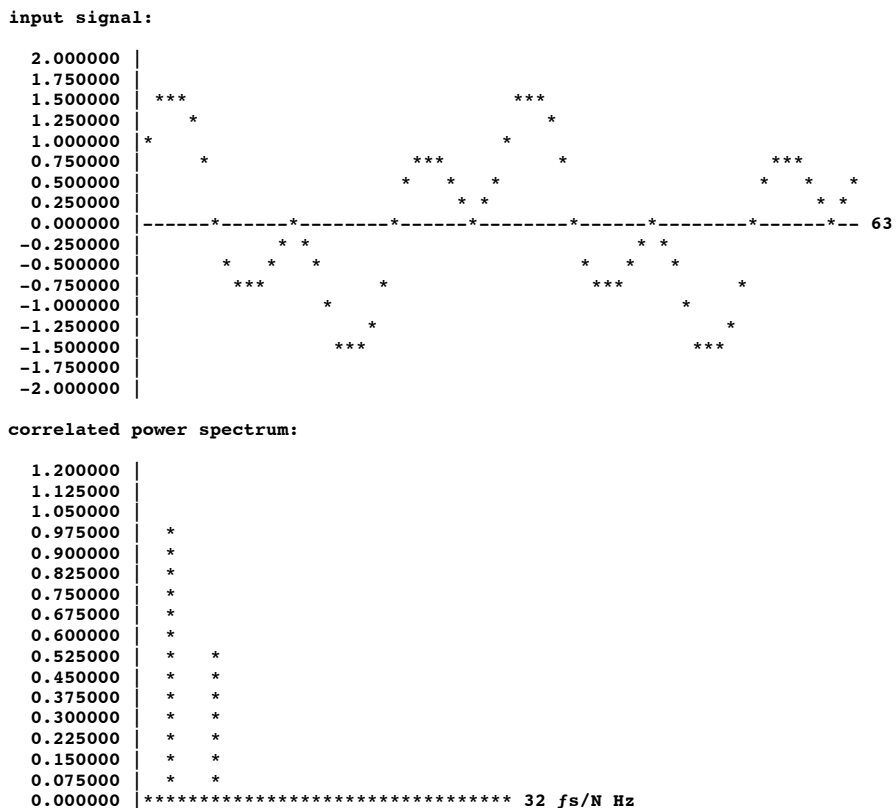
```

```

]
Array from: start to: stop graphFrom: lo to: hi labeled: label
[
  self from: start to: stop graphFrom: lo to: hi
  by: [:x :l :h | x between: l and: h] labeled: label
]
Array graphFrom: lo to: hi labeled: label
[
  self from: 0 to: self size - 1 graphFrom: lo to: hi labeled: label
]
Array testFFT
[
  | twopi isize fsize |
  isize := 64.
  twopi := 2.0 * Float pi.
  self := self new: isize.
  fsize := isize asFloat.
  0 to: isize - 1 do: [ :i |
    self at: i put:
      ((twopi * 2.0 * i asFloat / fsize) cos * 1.00)
      + ((twopi * 6.0 * i asFloat / fsize) sin * 0.75)
  ].
  '\ninput signal:\n' println.
  self graphFrom: -2 to: 2 labeled: ''.
  self fftForwardRealPower.
  '\ncorrelated power spectrum:\n' println.
  self from: 0 to: isize // 2 graphFrom: 0 to: 1.2
  by: [:x :l :h | x > l] labeled: ' \u0192s/N Hz'.
]
Array testFFT.
'\nThat''s all, folks' println.

```

2. Output from running the above Maru program



That's all, folks