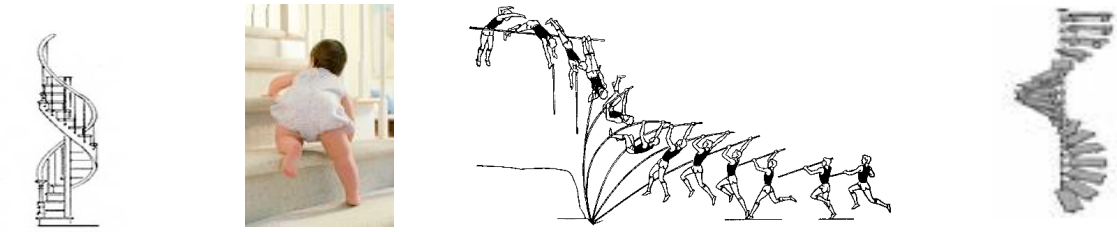


STEPS Toward The Reinvention of Programming

First Year Progress Report
Dec 2007

VPRI Technical Report TR-2007-008

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

STEPS Toward The Reinvention of Programming

First Year Progress Report, Dec 2007

The **STEPS** project is setting out to create "Moore's Law Software": a high-risk high-reward exploratory research effort to create a large-scope-and-range software system in 3-4 orders of magnitude less code than current practice. The detailed **STEPS** proposal can be found at http://www.vpri.org/pdf/NSF_prop_RN-2006-002.pdf. Other documentation on this research can be found in the Inventing Fundamental New Computing Technologies section of the VPRI web site: http://www.vpri.org/html/work/ifnct.htm

Table of Contents

Research Personnel for the STEPS Project

Principal Investigators


Alan Kay (PI)


Ian Piumarta (co-PI)


Kim Rose (co-PI)


Dan Ingalls (co-PI)
Sun Microsystems

Researchers


Dan Amelang


Ted Kaehler


Yoshiki Ohshima


Scott Wallace


Alex Warth


Takashi Yamamiya

Colleagues and Advisors


Jim Clune
UCLA


John Maloney
MIT Media Lab


Andreas Raab
qwaq, inc


Dave Smith
qwaq, inc


David Reed
MIT


Vishal Sikka
SAP


Chuck Thacker
Microsoft

Viewpoints Research Institute

Viewpoints Research Institute http://vpri.org/ is a 501(c)(3) nonprofit public benefit organization set up to conduct advanced computer related research energized by the romance and methods of ARPA-IPTO and Xerox PARC in the 1960s and 1970s.

Over the years we have been inspired by the overlap of "deep personal computing for children" and "dynamic systems building". This has brought forth inventions of systems that are close to end-users (GUIs and WYSIWYG authoring systems, programming languages and environments, etc.), fundamental software architectures, and many kinds of hardware organizations (personal computers, displays, multiple CPU architectures, microcode, FPGAs, etc.).

Funding and Funders in 2007

One of the highlights of late 2006 was receiving major multiyear funding from a variety of sources, that for the first time allowed several of the most difficult projects we've been interested in to be started and staffed.

The major funding for **STEPS** are 5 year grants from **NSF** (Grant# 0639876) and from **FATTOC** http://www.fattoc.com/static/overview.html. We would particularly like to thank the **NSF CISE** Program Managers who were instrumental in securing this grant, and the President of FATTOC for his generosity. **Intel** provided funding in 2006 that helped us put together the proposal for this grant.

Other critical support in 2007 came from **SAP**, **Nokia Labs**, **Sun Labs** and **Applied Minds, Inc.**

STEPS Toward The Reinvention of Programming

- A "Moore's Law" Leap in Expressiveness

We *make*, not just *to have*, but *to know*

About This Report

We have received surprisingly many inquiries about this project from outside the mainstream computer science research community – especially from students and from people involved in business computing. We think students are interested because this project seems new and a little unusual, and the business folk because the aim is to reduce the amount of code needed to make systems by a factor of 100, 1000, 10,000, or more.

Though quite a lot of this project is deeply technical (and especially mathematical), much of this first year is "doing big things in very simple ways". Besides being simple and understandable, many of the results are extremely pretty, some even beautiful. This tempted us to make some of these results more accessible to a wider group of readers. We have prepared several levels of detail.

- The report in your hands is a summary of the first year's results with a little technical detail.
- **Appendices A-G** contain more detailed examples (mostly short programs) of some of the results, and are referred to in the body of this report.
- Finally, we publish much more detailed technical papers and reports in the literature and our website http://www.vpri.org/html/work/ifnct.htm that contain deeper expositions of the work.

Motivation for This Project

Even non-computer professionals are aware of the huge and growing amounts of processing and storage that are required just to install basic operating systems before any applications (also enormous and growing) are added. For professionals, these large systems are difficult and expensive to create, maintain, modify, and improve. An important question is thus whether all this is actually demanded by the intrinsic nature of software functionality, or whether it is a "bloat" caused by weak and difficult-to-scale ideas and tools, laziness, lack of knowledge, etc. In any case, the astounding Moore's Law increase in most hardware-related things has been matched by the inverse process in software. A comment by Nicholas Negroponte: "Andy[1] giveth, but Bill[2] taketh away!"

However, we are not interested in complaining about Microsoft or any other software producers. As computer scientists we are interested in understanding and improving the important areas of our field. As Marshall McLuhan urged: "Don't ask whether it is right or wrong. Instead try to find out what is going on!"

Our questions about functionality are aimed at the *user's experience* while doing personal computing. They can use keyboards, various pointing devices, and other sensors, and usually have a nice big bitmap screen and high quality sound as their principal outputs. "Personal Computing" for typical users involves a variety of tasks (and not a big variety) that are mostly using simulations of old media (paper, film, recordings) with a few twists such as electronic transferal, hyperlinking, searches, and immersive games. Most users do little or no programming.

Science progresses by intertwining empirical investigations and theoretical models, so our first question as scientists is: if we made a working model of the *personal computing phenomena* could it collapse down to something as simple as Maxwell's Equations for all of the electromagnetic spectrum, or the US Constitution that can be carried in a shirt pocket, or is it so disorganized

[1] Andy Grove of Intel.
[2] Bill Gates of Microsoft.

(or actually complex) to require "3 cubic miles of case law", as in the US legal system (or perhaps current software practice)? The answer is almost certainly in between, and if so, it would be very interesting if it could be shown to be closer to the simple end than the huge chaotic other extreme.

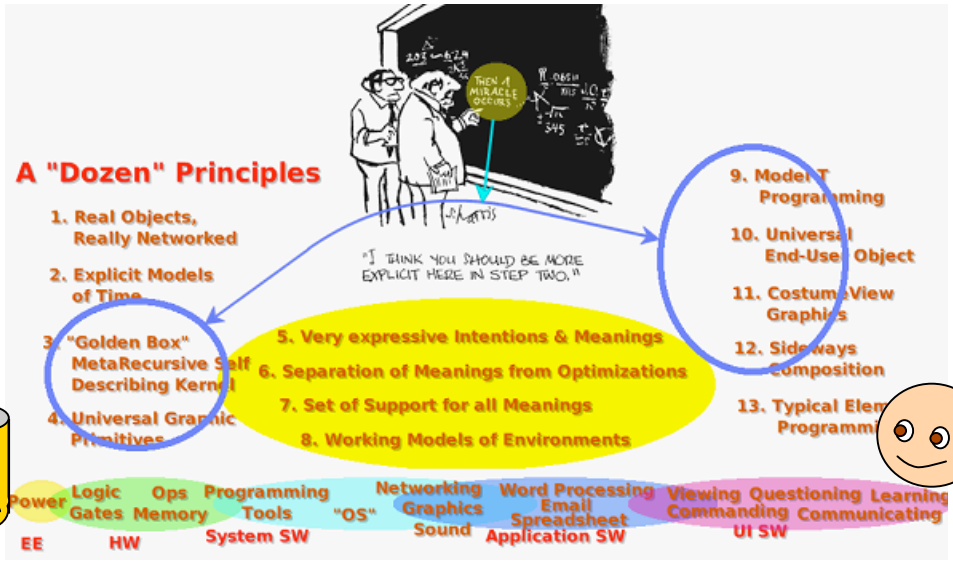
So we ask: is the *personal computing experience* (counting the equivalent of the OS, apps, and other supporting software) *intrinsically* 2 billion lines of code, 200 million, 20 million, 2 million, 200,000, 20,000, 2,000? There are apples vs. oranges kinds of comparisons here, and lots of wiggle room, but it is still an important and interesting question.

For example, suppose it might be only 20,000 lines of code in a new kind of programming system and architecture – this is a modest 400-page book, not the tiny "constitution in the pocket", but not a multivolume encyclopedia or a library of 1000 books (20 million lines of code) or 10,000 books (200 million lines of code). This enormous reduction of scale and effort would constitute a "Moore's Law" leap in software expressiveness of at least 3 and more like 4 orders of magnitude. It would also illuminate programming and possible futures for programming. It might not be enough to reach all the way to a "reinvention of programming", but it might take us far enough up the mountain to a new plateau that would allow the routes to the next qualitative change to be seen more clearly. This is the goal and mission of our project.

General Plan of Attack

The STEPS Proposal http://www.vpri.org/pdf/NSF_prop_RN-2006-002.pdf lays out the goals and sketches some of the dozen "powerful principles" we think can provide the architectural scaling and "dynamic math" that will allow the runnable model of the system to be both small and understandable.

The illustration below shows the "power supply" at the far left and the end-user at the far right, with a dozen principles more or less distributed to three areas. Some of the powerful principles date back into the 1960s and some were postulated more recently. A few have been used in earlier projects, but most of them have never been the guiding principles for a system of this level of comprehensiveness. One of our favorite cartoons **"THEN A MIRACLE OCCURS …"** is perched over the middle area, and this is apt since most of the unknowns of this project lie there.



Our plan of attack is to do many experiments and to work our way to the center from the outsides. This has some strategic value, particularly at the left where one could quickly use up 20,000 lines

of code just doing a tiny, but necessary, part like TCP/IP, or compilation, or state-of-the-art graphics generation. Things are a little easier on the right (at least eventually) because the miracle will have happened (the "**TAMO** Principle"). However, we need quite a few facilities at each end before the miracles are invented, and so part of the bootstrapping process involves making something *like* the system before we can make *the* system.

The desired "miracles" at the heart of the STEPS project have to do with coming up with more powerful "engines of meaning" that can cover wide areas of our large problem space. For example, there is a very large collection of cases in which objects are in multiple dynamic relationships to their "container" and to each other – graphic layout and construction, text handling and formatting, super Spreadsheets, "data bases", scheduling, even register allocation in code generation. We use the metaphor "particles and fields" for this general area. As with many other concerns in computing, each of the traditional trio of syntax, semantics and pragmatics needs to be addressed, but for us, the most important is to come up with a semantic-meaning-kernel that can have expressive syntactic forms defined for it, and for which extensive enough pragmatics can be devised. In other words, it is the runnable debuggable semantics of particles and fields that is central here.

Another example is the myriad of pattern-matching and transformation cases in all levels of the system from the very bottom level code generation, to publish and subscribe control structures, to a new of way of doing TCP, to forward and backwards inference, to the definition of all the languages used, to end-user facilities for scripting and search, etc. These are all instances from the same abstraction, which can be expressed very compactly (and need to be).

So, while we count the lines of code we use (which are expressed in languages with careful syntax that we define), the battles here are fought, won or lost on how much power of meaning lies under the syntactic forms. Because one of our main principles is to "separate meaning from optimizations" we only have to count the lines of meaning that are sufficient to make the system work. Because meaning has to be runnable and debuggable, there are some pragmatics involved whose code has to be counted. These are part of the definition of "active-math" meanings and are separate from optimizations that might be added later. The system must be able to run with all the separate optimizations turned off.

Quick Steep Slope From "Atoms to Life"

Most of today's computer hardware is very weak per instruction and provides few useful abstractions (in contrast to such venerable machines as the Burroughs B5000 [Barton 61]). So it is easy to use up thousands of lines of low-level code doing almost nothing. And the low-level language C needs to be avoided because it is essentially the abstraction of very simple hardware (and this is actually a deadly embrace these days since some recent hardware features have been put in just to run low-level C code more efficiently). Thus we need to have our own way to get to bare machine hardware that has an extremely steep slope upwards to the very high-level languages in which most of our system will be written. The chain of abstractions from high-level to machine-level will include a stage in the processing that is roughly what C abstracts, but this will always be written by automatic processes.

We also think that creating languages that fit the problems to be solved makes solving the problems easier, makes the solutions more understandable and smaller, and is directly in the spirit of our "active-math" approach. These "problem-oriented languages" will be created and used for large and small problems, and at different levels of abstraction and detail.

John von Neumann defined mathematics as "relationships about relationships"; Bertrand Russell was more succinct: *p* ***implies*** *q*. One way of approaching computing is mathematical, but because of the size and degrees of freedom expansion in computing, classical mathematics is only somewhat useful (and can even distract from important issues). On the other hand, making new

mathematical frameworks for dealing with representations and inferences in computing – let's call these *problem-oriented languages of sufficiently high level* – can make enormous differences in the quality and size of resulting designs and systems. The nature of this mathematics is that most artifacts of interest will require *debugging* – just as large theorems and proofs must be debugged as much as proved – and this means that all of our math has to be "runnable". Again, the central concern here is semantic, though we will want this "math" to be nicely human readable.

In addition to "runnable math" and ways to make it, we also need quite a bit of scaffolding for the different kinds of "arches" that are being constructed, and this leads to the organization of tools described below.

The central tool – called **IS** – is a pattern-directed transformation system with various levels of language descriptions from very high-level languages in which we write code, all the way to descriptions of the machine language instructions of our target machines. Two of the other dimensions of this system are protoabstractions of (a) structurings (meta-objects) and (b) evaluations (meta-code). Some of the translation systems are simple and very fast, some have great range and generality (and are less speedy). In the middle of the transformation pipeline are opportunities to make various kinds of interpreters, such as the byte-code VMs that we have employed since the 1960s (although this year we have concentrated exclusively on generating machine code).

A Few Comparisons and Orientations

JavaScript is not an Ultra High Level Language (it is a VHLL, a bit like Lisp with prototypes) but it is well and widely understood enough to make a useful vehicle for comparisons, and for various reasons we have used it as a kind of pivot point for a number of our activities this year. About 170 lines of meta-description in a language that looks like "BNF with transformations" (OMeta) is sufficient to make a JavaScript that runs fast compared to most of the versions in browsers (because IS actually generates speedy machine code rather than an interpreter).

The OMeta translator that is used to make human readable and writable languages can describe itself in about 100 lines of code (it is one of these languages).

IS can make itself from about 1000 lines of code (of itself described in itself).

One of the many targets we were interested in this year was to do a very compact workable version of TCP/IP that could take advantage of a rather different architecture expressed in a special language for non-deterministic processing using add-on heuristics. Our version of TCP this year was doable in these tools in a few tens of lines of code, and the entire apparatus of TCP/IP was less than 200 lines of code. (See ahead for more details.) We had aimed at a solution of this size and elegance because many TCP/IP packages run to 10,000 or 20,000 lines of code in C (and this would use up all of our code budget on just one little subsystem).

Modern anti-aliased text and graphics is another target that can use up lines of code very quickly. For example, the open source Cairo system (a comprehensibly done version of PostScript that is fast enough to be used for real-time interfaces) is about 44,000 lines of C code, most of which are various kinds of special case optimizations to achieve the desired speed. However, underlying Cairo (and most good graphics in the world) is a mathematical model of sampling and compositing that should be amenable to our approach. A very satisfying result this year was to be able to make an "active math" system to carry out a hefty and speedy subset of Cairo in less than 500 LOC (more on this ahead).

Language Strategies

The small size required to make useable versions of very high-level languages allows many throwaway experiments to be done. How the semantics of programming languages should be expressed has always been a much more difficult and less advanced part of the extensible language

field. (We are not satisfied with how we currently achieve this, even though it is relatively compact and powerful.) Each different kind of language provides an opportunity for distilling better semantic building blocks from all the languages implemented so far. At some point a more comprehensive approach to semantics is likely to appear, particularly in the mid-range between very high-level and low-level representations.

Relation to Extensible Languages Initiatives of the 1960s and 1970s

The advent of BNF and the first uses of it to define "translator writing systems" (for example "The Syntax Directed Compiler" by Ned Irons) led to the idea of statically (and then dynamically) extensible languages ("IMP", Smalltalk-72, etc.). Part and parcel of this was the belief that different problems were best expressed in somewhat custom dialects if not whole new language forms. Some of our very early work also traversed these paths, and we plan to see how this old dream fits to the needs and users of today. However, for the first few years of this project, most of our interests in easy extensions are aimed at finding succinct characterizations of the problem and solutions spaces – semantic architectures – for various systems problems that must be solved.

Relation to Domain-specific Languages of Today

In the last few years several good-sized initiatives (cf., Fowler-05) have arisen to retread the ground of "problem-oriented languages" (now called Domain-specific Languages). One of the most impressive is "Intentional Software" by Charles Simonyi ("Intentional Software", Simonyi, et al.). This project, like ours, came out of some "yet to be dones" from research originally carried out at Xerox PARC, and both the similarities and differences trace their routes back to that work. Similar are the mutual interests in having the surface level expressions of code be done in terms that closely fit the domain of interest, rather than some fixed arbitrary forms for expressing algorithms. Most different is the interest in STEPS of making an entire "from end-users to the metal" system in the most compact and understandable form "from scratch". The emphasis in STEPS is to make a big change in the level of meaning (both architectural and functional) that computers compute. This should *create* new domains and languages for them.

Relation to Specification Languages and Models

Some of the best work in specification and semantic languages – such as Larch, OBJ, etc. – has influenced the thinking of this project. Our approach is a little different. Every expression in any language requires debugging. Any language that is worth the effort of writing and debugging any kind of expression of meaning *should simply be made to run*, and just **be** the language. Similarly, the recent work in modeling (too bad this term got co-opted for this) is only convincing to us if the models can be automatically extracted (and if so, they then form a part of an underlying integrity system that could be a useful extension of a type system). Our approach is simply to make the expression of desirable meanings possible, and easy to write, run and debug. We use dynamic techniques and the architecture at all levels to ensure safety in rather simple ways.

2007 STEPS Research Activities

During the first year of the project we have concentrated on the extremities of the system: bootstrapping live systems from meta-descriptions, and making user experiences and interfaces using "unitarian" objects that can be composed indefinitely. For example, because our parsers can easily bootstrap themselves, they could easily be used as front ends for IS, Squeak and JavaScript.

- The IS version allows ultimate *utilities* to be made by compiling machine code.
- The Squeak version allows its considerable resources to be used to scaffold many experiments.
- The JavaScript version allows easy illustration of some of the experiments to be shown directly in a web browser.

Another example is found in the low-level rich-function graphics and mathematical transformations that can bring an entire visible object scheme to life with very little machinery. All of these will be described in more detail ahead.

We have built a number of "dumbbell models" this year using different architectures, each of which supported experiments on their component parts. We are building these models to learn and not necessarily to have. Many of them will ultimately be discarded once the invaluable experience of building them has been gained. This being said, in some cases the models have matured into stable subsystems that will continue to serve us throughout the remainder of the project.

Major Results in 2007 are listed below and descriptions follow:

- **several meta-parser/translators (Thesis work);**
- **IS, a parametric compiler to machine code that can handle a number of CPUs;**
- **a graphical compositing engine (Thesis work);**
- **a VG engine with speedy low-level mathematical rendering (Thesis work);**
- **a high-level graphics system using universal polygons, transforms and clipping windows;**
- **a number of languages including: JavaScript, Smallertalk, Logo, BASIC, Prolog, Toylog, Dataflow, CodeWorks, and specialty languages for metalinguistic processing, mathematics, graphics (SVG, Cairo) and systems (TCP/IP);**
- **an end-user authoring system made in JavaScript and SVG;**
- **a pretty complete HyperCard system using CodeWorks as the scripting language;**
- **control structure experiments in massive parallelism in our JavaScript and Logo;**
- **workable TCP/IP using non-deterministic inference in less than 200 lines of code;**
- **a major IDE system for IS;**
- **a working model of a tiny computer that can be instantiated on FPGA hardware;**
- **super high-level "compiling of agents" from declarative descriptions (Thesis work);**
- **architectural issues and designs.**

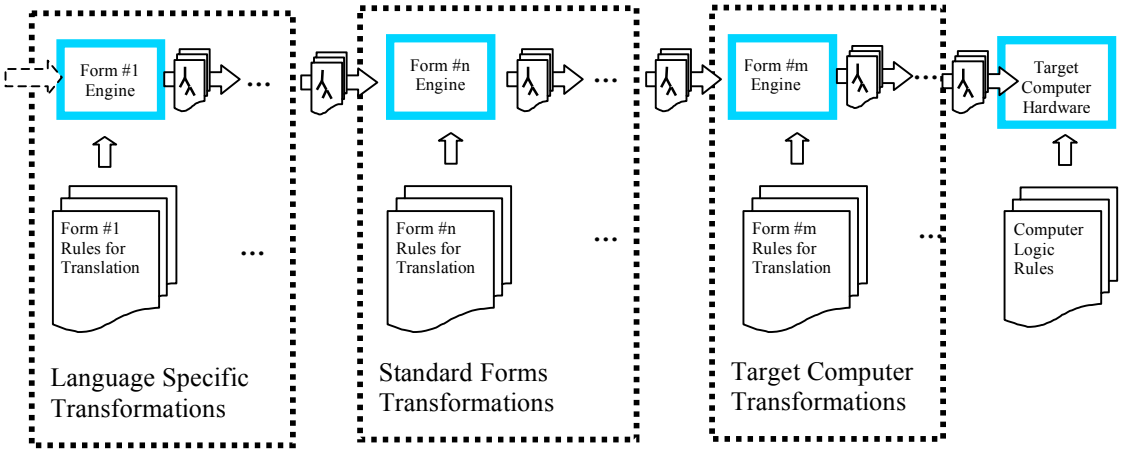
These experiments served to calibrate our "sense of entropy" for various parts of our task. For example, all the languages (including most of JavaScript) could be defined and made to run in under 200 lines of code fed to our metasystems. The graphical compositing engine can handle a hefty subset of Cairo (an open-source relative of PostScript) in less than 500 lines. This is critical because we have to be able to cover the entire bottom of the system with just a few thousands of lines of code, and thus we must validate the techniques we plan to use in the first phase of implementation.

Major Progress in 2007: Findings and Summary Explanations

IS — Meta-meta Language-language and Parametric Compiler

Principal Researcher: Ian Piumarta

IS can instantiate new programming paradigms and systems, including itself. It demonstrates the power of "extreme late binding" and treats many of the static vs. dynamic choices that are traditionally rigid (compilation, typing, deployment, etc.) more like orthogonal axes along which design decisions can be placed. A rapidly maturing prototype of **IS** has been made publicly available and several systems of significant complexity have been created with it.



The IS system can be thought of as a pipeline of transformation stages, all meta-extensible.

This is basically "good old-time computer science with a few important twists". **IS** can be thought of a "pipeline of transformations" coupled with resources – i.e., an "essence-procedure-framework", an "essence-object-framework", storage allocation, garbage collection, etc. Each of the transformational engines is made by providing meta-language rules. (The ones in the language-specific front ends look a little like BNF, etc.).

JavaScript: For making a complete JavaScript language translator and runtime, it takes only 170 lines of meta-language fed in at the "Form #1 Rules for Translation" stage. (We have to make much stronger languages than JavaScript in this project, but – because of the familiarity of JavaScript – being able to make an efficient version so easily provides some perspective into the "meta-ness" of our approach.)

This was partly easy because JavaScript is pretty simple mathematically and formally, and has nothing exotic in its semantics. The outputs of the first Language Specific stage are standard forms that can be thought of as tree or list structures. (See ahead for more description of this, and Appendix N to look at this code.)

The Standard (fancy term: "canonical") Form stage deals with semantic transformations into forms that are more like computers (we can think of something that is *like* an abstract, improved, dynamic C semantics at the end here).

The Target Computer stage is made from rules that specify the salient architectural forms (instructions, register set ups, etc.) and perhaps a few non-standard organizations the CPUs might have. We currently have three targets installed: Intel, PowerPC, StrongARM. The result here is actual machine code plus environment to help run it. As a result, JavaScript is quite speedy.

Note that the very last "engine and rules" of computer hardware and logic could be further decomposed if FPGAs are used. (See "Tiny Computer" example on page 29.)

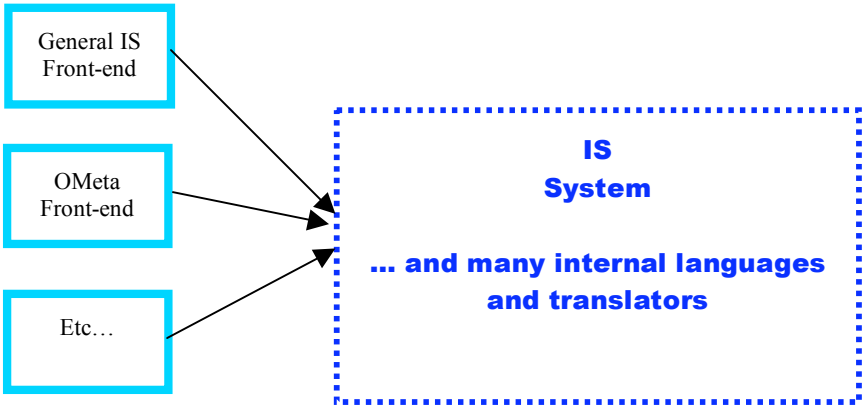
Prolog: A more exotic language like Prolog requires a little more environment to be supplied because of its unification and backtracking semantics. But still, this only takes about 90 lines of code total. This is because the syntax of Prolog is simple (9 lines of meta-language), and the additional semantics can easily be written in the somewhat Lisp-like IS language framework in about 80 more lines. (See ahead for more description of this, and Appendix C for the actual code.)

TCP/IP: One of the most fun projects this year was to make a tiny TCP/IP, thinking of it as a kind of a "parser with error handling". This is described a few pages ahead, took less than 200 lines of code to accomplish, and is a clear and efficient program written this way.

Graphics: Another fun project (described ahead) is a variety of efficient and compact graphical transformations in the genres associated with PostScript imaging for text and graphics. Again, the mathematics was organized for this: a language for expressing the mathematics was designed, the language translation was done by a front-end to IS, and the optimization stages of the pipeline were able to produce efficient machine code on-the-fly to do the operations. This means that special casing the code (as is usually done in such graphic systems – e.g., Cairo) is not necessary.

Itself: Since **IS** is made from parametric languages, it should be able to make itself from a meta-description. It takes about 1000 lines of code to make the engines and resources that will produce a new version of **IS**. (This is good, because we really have to count **IS** as part of the code base for this project.) Another way to look at this is that, for example, JavaScript from scratch really takes about 1170 lines of code to make a runnable system for 3 different CPUs (but with the side-benefit that other languages can be compactly made also with about 100–200 more lines of code each).

IS has a number of front-ends that are used for different purposes.



OMeta (described ahead) is the most general and wide-ranging front-end, and has been used for projects with the IS back-end, Squeak, and with JavaScript. TCP/IP used several interesting meta-forms and languages that were specially made to as the solution progressed.

This project requires much stronger yet-to-be-invented language forms than the ones we've been making in 2007. Quite of bit of the actual effort here will be to make the stronger semantic bases for these languages. We are confident that the apparatus we've made so far will be able to accommodate these stronger forms.

Graphical Compositing and Rendering

The story of this work has an interesting twist. The original plan was to deal with the immense amount of progress and work that has been done in modern computer graphics by using the very capable open source graphics package Cairo, which is a kind of second generation design and adaptation of PostScript.

Cairo is large, but our thought was that we could use Cairo as "optimizations" if we could build a small working model of the Cairo subset we intended to use. However, in a meeting with some of the main Cairo folks they explained that much of the "bloat" in Cairo was due to the many special case routines done for optimization of the compositing stage. What they really wanted to do was just-in-time compilation from the "math" that directly expressed the desired relationships. The IS system could certainly do this, and one of the Cairo folks – Dan Amelang (now part of Viewpoints) – volunteered to do the work (and write it up as his Master's Thesis for USCD).

So, the "twist" here is that the IS model Dan made is actually directly generating the high efficiency machine code for the compositing stage of Cairo. The relationship has been reversed: Cairo is using the IS system as the "optimization."

Jitblt

Principal Researcher: Dan Amelang

— a graphical compositing engine in which pixel combination operators are compiled on demand (done from meta-descriptions in **IS**). Traditional (static) compositing engines suffer from combinatorial explosion in the number of composition parameters that are possible. They are either large and fast (each combination is coded explicitly) or small and slow (the inner loops contain generic solutions that spend most of their time in tests and branches). Jitblt uses the dynamic "behavior instantiation" facilities of **IS** to convert a high-level compositing description into a complete compositing pipeline at runtime, when all the compositing parameters are known. The resulting engine is small (460 lines of code) and fast (it competes with hand-optimized, explicitly-coded functions). It has been deployed as an alternative compositing engine for the popular "pixman" library, which is what Cairo and the X server use to perform compositing.

Several specially designed "little languages" allow parts of the pipeline to be expressed compactly and readably. For example, the compositing operator **over** is quite simple:

compositing-operator: **over** : x+y*(1.0 – x.a)

Hundreds of lines of code become one here. The Jitblt compilation has to do automatic processing to efficiently make what is usually hand-written code cases. We can define the compositing operator **in** as:

compositing-operator: **in** : x*y.a

Another case is handling the enormous number of pixel formats in a way that can be automatically made into very efficient algorithms at the machine code level. A very simple syntax for specifying the makeup of a pixel is

four-component-case	:: component "," component ","component "," component
component	:: comp-name ":" component-size
comp-name	:: a | r | g | b
comp-size	:: integer

Notice that this grammar is context sensitive. Combining the two formulas, we can express the most used case in compositing for 32 bit pixels as the formula:

a:8, r:8, g:8, b:8 **in** a:8 **over** x:8, r:8, g:8, b:8

using the syntax definition:

formula :: source “**in**” mask “**over**” dest

Most of the spadework here is in the semantics (including the context sensitivity of the syntax) and especially the pragmatics of the compilation.

Two images digitally composited using Jitblt. The water texture image is masked by the anti-aliased text image and combined with the sand dune background image using the Porter-Duff over operation (i.e., water in text over dunes).

Gezira

Principal Researcher: Dan Amelang

— a small and elegant 2-D vector graphics engine. Gezira is meant to be used primarily for displaying graphical user interfaces but is well suited for displaying any 2-D graphical content such as SVG artwork or Adobe Flash animations.

Gezira replaces all external dependencies on third-party rendering libraries in IS. Only the most fundamental graphics components of the host windowing system are used. When desirable, Gezira will use the frame buffer directly.

Gezira draws its inspiration from the Cairo vector graphics library. Gezira is the name of a small, beautiful region in central Cairo (the city). Thus, the name "Gezira" is meant to suggest a small, elegant vector graphics implementation that is born out of the core concepts of the Cairo library.

The primary goal of Gezira is to express the fundamentals of modern 2-D graphics in the most succinct manner possible. At the same time, high-performance is also desirable where possible without interfering with the primary goal. Gezira employs a number of novel approaches to achieve this balance.

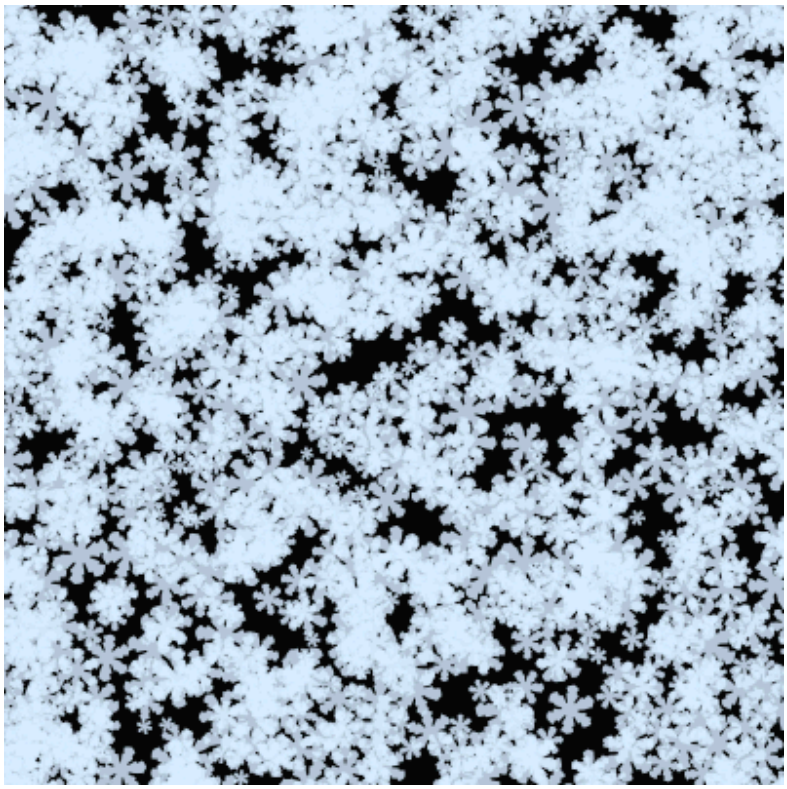
For example, the rasterization stage is often the most complex and performance-intensive part of the rendering pipeline. Typically, a scan-line polygon fill algorithm is employed, using some form of supersampling to provide anti-aliasing. Our goal was to avoid the complexity and performance disadvantages of this approach while maintaining adequate output quality for our purposes. To this end, Gezira uses an analytic pixel coverage technique for rasterization that can express exact pixel coverage via a mathematical formula.

This formula expresses the exact coverage contribution of a given polygon edge to a given pixel. The total coverage of a polygon is merely the linear combination of the edge contributions. (A variation of this formula allows for efficient rasterization by tracing the polygon edges, thus

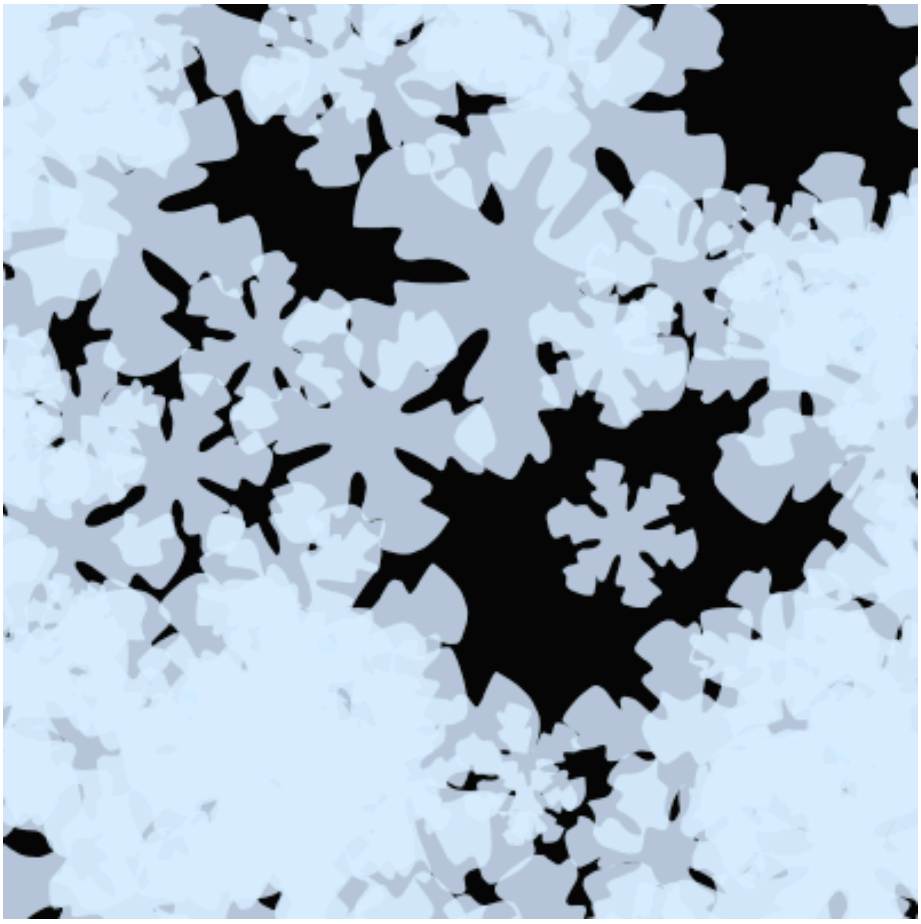
avoiding the "uninteresting" inner pixels.) This approach allows us to express this typically complex stage of the pipeline in only 50 lines of code instead of the 500+ lines of code seen in similar libraries. The Gezira rendering formula is presented mathematically in Appendix G.

Gezira, in its current state, already implements a good subset of standard vector graphics functionality in 450 lines of code. This functionality includes high-quality anti-aliased rasterization, alpha compositing, line and Bézier curve rendering, coordinate transformations, culling and clipping.

Once the core mathematics, algorithms and data structures of Gezira stabilize, we will set out to design a domain-specific language (or perhaps languages) for describing the graphics system. We hope to reduce the system size by an additional order of magnitude through this effort.



1400 animated "snowflakes". Each snowflake is composed of 64 cubic Bézier curves. The snowflakes are transformed, decomposed, rasterized (with anti-aliasing) and alpha-blended together. Each snowflake is assigned a psuedo-random scale, position, rotation, vertical velocity and angular velocity. The animation runs at ~10 frames per second on a 1.8 GHz Pentium M.



Detail of the same snowflake scene, zoomed by a factor of 9. The vector graphics really shine here, as raster graphics would display extreme pixelation at this scale.

Universal Polygons and Viewing

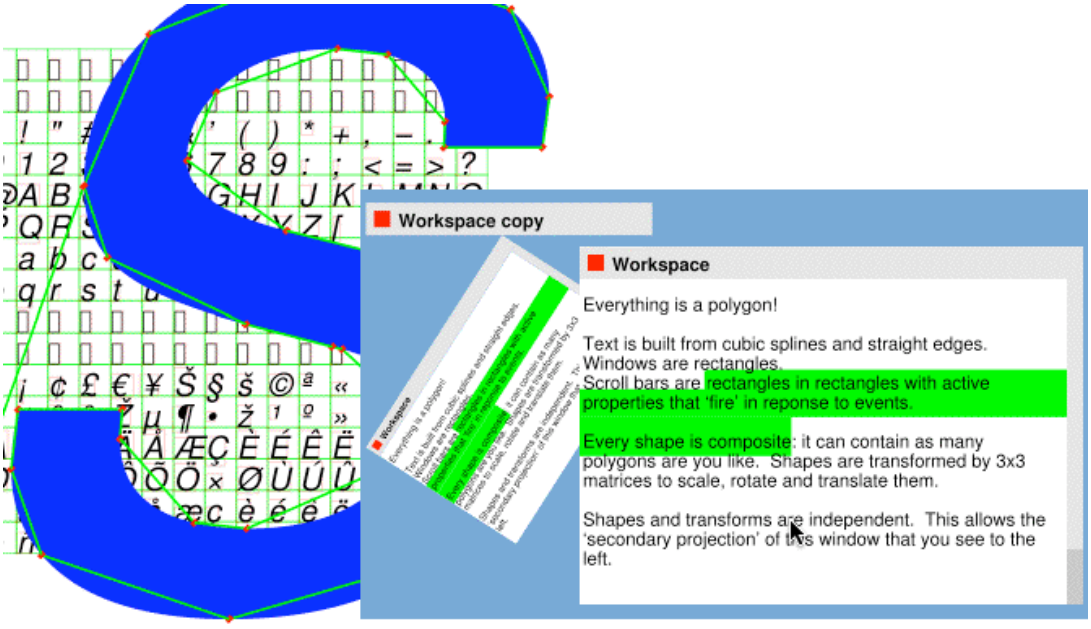
Principal Researcher: Ian Piumarta

— a runnable model of graphics, windows and user interaction. Here is an example of "making the many into one" at the level of meaning.

We chose 2D graphics for this experiment, but it would work just as well in 3D. The basic idea is to find an element of graphical meaning that can be used at all scales and for all cases, and to build everything else from it. A pragmatic solution could be triangles, since they can be (and often are) used to cover and approximate spatial regions for rendering (and some of today's graphics accelerators are triangle based). We chose polygons because they (along with curves interpolated between their vertices) can be used to make shapes that are meaningful to all levels of users (for example: triangles, rectangles, circles, ovals, text characters in many fonts, etc.). These can be positioned and manipulated by simple transforms, and the fills can be combinations of textures and mathematics. If we can composite and render them efficiently, then we have made the basis for the general graphics of personal computing.

The multiplicity of components and corresponding complexity found in most UI toolkits is eliminated by considering the UI as a "scene" described entirely by polygons and affine transformations. Even the characters making up regions of text are polygons, transformed into appropriate spatial relationships. This unifies, generalizes and simplifies every entity in the UI. An encouraging early result is that the Gezira graphics engine can render "glyphs-as-polygons" fast enough to support real-time scrolling of text without the usual retained bitmaps or other complicating optimizations. The current prototype is about 3,500 LOC (including models of geometry, color, typefaces, events, interaction and connections to platform windows), which will decrease as better abstractions are formulated for the primitive elements and algorithms.

This is a good measure for much of what we wish to accomplish with visible objects.



Just polygons and affine transformations produce everything on this desktop.

A Tiny TCP/IP Using Non-deterministic Parsing

Principal Researcher: Ian Piumarta

For many reasons this has been on our list as a prime target for extreme reduction.

- many implementations of TCP/IP are large enough to consume ½ to our entire code budget.
- there are many other low-level facilities that also need to be handled very compactly; for example, the somewhat similar extreme treatments of low-level graphics described above.
- there are alternative ways of thinking about what TCP does that should collapse code down to a kind of non-deterministic pattern recognition and transformation process that is similar to what we do with more conventional language-based representations.
- TCP/IP is also a metaphor for the way complex systems should be designed and implemented, and, aesthetically, it would be very satisfying to make a more “active math” formulation of it that would better reveal this kind of distributed architecture.

The protocols are separated into IP (which handles raw sending and receiving of packets, but with possible errors from vagaries of the networking machinery, such as out-of-order or dropped packets), and TCP (which is a collection of heuristics for error detection, correction and load balancing). This separation allows other strategies for dealing with packets to be attached to IP (for example, UDP is a simpler protocol that allows developers to deal with streams and retransmissions in their own manner).

In our “active math” version of this, the TCP stream and retransmission schemes are just a few lines of code each added to the simpler UDP mechanics. The header formats are actually parsed from the diagrams in the original specification documents.

Here, we give a glimpse of what the “programming with a grammar” looks like for the rejection of incoming packets with non-expected tcp-port or tcp-sequenceNumber, and which provides correct tcp-acknowledgementNumbers for outgoing packets.

```
['{ svc     = &->(svc? [self peek])
    syn     = &->(syn? [self peek]) .   ->(out ack-syn    -1 (+ sequenceNumber 1) (+ TCP_ACK TCP_SYN) 0)
    req     = &->(req? [self peek]) .   ->(out ack-psh-fin 0 (+ sequenceNumber datalen (fin-len tcp))
                                                             (+ TCP_ACK TCP_PSH TCP_FIN)
                                                             (up destinationPort dev ip tcp
                                                                 (tcp-payload tcp) datalen))
    ack     = &->(ack? [self peek]) .   ->(out ack           acknowledgementNumber
                                                             (+ sequenceNumber datalen (fin-len tcp))
                                                             TCP_ACK 0)
            ;
    ( svc (syn | req | ack | .) | .     ->(out ack-rst       acknowledgementNumber
                                                             (+ sequenceNumber 1)
                                                             (+ TCP_ACK TCP_RST) 0)
    ) *
  } < [NetworkPseudoInterface tunnel: '"/dev/tun0" from: '"10.0.0.1" to: '"10.0.0.2"]]
```

The text between curly braces defines a grammar object. The '<' message, with a network “tunnel” interface as argument, creates and runs a parser for the grammar, connected to a stream reading packets from the interface.

The first rule is a predicate that filters out unknown service port numbers. The next three describe the appropriate replies to SYN packets, connection data transfer packets, and ACK packets received from a connecting client. The 'out' function invoked from the actions in these rules reconstructs a TCP/IP packet with the given parameters, fills in the checksum, and writes the packet to the network interface.

See **Appendix E** for a more complete explanation of how this "Tiny TCP" was realized in well under 200 lines of code, including the definitions of the languages for decoding header format and for controlling the flow of packets.

OMeta — Principal Researcher: Alex Warth

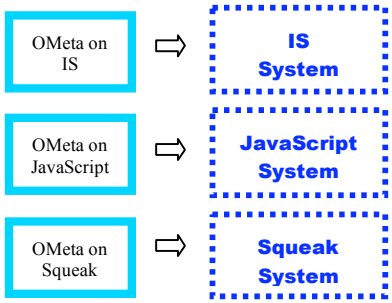
— a highly expressive, wide-spectrum meta-translator. Tokenization, phrase recognition and structural rewriting are central to many programming systems, and ours is no exception. They are traditionally implemented as separate translation stages, necessitating incompatible intermediate representations between each stage and a distinct engine to implement each transformation. We have built a prototype pattern recognition and translation engine that unifies these three activities within a single framework. These three (or more) phases communicate information in both directions, can be arbitrarily overlapped, and are used to describe (and bootstrap) their own implementation (inspired by the Meta II system, dating back to the 1960s). This latter point, in conjunction with the dynamic features of IS, implies that the structures the translator manipulates are within the domain of its own implementation, making "introspective" modification of the system's "chain of meaning" available to, and natural for, system designers and end-users alike.

The general form of an OMeta "rule" resembles BNF plus a translation/transformation meaning. For example, the classical definition of addition syntax looks like:

```
addExpr      = addExpr:x "+" mulExpr:y  -> (x, '+', y)
```

The transformation part of this rule is an executable expression whose meaning depends on the context. In this case, it could produce either a direct expression in another language (such as Squeak or JavaScript), or it could produce a node in a tree to be further processed by **IS**.

Just as **IS** can have several front-ends, including this system, OMeta can use a variety of back-ends.



In the next few examples we will show some of the languages that have been brought to life with just a few lines of code in OMeta. We list them here:

JavaScript - 170 LOC in **IS** and **Squeak**. Compatible with version 1.4 that runs in all browsers.

Smalltalk - 200 LOC in **JavaScript** with a few changes.

Logo - 50 LOC in **Squeak** and **JavaScript**. See ahead, then **Appendix A** for more details.

Prolog - less than 100 LOC in **JavaScript**. See ahead, then **Appendix B** for more details.

Toylog - about 120 LOC in **JavaScript**. See ahead, then **Appendix C** for more details.

Codeworks - about 100 LOC in **Squeak**. Used in HyperCard as scripting language.

OMeta - about 100 LOC in **IS, Squeak, JavaScript**. See **Appendix D** for more details.

JavaScript

Principal Researchers: Alex Warth and Yoshiki Ohshima

— a complete implementation in less than 200 lines of code. We built this to investigate and inspire our "unified, homogeneous" approach to the "chain of meaning" in programming system implementation. JavaScript programs are parsed and translated into structures representing JS semantics according to an OMeta-like "grammar". Simple rewrite rules convert these structures into the native structures of IS.

Most of the heavy lifting (converting these into executable machine code) is therefore completely hidden from the system implementer, by presenting the implementation of IS itself as a reusable component of any programming system.

This experiment suggests to us that a serious "standards-conforming" implementation would not be significantly larger or more complex. We are particularly encouraged by the overall simplicity of the implementation and the potential it represents for end-users to create their own expressive programming languages and systems specifically targeted at their particular problem.

Dynamically Extensible JavaScript

If we add to the 170 lines of definition for JavaScript, the 40 lines of definition that it takes to make OMeta itself (see ahead), we can create a JavaScript that can extend itself on-the-fly. This is useful both for trivial needs (such as making more readable syntax in place of the awkward JavaScript constructions), and for embedding new computing paradigms (such as Prolog) which make problem solving easier and more expressive.

Simple Example: An expressive case statement for JavaScript

One example for an expressive case statement could look like this:

```
case f(x) + 2 of
  <  0: alert("this is a less than");
  == 0: alert("this is an ugly equals");
  >  0: alert("this is a greater than");
```

The basic idea is that we can separate the calculation of the value from the various tests and cases, making the result easier to read, write and understand. This is easy to add to JavaScript using OMeta-within-JavaScript.

```
ometa CaseJSParser : JSParser {
  stmt     = "case" expr:v "of" caseBody:b                    -> {#call. {#call. {#func. {'_cv_'}.
                                                                    {#begin. {#return. b}}}. v}}.

  operator = "<" | "<=" | "==" | ">=" | ">".
  caseBody = operator:op addExpr:x ":" srcElem:t caseBody:f  -> {#condExpr. {#binop. op. {#get. '_cv_'}. x}.
                                                                   {#func. {}. {#begin. t}}.
                                                                   f}
           |                                                 -> {#func. {}. {#begin. t}}.
}
```

Since we have all of JavaScript + OMeta defined in OMeta, it is easy to consider the idea of giving all of JavaScript a nicer look and feel. But we can go further – adding new bases to JavaScript itself. The new case structure is an example of making a new control structure and giving it a simple expressive look. A more comprehensive extension would be to define a number of deep control structures that introduce non-deterministic processing (see the next example) or implement powerful ways to do massively parallel programming.

Prolog and Toylog in OMeta

Principal Researcher: Alex Warth

Prolog has a simple, uniform syntax and it is straightforward to write an OMeta translator for it in just a few lines.

```
Ometa PrologTranslator : Parser {
variable   =  spaces firstAndRest(#upper, #letterOrDigit):name      -> (Var new: name mash).
symbol     =  spaces firstAndRest(#lower, #letterOrDigit):name      -> (Sym new: name mash).
clause     =  symbol:sym "(" listOf(#expr, ',':args ")"             -> (Clause new: sym : args).
expr       =  clause | variable | symbol.
clauses    =  listOf(#clause, ',').
rule       =  clause:head ":-" clauses:body "."                     -> (Rule new: head : body)
           |  clause:head                   "."                     -> (Rule new: head : {}).
rules      =  rule*:rs spaces end                                   -> rs.
query      =  clause:c spaces end                                   -> c.
}.
```

However, the semantics of Prolog are rather different from most programming languages, particularly with its use of variables and ability to "fail" and try other routes to success.

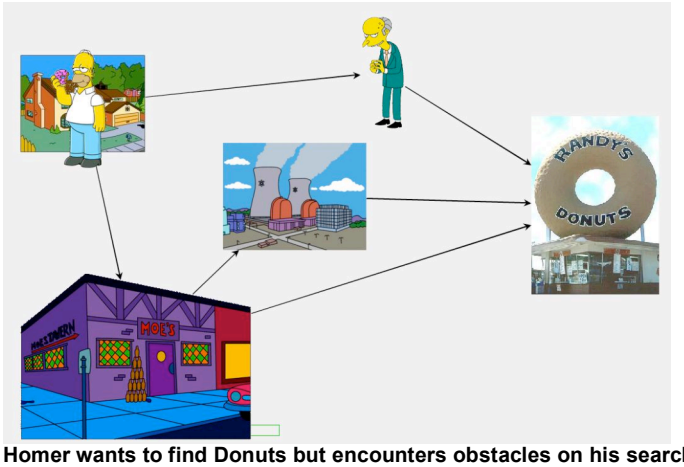
```
father(abe, homer).
father(homer, lisa).
father(homer, bart).
grandfather(X, Y) :- father(X, Z), father(Z, Y).
? grandfather(abe, X)
```

Our version of the semantics of Prolog came to about 80 lines of code (and in the process we discovered a number of ways to make this quite a bit smaller in 2008). See **Appendix C** for what this looks like written in JavaScript. This investigation leads to issues of "the control of control" and suggests the existence of much better control primitives for language building.

The English syntax of Toylog is a little more involved and required about 35 lines.

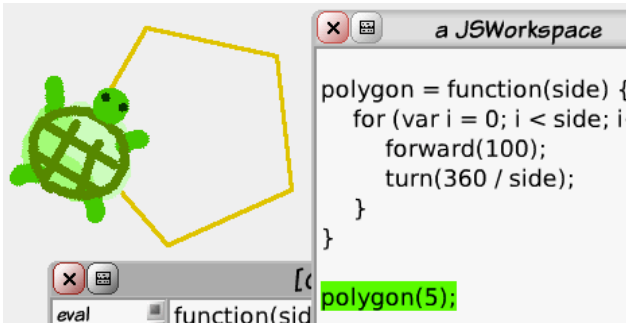
```
Abe is Homer's father.
Homer is Lisa's father.
Homer is Bart's father.
x is y's grandfather
    if x is z's father and z is y's father.
? x is y's grandfather
```

In Toylog, which is aimed at children, questioning, answering and problem solving were connected with animations to make the underlying mechanisms tangible and evident for a non-expert audience.



Homer wants to find Donuts but encounters obstacles on his search.

One of the several JavaScript implementations was done using the Squeak back-end (see to the right), and another was done "all the way to metal" in IS.



OMeta in Itself

Principal Researcher: Alex Warth

OMeta is described in itself. Its self-definition, used to generate an OMeta parser within any "reasonable" target language, is about 40 lines of code. See **Appendix D** for more details.

Embeddable OMeta

Besides being compact and easily transportable, the self-definition can be included with any language description so that the OMeta translations can be used on-the-fly along with the normal statements and expressions in the language. This is useful when dealing with a language of low expressive power (like JavaScript) but whose underlying semantics allow much more to be done.

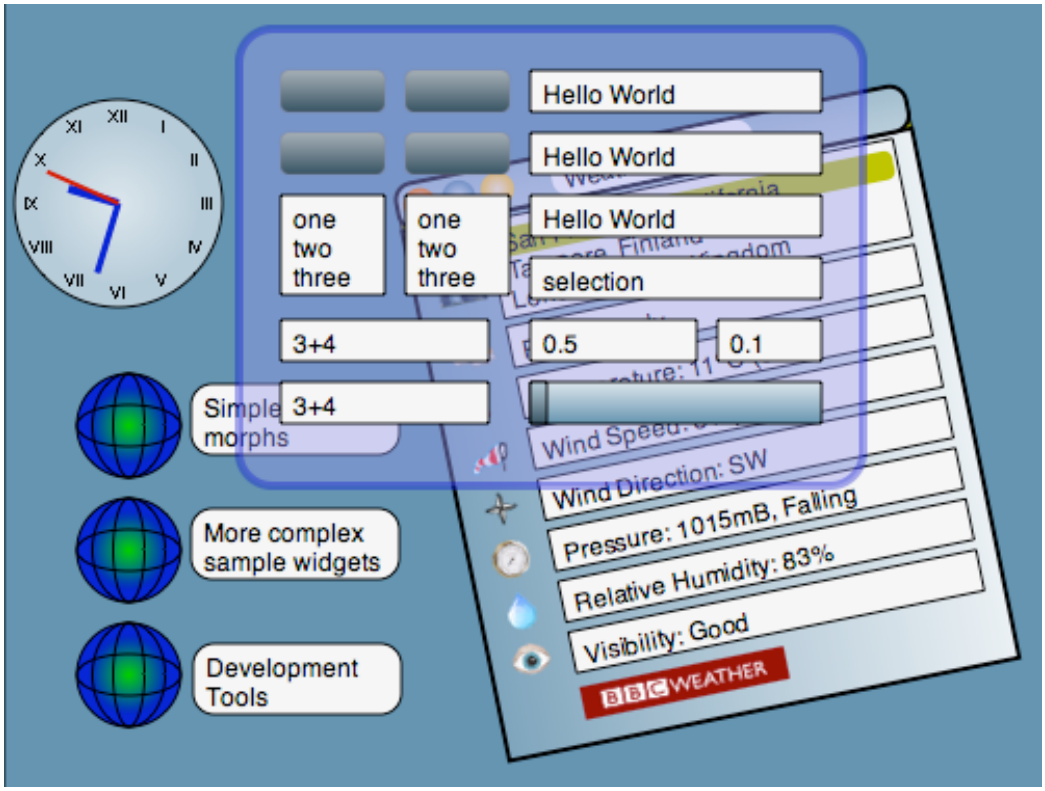
This expressibility has also made it easy to make many improved versions of OMeta over the last year, and allows us to investigate a more integrated notion of top-down non-deterministic programming that unifies ideas from procedural control, pattern-directed invocation, and logic programming.

"Lively Kernel": Morphic Architecture and Development Environment in JavaScript
Principal Researcher: Dan Ingalls (Sun Microsystems)

— a Squeak-like UI, media and development environment. Morphic is the screen level graphical toolset of Squeak Smalltalk and has been used to make many systems over the last decade, including the widely-distributed Etoys authoring environment. Both as an experiment in expressibility, and one that might have some great practical use, this graphical architecture was redesigned and rewritten in JavaScript and made to run both in the browser-based JavaScripts and in our 200-line OMeta-based version.

The practical significance is that this powerful architecture is for the first time usable inside web browsers without having to download plugins. All authoring can be done on-the-fly, no servers are needed, and none of the paraphernalia of the xxMLs need be touched by human hands.

A widget set built from the graphical elements provides a kit for user interface construction and tools are provided for extending and altering the widget set. A modest window-based IDE, built from these widgets, allows users to edit their application programs and even the code of the system itself.



Morphic environment running in a browser-based JavaScript.

What makes the Lively Kernel especially interesting is that it is simply a web page (http://research.sun.com/projects/lively/). When a user visits this page, the kernel loads and begins to run with no need for installation. The user can immediately construct new objects or applications and manipulate the environment.

The Lively Kernel is able to save its creations, even to clone itself, onto web pages. In so doing, it defines a new form of dynamic content on the web, and affords a chance to look at the web in a

new way. Moreover, since it can run in any browser, it promises that wherever there is the Internet, there can be authoring.

In and of itself, the Lively Kernel is a useful artifact. But beyond its utility, its simplicity and completeness make it a practical benchmark of system complexity, and a flexible laboratory for exploring new system models such as secure and distributed processing and radically simple graphics.

The download is about 10,000 lines of code (about 300k bytes), but it requires no installation, and is smaller than many simple pictures and most document downloads from the web. This is both a step forward for the web community, but is also a good metric for expressibility for JavaScript (which as expected is fairly poor[3]). However, it is real, can run on top of our minimal bases, and is a good target for our ongoing efforts to improve expressibility.

A Quick Summary of the Morphic Architecture

The Morphic architecture defines a class of graphical objects, or "morphs", each of which has some or all of the following properties:

A shape, or graphical appearance

A set of submorphs

A coordinate transformation that affects its shape and any submorphs

An event handler for mouse and keyboard events

An editor for changing its shape

A layout manager for laying out its submorphs

A WorldMorph captures the notion of an entire web page; its shape defines its background appearance if any, and its submorphs comprise all the remaining content of the page. A HandMorph is the Morphic generalization of the cursor; it can be used to pick up, move, and deposit other morphs, its shape may change to indicate different cursor states, and it is the source of user events in the architecture. A morphic world may have several hands active at the same time, corresponding to several active users of that page.

Interested readers are referred to the original papers on Morphic [Maloney and Smith 95, Maloney 02], and to the Lively Kernel technical documentation.

The twist in the Lively Kernel is to use JavaScript for all the machinery of activity, thus avoiding the need to install a plugin. Other libraries such as Dojo, Scriptaculous and others operate in the same way without needing installation, but the Lively Kernel goes several steps further. First, since its graphical library is built from the ground up in JavaScript, it sets the stage for a world without HTML and the epicycles that revolve around it. Second, it brings with it a world model in which everything is active and reflective from the beginning, a world of concrete manipulation that is immediately empowering to developers and users alike.

[3] Actually JavaScript is only part of the problem. Of course (and unfortunately) the JS *resources* in the many varieties of web-browsers are not identical and compatibility kits don't track very well. This is why we like to do the entire package ourselves. However, because of MS and browser security problems, many SysAdmins will not allow browser plugins to be downloaded. This is despite the fact that a plugin can be run in a completely confined address space, whose only output could be non-executables (display pane bitmaps, values and processes encoded as strings, etc.). It is quite maddening that there is a simple solution to getting around the many poor choices in browsers, but that one of the poor choices (not to allow downloads into confined processes) disallows the solution!

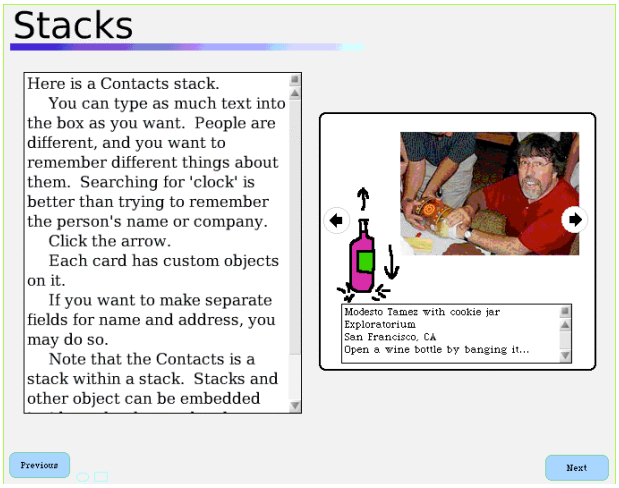
“HyperCard” Model

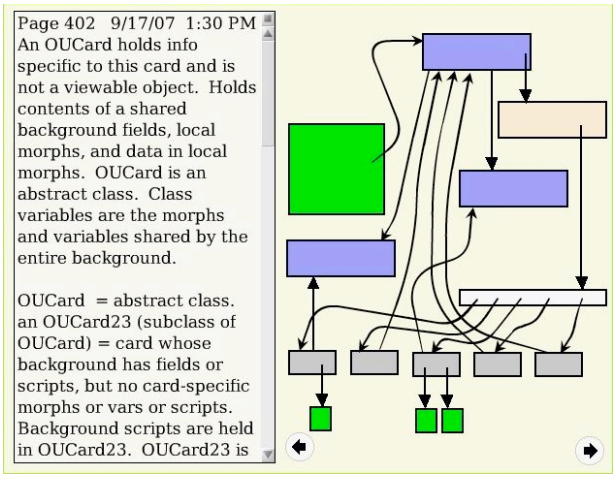
Principal Researcher: Ted Kaehler

Many of our systems over the years have been influenced by HyperCard (Etoys projects and “bookmorphs”, for example), and we feel that a strong reconciliation of a Like-like model with modern object architectures and UIs will point toward one of the central end-user models for the **STEPS** project. (We also feel that the web browsers and many other deployed systems today would have been much better if the developers had been willing to learn from HyperCard.)

We have implemented an HC-II that is very Like-like in spirit but has introduced a few changes and improvements.

- First, it is page-oriented rather than stack (or document) oriented (a stack or doc is just an organization of card/pages, and a card/page can be in any number of stack/docs).
- The HyperCard notion of foreground and background has been generalized in several ways, including identifying a background with the class idea, and the foreground on a card with the instance state of an object.
- The UI is modeless, so there are no separate “button” or “field” or “background”, etc., modes that have to be remembered and exited from. Instead, a variation of the Etoys UI conventions is used.
- The scripting is done in a much more powerful, but still user-friendly scripting language that has the semantic power of Squeak Smalltalk with a scripting syntax adapted from Codeworks.
- “card/pages” and “stack/docs” are “just objects” and thus can be recursively embedded in themselves.
- All visible objects are formed from a universal visible object via shape changing and embedding. Thus, they are all scripted the same way; they can be searched the same way.
- Many of the available “supply objects” go considerably beyond classic HC: for example, “fields” in this version readily flow from container to container and thus provide real DTP layout possibilities for text in documents.
- “card/pages” can be used as web pages via a plugin, and this allows Style-style authoring to be used on the web/internet.

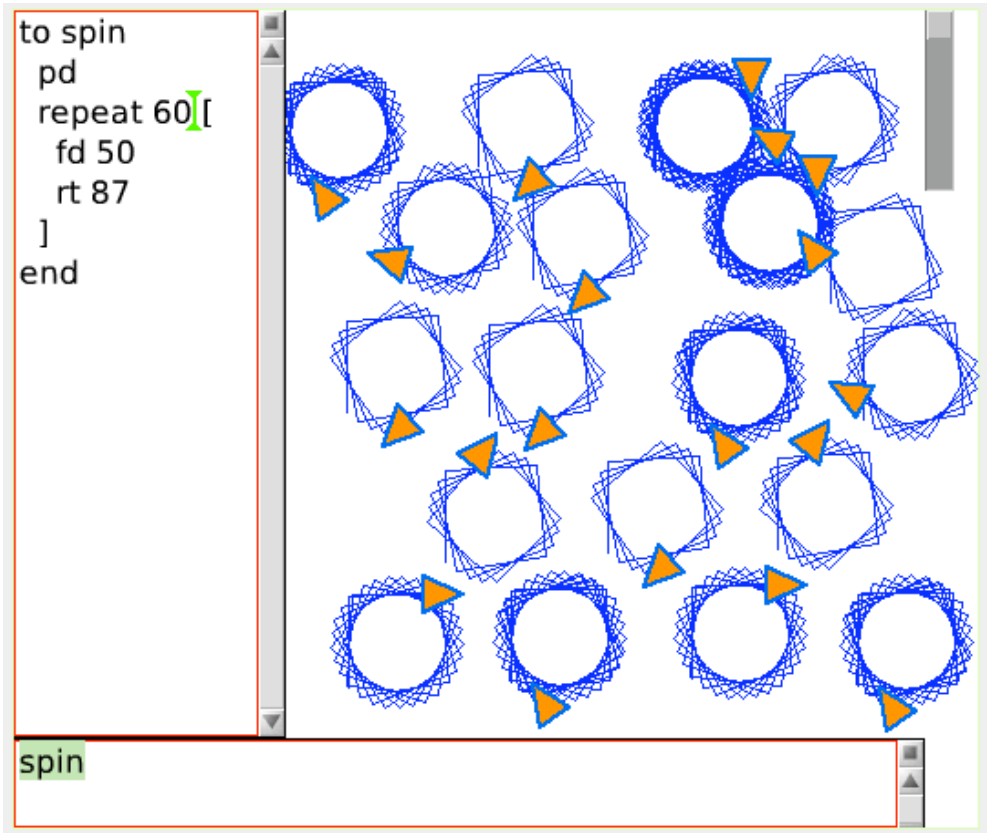




Logo Here, Logo There

Researchers: Alex Warth and Yoshiki Ohshima

— with control structures as Croquet Tasks. An experiment of interest was to use the OMeta sytem to make a parallel Logo system that used the Croquet-style [http://opencroquet.org/index.php/System_Overview] model of "race-free" pseudo-time coordination to allow many thousands of turtles to be made and coordinated in real-time. This required about 260 lines of OMeta and about 290 lines of supporting code to make. This is part of a large number of experiments to make sure that our mechanism for parallelism is rich, safe, and will scale to the entire Internet.

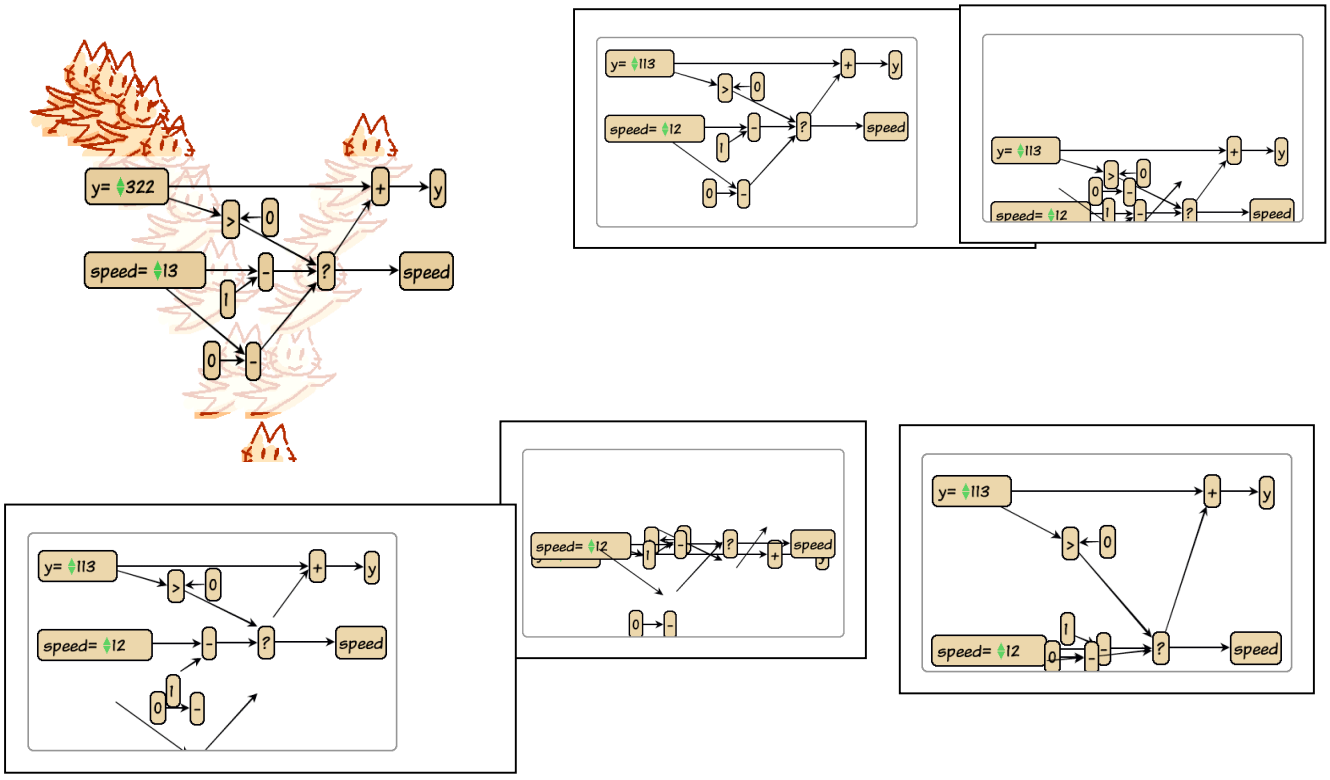


A logo program for spinning turtles is shown on the left. The slider governs the rate of flow between the real-time and pseudo-time. Each turtle executes the program at a different speed.

Visual Dataflow Programming

Principal Researcher: Takashi Yamamiya

— Dataflow is another style of programming we investigated by making a programming system and user interface. This allowed us to explore the variable-free and path-oriented properties of combinators as alternate parallel computation structures for expressions.



The program above is constructed from dataflow expressions. Dataflow is concerned with the dependency relationships between data, not the flow of process. For example, in a physics simulation, a moving bouncing ball is more clearly represented as data relationships between the X axis, the Y axis, and the acceleration of gravity than as imperative statements in an "ordinary" language. Moreover, within our live dataflow system one can modify any data item to see dynamically and immediately its effect on the whole system.

A subset of the "Joy" language is used for the internal representation of dataflow programs. Joy is a language based on Combinatory Logic, useful for program transformation because it has no free variables. We used a simplified Joy based on Linear Logic that requires the programmer to make explicit any duplication or deletion.

Tiny BASIC

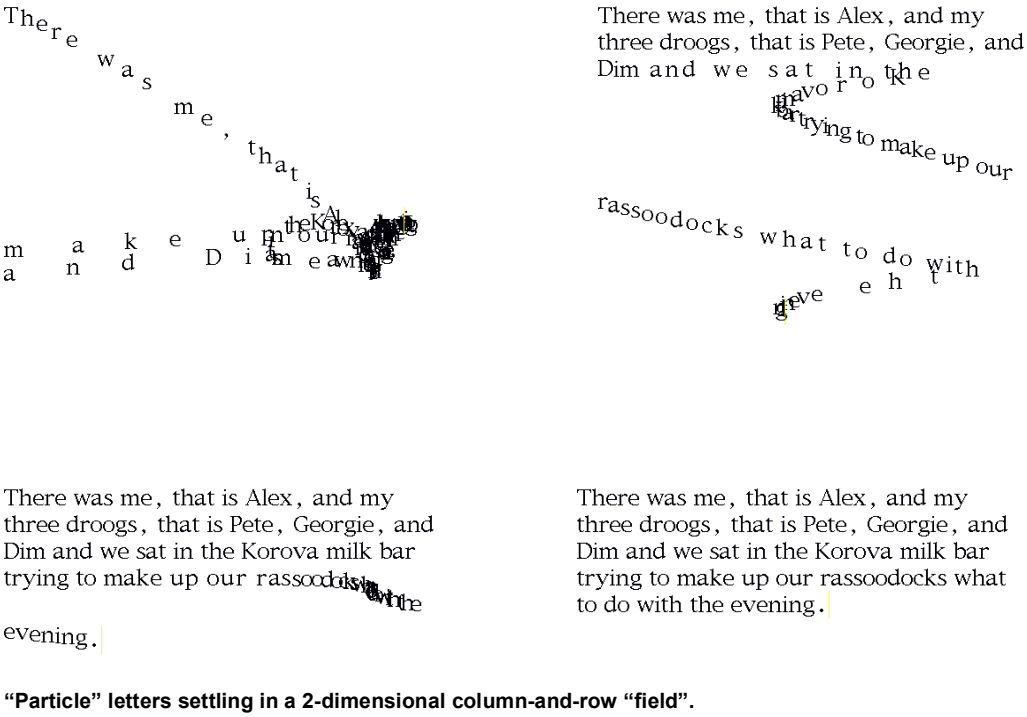
Principal Researcher: Ian Piumarta

— program execution as a side effect of parsing. To set a performance bar we implemented a much more efficient parser than OMeta, based on parsing expression grammars [http://pdos.csail.mit.edu/~baford/packrat/] whose goal was to maximize speed at all costs. The benchmark was an 85-line grammar for Tiny BASIC with another 200 lines of code to model the runtime state (subroutine stack, variables, and so on). Placing appropriate side effects in the grammar as semantic actions turns the parser into a "grammar-directed interpreter". Repeatedly parsing successive source lines to obtain the useful side effects in the semantic actions "runs" a BASIC program at a rate of 250,000 lines per second (parsing about 3 megabytes of BASIC source code per second) on modest hardware. This opened the door to more interesting experiments, such as recognition-based approaches to TCP/IP and other network protocols described elsewhere in this document.

Particles and Fields

Principal Researcher: Alex Warth, using ideas by Ted Kaehler

— the start of a comprehensive real-time constraint system. This is one of the dozen fundamental principles of this project. It is a metaphor that covers multiple active objects (particles) that are coordinated by a wide variety of local and global messaging (fields). A critical goal of our system is to achieve massive scaling of numbers of objects, efficiency of coordination, safety, and kinds of applications that can be easily made using the metaphor. One experiment in massive, coordinated parallelism extended our 200-line JavaScript so that "particle and field" programs could be easily written. A particularly compelling example of this metaphor is a "decentralized" text formatter in just a few lines of code that expresses the constraints (successive placement, line wrapping) that act on each letter and between adjacent pairs of letters. This is a new implementation of the example in the proposal to NSF for this project.



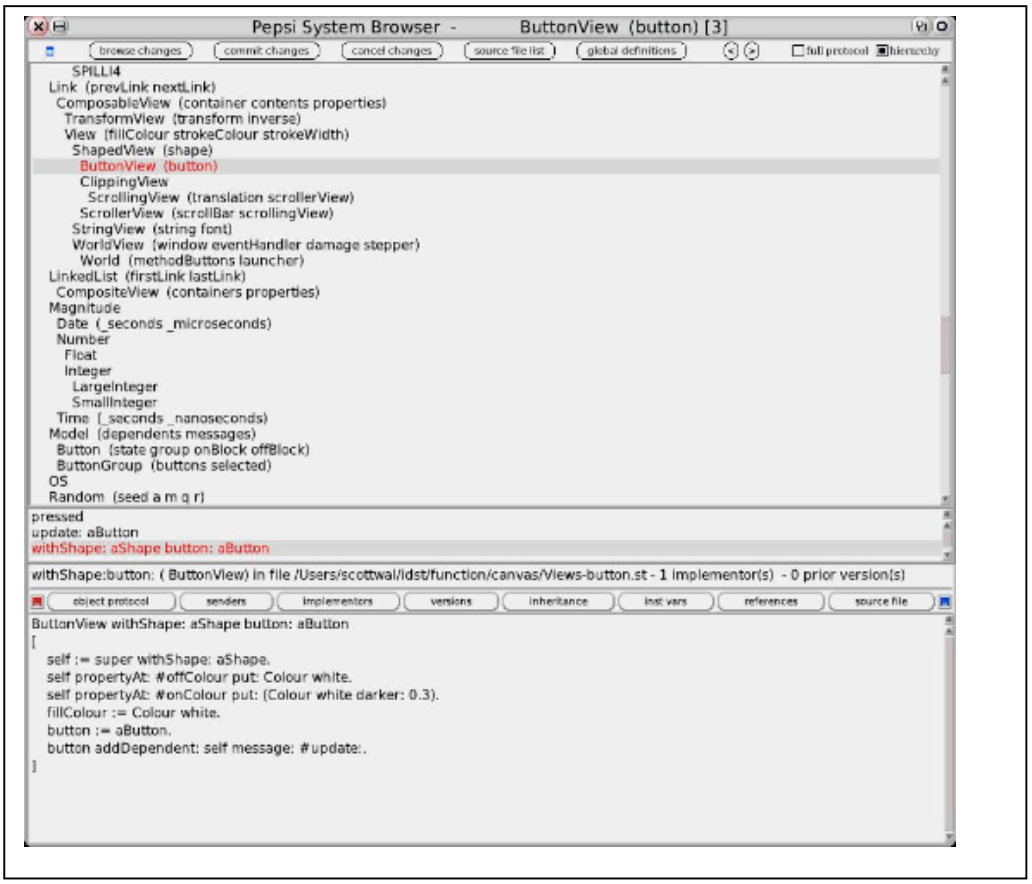
"Particle" letters settling in a 2-dimensional column-and-row "field".

— bringing "live environment" features, traditionally only found in self-hosting dynamic environments such as Smalltalk, into a source-code editing environment compatible with external source files organized into a source tree.

The simple front-end facilities of the **IS** system work with flat files. A "source tree" is an interlocking set of source-code files, linked by compiler directives such as "{ include <filename> }" and "{ import <filename> }" distributed across a number of directories. Many tens of files, in several directories, are present in the source tree from which IS bootstraps itself.

Phase 1 of the IDE is a Squeak-hosted "semi-live environment" for viewing, traversing, querying, and editing the code within a source-tree. The IDE constructs an internal "map" of all the objects, methods, and other structures defined in the tree, and uses that map to provide various views of the system. Each of the many query tools supports a source-code editing environment that includes:

- Ability to view and assess the system at any point along the source-editing development line.
- Ability to "commit' an arbitrary set of changes back to the source tree.
- Ability to browse and edit changes made since the latest "commit".
- Ability to view and edit code both as individual units (e.g., methods) and on a whole-file basis.
- Selective rollback both at the individual method or prototype level, and at the file level.
- Ability to pursue chains of enquiry at any point in the development process.



The above figure shows one of the query tools – a "system browser" showing some of the types and methods within the Universal Polygons source tree. The full suite of query tools is described in **Appendix F**.

A Tiny FPGA Computer

Principal Researcher: Chuck Thacker (Microsoft, Inc.)

— implemented in Field Programmable Gate Arrays. We eventually plan to go "below the metal" to make efficient parallel computational structures for some of the more adventurous solutions to the problems of this project. FPGAs are the microcode of the 21st century, and our colleague Chuck Thacker made a really tiny computer in FPGAs as a first test that can serve as an initial target for our meta-translators. This required about 100 lines of Verilog to create a working example. Our hope is that system hardware, just like system software, can be made accessible to, and programmable by, a non-expert audience.

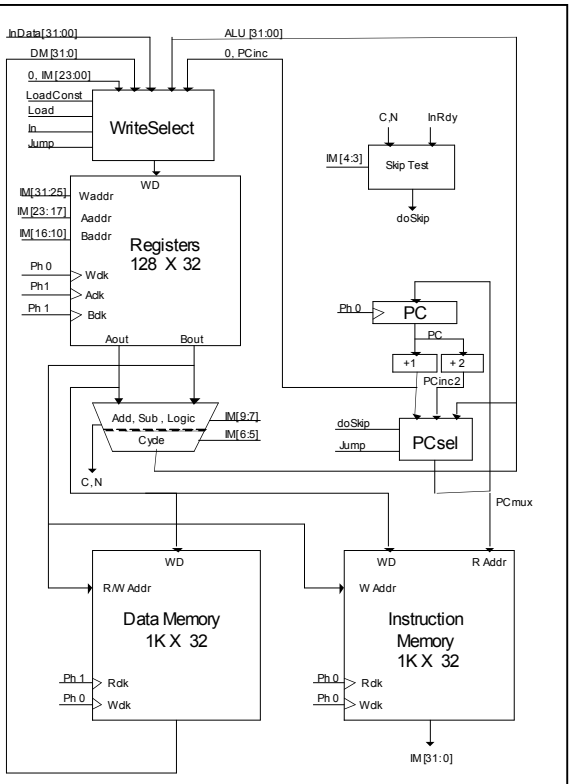


Figure 1: The Tiniest Computer ?

Architectural Issues and Lookaheads

Principal Researchers: Alan Kay and Ian Piumarta

Many of the experiments done this year were manipulative and transformational, at the level of making various kinds of languages with a range of efficiencies, with most of the larger architectural issues still to be taken up in 2008 and 2009. But, since much of the expressiveness and code reduction we are hoping for will come from new architectural paradigms, we have done quite a bit of thinking this year about the issues in preparation.

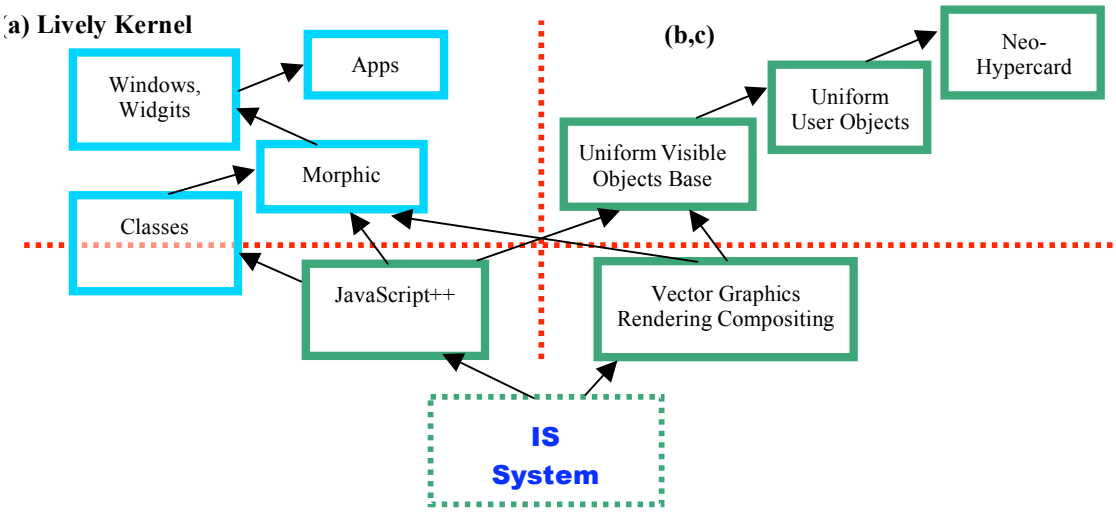
One really interesting architectural experiment this year was the "Tiny TCP/IP" done as a kind of non-deterministic parser and filter. Another was a bare start at "particles and fields" in the redo of "Wandering Letters". Quite a bit more will be done with each of these styles next year.

Several of the larger architectural issues for 2008 are at rather different levels of the system. One is to make a proto system, building only on the foundational primitives for language semantics and graphics that were done in experiment in 2007. At the other end of the system, a timely project would be to take a set of criteria for one of the "real applications" we have to do and make all work in a very compact form. Both of these projects would be advanced by inventing a better uniform object model. Finally, we need to pick one of the several TAMO meta-issues – choosing from "separation of meaning and optimizations", or "semantic publish and subscribe all the way down" – and try to materially advance our understanding. The latter could be thought of as "an Internet all the way down" object architecture using higher level semantic messaging.

A Complete Proto System Bootstrap

The "Lively Kernel" (LK) work (page 22) creates quite a few system structures on top of the browser resident JavaScript and Simple Vector Graphics package. First are classes with inheritance, then a version of the Morphic graphics environment from Squeak, then a 2.5D windows and visible objects compositing and authoring environment, and finally a number of widgets and mini-applications.

A good learning curve would be to (a) replace the browser JavaScript and graphics primitives with the STEPS versions of these, and in parallel to (b) try to make a better implementation environment than Morphic to do the same kinds of objects in LK, then (c) to design a better "application" architecture for larger scale user-object-systems (see next section).



A Complete "Application" Done with Minimal Code

It is hard to think abstractly about computing without real examples that have to be programmed and debugged. This usually brings the castles in the clouds slamming down to earth! Here is another example of an earlier requirement in this report: part of the bootstrapping process involves making something *like* the system before we can make *the* system.

The trick is to take something that is understood well enough to clearly be part of the learning curve and then try to finesse (or force) the next versions out by doing a real example that is outside the current bounds. This has been done very successfully this year in language-language systems. For next year, we need to make a real application that will be *like* the actual applications in this system, *so we can learn how to make them*.

A nice one would be to cover the needs for word processing and desktop publishing with a very powerful facility but with an easy to use and learn user interface. We can get *requirements* by examining existing systems such as MS Word, PageMaker, etc. But we don't want to try to reverse engineer either their exact functionalities or their user interfaces. Instead, we want to come up with equivalent functionalities and UIs.

One of the interesting aspects of this specific problem is that there exist a number of extreme abstract simplifications. For example, if we take a TEX squint at this, we can see that the layout problems mostly fall within TEX's constraint and dynamic programming model [Knuth 84] (and that this model is a subset of the "particles and fields" fundamental principles we posit will make this project possible). But the TEX *user* model is not suitable for most users in personal computing. If we look at the problem of desktop publishing with our end-user hat on, we see that the similarities that allow TEX's style of constraint solving to be done, can also be exploited in a much simpler UI design in which most things can be approached through a single set of UI principles.

A concomitant question has to do with user-oriented features and their interactions with a uniform UI design. In theory, a DTP type application should "almost not exist", in the sense that there is no end of the kinds of visible objects that we would like to make, organize, composite and publish. This brings up the important issue of whether there could be a mostly automatic matching of a *style* for UI interaction and recognizable polymorphisms in the uniform objects (probably). Then the question turns to how much work and how messy will it be to try to handle necessary idiosyncrasies by exception.

Multiple Perspective Object and Viewing Model

In the middle of all of this is the need for a very simple uniform object model, but that paradoxically permits more flexibility than most of the classic object models of the past. A very interesting *multiple perspectives* object model was devised in the late 70s by Danny Bobrow and Ira Goldstein as a significant extension to Xerox PARC Smalltalk [cf., the PIE papers] that was inspired in part by one of the PARC AI languages. The idea is that in the real world we don't just look at an "object" (such as a person) as a flat collection of all its properties and behaviors. Instead, we choose *perspectives* such as "physical object", "mammal", "father", "son", "manager", "employee", etc. Each one of these perspectives will have characteristic properties and behaviors. Some of these will be quite separate from all the others, and changes in some will effect changes in others.

An interesting problem with multiple perspectives (which also appears in the somewhat parallel devices of multiple inheritance) is whether the programmer has to *qualify* (to state somehow) the perspective of the object to which messages are going to be directed. This is a pain, and the oppo-

site ploy of having the system try to figure out what is desired (for example, because there is only one method with a particular name) is fraught with difficulties (contexts and change, new methods can be added, etc.). One experiment we will try this year is the idea of having a binding to an object also be a *view* of it that automatically provides a local nomenclature without ambiguity if the match succeeds. We are not fans of traditional typing systems (too much extra work for too little return), but the notion of *expectations at the meaning level* is important and useful, and this starts to motivate a kind of typing that we think can finally pay its way.

We can see that a *perspective* is rather like a *graphical view*, in that it presents a subset of the object organized for best understanding and use. A goal for 2008 is to actually unify these ideas and assess the benefits.

Internet All The Way Down

Part of the solution to "an Internet all the way down" has interesting conflicts with today's hardware and we are curious to see just how far this can be taken without having to posit a different (but pretty minimal) set of machinery to help out. Basically, we would like to make a distributed object system that is (a) so protected that it can allow completely foreign objects to be brought in from elsewhere without causing harm, and (b) so efficient that a much higher level of abstract communication can be used between objects (perhaps an advanced form of "publish and subscribe" or "forward inferencing using knowledge patterns").

Besides encapsulation, the key to a real object system lies in the messaging system design and realization. The actual language in a real object system **is** the messages that are sent around. Using the metaphor of "Internet All The Way Down" we can see that it would have been really nice if the folks who created **HTTP** had thought more about what they were actually doing. They could have created an excellent extensible language that would make both the web and the underlying Internet much more simple and useful.

On the other hand, since we don't really know yet what we are doing here in this part of the project (some of the NSF reviewers hated this!), we want to avoid making language commitments we don't want to keep. So we have to think about "meta-messaging" and what the mechanisms might be for this, especially given that we would like to have however our messaging system turns out, to work all the way down to the lowest primitive objects.

We don't think that *imperatives* scale well, even if they are removed from the simple (and bad) "getting and setting" uses of objects as abstract data types. *Names* don't scale well either. So, in part, we are looking for *ways to get things to happen* without (a) having to know who or how they are going to be done, (b) without having to tell any object to "just go do this", and (c) without turning what is said into a big mystery that requires lots of cleverness to set up.

Part of this has been done pretty successfully in the past – but without big pressure on scaling – in several of the better expert systems (for example, the Automated Reasoning Tool (ART) by Inference Corp). And it has been done for billions of years very successfully in cellular biology. The basic idea is to design a system so that each module knows what, when and why it needs to do its thing, and the doing of its own thing, a very large part of the time, will be a positive contribution to the goals and dynamics of the whole system. We can see that this overlaps to some extent with "publish and subscribe" architectures, and also with one way to use Gelernter's Linda coordination system.

The idea here is to have objects that *notice* what is going on around them, and can produce internal changes *only of themselves* as appropriately as possible. The part of what is going on around an object that it is interested in can be thought of as *what it needs*, and this can provide a basis for automatically setting up mechanisms to detect these needs and provide triggering events when

appropriate. Some years ago we did a children's version of this called "Playground" that worked well on a small scale and was very expressive.

The metaphor of *noticing and reacting* has interesting parallels with the issues of finding and using external resources. A perhaps too simple example: suppose we need a super fast sine routine and we don't want to go to the effort of doing the necessary work (let this be a stand-in for any such external resource). We can draw from the entire Internet, but how do we know that we have found the external resource that will help us? Type signatures do not have much power here. Knowing a name for such a resource (sine) might not help (this function is called "circle 4" in APL). We really want "semantic types" that can match desired meanings with promised meanings. How can a matcher find and bind?

(The "Internet" as a place to find resources is meant concretely, but also as an analogy to "really large collections of resources" such as are found in the software at any large computing intensive business, government, university, etc. With lines of code running in the hundreds of millions – literally – and possibly millions of modules, the need for scalable matching and binding is manifest.)

One of our preliminary answers to this is that it is time for modules to include a working model of their external environment and to be debugged by using this model. The "working model" is at the same level of semantics as the "meanings" in the STEPS system. So, what is a model for sine? We think it is the simplest way to produce an accurate functional result "quickly enough" but perhaps not blindingly fast. The result is not just a number, but also a notion of acceptable tolerance. A matcher will have to do something very similar to what has to be done to test optimizations against meanings, which in this case probably involves sampling and close but non-exact matching. This reminds us very much of the MO of Lenat's AM and Eurisko systems, which tried to gather evidence and make stub functional relationships as it was finding stronger versions of what relationships meant.

In any case, the working model of environment of a module should be of sufficient richness to allow the module to be debugged, to allow it to find more suitable modules to provide the external resources needed, and to function as both checks and defaults.

A pretty good metaphorical term for *noticing modules – monads* – has already been hijacked from Leibniz by category theorists, was used more as Leibniz intended in early ideas about objects, and then got colonized by functional language folks for their own somewhat different uses.

Quite a bit of this style scales, but it is not so clear if the mechanisms do. We know from the forward inferencing work at CMU and with ART that quite a bit can be done to compile the detectors and triggers into very efficient structures. We think that one part of an FPGA architecture could be fashioned to do this even faster and with larger scaling. It remains to be seen how far this can actually go on the standard HW of today.

This experiment to come is somewhat reminiscent of early object days when it was realized that good things would happen if everything could be an object intercommunicating via messaging – even the number "3". Quite a bit of work was required to make "3" behave, but then the whole system got very simple and easy to make things with. A major question is whether this style can stay pure enough to make a big difference in expressibility in the large (because any event driven style can be used to mimic direct subroutine calls and thus subvert the more scalable "blind behavior" architecture). This is very similar to the problem of the subversion of pure object styles by using objects to simulate data with abstract data types.

Comments

There are overlaps of goals and techniques in some of these experiments. For example, the "Universal Polygons" experiment and the "Morphic Architecture" experiment both produce graphic, media rich, end-user authoring environments. The latter uses ideas that have worked in the past but put in a new bottle, while the former completely starts from scratch with an extremely simple and wide-ranging graphics model that – while needing much more work – is more like we expect the final system to turn out. As another example, we currently use several meta-pattern-matching systems in different parts of our fundamental mechanisms.

As already mentioned, our "meta parser" is written in itself (this is an essential characteristic of our entire system). Languages built with it (such as the JavaScript described above) can be trivially extended to include semantics whose domain is the "meta expressions" describing that language. We can expose (make reflective) these language implementations in themselves. If we use the underlying dynamic late-binding system, then we can make deep changes to these languages "on-the-fly" by changing their own syntax/semantic functions from within themselves. This technique can reach all the way down to the IS "quantum mechanics" (machine instructions, or even programmable hardware below them) that makes everything happen. This kind of flexibility is almost a guaranteed characteristic of any system that is dynamically changeable and meta-described entirely in itself.

It should be emphasized that our eventual ultra-high-level "miracle in the middle" will have to be *qualitatively better than languages that are already qualitatively better than JavaScript*. On the other hand, the JavaScript example does have the merit of dealing with a widely known (and used) dynamic language so that comparisons made are much more easily understandable by the outside world. JavaScript lacks so many features that it is illuminating to see what it takes both to make the base system itself and to make many powerful extensions (and some actual reformulations of weaknesses in it) that help us to think about what our eventual system should be like.

Opportunities for Training and Development

Graduate students at UCLA and UCSD are actively participating in the design and development activities of the project.

- Alessandro Warth gained experience with our COLA architecture and built a prototype meta-parser that unifies parsing and structural pattern matching into a single activity. Alex is a Viewpoints full-time researcher as well as a Ph.D. candidate at UCLA.

- Daniel Amelang gained valuable experience with the COLA architecture and used it to develop a framework for graphical compositing in which the "pipeline" is instantiated on-demand and "just-in-time". He also developed the elegant, mathematical model of 2D rendering that we are using in Gezira. Dan will submit a M.Sc. thesis (UCSD) later this year describing some of this work. Dan is a full-time Viewpoints researcher.

- Narichika Hamaguchi was a visiting researcher from NHK (the Japan Broadcasting Corporation). He has gained experience with several of our prototypical tools and is combining that with his considerable knowledge in the area of TVML to create new authoring tools for interactive media development. These tools will likely feature as part of the end-user "exploratorium" experience within our system.

Outreach

Ian Piumarta was an invited lecturer for Stanford University's Computer Systems Colloquium series (EE308, 14 February 2007) at which he presented many of the ideas underpinning the "power supply" end of this project.

Viewpoints Research Institute organizes a three-day "Learning Lab" retreat in August every year where we exchange research directions and experience with several tens of our colleagues from academia and industry. Three of the presentations this year concerned this project.

Daniel Amelang formally presented our Jitblt and Gezira work at the X.Org Developers' Summit (October 2007, Cambridge, UK).

We host a mailing list for discussions about our publicly available software platform. The list now has more than 200 subscribers. http://vpri.org/mailman/listinfo/fonc

Service

Alan Kay serves on the NSF CISE Advisory Committee, the ACM Turing Award Committee, and the Computer History Museum Fellow Awards Committee.

References

Acknowledgements

The images featured in the Jitblt example (page 13) are both released under the Creative Commons license Attribution-Noncommercial-Share Alike 2.0. They are "Catch the Sun" by Evan Leeson and "Arena y Viento" by Pablo Arroyo. As required by this license, this composition is also released under the CC Attribution-Noncommercial-Share Alike 2.0 license.

Refereed Publications

Alessandro Warth and Ian Piumarta, *OMeta: an Object-Oriented Language for Pattern Matching*, ACM SIGPLAN Dynamic Language Symposium, 2007, Montréal, Canada.
http://www.cs.ucla.edu/~awarth/papers/dls07.pdf

Technical Reports, Research Notes, Memos, Online Demonstrations

Ted Kaehler, *Bare Blocks with a Thin Object Table: An Object Memory for COLA*. http://vpri.org/pdf/BareBlocks_RM-2007-005-a.pdf

Ian Piumarta, Efficient Sideways Composition in COLAs via 'Lieberman' Prototypes. http://vpri.org/pdf/lieberman_proto_RM-2007-002-a.pdf

Ian Piumarta and Alessandro Warth, *Open, Reusable Object Models*. http://vpri.org/pdf/obj_mod_RN-2006-003-a.pdf

Ian Piumarta, Accessible Language-Based Environments of Recursive Theories (a white paper advocating widespread unreasonable behavior). http://vpri.org/pdf/colas_wp_RN-2006-001-a.pdf

Dan Ingalls, et al., A Lively Kernel Implemented in a Web Browser (live demo). http://research.sun.com/projects/lively/

Dan Ingalls, et al., The Sun Labs Lively Kernel, A Technical Overview. http://research.sun.com/projects/lively/LivelyKernel-TechnicalOverview.pdf

Alessandro Warth, *OMeta Demonstration Pages*. http://www.cs.ucla.edu/%7Eawarth/ometa/ometa-js/

In Preparation: Theses, Technical Reports, Research Notes and Memos

Dan Amelang, *Highly Mathematical Efficient and Compact Graphics Primitives* (MS Thesis UCSD).

Dan Ingalls, et al., *The Lively Kernel: A self-supporting system for web programming,* to appear in the proceedings ECOOP 2008.

Ian Piumarta, *A Minimal Architecture for TCP/IP*.

Ted Kaehler, *A New Approach to "HyperCard."*

Scott Wallace, *A Development System for IS*.

Yoshiki Ohshima, *Experiments in Control Primitives and Scheduling.*

Web/Internet Sites

Organization home page: http://vpri.org/

Project home page: http://vpri.org/html/work/ifnct.htm

Core software distribution page: http://piumarta.com/software/cola/

Core software mailing list: http://vpri.org/mailman/listinfo/fonc

OMeta demo pages: http://www.cs.ucla.edu/%7Eawarth/ometa/ometa-js/

Morphic in JavaScript home page: http://research.sun.com/projects/lively/

References for Non-Viewpoints Work Cited

Baker, Henry G., Linear Logic and Permutation Stacks – The Forth Shall Be First, 1993.

Barton, R.S., "A new approach to the functional design of a digital computer", Proceedings of the WJCC, 1961.

Carriero, Nicholas and Gelernter, David, "Linda in Context", *Communications of the ACM* 32(4): pp. 444-458, 1989.

Fenton, Jay and Beck, K., Playground: An object-oriented simulation system with agent rules for children of all ages, ACM Sigplan Notices, OOPSLA 1989, pp. 123-137.

Ford, B., Packrat parsing: simple, powerful, lazy, linear time, functional pearl. In *ICFP '02: Proceedings of the Seventh ACM SIGPLAN international conference on Functional programming*, pp. 36–47, New York, NY, USA, 2002.

Fowler, Martin, Language Workbenches: The Killer-App for Domain Specific Languages? 2005. http://martinfowler.com/articles/languageWorkbench.html

Garland, Stephen J., Guttag, John V., Horning, James J.: An Overview of Larch. Functional Programming, Concurrency, Simulation and Automated Reasoning, pp. 329-348, 1993.

Goguen, J.A., Winker, T., Meseguer, J., Futatsugi, K., and Jouannuad, J.P., Introducing OBJ, October 1993.

Ingalls, D., Kaehler, T., Maloney, J., Wallace, S. and Kay, Alan, "Back to the Future – The Story of Squeak, A Practical Smalltalk Written in Itself", Proceedings of the ACM, OOPSLA, October 1997. SIGPLAN Notices 32(10), October 1997.

Irons, Edgar T., IMP, Communications of the ACM, Jan. 1970.

Irons, Edgar T., The Syntax Directed Compiler, *Communications of the ACM*, ca. 1960.

Knuth, Donald E., "The TeXbook" (Computers and Typesetting, Volume A). Addison-Wesley, 1984. ISBN 0-201-13448-9.

Lenat, Douglas B., "EURISKO: A Program That Learns New Heuristics and Domain Concepts," Vol. 21, Artificial Intelligence Journal, 1983.

Maloney, J. and Smith, R., "Directness and Liveness in the Morphic User Interface Construction Environment," *Proceedings UIST '95*, pp. 21-28, November 1995.

Maloney, John, "An Introduction to Morphic: The Squeak User Interface Framework", *Squeak Open Personal Computing and Multimedia*, ed., Mark Guzdial and Kim Rose, Prentice Hall, NJ, 2002.

Shorre, D.V., META II a syntax-oriented compiler writing language, *Proceedings of the 1964 19th ACM National Conference*, pp. 41.301-41.3011, 1964.

Simonyi, C., Christerson, M., and Clifford, S., *Intentional Software*, OOPSLA 2006, October 22-26, Portland, OR, ACM, 1-59593-348-4/06/0010.

von Thun, Manfred, A short overview Joy, 1994.
http://www.latrobe.edu.au/philosophy/phimvt/joy.html

Wing, Jeanette, N., A two-tiered approach to specifying programs. Technical Report LCS/TR{299} MIT, May 1983. Ph.D. Thesis, Depts. of EE and Computer Science.

Appendix A: Extended Example: An OMeta Translator from Logo to JavaScript
(by Alex Warth)

```
Turtle rt: n [self turnBy: n].
Turtle lt: n [self turnBy: n negated].
Turtle fd: n [self forwardBy: n].
Turtle bk: n [self forwardBy: n negated].
Turtle pu    [self setPenDown: false].
Turtle pd    [self setPenDown: true].

ometa LogoTranslator : Parser {
  name         = spaces firstAndRest(#letter, #letterOrDigit):xs -> (xs squish mash).
  cmdName      = name:n
                   ?(n ~= 'to') ?(n ~= 'end') ?(n ~= 'output')   -> n.
  number       = spaces digit+:ds                                -> (ds mash).
  arg          = ":" name.
  cmds         = cmd*:xs                                         -> (xs join: ';').
  block        = "[" cmds:xs "]"                                 -> ('(function() {', xs, '})').
  primExpr     = arg | number | block
               | "(" (expr | cmd):x ")"                          -> x.
  mulExpr      = mulExpr:x "*" primExpr:y                        -> (x, '*', y)
               | mulExpr:x "/" primExpr:y                        -> (x, '/', y)
               | primExpr.
  addExpr      = addExpr:x "+" mulExpr:y                         -> (x, '+', y)
               | addExpr:x "-" mulExpr:y                         -> (x, '-', y)
               | mulExpr.
  relExpr      = addExpr:x "<"  addExpr:y                        -> (x, '<', y)
               | addExpr:x "<=" addExpr:y                        -> (x, '<=', y)
               | addExpr:x ">"  addExpr:y                        -> (x, '>', y)
               | addExpr:x ">=" addExpr:y                        -> (x, '>=', y)
               | addExpr.
  expr         = relExpr.
  cmd          = "output" expr:x                                 -> ('return ', x)
               | cmdName:n expr*:args                            -> ('$self.performwithArguments("', n, '", [',
                                                                      (args join: ','),
                                                                     '])').
  decl         = "to" cmdName:n arg*:args cmds:body  "end"       -> ('$self.', n, ' = ',
                                                                      'function(', (args join: ','), ') {',
                                                                        body,
                                                                      '}').
  topLevelCmd  = decl | cmd.
  topLevelCmds = topLevelCmd*:xs spaces end                      -> ('(function() { var $self = this;
                                                                            ', (xs join: ';'), '})').
                                                                 }.
```

Appendix B: Extended Example: An OMeta Translator from Prolog to JavaScript
(by Alex Warth)

Prolog has a very simple syntax, needing 9 lines of OMeta for translation into JavaScript.

```
  Ometa PrologTranslator : Parser {
  variable   =  spaces firstAndRest(#upper, #letterOrDigit):name     -> (Var new: name mash).
  symbol     =  spaces firstAndRest(#lower, #letterOrDigit):name     -> (Sym new: name mash).
  clause     =  symbol:sym "(" listOf(#expr, ','):args ")"           -> (Clause new: sym : args).
  expr       =  clause | variable | symbol.
  clauses    =  listOf(#clause, ',').
  rule       =  clause:head ":-" clauses:body "."                    -> (Rule new: head : body)
              | clause:head                  "."                    -> (Rule new: head : {}).
  rules      =  rule*:rs spaces end                                  -> rs.
  query      =  clause:c spaces end                                  -> c.
  }.
```

However, Prolog is rather different from JavaScript, so we write some JavaScript code to provide the meanings for Prolog's searching and matching semantics. Less than 80 lines of code are required for this support.

```
function Sym(name) { this.name = name }
Sym.prototype.rename  = function(nm)  { return this }
Sym.prototype.rewrite = function(env) { return this }
Sym.prototype.toAnswerString = function() { return this.name }

function Var(name) { this.name = name }
Var.prototype.rename  = function(nm)  { return new Var(this.name + nm) }
Var.prototype.rewrite = function(env) { return env[this.name] ? env[this.name] : this }
Var.prototype.toAnswerString = function() { return this.name }

function Clause(sym, args) { this.sym  = sym; this.args = args }
Clause.prototype.rename   =  function(nm)   {  return  new  Clause(this.sym,  this.args.map(function(x)
{ return x.rename(nm) })) }
Clause.prototype.rewrite  =  function(env)  {  return  new  Clause(this.sym,  this.args.map(function(x)
{ return x.rewrite(env) })) }
Clause.prototype.toAnswerString = function() {
  return     this.sym.toAnswerString()     +     "("     +     this.args.map(function(x)     {     return
x.toAnswerString() }).join(", ") + ")"
}

Array.prototype.rename  = function(n)   { return this.map(function(x) { return x.rename(n) }) }
Array.prototype.rewrite = function(env) { return this.map(function(x) { return x.rewrite(env) }) }
Array.prototype.toAnswerString    =    function()    {    return    this.map(function(x)    {    return
x.toAnswerString() }).join(", ") }

function Rule(head, clauses) { this.head = head; this.clauses = clauses }
Rule.prototype.rename       =    function(n)           {    return    new    Rule(this.head.rename(n),
this.clauses.rename(n)) }

function addBinding(env, name, value) {
  var subst = {}
  subst[name] = value
  for (var n in env)
    if (env.hasOwnProperty(n))
      env[n] = env[n].rewrite(subst)
  env[name] = value
}
function assert(cond) { if (!cond) throw "unification failed" }

Sym.prototype.unify = function(that, env) {
  if (that instanceof Sym)
    assert(this.name == that.name)
  else {
    assert(that instanceof Var)
    if (env[that.name])
      this.unify(env[that.name], env)
    else
      addBinding(env, that.name, this.rewrite(env))
  }
}
Var.prototype.unify = function(that, env) {
```

```
  if (env[this.name])
    env[this.name].unify(that, env)
  else
    addBinding(env, this.name, that.rewrite(env))
}
Clause.prototype.unify = function(that, env) {
  if (that instanceof Clause) {
    assert(that.args.length == this.args.length)
    this.sym.unify(that.sym, env)
    for (var idx = 0; idx < this.args.length; idx++)
      this.args[idx].unify(that.args[idx], env)
  }
  else
    that.unify(this, env)
}

function State(query, goals) { this.query = query; this.goals = goals }

function nextSolution(nameMangler, rules, stateStack) {
  while (true) {
    if (stateStack.length == 0)
      return false
    var state = stateStack.pop(),
        query = state.query,
        goals = state.goals
    if (goals.length == 0)
      return !window.confirm(query.toAnswerString())
    var goal = goals.pop()
    for (var idx = rules.length - 1; idx >= 0; idx--) {
      var rule = rules[idx].rename(nameMangler), env
      try { rule.head.unify(goal, env = {}) }
      catch (e) { continue }
      var newQuery = query.rewrite(env),
          newGoals = goals.rewrite(env),
          newBody  = rule.clauses.rewrite(env)
      for (var idx2 = newBody.length - 1; idx2 >= 0; idx2--)
        newGoals.push(newBody[idx2])
      stateStack.push(new State(newQuery, newGoals))
    }
  }
}

function solve(query, rules) {
  var stateStack = [new State(query, [query])], n = 0
  while (nextSolution(n++, rules, stateStack)) {}
  alert("no more solutions")
}
```

Appendix C: Extended Example: Toylog: An English Language Prolog
(by Alex Warth)

This example uses a different OMeta front-end syntax translation.

```
ometa ToylogTranslator : Parser {
  rule                    = clause:head "if" conj:body "."              -> (Rule new: head : body)
                          | clause:head                "."              -> (Rule new: head : {}).
  clause                  = iClause('', {}, false).
  iClause :rel :args :not = ( "not"  !(not := not not)
                            | var:x  !(args add: x)
                            | word:x !(rel := rel,
                                       (rel size > 0
                                         ifTrue:  [x capitalized]
                                         ifFalse: [x]))
                            | thing:x !(args add: x)
                            )+
                            !(rel := Clause new: (Sym new: rel) : args)  -> (not
                                                                                ifTrue:
                                                                                        [Clause new:
                                                                                          (Sym new: 'not') :
                                                                                             {rel}]
                                                                                ifFalse: [rel]).
  var                     = ( ("who" | "what" | "when"):ans
                            | spaces lower+:xs !(xs join: ''):ans
                                ?(ans size = 1 and: [(ans at: 0) ~= $a])
                            )                                           -> (Var new: ans).
  wordPart                = spaces lower+:xs                            -> (xs join: '').
  word                    = wordPart:xs $' wordPart:ys                  -> (xs, ys capitalized)
                          | ~("if" | "not" | "and") wordPart
                          | $' wordPart:xs                              -> (xs capitalized).
  thing                   = spaces firstAndRest(#upper, #lower):xs      -> (Sym new: (xs join: '')).
  conj                    = listOf(#clause, 'and').
  rules                   = rule*:rs spaces end                         -> rs.
  query                   = clause:c spaces end                         -> c.
}.
```

Typical Toylog facts and definitions

```
Abe is Homer's father.
Homer is Lisa's father.
Homer is Bart's father.
x is y's grandfather if x is z's father and z is y's father.
```

Typical Toylog query

```
Abe is y’s grandfather?
```

Appendix D: Extended Example: An OMeta Translator from OMeta to JavaScript (by Alex Warth)

This is the OMeta translator that defines OMeta and translates its definitions to JavaScript code. The grammar does not generate JavaScript directly; instead it generates an intermediate abstract syntax tree (AST) that can be further analyzed and manipulated by subsequent OMeta grammars.

```
ometa NewOMetaParser : Parser {
  tsName       = listOf(#letter, #letterOrDigit):xs               -> (xs mash).
  name         = spaces tsName.
  tsString     = $' (~$' char)*:xs $'                             -> (xs mash).
  character    = $$ char:x                                        -> {#App. #exactly. x printString}.
  characters   = $` $` (~($' $') char)*:xs $' $'                  -> {#App. #seq.     xs mash printString}.
  sCharacters  = $"    (~$"      char)*:xs $"                     -> {#App. #token.   xs mash printString}.
  string       = ($# tsName | $# tsString | tsString):s           -> {#App. #exactly. s}.
  number       = ('-' | -> ''):sign digit+:ds                     -> {#App. #exactly. sign, ds mash}.
  keyword :xs  = token(xs) ~letterOrDigit                         -> xs.
  hostExpr     = foreign(self.SqueakParser, #unit):x              -> (x squish mash).
  args         = "(" listOf(#hostExpr, ','):xs ")"                -> xs
               |                                                  -> {}.
  application  = name:rule args:as                                -> ({#App. rule}, as).
  semAction    = ("!" | "->") hostExpr:x                          -> {#Act. x}.
  semPred      = "?" hostExpr:x                                   -> {#Pred. x}.
  expr         = listOf(#expr4, '|'):xs                           -> ({#Or}, xs).
  expr4        = expr3*:xs                                        -> ({#And}, xs).
  optIter :x   = "*"                                              -> {#Many. x}
               | "+"                                              -> {#Many1. x}
               |                                                  -> x.
  expr3        = expr2:x optIter(x):x (":" name:n                 -> {#Set. n. x}
                                      |                           -> x
                                      )
               | ":" name:n                                       -> {#Set. n. {#App. #anything}}.
  expr2        = "~" expr2:x                                      -> {#Not. x}
               | "&" expr1:x                                      -> {#Lookahead. x}
               | expr1.
  expr1        = application | semAction | semPred
                 | ( keyword('undefined') | keyword('nil')
                   | keyword('true')      | keyword('false') ):x   -> {#App. #exactly. x}
                 | spaces ( character | characters | sCharacters
                          | string    | number                  )
                 | "{" expr:x "}"                                 -> {#Form. x}
                 | "(" expr:x ")"                                 -> x.
  rule         = &name:n rulePart(n):x (";" rulePart(n))*:xs "."  -> {#Rule. n. {#Or. x}, xs}.
  rulePart :rn = name:n ?(n = rn) expr4:b1 ( "=" expr:b2           -> {#And. b1. b2}
                                           |                       -> b1
                                           ).
  grammar      = keyword('ometa') name:n
                   ( ":" name
                   | -> 'OMeta' ):sn
                   "{" rule*:rs "}"                               -> ({#Grammar. n. sn}, rs).
}.
```

The AST structures produced by the above translator are converted into JavaScript by another OMeta translator, shown below. (Separating the abstract syntax makes the underlying semantics that have to be implemented clearer.)

The OMeta/JS Code Generator

```
" By dispatching on the head of a list, the following idiom allows
  translators to avoid checking for different kinds of lists in order. "
ometa Translator {
  trans = {:x apply(x):answer}             -> answer.
}.

ometa NewOMetaCompiler : Translator {
  App 'super' anything+:args               -> (self.sName, '._superApplyWithArgs($self,', (args join: ','), ')');
  App  :rule  anything+:args               -> ('$self._applyWithArgs("', rule, '", ', (args join: ', '), ')');
  App  :rule -> ('$self._apply("', rule, '")').
  Act  :expr -> expr.
  Pred :expr -> ('$self._pred(', expr, ')').
  Or   transFn*:xs                         -> ('$self._or(', (xs join: ','), ')').
  And  notLast(#trans)*:xs trans:y
       !(xs addLast: 'return ', y)         -> ('(function(){', (xs join: ';'), '})()');
  And  -> '(function(){})'.
  Many  trans:x                            -> ('$self._many(function(){return ', x, '})').
  Many1 trans:x                            -> ('$self._many1(function(){return ', x, '})').
  Set   :n trans:v                         -> (n, '=', v).
  Not   trans:x                            -> ('$self._not(function(){return ', x, '})').
  Lookahead trans:x                        -> ('$self._lookahead(function(){return ', x, '})').
  Form  trans:x                            -> ('$self._form(function(){return ', x, '})').
  Rule  :name locals:ls trans:body         -> (self.gName, '[''', name, ''']=function() {', ls, 'return ',
                                                 body, '};').

  Grammar :n :s !(self at: #gName put: n;
                       at: #sName put: s)
          trans*:rules                     -> (self.gName, '=', self.sName, '.delegated();', (rules join: ''),
                                               self.gName, '.prototype=', self.gName, ';').

  locals  = {anything*:vs}                 -> ('var ', (vs join: ','), ';')
          | {}                             -> ''.
  transFn = trans:x                        -> ('(function(){return ', x, '})').
}.
```

Appendix E: Extended Example: A Tiny TCP/IP Done as a Parser (by Ian Piumarta)

Elevating syntax to a 'first-class citizen' of the programmer's toolset suggests some unusually expressive alternatives to complex, repetitive, opaque and/or error-prone code. Network protocols are a perfect example of the clumsiness of traditional programming languages obfuscating the simplicity of the protocols and the internal structure of the packets they exchange. We thought it would be instructive to see just how transparent we could make a simple TCP/IP implementation.

Our first task is to describe the format of network packets. Perfectly good descriptions already exist in the various IETF Requests For Comments (RFCs) in the form of "ASCII-art diagrams". This form was probably chosen because the structure of a packet is immediately obvious just from glancing at the pictogram. For example:

```
+-------------+-------------+-------------------------+----------+--------------------------------------+
| 00 01 02 03 | 04 05 06 07 | 08 09 10 11 12 13 14 15 | 16 17 18 | 19 20 21 22 23 24 25 26 27 28 29 30 31 |
+-------------+-------------+-------------------------+----------+--------------------------------------+
|   version   |  headerSize |      typeOfService      |                     length                         |
+-------------+-------------+-------------------------+----------+--------------------------------------+
|                    identification                  |  flags   |                 offset                 |
+---------------------------+-------------------------+----------+----------------------------------------+
|       timeToLive          |         protocol        |                    checksum                    |
+---------------------------+-------------------------+------------------------------------------------+
|                                              sourceAddress                                             |
+--------------------------------------------------------------------------------------------------------+
|                                           destinationAddress                                           |
+--------------------------------------------------------------------------------------------------------+
```

If we teach our programming language to recognize pictograms as *definitions of accessors* for bit fields within structures, our program is the clearest of its own meaning. The following expression creates an IS grammar that describes ASCII art diagrams.

```
'{
  structure :=
    error       = ->[self error: ['"structure syntax error near: " , [self contents]]]
    eol         = '\r''\n'* | '\n''\r'*
    space       = [ \t]
    comment     = [-+] (!eol .)* eol
    ws          = (space | comment | eol)*
    _           = space*
    letter      = [a-zA-Z]
    digit       = [0-9]
    identifier  = id:$(letter (letter | digit)*) _  -> [id asSymbol]
    number      = num:$digit+ _                     -> [Integer fromString: num base: '10]
    columns     = '|'                               -> (structure-begin self)
                   ( _ num:number                   -> [bitmap at: column put: (set bitpos num)]
                       (num:number)* '|'            -> (let ()
                                                         (set bitpos num)
                                                         (set column [[self readPosition] - anchor]))
                   )+ eol ws                        -> [bitmap at: column put: (set width [bitpos + '1])]
    row         = ( n:number                        -> (set row n)
                    ) ? '|'                         -> (let ()
                                                         (set anchor [self readPosition])
                                                         (set column '0))
                  _ ( id:identifier '|'             -> (structure-field self id)
                        _ )+ eol ws                 -> (set row [row + width])
    name        = id:identifier (!eol .)* eol       -> (structure-end id)
    diagram     = ws columns row+ name | error
 }
```

It scans a pictogram whose first line contains numbers (identifying bit positions) separated by vertical bars (anchor points, '|'). Subsequent lines contain vertical bars (matching some subset of the anchors in the first line) separated by field names that will become the names of accessors for the bits between the anchors. Any line beginning with a dash '-' is a comment, letting us create the horizontal lines in the pictogram. The final line of input recognized contains a single identifier that is a prefix to the structure accessors; this lets us write a 'caption' on a pictogram whose first word is the name of the structure depicted. The first line of the grammar gives it the name 'structure' and the final rule can be

referred to from within any other grammar by the name 'structure-diagram'.

We can now define accessors for the fields of an IP packet header simply by drawing its structure. The following looks like documentation, but it's a valid *program*. It declares and defines accessors called ip-version, ip-headerSize, and so on through ip-destinationAddress.

```
                                    { structure-diagram }
+-------------+-------------+------------------------+----------+---------------------------------------+
| 00 01 02 03 | 04 05 06 07 | 08 09 10 11 12 13 14 15 | 16 17 18 | 19 20 21 22 23 24 25 26 27 28 29 30 31 |
+-------------+-------------+------------------------+----------+---------------------------------------+
|   version   |  headerSize |      typeOfService      |                    length                       |
+-------------+-------------+------------------------+----------+---------------------------------------+
|                     identification                  |  flags   |                 offset                |
+---------------------------+------------------------+----------+---------------------------------------+
|       timeToLive          |         protocol       |                   checksum                      |
+---------------------------+------------------------+---------------------------------------------------+
|                                        sourceAddress                                                    |
+-----------------------------------------------------------------------------------------------------------+
|                                      destinationAddress                                                   |
+-----------------------------------------------------------------------------------------------------------+
                          ip -- Internet Protocol packet header [RFC 791]
```

The first line '{ structure-diagram }' is a top-level COLA expression representing an anonymous grammar object. This grammar has a trivial default rule that matches the 'diagram' rule defined in the 'structure' grammar. The anonymous grammar object is evaluated by the COLA shell, and immediately starts to consume text from the program until it satisfies the structure-diagram rule. In doing so, it defines the ip-* accessors of our packet header structure. The COLA read-eval-print loop regains control after the entire structure diagram has been read.

Given a packet p read from a network interface, we can check that (ip-version p) is 4, (ip-destinationAddress p) is our interface's address and (ip-protocol p) is 6, indicating a TCP packet. The payload begins at p + (4 * (ip-headerSize p)) and will be a TCP header, which we also choose to declare and define by drawing its contents:

```
                                    { structure-diagram }
+-------------+---------+---------+------------------+-------------------------------------------------+
| 00 01 02 03 | 04 05 06 | 07 08 09 | 10 11 12 13 14 15 | 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 |
+-------------+---------+---------+------------------+-------------------------------------------------+
|                     sourcePort                      |                destinationPort                  |
+-----------------------------------------------------+-------------------------------------------------+
|                                          sequenceNumber                                                 |
+---------------------------------------------------------------------------------------------------------+
|                                       acknowledgementNumber                                             |
+-------------+---------+---------+------------------+-------------------------------------------------+
|    offset   | reserved |   ecn   |    controlBits   |                   window                        |
+-------------+---------+---------+------------------+-------------------------------------------------+
|                     checksum                        |                urgentPointer                    |
+-----------------------------------------------------+-------------------------------------------------+
                  tcp -- Transmission Control Protocol packet header [RFC 793]
```

If we provide a single service then it is enough to reject incoming packets having an unexpected tcp-port or tcp-sequenceNumber and to provide correct tcp-acknowledgementNumbers in outgoing packets. The state of the tcp-controlBits (containing the TCP SYN, ACK, PSH and FIN bits) is sufficient to determine unambiguously the appropriate reply. Although overkill for such a simplified TCP implementation, we can write the control structure as a trivial grammar:

```
['{ svc     = &->(svc? [self peek])
    syn     = &->(syn? [self peek]) .   ->(out ack-syn    -1 (+ sequenceNumber 1) (+ TCP_ACK TCP_SYN) 0)
    req     = &->(req? [self peek]) .   ->(out ack-psh-fin 0 (+ sequenceNumber datalen (fin-len tcp))
                                                             (+ TCP_ACK TCP_PSH TCP_FIN)
                                                             (up destinationPort dev ip tcp
                                                                 (tcp-payload tcp) datalen))
    ack     = &->(ack? [self peek]) .   ->(out ack           acknowledgementNumber
                                                             (+ sequenceNumber datalen (fin-len tcp))
                                                             TCP_ACK 0)
            ;
    ( svc (syn | req | ack | .) | .     ->(out ack-rst       acknowledgementNumber
                                                             (+ sequenceNumber 1)
                                                             (+ TCP_ACK TCP_RST) 0)
    ) *
  } < [NetworkPseudoInterface tunnel: '"/dev/tun0" from: '"10.0.0.1" to: '"10.0.0.2"]]
```

As before, the text between curly braces defines a grammar object. Quoting that object and then sending it a '<' message with a network interface as argument will create and run a parser for the grammar connected to a stream reading packets from the interface.

The first rule is a predicate that filters out unknown service port numbers. The next three describe the appropriate replies to SYN packets, connection data transfer packets, and ACK packets received from a connecting client. The 'out' function invoked from the actions in these rules reconstructs a TCP/IP packet with the given parameters, fills in the checksum, and writes the packet to the network interface. The 'up' function delivers the packet and its payload to a local service provider. The final (unnamed) rule in the grammar is the start rule that says to reset the connection if the service is unknown, to reply appropriately to SYN, data transfer and ACK packets, and to ignore everything else. A few functions called from the above code are not shown; they are short and their names should indicate clearly what they do. The four helper functions referred to within the named rules are not as obvious, and are defined as follows:

```
(define tcp?  (lambda (p) (== 6 (ip-protocol p))))

(define svc?  (lambda (p) (let ((ip [p _bytes]))
                (and (tcp? ip) (tcp-service-at (tcp-destinationPort (ip-payload ip)))))))
(define syn?  (lambda (p) (let ((ip [p _bytes]))
                (and (tcp? ip) (& TCP_SYN (tcp-controlBits (ip-payload ip)))))))
(define req?  (lambda (p) (let ((qi [p _bytes]))
                (and (tcp? qi) (with-tcp-ip qi (and (== 0 acknowledgementNumber) datalen))))))
(define ack?  (lambda (p) (let ((ip [p _bytes]))
                (and (tcp? ip) (or (> (ip-length ip)
                                      (* 4 (+ (ip-headerSize ip) (tcp-offset (ip-payload ip)))))
                                   (& TCP_FIN (tcp-controlBits (ip-payload ip))))))))
```

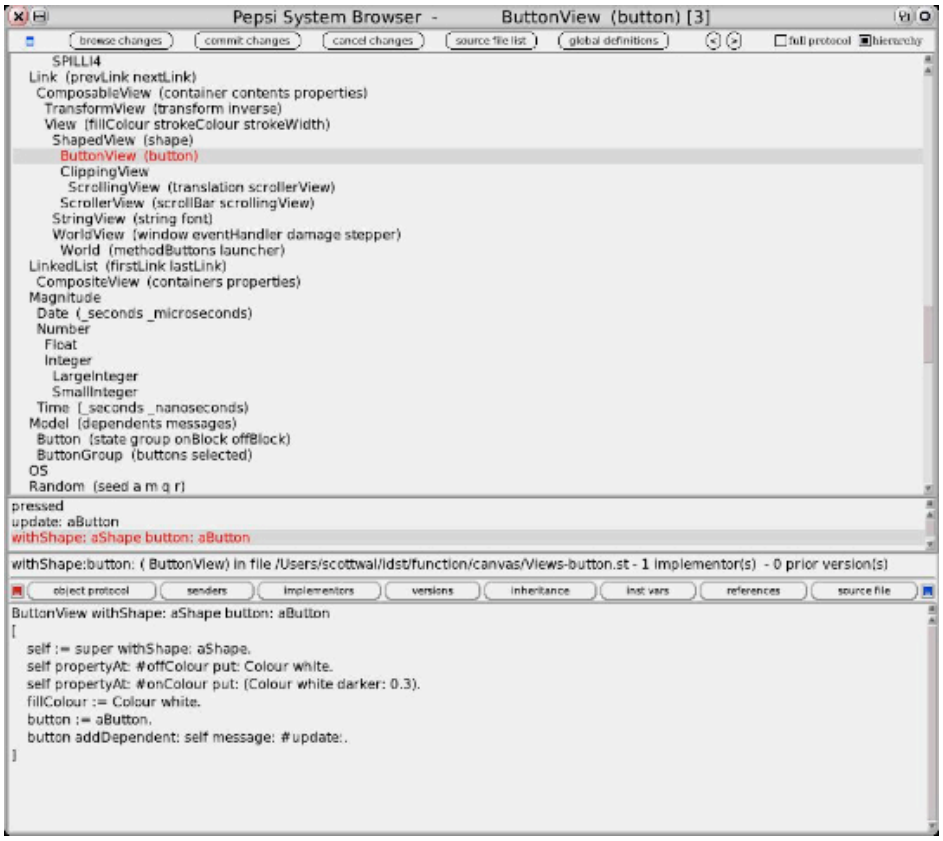
The with-tcp-ip syntax is a combination of with-ip and with-tcp, which are defined by the structure diagrams to run their body code in a context in which the fields of a particular instance of the named structure are available without explicit use of the prefix or the name of the instance.

Using the above network stack, we can now provide a simple 'daytime' service on port 13 like this:

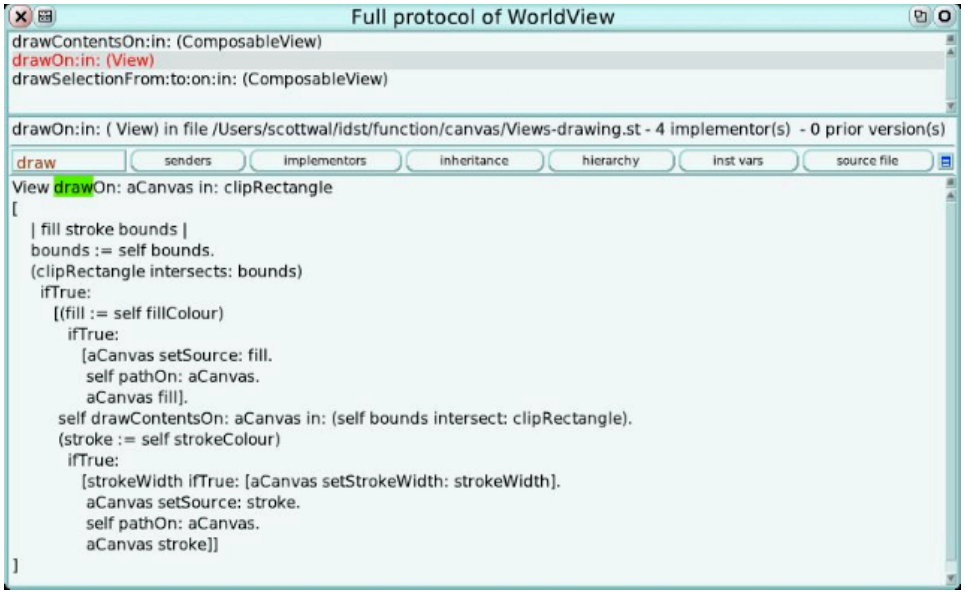
```
(define daytime-receive
  (lambda (if ip tcp data len)
    (set (char@ data len) 0)
    (printf "DAYTIME client sent %d bytes:\n%s\n" len data)
    (strlen (strcpy data [[Time now] _formatted_: "%a %d %b %Y %T %Z\n"]))))

(tcp-service-at-put 13 daytime-receive)
```

Appendix F: Interactive Development Environment for IS (by Scott Wallace)

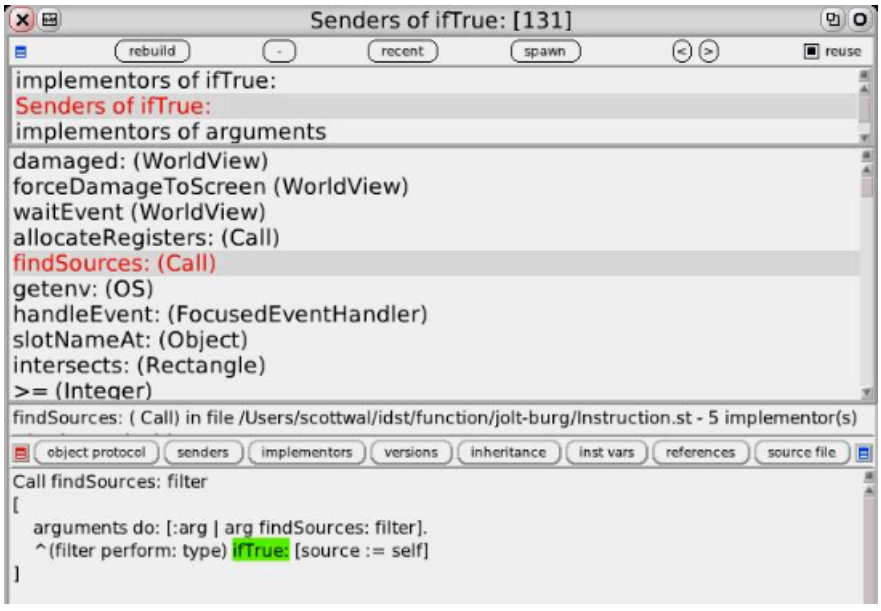
(a) System Browser For a given source tree, shows all the object type-definitions, and all the code of each type represented. Checkboxes at top allow the alternatives of viewing by hierarchy (as illustrated) or viewing alphabetically by type-name, and also the alternatives of listing under each type only the methods explicitly defined by it or listing its "full protocol" (including inherited methods.) An "annotation pane" provides collateral information about the currently selected method, and buttons and menu-items in the tool allow for queries to be issued.



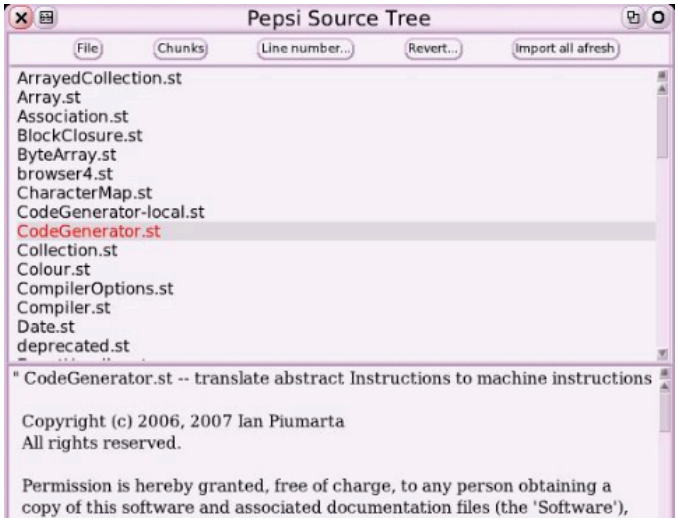
(b) Individual Object Lexicon For a given object, this tool shows all of its code (including inherited methods) and (once it is running in the target system) all of its state, and allows everything – all code and all state – to be edited. The centerpiece of this tool is its "search" capability, allowing the user to find methods, within the protocol of a given object, by name/keyword. In the example below, the user has typed "draw" into the Search pane, and in consequence only the three methods in the object's protocol whose selectors contain the substring "draw" are displayed:



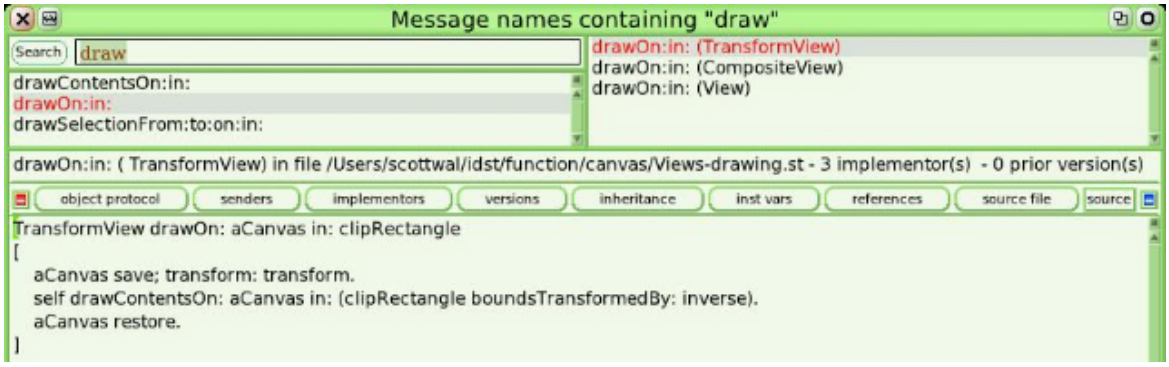
(c) Message-Set Lists These provide a uniform mechanism for browsing the results of multiple queries within the bounds of the same tool. Typical queries include: senders, implementers, references... (see "Facile Queries" below.) In the example below, the results of three different queries are all seen in the tool. The < and > buttons in the control panel at the top of the tool allow the user to retrace his browsing history within the tool, and the "reuse" checkbox allows the user to decide whether further queries should be serviced within the tool or whether new windows should be used for them.



(d) "Flattened File-List" This tool presents all the files that participate in the source tree in a single, flat list, and allows the user to see and change the contents of the files at any time, using the highly-evolved Squeak text-editing tools. Automatic versioning is provided, and the "Revert..." button allows for selective rollback.



(e) "Message Names" The entire system can be searched for methods whose names (selectors) match any given string pattern; the retrieved methods can be viewed and edited in-place within the tool, and all the usual queries can be initiated within the tool as well. In the example below, the user has searched for methods whose selectors contain the fragment "draw". Three selectors were found. One of these, #drawOn:in:, has been clicked on by the user, revealing three object types which implement that method. One of these implementations, that of TransformView, has been selected, and the source code we see in the bottom pane is the code for TransformView's implementation. of #drawOn:in:.



Facile queries – An important property of effective code development in live IDE's is that most of the plausible queries that the programmer needs to make during development can be posed from within the tool currently being used, and (ideally) the results of most such queries can be viewed within the tool from which the query was launched. Some of the queries that can be invoked at any point from within any tool of the IDE are: Browse all methods:

- that implement a given message
- that send a given message
- that reference a given instance variable of a given object type
- reference a given global entity
- whose selectors contain a given string pattern
- for the inheritance hierarchy of a given method
- that have been changed since the last commit.

Appendix G: Gezira Rendering Formulas (by Dan Amelang)

Given the x and y coordinates of the lower-left corner of a pixel, the coverage contribution of an edge AB can be calculated as follows:

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

$$\begin{aligned} \gamma(P) &= min(x + 1, max(x, P_x)), \\ & \quad min(y + 1, max(y, P_y)) \end{aligned}$$

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

$$\begin{aligned} coverage(\overrightarrow{AB}) &= \sigma(\gamma(A), \gamma(\omega(A))) + \\ & \quad \sigma(\gamma(\omega(A)), \gamma(\omega(B))) + \\ & \quad \sigma(\gamma(\omega(B)), \gamma(B)) \end{aligned}$$

The total coverage contribution of a polygon is the linear combination of the edge contributions, with some additional adjustment:

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