STEPS Toward The Reinvention of Programming, 2008 Progress Report Submitted to the National Science Foundation (NSF) October 2008

A. Kay, I. Piumarta, K. Rose, D. Ingalls, D. Amelang, T. Kaehler, Y. Ohshima, H. Samimi, C. Thacker, S. Wallace, A. Warth, T. Yamamiya

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

VPRI Technical Report TR-2008-004

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

Figure 1: The first page of this report prepared in the STEPS system itself

Jeannette Wing characterizes computing as "the automation of abstractions". If science can be looked at as "finding the best abstractions to represent and understand phenomena", then the science in STEPS is to find those fruitful abstractions throughout the range of personal computing, and the engineering in STEPS is to find out how to actually automate those abstractions to create a working model that is both illuminating in its form and that works in useful practical fashions with users.

If the abstractions chosen and invented are powerful and parsimonious, then we are betting that the realization of this system will not require much actual code.

Major Findings

An example that is at once a major result of this project, serves as an excellent analogy to the rest of the project, and is relatively easy to exhibit because of its graphical nature, is the universal polygon rendering system we invented and use for all graphics. We started by examining one of the main open source renderers – Cairo – which is essentially a variation of PostScript made for both screen display and for printing. Cairo is good, but quite large and consists of an enormous number of procedures that handle special cases for reasons of efficiency.

One way to state the general problem is that every shape we are interested in can be modeled by a "polygon with polygonal holes" and from these we have to find the *shadings* to assign to the squarish pixels of the display devices that will most approximate a perfect display of perfect curves.

This raises the question as to whether this problem can be mathematized (a) to capture the relationships compactly and understandably, and (b) in a way that the math can be automatically transformed into an efficient running program. We call mathematical expressions that fill both of these requirements "dynamic math".

Our compact mathematical characterization of rendering measures the contribution of an arbitrary polygon edge with an arbitrary rectangle (in practice a square pixel):

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

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

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

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

Our meta-representation-and-transformation engine allows us to easily make up languages and to guide their automatic transformation into efficient lower-level code.

About 500 lines of code are needed to express the above rendering formula in a conventional programming language; we expect this to drop to a few tens of lines when the mathematics *is* the program.

We can turn this into a visual example by creating a TEX-like constraint engine to fit sequences of polygons that describe the outlines of text characters into a justified paragraph, and produce images like Figure 1 of this report.

Two of the results from this work surprised us. The first was that the rendering was much faster than we had anticipated. The second was that the rendering looked much better than we had expected. The latter is particularly striking when very small sizes are chosen for the text (but is difficult to reproduce here because of the limitations of graphical import).

Most high-quality rendering engines use hinting and other tricks such as snapping to pixel boundaries and "color picking" to use the actual sub-pixel resolution of flat-screen displays. In contrast, the direct calculation of shading in our rendering engine seems to produce near-perfect mathematical anti-aliasing without any of the artifacts associated with conventional methods such as sub-sampling from higher resolutions.

A second striking example from this year's research is the handling of communications protocols, specifically TCP/IP. We need this to communicate over the Internet, and it must be very small and fast. Typical implementations are between 2000 and 20,000 lines of (usually) C code. Since our entire system is aimed at no more than 20,000 lines of code, and has wide scope and depth, we can budget only a hundred or so lines of code for TCP/IP.

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 most or all of 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 unreliable exchange of raw packets), and TCP (which adds 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, which delivers packets to a specific service but does not address any of the unreliability of the underlying IP, leaving error detection and recovery to the programmer).

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 IP mechanics. The header formats are parsed from the diagrams in the original specification documents, converting “ascii art” into code to manipulate the packet headers.

Here we give a glimpse of what the “programming with grammar” looks like for the rejection of incoming packets with unexpected TCP port or sequence number, and the calculation of correct acknowledgement numbers 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 raw packet data 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 packet with the given parameters, fills in the checksum, and writes the packet to the network interface.

The above two examples illustrate our thesis that inventing new dynamic mathematics will vastly reduce code and raise expressibility and understanding. Over the last two years we have made and demonstrated examples and “learning-projects” at every level of the system: from bootstrapping on bare hardware to fundamental meta-mechanisms for creating process, contexts, languages, communications systems, and graphical rendering. We have created prototypes of integrated development environments, user interfaces, and end-user applications.

It is worth saying a little more about two of these areas: contextual control mechanisms and the use of inferential “expert system” techniques for doing systems design and programming.

For example, we believe there is much (albeit scattered) evidence that being able to safely and efficiently retract computation at any level of granularity will have a positive effect on how we structure and write code. Certainly, having a comprehensive UNDO at the user level is part and parcel of personal computing itself. The level of comprehensiveness here has a direct bearing on the kinds of experiments that end-users

can be encouraged to do; for learning and also for providing peace of mind in the face of the myriad errors that naturally occur in systems the size of the Internet. An effective UNDO at the next programming level will allow programmers to do much better heuristic programming and to deal with errors by recovering the system to a safe and predictable previous state. At another level, it is the essence of both backtracking algorithms and possible worlds reasoning systems to reach dead-ends and have to try other branches. A comprehensive UNDO would allow actual experiments involving any and all states in the system to be looked at speculatively before any commitments have to be made to the results of such speculation. This could be thought of as "system dreaming".

These and other considerations lead to a deeper set of convictions, some of which we've started exploring this year: that quite a lot of the structure of the STEPS system will resemble a "next generation expert system" for representing and reasoning as a "Society Of Mind" (to use Marvin Minsky's provocative phrase), or at least as an "Ecology Of Abilities" or "Meanings" (to use our less-charged phrasing).

The point here is that our goals are not to build a system that can learn, represent and reason like a human being (this larger goal is still over the horizon for computing), but that we should be able to use the best "architectures and math" that would be employed in a serious attempt at the larger goal in order to do a good job of handling our much smaller system design and programming goals.

So, for example, it would be very useful if the "set of support" ideas from expert systems of the past – that deduced "knowledge" should dynamically rest on its evidence and reasoning chains, and should be maintained or retracted on whether the deductions still hold – could be extended to all code dependencies in the system.

We would also like the system to be able to do experiments with its own "abilities and meanings" – in the manner of several "discovery systems" of the past – so a kind of "semantic typing of meanings" can be set up and used for both local abilities and new abilities that might be found dynamically over the Internet.

One of the architectural "powerful principles", described in the original proposal, that would make a huge difference in realizing our goals is *the separation of meaning from opitimizations*. We can see that this goal meets up nicely with many of the representation issues stated above that require our system to be able to reason about itself and what it can do.

Research and Education Activities

The STEPS project requires both new inventions of computer structures in terms of "dynamic mathematics" and the engineering of the math in the transformation system so it can be used pragmatically in a computer system. Often, these stages are done in subsequent years. For example, the Gezira rendering system was invented and tested in year 2007, but engineered to be an efficient and generally useful component in 2008.

For 2008, we present both new inventions that have been carried forward enough to be demonstrated, and a number of next-pass engineering versions of the major findings of 2007.

Communications Strategy Language (CSL) – a first example: our very compact TCP/IP, briefly described earlier in this report. The premise: since TCP/IP is essentially a collection of heuristic strategies combined with various kinds of parsings and assemblies that must be robust in the presence of errors, a special language which is directly aimed at the problem space should be much more compact and understandable, and would be very useful if it could run fast enough to deal with the realities of the Internet. This turned out to be the case, and only 160 lines of metalanguage and language total were required to make a working TCP/IP and to have it serve as the basis of a simple website. A close examination of this result indicates that it could possibly collapse by half again to 80 lines of even clearer code in the next attempt at this.

An important design criterion for making full use of "problem oriented languages" is to find ways to avoid the "Tower of Babel" problem of winding up with a system composed of many different languages which – although small – would be extremely difficult to learn and decipher. Our approach is to gradually seek an underlying style for these languages that will eventually wind them up looking very much like each other as at least a "family of expression". Quite a bit of this has happened already because the way we talk about these problems has incorporated our general outlook on how the system will eventually function. Many of the experiments over the last few years use pattern-directed non-deterministic "discovery and transformation" coupled with constraint-solving as their central style of approach.

Worlds – an experiment in providing language support for controlling the scope of side effects. In this model, all computation takes place inside a world, which captures the side effects (i.e., changes to global, local, and instance variables, arrays, etc.) that happen inside it. A new world can be "sprouted" from an existing world at will. The state of a child world is derived from the state of its parent, but the side effects that happen inside the child do not affect the parent. At any time, the side effects captured inside the child world can be propagated to its parent via a *commit* operation.

We have prototyped this idea as an extension of JavaScript, and our early experiments indicate that worlds have a wide range of applications including sandboxing, undo, tentative execution, improving exception-handling mechanisms, and may even be used as the basis of a powerful module system. We also plan to investigate worlds in a parallel

setting, as we believe that this may lead to a more tractable model for programming multi-core architectures.

An important idea here is that the "Worlds" mechanism is efficient enough to work well at fine levels of grain in the system as well as for much larger system organizations.

JOHN – (for John McCarthy) a representation, reasoning and programming system for expressive and efficient realization of goals, automated planning (similar to STRIPS, an early planning program), and separating meanings from optimizations.

It is an object-oriented model of microworlds which uses situation calculus, possible worlds models, and first-order logic to reason about past and future states of the objects. The system uses search of possible worlds along with the aid of heuristics and optimizations to find a rather optimum path to a goal of an object in the microworld.

The programming language part of JOHN has three main objectives:

1) To offer a higher level of intelligence to everyday programs by bringing traditional AI techniques such as search, heuristics, and planning to everyday programming.

We can add a higher level of control and readability to everyday programs if the procedures do not run by themselves, but a higher level agent chooses which combination of procedures to run and when. The agent can use the aid of heuristics and optimizations to decide the most relevant set of procedures to run, as well as search for exploring multiple possibilities.

2) To explore ways to *separate meanings from optimizations*. The goal-driven methodology offers more coherent, readable, and manageable programs by separating the meanings and invariants of problems (i.e., what it is they're trying to accomplish) from procedures and optimizations (i.e., how they accomplish that). The agent can determine when the task is accomplished by asking the meaning part of the program.

3) To be useful as an educational programming tool for children and non-programmers.

The syntax aims to read like natural language and mathematics. It follows the belief that if the statements fairly resemble sentences of a natural language, a non-technical person can understand the system by reading the program itself. The most valuable learning aspect for children would likely be how they can "model" a specific domain problem as a "microworld" in the most natural way.

Many examples were written and run in JOHN this year including traditional programming problems (such as sorting and searching), games (chess, checkers, etc.) and question answering.

XOOS – is an initial excursion into booting the STEPS system on bare hardware. OpenFirmware is used to load STEPS onto the machine and start it running; subsequent access to system resources is made either by direct interaction (we are developing simple device drivers for low-cost, high-payoff access to display hardware acceleration) or by reusing existing OpenFirmware support. One early result from these experiments on the One Laptop Per Child XO machine shows that a simple hardware-accelerated framebuffer driver supporting the AMD GeodeLX BitBlt operation can be written in 20 lines of understandable code. Following our "right abstraction for the job" philosophy,

we intend to invent fundamentally more compact, mathematical and elegant ways to describe and program the interactions between software and hardware.

Bottle – is an experiment to determine the requirements for a Lisp/Smalltalk-like listener/workspace tool in the STEPS system. (A listener or workspace is a text editor in which a text selection can be evaluated as an expression and the result pasted back into the workspace.)

IS – the core of the STEPS architecture. Work this year has concentrated on re-engineering the "transformational pipeline" that connects high-level abstractions at one end to low-level implementations at the other. A single transformation framework is nearing completion that is powerful enough to support top-down parsing of unstructured text (such as program source), structure-to-structure transformations (for reasoning about the meaning of programs), and bottom-up rewriting of structures (to generate unstructured output, such as machine code) within a single abstraction, and which is efficient enough to be deployed in an interactive or on-demand execution environment.

Lively Kernel. The Lively Kernel (LK) as of last year was a project to impose a much more powerful architecture for building whole systems on top of a weaker contemporary language system and environment – in this case, the Javascript and DOMs that run in web browsers. About 10,000 lines of code were required to make a powerful virtual environment for programming and graphics that includes real classes, the morphic graphics architecture, an overlapping windowing and development system, widgets, apps, etc. Lively is an example of a compact powerful "environment building architecture" which can be built on a substrate that "virtualizes" more mundane building blocks (e.g., Javascript or STEPS).

The Lively Kernel in 2008 has focused on three main explorations of higher-level architectures: development environments, end-user authoring, and stand-alone kernels.

In the end-user hosting space, we have developed a client-side wiki which uses LK's WebDAV facility to do auto-commits into an SVN repository. Thus, instead of the usual edit/use mode wiki process, the Lively Kernel is always in use mode, and one can save at any time, with SVN providing recourse to previous versions for undo, or for historical inquiry. This wiki, coupled with a page-local changeSet facility, offers a host environment for numerous Lively kernel projects.

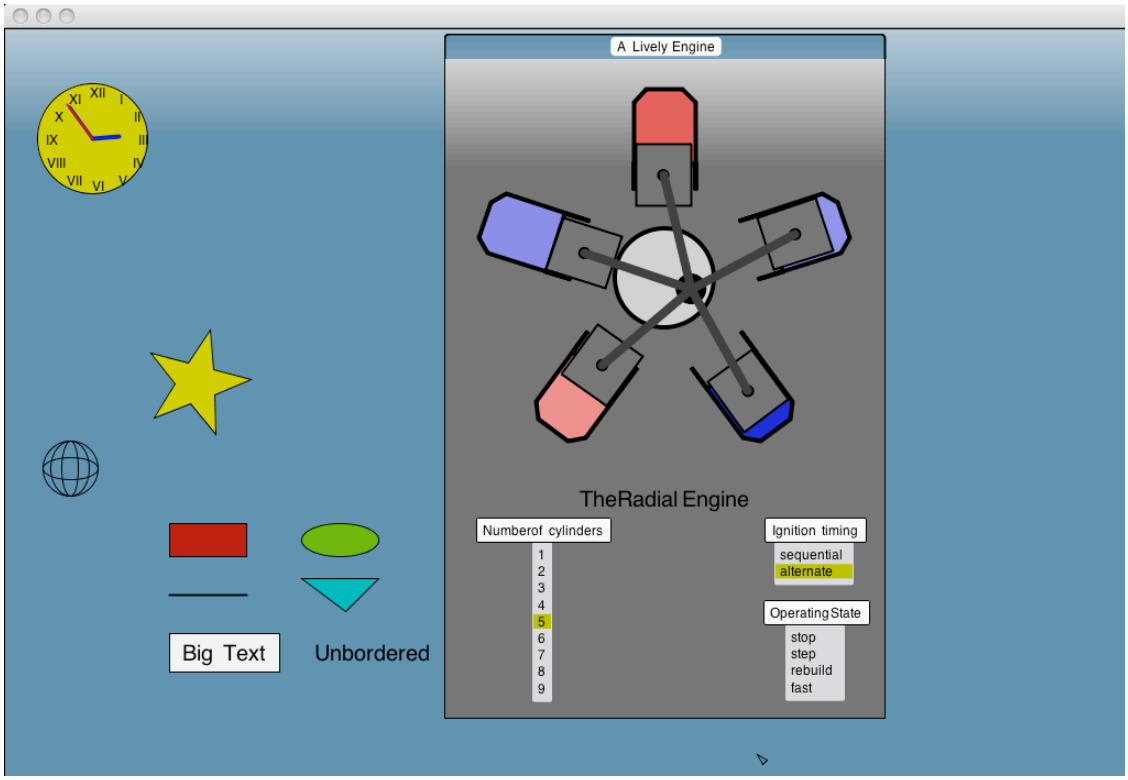
We have initiated two end-user programming projects, one similar to the Fabrik "hook-together" visual programming system, the other similar to the EToys tile scripting environment. The Fabrik project, like its ancestor, is mainly appropriate to UI-intensive applications such as desk accessories for viewing weather, stock prices and other information gleaned from the web. The EToy-style work is being used to program the function components in that Fabrik system, and also to experiment with various amusing and educational Flash-like animations and simulations.

On the developer side we have the beginnings of a Smalltalk- (or emacs-) style development environment within the Lively Kernel itself. Originally only a toy IDE built around JavaScript's eval, we have now built an entire reflective runtime achieved by

wrapping all methods to provide an emulated stack even though, for security reasons, this is not available in most JavaScript VMs. Serious capabilities include debugger stack access, trace-to-log, and full execution profiles with exact counts and millisecond tick tallies.

Light and Lively – One of the ancillary projects last year was to completely replace the browser Javascript with one of ours made in less than 200 lines of code in the OMeta and IS systems. This year we replaced the SVG graphics and rendering with our own Gezira system to make a standalone version.

Beginning with the emulated DOM object project (see above), we have executed a relatively complete port of the Lively Kernel to an unanticipated native environment in less than two weeks (see figure). The host environment in this case is the Java FX scene graph which, fortunately, bears a very close resemblance to SVG in form as well as function.



Although our final STEPS system is going to look rather different from Lively (which mainly uses very powerful techniques from 20 and 30 years ago), we are gaining valuable experience with our tools and components by being able to test them by driving real systems.

Lesserphic – an experimental GUI framework that gives special prominence to the notion of multiple perspectives. Graphical entities in a GUI often have a range of functionality, of which display-related code is only one part. Conceptually, appearance is not necessarily bound to functionality and units of functionality usually fall into a range

of well-separated categories of behavior. In practice, code governing appearance and behavior is often intertwined, and behavioral concerns are usually not cleanly separated.

Graphical objects in Lesserphic have multiple perspectives, which we call "behaviors". Each behavior is a set of methods and related state, documentation, and (potentially) unique tools. Behaviors are "first-class" objects, allowing a user to move them between widgets in a straightforward manner.

Exploiting the strengths of Lesserphic requires supporting tools. Drawing on concepts that evolved from the Etoys system (as well as conventional IDE notions) the "elementary particles" from which familiar and powerful tools (such as "viewers", "scriptors", and "watchers") are built were identified and implemented in Lesserphic, to serve as an intermediate layer between the underlying structures and the end user. Upon these we can build analogs not only to all the basic Etoy widgets, but also to programming tools such as browsers and inspectors.

Leastphic – the continuing quest to find the minimal UI framework. The ideas from the Lessphic UI project reported last year are here taken to extremes: everything is still a polygon, but now there is just a single organizational primitive – the "Box". This useful UI framework, sufficient to allow the authoring of Figure 1 in this report and providing many of the usual widgets (buttons, menus, etc.), has been reduced to less than 1000 lines of STEPS code. The final STEPS UI framework will be borne of the "tension" between the goals of Lesserphic and Leastphic.

Algorithmics toolbox – amongst the various algorithms available to programmers, a few stand out for being so powerful and general in application as to constitute fundamental building blocks for the construction of elegant solutions to a wide variety of problems. Examples include linear programming and dynamic planning. Making these available as "generic" tools for parameterization and reuse in arbitrary contexts will be a focus of future work. This year we took a brief look at dynamic planning, inspired by Donald Knuth's solution to breaking paragraphs into lines (needed for our top-to-bottom authoring tool) and incorporated it into the Leastphic architecture as one of several layout algorithms available for organizing the contents of a "box".

Source IDE – The "Source-IDE" development environment for STEPS, begun at the end of 2007, continued to evolve. A first usable version, hosted in Squeak as a cross-development platform, was released in early 2008.

Current areas of focus, in addition to advances in usability and robustness whose need became evident as a user community began to form, include adding abilities both to submit files for compilation and to run an executable built in this fashion, directly from within the IDE. Additionally, several new, higher-level tools for traversing and querying a large source-tree are in prospect for development in early 2009.

We anticipate that in 2009, the ongoing work on UI frameworks, such as "Lesserphic", combined with a further-evolved Source-IDE, should lead to the development of a first, base-level native STEPS IDE, which will lead to a durable, self-sustaining set of native

tools. Experience has shown that being able to develop productively within the system itself will result in a snowballing effect with regard to usability of the system and productivity of its users.

OMeta/JS – a pattern-directed meta-programming language for JavaScript. This makes it possible for other researchers and programmers to experiment with some aspects of the STEPS project from within their web browser, without having to install any additional software. Programming language experiments can be performed within the system by source-to-source transformation to JavaScript. A wiki-style web site (called the OMeta/JS Workspace) allows users to create their own programming language experiment projects and then make them accessible to other users.

TileScript – an end-user active essay programming environment using JavaScript within a web browser. With it authors can seamlessly combine computer programs and text within a single document. A TileScript document consists of one or more paragraphs where a paragraph is either JavaScript code, a "tile script", or an HTML expression. TileScript documents support Wiki-style hyperlinks and can be saved on a web server using the WebDAV protocol.

A tile script is a set of draggable tile objects where each tile represents a syntactic element in the programming language. Tiles can be combined to construct a program using drag and drop. This is an easy way to write programs without syntax errors.

HTML is used for annotation and explanation. Any HTML elements can be placed around tile scripts and programs to construct rich narrative explanations.

Chalkboard – a lightweight version of TileScript providing hyperlinks and dynamic JavaScript execution. The goal is also to support authoring Active Essays while keeping the system as simple as possible. Contents in Chalkboard are text oriented and fit naturally into HTML's document object model.

The screen has three areas: an editor, a play area, and a transcript. In the editor, a user writes text or source code that can be executed as a JavaScript program. The play area provides a free-form space for the user that can be used for any purpose; for example, a drawing tool might attach a canvas element to it for graphical output. The transcript area displays the results from evaluating code in the editor.

DynaBook Jr. – is the name of our modern DesktopPublishing/HyperCard-like system. It borrows the idea of "card stacks" from HyperCard and allows arbitrarily complex objects to be included in cards. Text is "journaled" to provide resilience against crashes.

BoldlyCode. To develop Dynabook Jr, we made an end-user scripting language called BoldlyCode, derived from Mark Lentczner's Glyphic Codeworks (1995), that uses text emphasis to denote syntactic forms. In BoldlyCode, compound messages are in bold, unary messages are underlined, and comments are in italics.

```
Spaceship motor.
    when burnButton color is Color green
        do [make my ySpeed increaseBy 3.
            make flame's locationBe my location.
            flame show]
        otherwise [flame hide]
```

Running Parsers Backwards. Translation of source code from one computer language to another is normally done by using a parser to convert the text to an Abstract Syntax Tree (AST), and then a "pretty printer" to convert the tree to the target language. Going in both directions between two languages therefore requires four components: two parsers and two pretty printers. We have investigated techniques for running the rules of a parser in reverse to convert an output AST back to its input source code. This makes it possible to translate programs back and forth between two languages using only two parsers. Some of the parser's rules must be written in a restricted form, so that our system can automatically extract templates from them. The parsers are written using a PEG-based parser generator so that we can analyze the ASTs of the parser's own rules.

FPGAs and "Hardware by Programming". Last year we reported a "tiny computer" realized in FPGAs by Chuck Thacker (now at Microsoft) a longtime colleague (reaching all the way back to Xerox PARC in the 1970s). One of the secret weapons back then was the use of microcoded inherently-DPU (Distributed Processing Unit) architectures. This was facilitated by the combinations of MSI and LSI available just as the "Moore's Law" revolution hit in 1971. This allowed raw hardware to be semantically and pragmatically restructured to be a better fit for problem spaces whose code needed to be small and efficient. (The basic idea is that if things could be arranged so that main memory was inescapably the bottleneck then nothing faster than microcode would help, and thus the goal of "hardware via program" would be achieved.)

Today, the analogous architectures to provide similar leverage would not be realized in discrete components whose artful arrangements could be organized by a master designer, but would have to be realized on a new chip itself, and this is just too expensive for experimentation (and unfortunately also is perceived to be too big a bet even for large-scale commercial changes).

However, the very promising technology of Field Programmable Gate Arrays has come of age, and Chuck Thacker (the central hardware designer at Xerox PARC in the '70s) has this year made a new kind of hardware platform – BEE3 – whose goal is to "return hardware architecture to the cutting edge of computer-science research". This platform would be our first choice to investigate the issues "a little below the metal" related to the STEPS project.