



STEPS Toward Expressive Programming Systems, 2010 Progress Report Submitted to the National Science Foundation (NSF) October 2010

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

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

VPRI Technical Report TR-2010-004

NSF Award: 0639876

Year 4 Annual Report: October 2010

STEPS Toward Expressive Programming Systems

Viewpoints Research Institute, Glendale CA

Important Note For Viewing The PDF Of This Report

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

Table Of Contents

STEPS For The General Public	2
STEPS Project Introduction	3
STEPS In 2010	5
The “Frank” Personal Computing System	5
The Parts Of Frank	7
1. DBjr	7
a. User Interface Approach	9
b. Some of the Document Styles We’ve Made	11
2. LWorld	17
3. Gezira/Nile	18
4. NotSqueak	20
5. Networking	20
6. OMeta	22
7. Nothing	22
2010 STEPS Experiments And Papers	23
Training and Development	26
Outreach Activities	26
References	27
Appendix 1: Nothing Grammar in OMeta	28
Appendix 2: A Copying Garbage Collector in Nothing	30

A new requirement for NSF reports is to include a few jargon-free paragraphs to help the general public understand the nature of the research. This seems like a very good idea. Here is our first attempt.

STEPS For The General Public

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

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

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

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

STEPS Project Introduction

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

Science of Art

Part of our aim is to practice a “science of the artificial” [1], paralleling how natural science seeks to understand complex phenomena through careful observations leading to theories in the form of “machinery” (models) – classically using mathematics – that provide understanding by recreating the phenomena and having the machinery be as simple, powerful and clear as possible. We do the same, but draw our phenomena from *artifacts*, such as human-made computer systems.

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

Art of Design

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

STEPS Aims At “Personal Computing”

STEPS takes as its prime focus the modeling of “personal computing” as most people think of it, limiting itself to the kinds of user interactions and general applications that are considered “standard”. So: a GUI of “2½D” views of graphical objects, with abilities to make and script and read and send and receive typical documents, emails and web pages made from text, pictures, graphical objects, spreadsheet cells, sounds, etc., plus all the development systems and underlying machinery down to typical personal computer hardware.

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

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

Evaluating STEPS

We set a limit of 20,000 lines of code to express all of the “runable meaning” of personal computing (as gisted above) “from the end-user down to the metal”, where “runable meaning” means that the system will run with just this code (but could have added optimizations to make it run faster). In fact we have added optimizations here and there to make STEPS run acceptably on the typical laptops we use. Cur-

rently we include these optimizations in our code counts for simplicity, but technically they are not part of the 20,000 lines of meaning.

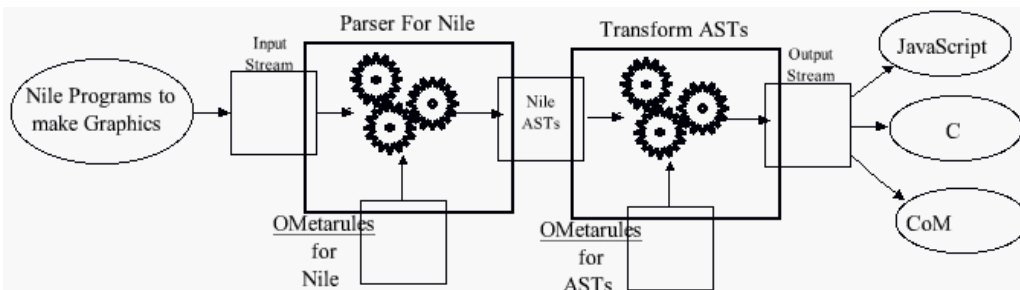
Thus one measure will be what did and did not get accomplished by the end of the project with the 20,000 lines budget. Another measure will be typical lines of code ratios compared to existing systems. We shoot for factors of 100, 1000, and perhaps the next order of magnitude as well. The basic idea here is that we are interested in very large qualitative comparisons, not small ones.

Another measure is understandability. Are the designs and their code clear as well as small? Can the system be used as a live example of how to do this art? Is it clear enough to bring to mind other, perhaps better, approaches?

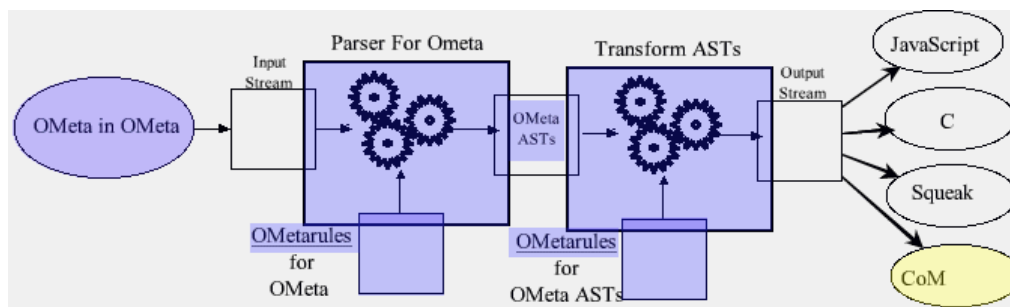
Previous STEPS Results

The first three years were devoted to making much smaller, simpler, and more readable versions of many of the *prime parts* of personal computing, including: graphics and sound, viewing/windowing, UIs, text, composition, cells, TCP/IP, etc. These have turned out well (they are chronicled in previous NSF reports and in our papers and memos).

For example, essentially all of standard personal computing graphics can be created from scratch in the Nile language in a little more than 300 lines of code. Nile itself can be made in a little over 100 lines of code in the OMeta metalanguage, and optimized to run acceptably in real-time (also in OMeta) in another 700 lines. OMeta can be made in itself and optimized in about 100 lines of code.



The two OMeta translators for Nile (as of 2009)



The two OMeta translators for OMeta (as of 2009)

The results so far show that many individual parts of personal computing can be created and run pragmatically in smaller amounts of more expressive program code (roughly factors of 100 to 1000 less code).

STEPS In 2010



This year has been aimed at “stitching together” the parts we’ve been working on to function as a kind of “Cute Frankenstein’s Monster”. This will give us a comprehensive platform to aid thinking. For example, to find better integrations to make the systems structures more loosely coupled than most personal computing architectures.

The current system—known as “Frank”—is starting to cover a wide territory of the standard personal computing landscape. Frank is occasionally pretty scary, but is friendly enough to allow us to make many examples to test the architecture at all levels.

rently we include these optimizations in our code counts for simplicity, but technically they are not part of the 20,000 lines of meaning.

Thus one measure will be what did and did not get accomplished by the end of the project with the 20,000 lines budget. Another measure will be typical lines of code ratios compared to existing systems. We shoot for factors of 100, 1000, and perhaps the next order of magnitude as well. The basic idea here is that we are interested in very large qualitative comparisons, not small ones.

Another measure is understandability. Are the designs and their code clear as well as small? Can the system be used as a live example of how to do this art? Is it clear enough to bring to mind other, perhaps better, approaches?

Previous STEPS Results

The first three years were devoted to making much smaller, simpler, and more readable versions of many of the prime parts of personal computing, including: graphics and sound, viewing/windowing, UIs, text, composition, cells, TCP/IP, etc. These have turned out well (they are chronicled in previous NSF reports and in our papers and memos).

For example, essentially all of standard personal computing graphics can be created from scratch in the Nile language in a little more than 300 lines of code. Nile itself can be made in a little over 100 lines of code in the OMeta metalanguage, and optimized to run acceptably in real-time (also in OMeta) in another 700 lines. OMeta can be made in itself and optimized in about 100 lines of code.

The two OMeta translators for Nile (as of 2009)

The two OMeta translators for OMeta (as of 2009)

The results so far show that many individual parts of personal computing can be created and run pragmatically in smaller amounts of more expressive program code (roughly factors of 100 to 1000 less code).

4

Writing the previous page of this report in Frank, with a “Halloween themed” user interface look

The visual part of personal computing coincides with “desktop publishing” and both of these were originally invented as the same thing at the same time. The same viewing mechanisms are used, the same display, compositing and layout mechanisms, etc. For visible objects, Frank uses a “foreground-background” scheme derived from Hypercard to create both content and its user interface.

The overall user interface scheme is new, but drawn from already invented UIs (particularly a combination of Etoys and MS Office 2007). This is partly because STEPS is about “personal computing” as users

generally see it, and a completely new UI (which we were sorely tempted to design) would make it more difficult to assess just how much of personal computing is actually subsumed by the STEPS designs. Frank's UI is not just an amalgam or clean up, but has enough new design to make it much easier, comprehensive, learnable and useful than its roots.

The Large Scale Architecture

The overall architectural scheme stems from some of the earliest ideas about objects and networks – where objects were considered to be “software computers” communicating via messages with a very similar architecture to the proposed ARPANet and later Internet. The object system would be a virtual-ARPANet that would use the hardware as caches. Interobject messages would span both local and intercomputer communications. Locality and migration could be used for efficient interobject coordination and for load balancing.

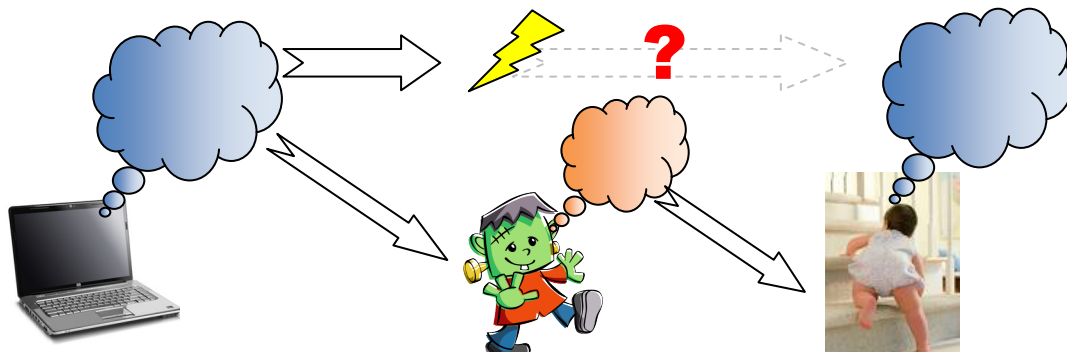
In the current Frank, only part of this architecture is implemented so far – basically enough to test the overall design, and to unify and subsume the ideas of files, documents, web-pages, etc. In Frank, there is one way to find things (they may be cached locally or be stored on other computers), and these “things” are self-contained encapsulated virtual computers whose contents need not be understood by Frank, but which can be used by a Frank user, can generate views which Frank can integrate on its desktop, etc.

Design Approach

This time around, in contrast to some of our earlier systems, we decided to try to work with “compelling examples” from the bottom-up. The examples give rise to mathematical relationships that model the meanings as elegantly as possible, and then we design and build a programming language – “runable math” – that is as much like the math as possible, but which will run without having to state much beyond the math.

The tradeoffs are interesting. It is really liberating to choose and invent paradigms rather than having to use any particular one as a given. The dues for this freedom are that one is trying to solve problems *and* invent and maintain new programming languages *and* make everything run well *and* integrate with the other ways of doing things. We feel this was a very good choice for this project: getting more fluent in quickly making well designed languages that can be real workhorses has been the key to success so far.

Frank is the next level of architecture for this approach. We were not sure exactly what loose coupling scheme would work the best (or even if we knew one that would work the best), but STEPS' small size made it feasible to suture together a Frank, and then try to see what a more elegant integration scheme might be. This constitutes a level of recursion in STEPS, where we sometimes have to implement needed behaviors in order to have enough “phenomena” to guide more elegant mathematical approaches.



Frank, coupled with the larger scale Internet wide architecture, is likely to yield real dividends. We expect that this larger perspective will allow us to replace “the sutures” with a language which “will receive messages but not have to send them”.

The Parts of Frank

Oversimplifying (and not mentioning scaffolding that will be eventually removed), Frank is made from the following main parts:

1. *DBjr* — A universal media system inspired in part by Hypercard and Etoys. It serves as the user interface and media: using, making, saving, sending, finding, etc.
2. *LWorld* — The raw materials to make user-usable systems like *DBjr*. Includes viewing, event-handling, scheduling, universal graphic objects, etc.
3. *Nile/Gezira* — a universal graphics (and sound) processing system for massively parallel processing to render and composite graphics and sound.
4. *NotSqueak* — a simple workhorse object-oriented system used for making facilities such as *LWorld*.
5. *Networking* — includes facilities for including the Internet as part of the STEPS resources
6. *OMeta* — a versatile metalanguage translator for matching and transforming many kinds of patterns. Used for making all of the language systems in STEPS
7. *Nothing* — a “higher level language with only low level semantics”. This is a highly portable, efficient universal target for all of the other systems. All machine code generated in STEPS is generated “from nothing”.

Nile is a little like a “parallel stream-based APL”. *NotSqueak* is a little like a “Smalltalk with publish and subscribe”. *OMeta* is a little like a “BNF that transforms”. *Nothing* is a little like a “symbolic computer”. All the languages are made from *OMeta*, and will be translated into *Nothing*, whose back-end creates machine codes.

We will start with what the end-user deals with—the user interface and personal computing “applications”—and work our way through the current Frank systems organization. Finally we will summarize some of the many tasks that remain to be done.

1. *DBjr*

We have made the user experience and “user illusion” to be highly similar to what most users think of as “personal computing”. One of the main changes is that unnecessary differences between applications that are almost the same (e.g. such as a “document” and a “presentation”) have been removed as much as possible. In most personal computing systems, the kind of document you can make in email is different from the kind you can make in the web browser is different from the kind of document you can make in the presentation system is different from the kind of document you can make in word processing is different from the kind of document you can make in a desktop publishing app. Besides being annoying, each one of these systems has a somewhat different user interface to learn. In Frank all these documents are one kind of document, and these can be “emailed” to others, “posted” on the Internet, printed as books and brochures, used in presentations, etc.

The document styles we’ve made so far include:

1. Desktop Publishing
2. Presentations
3. “Email”
4. “Web Pages”
5. “Cells” and “Sheets”
6. Scripting
7. Data Base

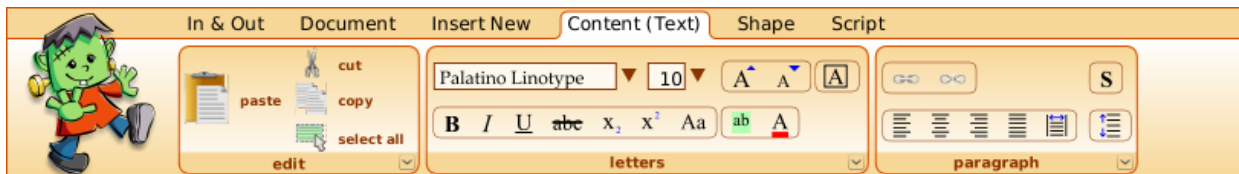
Examples of these will be given in Section 1.b

Backgrounds

We have extended Hypercard's idea of "backgrounds" and DTP's ideas about "master pages" to every object in Frank. For documents, this allows a "mood" to be chosen from existing backgrounds for a particular purpose, from something as simple as personalized stationery to making a presentation or a "web" page.



But this is also used to make Frank's user interface. For example, the menu bar is made from a "page" and the style, coloration, placement of the buttons and readouts are all part of the "background".



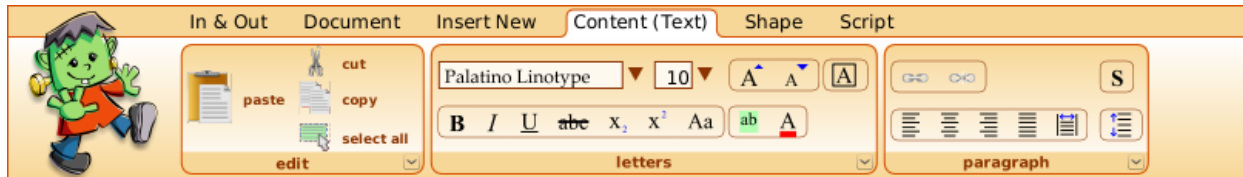
The "slots to be filled" in a background are processes that work like spreadsheet cells: they actively "look around" in their environment to find likely objects to bind to. For example, the cells with "Palatino Linotype" and size "10" are looking for "information like this" from the current selection. This is a generalization of the "Models and Views" architecture from Xerox PARC. Similarly the "buttons" are looking for the corresponding functional descriptions to fire off when pressed.

More explanation of how this works will be found further ahead in this report.

“User interface design is the little things –
the hundreds and hundreds of little things”
—Cliff Shaw
The first great UI designer

1.a DBjr User Interface Approach

User interface design is what makes personal computing work as well as it does for billions of users, but despite relative successes it is far from being understood and presented in as simple and comprehensive a manner as needed. The STEPS project does not have to be original in any aspect of look and feel, but we have generally taken the tack that being like traditional personal computing UIs when possible will help comparisons and judgments about how much ground we’ve covered, and each time we can do this we save ourselves a lot of effort in trying to come up with better solutions.

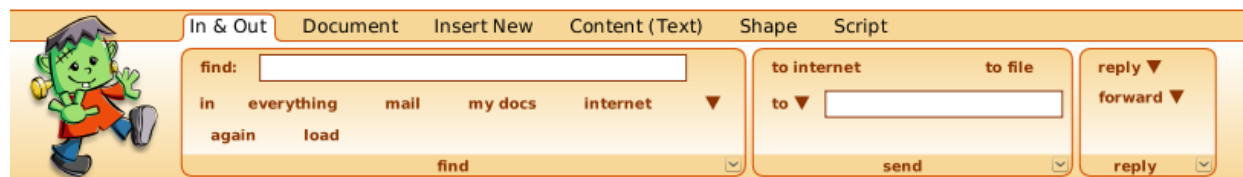


Universal menu bar with visible full width “pane” holding “bubbles”. The shaded area at the bottom is the “spill” for more commands

Our current scheme draws from designs at PARC, Mac, MS, Etoys, plus a few new ideas. For example, we use text editing, viewing, windowing, icons, pop-ups and pull-downs, etc., conventions from PARC; a top of the screen menu bar idea from the Mac, a visible “menu panel” idea used in MS Office 2007, and the “category” and “drag and drop menu item ‘tiles’ to scripts” used in Etoys and Scratch. This UI tries to make as many options visible as possible, and so is aimed at the casual (but daily) end-user.

We’ve stayed with many elementary conventions (e.g. those used for editing text) even though there might be better approaches. We have simplified and eliminated many “almost but not quite the same” problems by going after a universal document solution to standard personal computing media. On the one hand, the burden of UI is lessened because there is just one “idea of document” with the very same properties no matter how a document is used.

On the other hand, this increases the choices and also the possibilities for confusion. For example, in MS Word one doesn’t have to worry about using a Word doc as an email doc with the increased UI for dealing with emails, etc. However, if we could work out a simple unified scheme for each “mood” (no “modes”) of document use, then there would be one set of conventions for “saving and sending” and one set of conventions for “finding and fetching” which would eliminate currently separate modes and categories for files, folders, emails, web pages, etc.



“Finding” includes searching “files”, emails, the Internet, etc. “Sending” includes “saving”, sending emails, posting “web pages”.

The first time around at Xerox PARC, inventing a comprehensive UI that actually worked for end-users required by far the most effort and experimentation, and we have little doubt that we will still be tinkering with the STEPS approach to UI as our research period comes to an end. So far we have found that a judicious combination of approaches from the original PARC UI, the more recent Etoys UI, and an “office suite redux” UI is working pretty well in providing comprehensive coverage for all the “moods” without adding much burden to any of them, while in the process removing many of the “almost but not quite” confusions in current day systems.

General Approach To The STEPS UI

1. Browsing and icons – basically: “browsing is a fast way to deal with remembered paths” and “icons are often more memorable and easier to spot than words”. The learning curve for this breaks down if there are more than 100 visible items, or if most of the items are hidden below menu levels. This is the case for this UI. So considerable auxiliary work has to be done in the design to allow it to be used by beginners until the paths and controls become more memorable.

Some of the “how do I ...?” questions can be answered by exploring and trying. Some are answered by using conventions that are like most personal computing systems (e.g. for copying, cutting and pasting, icons for formatting, etc.) However, we really want the user to be able to ask questions like “How can I send (etc) email?” and have the gist of the answer brought to them. Many of these niceties are part of the goals for next year.

2. Universal Document/Media – this has worked out very well with Etoys, and is gestured at in MS and Open Office (they don’t really do it, but try to have the UIs for their different document types be similar). We have found that a combination of redesigning the Etoys tools UI and organizing it sideways and redesigning the MSO 2007 scheme (which exhibits some similarities, and is kind of like a Mac UI with full width sticky pulldowns), covers most of the document “moods” that are needed.

We are experimenting with having every aspect of the user experience be in terms of the universal document idea (including “moods” such as “desktop” and “windows”). The actual underlying system is built this way, but sometimes having too much uniformity (e.g. LISP) adds confusion and “noise” rather than helping. We are interested to see how this works out.

3. Use “video player” approach to getting/hiding tools. Personal computer users are used to showing their videos full screen but having a “casual mode” switch to showing tools via clicking on the screen or moving the mouse up to the top of the screen to show a tool bar with a slightly smaller window for the video. This is different from the much stronger mode in (say) PowerPoint, which sharply distinguishes between “presentation” and “construction”. (One could imagine an iPad gesture for this which distinguishes between casual pan and zoom of the content, and a contraction of the window view the content is in.) Another wrinkle is that the “tools casual mode” could be an excellent default for beginners (as it usually is when people first start playing videos on a PC).
4. “No modes” – meaning that “authoring is always on” but perhaps “with fences”. I.e. we have found with Etoys that always having the ability to e.g. move things around, get viewers, etc., is almost always a good thing, and that fences – such as “avoid being picked up” are sufficient to keep the content stable even while allowing it to be edited.
5. Universal UNDO – we use the “Worlds” mechanism to provide not just comprehensive UNDO of all steps, but to also deal with multiple versions in a variety of ways.
6. Halos – are simpler than in Etoys and provide mostly graphical handles (resizing, corners, rotation, etc.) The automatic “menu panels” at the top of the desktop are used for the rest of the properties.
7. “Menu Panels” – both “full page” and “individual object” menus kept visible.
8. “Most used” plus “escapes” – this allows a lot of work to be done at the first level of UI, but provides full access to the (usually too many) features of any modern personal computer application.
9. Contextualized help – is an attempt to do this at least a little better than most current systems do today – a much better version of this should be part of personal computing, but we don’t know of one.

1.b DBjr: Some of the document styles we've made

1. Desktop Publishing

Here is the “full screen” view of the previous “Page 4” example in the DBjr interface and media system.

rently we include these optimizations in our code counts for simplicity, but technically they are not part of the 20,000 lines of meaning.

Thus one measure will be what did and did not get accomplished by the end of the project with the 20,000 lines budget. Another measure will be typical lines of code ratios compared to existing systems. We shoot for factors of 100, 1000, and perhaps the next order of magnitude as well. The basic idea here is that we are interested in very large qualitative comparisons, not small ones.

Another measure is understandability. Are the designs and their code clear as well as small? Can the system be used as a live example of how to do this art? Is it clear enough to bring to mind other, perhaps better, approaches?

Previous STEPS Results

The first three years were devoted to making much smaller, simpler, and more readable versions of many of the prime parts of personal computing, including: graphics and sound, viewing/windowing, UIs, text, composition, cells, TCP/IP, etc. These have turned out well (they are chronicled in previous NSF reports and in our papers and memos).

For example, essentially all of standard personal computing graphics can be created from scratch in the Nile language in a little more than 300 lines of code. Nile itself can be made in a little over 100 lines of code in the OMeta metalanguage, and optimized to run acceptably in real-time (also in OMeta) in another 700 lines. OMeta can be made in itself and optimized in about 100 lines of code.

The diagram illustrates the Nile translators. It shows an input stream of Nile programs for making graphics entering a 'Parser For Nile' box. This box contains gears and is supported by 'OMetarules for Nile'. The output is Nile ASTs, which enter a 'Transform ASTs' box. This box also contains gears and is supported by 'OMetarules for ASTs'. The final output stream includes JavaScript, C, and CoM.

The two OMeta translators for Nile (as of 2009)

The diagram illustrates the OMeta translators. It shows an input stream of OMeta in OMeta entering a 'Parser For Ometa' box. This box contains gears and is supported by 'OMetarules for OMeta'. The output is OMeta ASTs, which enter a 'Transform ASTs' box. This box also contains gears and is supported by 'OMetarules for OMeta ASTs'. The final output stream includes JavaScript, C, Squeak, and CoM.

The two OMeta translators for OMeta (as of 2009)

The results so far show that many individual parts of personal computing can be created and run pragmatically in smaller amounts of more expressive program code (roughly factors of 100 to 1000 less code).

4

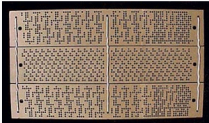
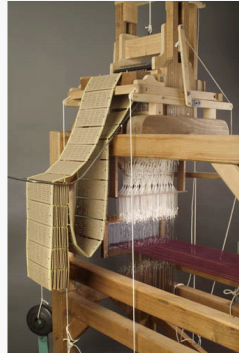
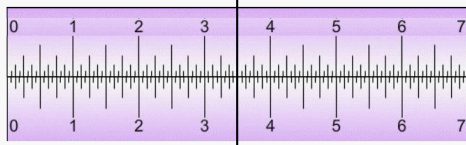
2. Presentations

Making a presentation in STEPS is “not a mode, but a *mood*”. In other words, much of it is a subset of making a general document with less connected writing, but making use of dynamic “builds” or “appearances” of individual objects on each page, and of page turns. So we make a “slide” in the very same way we make a “page” (they are the same).

The basic user action in giving a presentation is “Next”, and this will turn the page or invoke “builds” and other effects. The latter in most presentation systems are usually fixed features, but in DBjr we use its more general scripting system to allow a much wider variety of things that can be done during a presentation. The general scripting approach is discussed in more detail below. Here, we will just gist how this is used and how it looks when used for presentations.

Computing is Transforming

Programming is making a machine into another machine by changing relationships



Here is a typical “slide” in a talk. The elements were brought in one by one, and some of them (such as the titles) were changed on the fly. The “addition slide rule” is dynamic and can be moved by fingers or the mouse during the talk

The screenshot shows a presentation software interface. The main slide area displays the same content as the previous image: the title "Computing is ... ?", the subtitle "Programming is making a machine into another machine by changing relationships", the ruler with waveform, the stack of punched cards, the loom images, and the Jacquard punchcard. The interface includes a menu bar at the top with options like "In & Out", "Document", "Insert New", "Content (Text)", "Shape", and "Script". A toolbar below the menu bar contains icons for "paste", "cut", "copy", "select all", "edit", "letters", and "paragraph". On the right side, there is a "Simple UI Panel for STEPS" with a list of elements and their actions:

- start
- click topRuler show
- click textBox1 show
- textBox hide
- click loomwork show
- jacquardpunchcard show
- click jacquardcards show
- click jacquardmemoire show

At the bottom right of the panel, there are "go" and "stop" buttons. The slide area shows a zoom level of 51%.

The slide in “menu mood”. This is a “mood” because the slide elements are live during the talk. So the change of moods is more like showing a video and moving from full screen to showing menus without having to stop or click. The “custom animation” for showing the slide elements in sequence is effected by using the general scripting system of Frank. The script is shown in the right hand pane.

One of the old time mantras for personal computing is “simple things should be simple, complex things should be possible”, so the design task here is to not make things more complex than what the user is already used to. On the other hand, this is a great way to get end-users to write simple programs without even realizing that they are starting to learn deeper powers of computing.

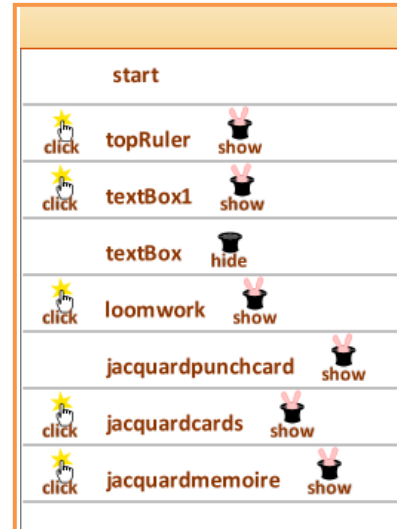
The “build sequence” (sometimes called “custom animation”) is just a script that is single stepped by a “next” input from the presenter (often a mouse click or space bar push).

The script lines fit very well with the tile design of our scripting system. Each tile line has 4 sections:

condition to trigger—object(s) affected—action—modifier

So a typical script for building the previous slide example during a presentation would look like this:

And it would be created by clicking on or dragging out tiles presented in the menu bar and panel.

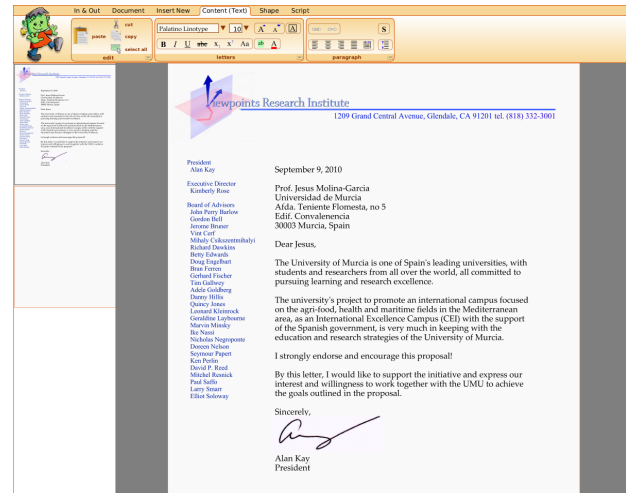


3. “Email”

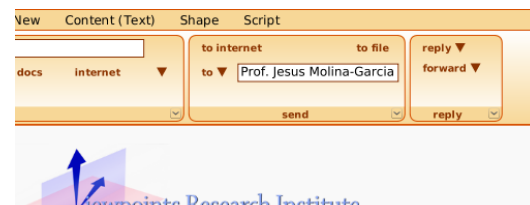
The “email” system in Frank is really a service for sending and receiving any kind of documents so it is essentially the same as a web search which looks for search keys of the form “To: the user’s name or id”.



The “letter”



Composing the “letter”



Sending the “letter” as “email”

Since any Frank document is “sendable” directly as email or a “web page” we can make graphical letterhead stationery and use it directly to write and send a letter as an email, or it can be directly posted to the “web”.

4. "Web Pages"

The web should be a service for locating and making available objects which can be used in a safe manner. Without taking space to criticize the existing poor design of the web, we have made an alternate web that supplies more in a safer fashion. The basic problem of approach of the existing web is that the creators thought web browsers were a kind of application, and that web content would be simple content. A more fruitful way to look at the first order metes and bounds of a "world wide web" is to think of it as a system of objects done by many people that somehow has to be run safely and coordinated with other content on one's personal computer. This makes it an "Operating System 101" problem (since the main job of a personal computer operating system is to take unknown content from outside, run it safely, and allow its outputs (mostly visual and sound) to be coordinated, composited, and accessed in a legible and usable manner.

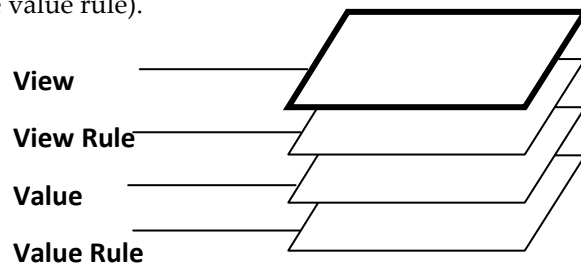
We defer more discussion of this architectural issue and solution to later in this report. For now, we can just say that the equivalent of web content in STEPS is actually treated as an encapsulated virtual machine on the Internet, and is run in a confined manner. Though it can use STEPS tools, etc., it is also possible for the content to have all its own tools, including how it does graphics, and sound. In this case, the process is simply given user inputs, and a bit-map, etc., to create output on. This output is then taken by the DBjr UI and coordinated with other views on the display.

For STEPS content, the universal document authoring allows every kind of existing web formatting to be used, but also allows formatting that is beyond current conventions and HTML.

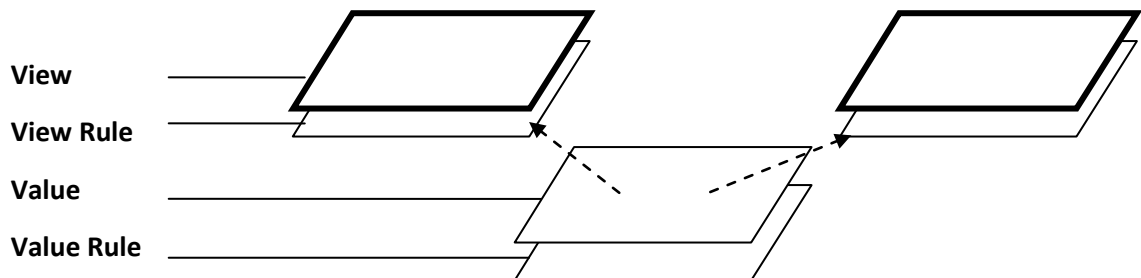
5. "Cells and Sheets"

Standard "productivity suites" include "spreadsheets" but in a very weak form that has not been materially improved commercially since the commercial introduction of this very powerful concept. The Xerox "Analyst" was made in the early 80s for the CIA and NSA by Xerox PARC/XSIS, but the many important ideas in it were ignored by the mainstream when it was later offered on the market. Some of the Analyst ideas and new forms of programming were discussed [2] in the early 80s, and a children's system [3] using some of these ideas was made a few years later.


The key idea of a spreadsheet cell is to unify computation with display. We can see that a "model" and its "view" in a Model-View-Controller could be expressed in spreadsheet cell form. The Analyst was the first spreadsheet design to allow arbitrary models and views in a cell. We can get still more power by loosely coupling the view to the model (one way could be through using search or publish and subscribe mechanisms in the value rule).



In STEPs, a value can have more than one view.




The simplest use of the generalized cells in STEPS is to make an extended spreadsheet++, where the “++” includes arbitrary expressions for the value rule e.g. the value rule for cell B6 will cause it to change color towards red as the expenses are deemed too high. The value rule of cell C6 shows a “thumbs up” image if the expenses are less than \$300.

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

C6 = self graphicFrom: ((C5 value < 300) ifTrue: ['thumbUp'] ifFalse: ['thumbDown'])

6		
---	---	--

A6 = self graphicFrom: 'frank'

	A	B	C
1	date	item	amount
2	April 23, 2010	air fare, southwest	198
3	April 23, 2010	taxi, BUR to AMI	28
4	April 24, 2010	parking, SJC	300
5	-----	TOTAL	527
6			

C6 = self graphicFrom: ((C5 value < 300) ifTrue: ['thumbUp'] ifFalse: ['thumbDown'])

Since anything can be used as the value rule for a cell, we can even put in a problem solving system (such as the Toylog English language version of Prolog discussed in the 2007 report).

But we also use these mechanisms everywhere. For example, the STEPS menus are made from *pages* whose *background* includes *buttons*, which are *views* which “look” for the actual methods that are part of the underlying object. Thus when a particular object is clicked on, it becomes the *purview* of the UI menu system which forms loose couplings between the menu items and the underlying mechanisms.

The retrieval metaphor that characterizes spreadsheet cells can be used to *find* things and *present views* of those things. This is used in windowing, retrieving from the Internet, etc. For example, the “email” *inbox* is a spreadsheet cell whose value rule is a pattern that will only match on the end-user’s mail documents, and whose formatting rule is (usually) “list”.

Generalized spreadsheet cells and ways to program them are wide and varied. This year we have mainly constructed the general functionality and a few uses. A comprehensive user interface for scripting in general and especially for allowing end-users to make wider use of what cells can do is a goal for 2011.

6. Scripting

Right now this is still a “lick and a promise” (the rest is planned for 2011), but we have done part of the scripting UI design. This is inspired by the two children’s scripting systems built in Squeak – Etoys and Scratch – and by several Hypercard ideas. We have been experimenting with tile-based scripting since our MIT graduate student Mike Travers created Agar, an agent-based, tile-scripted, authoring system for making behavior creatures for the Vivarium project [4]. These ideas were adapted for Etoys in 1997, and then adopted by Alice, and later by Scratch. This has generated considerable end-user experience in the strengths and weaknesses of tile-based scripting systems, and we have tried to maximize the former and minimize the latter in this design.

The basic idea here is that many of the properties and behaviors to be dealt with when scripting already exist in the media part of the UI. For example, the location of a particular object in a document or on the desktop, or to change the font size or insert a new object or text into text, etc. This can just be dragged out of the general UI onto the desktop or into scripts.

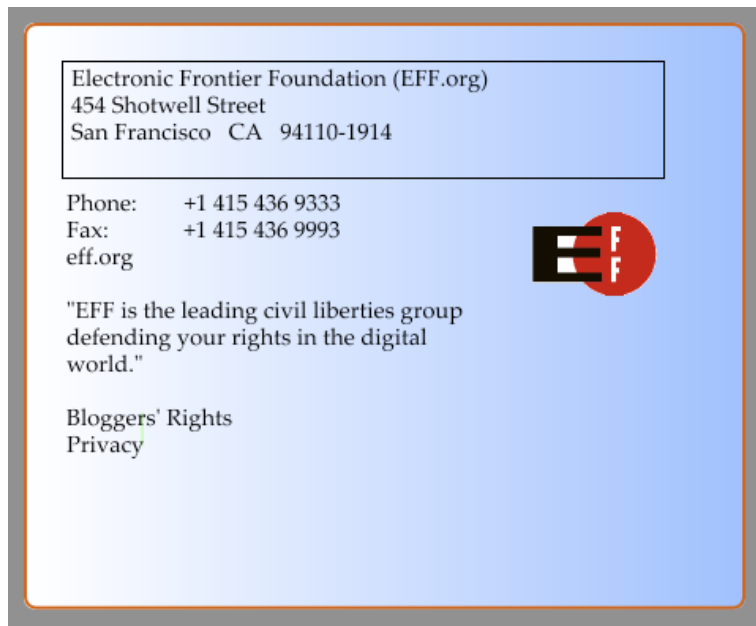
Other scripting objects are dragged from several scripting panels that are incorporated into the general menu interface. The visibility of the scripting elements is a little more like the original Etoys interface (and like the Scratch interface today). For fun, we have made some Scratch-like tiles to use in this example. We expect next year to design tiles that have much of the same tactile impact when picking, dragging and dropping, but are much easier to read when conglomered in an actual script.

Scripting is seen out of the corner of the user's eye in the presentation interface but also in the feedback pane, which keeps a log of what the user is doing as scripting actions. The basic idea here is that quite a bit of scripting is a mechanization of actions the user can do, and we endeavor to show these in scripting terms, and in a way they can be used as scripting elements.

Tile-based scripting is at its best for beginners and small scripts. As a script gets larger, the extra visual cues that help beginners start to compete with being able to "gist-read" the script for general content. Also, a more fluent scriptwriter can often type faster than finding, dragging and dropping. This leads to a number of important secondary designs for scripting, that accommodate to the learning of the user.

7. Data Base

As in Hypercard, every page (and in STEPS, every document) is automatically indexed for retrieval. So every multipage document is already a kind of database. The use of backgrounds, properties and spreadsheets/cells allow most kinds of enduser databases to be made. (We can also see that this is completely compatible with the idea of "web pages".)

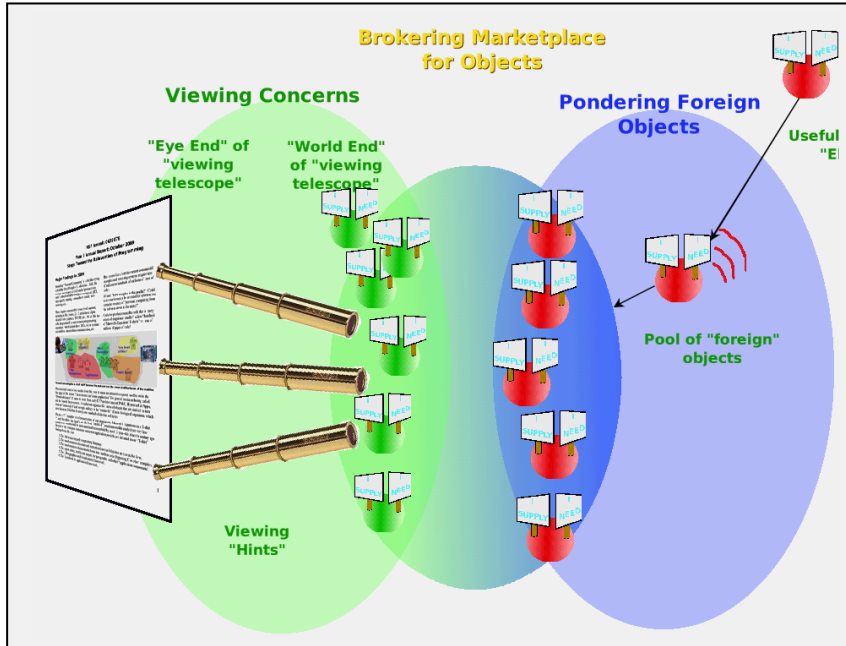


A simple Frank data base that acts like a Rolodex

2. LWorld

LWorld is the tools framework used to build application and user interface frameworks such as DBjr.

The base object model uses fully reified Announcements for events-notification and the ``components`` for the entry object, as described in the previous year's report.



The basic LWorld scheme of loosely coupled interactions and viewing of independent objects

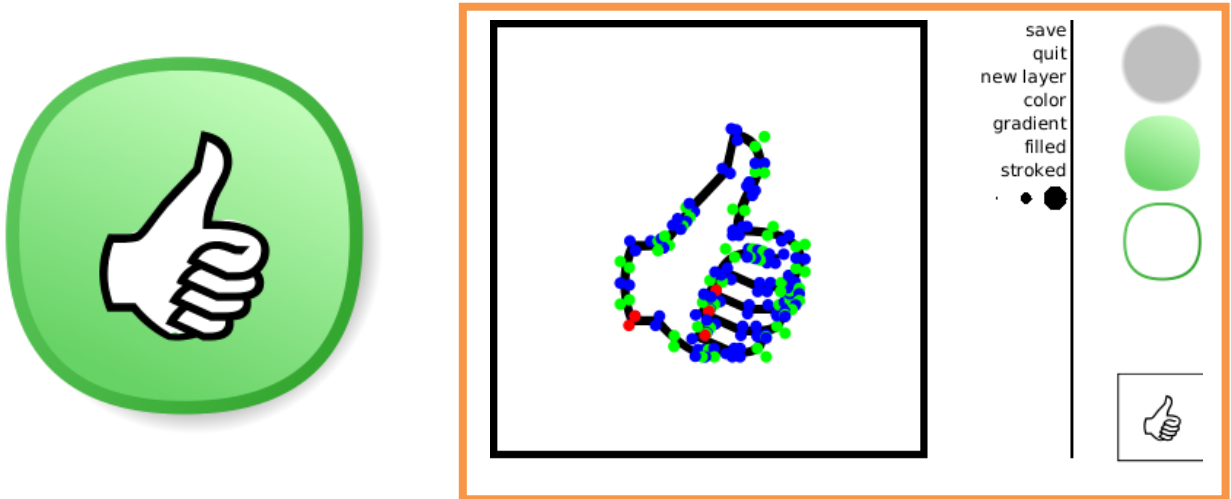
The LWorld framework provides the basis of the event notification and the LBox framework is the graphical application framework on top of it.

One of the major milestones achieved is the full-integration with the Gezira graphics engine. The graphical aspect of widgets (called Shapes) in the LBox uses the geometry, fill and stroke data that is used by the Gezira engine. The rendering is done by constructing Gezira pipelines for Boxes in the display tree and feed the data through the pipelines.

In 2010 the framework has been further extended with new tools:

- The bitmap editor (described in VPRI TR-2010-001 [8]) that explores the practicality of Worlds in an application with large data.
- Spreadsheets (see the description above). These take advantage of the Announcements event-mechanism to propagate the recalculation requests. The cells of the table can contain various data types and expressions.
- Inspectors. An inspector is just the table with cells that are waiting on the target objects particular value change events. Since the visual representations of cells are just regular LBoxes and are not restricted to the table-like setting, we assembled them with explanation text to make a free-form dynamic essay on the object.
- A document reader. The EPub parser is written in OMeta and the reader displays the contents

- A Shape editor. The Gezira data in a Shape can be interactively edited. Here is how the “Thumbs Up” in the spreadsheet example was made.



The “Thumbs Up” icon, and how it is made in the Shape Editor

3. Gezira/ Nile

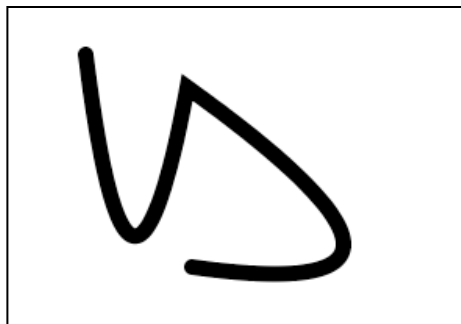
Gezira is the STEPS graphics system, and Nile is the “runable math” programming language we invented to express the relationships compactly and run them quickly enough to be useful (see the previous 2009 report to NSF for more details).

Much of the 2010 work with Gezira/Nile has been to complete, reengineer, and integrate.

- This year Gezira/Nile was operationally integrated with the rest of the STEPS system, and is used for all graphical operations in Frank.
- The meta-compilation of all of the Nile code via OMeta was completed.
- Considerable work was done to integrate the Nile facilities into “NotSqueak” so they could be used for all the viewing and rendering needs of DBjr and LWorlds.

These are all illustrated in the earlier part of this document. A few new features were added as well:

- Pen stroking was extended to include different joins and cap types.

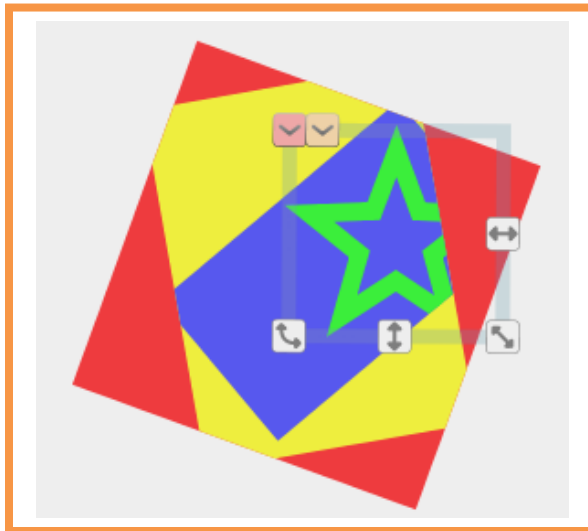


- Bilinear and bicubic transformations were added to provide very high quality image (bitmap) scaling. “Mipmapping” was added to make extreme reductions (e.g. for thumbnails) more legible. Examples of thumbnailing can be seen on pages 5, 12, and 14.

- A Gaussian filter was added, especially to make a wide variety of useful soft shadows.



- A new kind of fill that is “transparent and contrastive” has been devised. An example goal here is to be able to render images over any background and have them be readable without occluding the background. (This example doesn’t transfer to MS Word very well.)



- An experimental trial of massively parallel processing for Nile is in progress.

As with other stream processing languages, Nile programs are implicitly concurrent. Most Nile processes (previously called "kernels") execute in complete isolation from each other, interacting exclusively through input and output streams (channels). In addition, the production and processing of stream data

can happen piecemeal. This strictly defined encapsulation and communication lead naturally to parallel execution of Nile processes. But besides pipeline-level parallelism, Nile programs also lend themselves to data-level parallelism. In practice, Nile processes are often stateless. This allows the duplication of processes to handle different sections of the same stream in parallel.

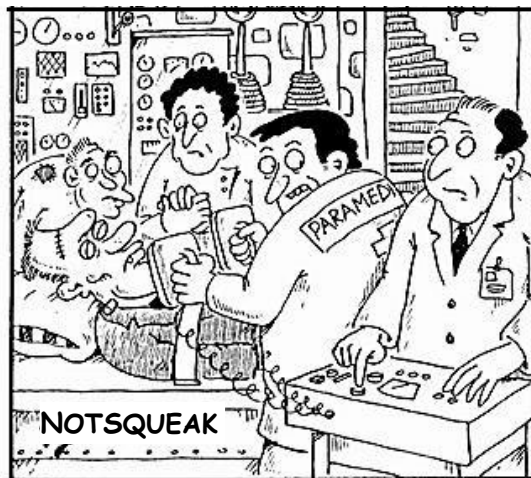
We are investigating parallel execution of Nile programs on commodity multi-core processors. The Nile scheduler is currently being adapted to assign processes to hardware threads. The threads communicate through stream buffers that, once filled, are immediately delivered to a process' "inbox." Once a process has at least one buffer of input, it is put on a ready queue to be executed by an available thread. This implements pipeline-level parallelism. Data-level parallelism is provided by duplicating stateless processes when a process' inbox becomes "too full." Furthermore, we will support simultaneous execution of multiple pipelines, which adds an additional axis of parallelism.

We plan to use our 2D vector graphics renderer as a real world benchmark for our parallelization efforts. We are optimistic that, given the many deep pipelines and data-parallel processes in the Gezira code base, there will be much room for parallel execution.

4. NotSqueak

This is a considerable subset of Squeak Smalltalk (without the Squeak libraries and most features) used as a placeholder object-oriented language to make frameworks for the graphics and document system. Part of the design of the subset has been done to facilitate bootstrapping the entire top part of Frank over to the new foundations which are being built in "Nothing" (see section 7 ahead). Another perspective on the subset was to make it possible to replace the "announcements" loose coupling used in LWorlds with a more general and comprehensive framework when we understand it better.

NotSqueak's connection to Squeak has allowed an "intensive care unit" to be made for Frank which allows us to make Frank objects, use the Frank User Interface, employ Frank graphics, etc., without having to do a complete bootstrap from scratch. This has allowed quite a bit of progress to be made on the design.



"Clear!"

We expect that this scaffolding language will be completely replaced by the end of next year.

5. Networking

STEPS is about the code required to manifest personal computing on a personal computer, and this is the code that we count. But personal computers are usually connected to networks including the Internet. Thus we have to make a TCP/IP and other systems to communicate with Internet services. We also need

to supply services that are like those available via the Internet (such as the world wide web).

The Web Should Be About Objects (And Especially The Same Objects As Personal Computing)

The web was not thought through as a systems design. DTP already existed with symmetric WYSIWYG authoring, and related models for hypermedia – such as Hypercard – were already used extensively. The LOCUS [5] distributed operating system had demonstrated how to make up computations as combinations of virtual machines and to successfully distribute them to multiple heterogeneous platforms with automatic load balancing, so that computer hardware on the Internet could now simply be used as caches for a “virtual Internet” of intercommunicating virtual machines.

Once looked at this way, we can readily see that the web browser which is currently an enormous mish-mash of weak legacy standards approached as an application, could have been “almost nothing” and much more extensible and powerful if it could have been seen as a kind of operating system, whose job it is to safely run processes made by others, and to combine and compose the outputs from these processes at the user level, and to transmit user commands back to them. In other words, the browser should have been the exact opposite of what it is now.

Similarly, the whole web media could be easily made and more easily authored if it were just the super-DTP/GUI mentioned above. And, the comprehensive viewing mechanism and distributed process scheme allows any new media to be invented and integrated (it doesn’t have to be “mashed”!).

“Operating Systems” Should Only Be About Allocating Resources Safely To Cached Objects

Finally, we can see from the above that much of what is in currently operating systems should not be there. A more “Internet perspective” will encourage distribution and the integration of heterogeneous elements rather than trying to make a conglomeration of features that is more difficult to integrate.

For example, there is no reason for an OS to contain device drivers - there are too many devices, and all have their own resources. What is needed is for the devices to furnish their own drivers in a way that can be handled universally.

There is no reason for an OS to supply a graphics system. This can be distributed to the virtual machines on the virtual Internet. The “OS” needs to coordinate the results of the distributed loci of functionality, it doesn’t need to supply resources for them.

What was a mega-monolith quickly sublimates away, and we wind up with a simple plan to run hardware only as a cache for encapsulated message exchanging processes (otherwise known as “real objects”), some of which are processes that can coordinate views generated by other processes. We do load balancing (and this can be seen as a next level of semantics of TCP/IP, and part of the same family tree).

This approach gets rid of a lot of needless stuff, but it only helps part way for the large spectrum of functional “applications” needs and possibilities in personal computing. In our project, we don’t have to cope with this much because STEPS is primarily aimed at the standard productivity suite software – which is pretty much about making and sending and receiving documents in various ways – and not at the hundreds of thousands of applications that have other aims and are not so document oriented. Not a lot of functionality needs to be added to the basic “universal documents” to make them behave as word processors, DTPs, presentations, graphics editors, email clients, web browsers, spreadsheets, etc., or any combination. (A separate attack on the “general application making problem” would be a good subject for a follow-up research project.)

“Architecture Dominates Materials” – “Design Wins”

So, this part of STEPS is not particularly new, but is simply a return to past design insights and principles—some more than 40 years old. However, we think these “findings” are still significant results, partly because STEPS is a “science of design” project, but also as a way of (re)calling attention to the need

for better more comprehensive design thinking before plunging in willynilly to write random code and create ungainly non-scalable *defacto* standards that serve mostly as barriers to progress.

“File Systems”

This year we have implemented enough of the “external network environment” to handle our own filing, searching, email, and “web” needs. Next year we will implement the “protected address space” architecture that will potentially allow any object done in any fashion to be cached and run safely under STEPS.

6. OMeta

OMeta has been the workhorse of STEPS, and is currently used to make the STEPS languages: Nile, NotSqueak, Nothing, and itself. It runs on top of Squeak, Javascript, C, and Nothing.

We have not needed to improve it this year, and the main new deployment has been to use it to help make Nothing (see the next section 7., and Appendix 1), and then bootstrap itself into Nothing (Appendix 2 shows a copying garbage collector that is part of this process). The next task will be to use OMeta to move Nile from its current main target C to use Nothing as its high performance back end.

“Hey Alex, what did you make progress on today?”

“Oh, Nothing”

“Wow, that’s really great!”

7. Nothing¹

“Nothing” is a *symbolic computer* that is used as the lowest level target for all other “machine code” generation in STEPS. (“Machine code” here refers to actual machine code for various CPUs, and, for convenience, we occasionally translate to Smalltalk or C.)

It can be thought of as a higher level language with only low level features—to be looked at by humans but not to be programmed directly (perhaps occasionally by *superhumans*).

One of the reasons we decided to insert this extra layer was pedagogical; it provides a “hardware semantics” that will make it much easier for students to trace the chain of meaning down to computer hardware. The semantics of Nothing are simple, and the translations from Nothing to actual hardware are also simple enough to allow a backend to be readily understood by a student, and to allow a new backend to be created by a student.

One pedagogical illustration that we wanted to be very clear is how one can bootstrap from a lower level of expression to a higher level (in terms of bricks, going from making walls to making arches). By having Nothing be “almost nothing”, it is easy to show how nicely “bootstrapping upwards” can be done.

We were inspired by two “high level low level” designs from the 1960s: BCPL [6] by Martin Richards, and by the “elementary notation for algorithms” devised by Niklaus Wirth for implementing and describing his Euler language [7]. Syntactically Nothing is a little closer to the former, and semantically is a little closer to the latter.

The grammar for transforming Nothing into ASTs is a very simple 150 lines of OMeta (see **Appendix 1**).

An example Nothing program to make a copying garbage collector in less than 100 lines of code is shown in **Appendix 2**.

Each back-end from ASTs to final target is also implemented in OMeta.

¹ It was originally called *Ex Nihilo*, but this seemed too fancy for this project

OMeta is implemented on top of many different languages (Smalltalk, JavaScript, etc.) But in the final STEPS, it is implemented in terms of Nothing, and this also provides a nice case study of bootstrapping because it is recursive and requires a virtual machine with a stack. We had a lot of fun doing this because it echoes one of our favorite papers of all time: Meta II by Val Shorre in 1964, who pioneered this style of translation, and in his short paper was able to provide three complete working examples – two Algol-like languages and Meta II itself, and the virtual machine for Meta II, all done on an 8K 1401!

One of the many benefits of this approach is that it allows us to experiment with “the bottoms” even as we are implementing and experimenting with “the top”. We can produce different virtual machines for the various DSLs we have created to try to understand the most fruitful ways to run them efficiently (but simply). Eventually this will lead (perhaps) to a more consolidated design for the VMs.

The next project is to replace the back-end of the Nile/Gezira graphics engine (which now has both C and Smalltalk back-ends) with a Nothing back-end, to completely remove all code but ours from the Nile “chain of meaning.”

2010 STEPS Experiments and Papers

Here are three experimental projects done to learn more about several important areas of the STEPS research:

- *an experimental high speed universal “speculative programming and undoing” system;*
- *a metalanguage port of some of our work to the Flash virtual machine; and*
- *a complete “chain of meaning” from high level to machine code using a single transformation language.*

Worlds: Controlling The Scope Of Side Effects [8]

Alex Warth, Yoshiki Ohshima, Ted Kaehler, Alan Kay

This paper introduces worlds, a language construct that reifies the notion of program state and enables programmers to control the scope of side effects.

The state of an imperative program—e.g., the values stored in global and local variables, arrays, and objects’ instance variables— changes as its statements are executed. These changes, or side effects, are visible globally: when one part of the program modifies an object, every other part that holds a reference to the same object (either directly or indirectly) is also affected.

We investigate this idea by extending both JavaScript and Squeak Smalltalk with support for worlds, provide examples of some of the interesting idioms this construct makes possible, and formalize the semantics of property/field lookup in the presence of worlds. We also describe an efficient implementation strategy (used in our Squeak-based prototype), and illustrate the practical benefits of worlds with two case studies.

Solutions to many problems in computing start with incomplete information and must gather more while the solution is in progress

An important class of problems have to perform *speculations* and *experiments*, often in parallel, to discover how to proceed. These include classical non-deterministic problems such as certain kinds of parsing, search and reasoning, dealing with potential and actual error conditions, doing, undoing, and redoing in user interfaces, supporting multiple forked versions of files and other structures that may need to be both ramified and retracted, etc. The “need to undo” operates at all levels of scale in computing and goes beyond simple backtracking to being able to support multiple speculative worldlines

Most of the ploys historically used to deal with “undoing” have been ad hoc and incomplete. For example, features such as try/catch enable some speculation, but only unwind the stack on failure; side effects are

not undone automatically. Programmers have little choice but to rely on error-prone idioms such as the command design pattern. This is analogous to the manual storage management mechanisms found in low-level languages (e.g., malloc and free in C). In contrast, garbage collection trades a little efficiency for enormous safety and convenience, and the *worlds* mechanism we present in this paper provides a similar service for all levels of “doing-and-undoing.” Web surfing is a useful analogy for thinking about worlds: during a simple exploration of the web, you might just use the back button, but more complex explorations (speculations) are more easily done with multiple tabs. All the changes you made during your explorations remain local to the tab that you used, and can be made “global” or not by your choice

This is somewhat similar to *transactions*, which are another example of a general mechanism that can handle some of the “computing before committing” problems at hand here. But whereas the purpose of transactions is to provide a simple model for parallel programming, the goal of worlds is to provide a clean and flexible mechanism for *controlling the scope of side effects*. Unlike transactions, worlds are first-class values and are not tied to any particular control structure—a world can be stored in a variable to be revisited at a later time. This novel combination of design properties makes worlds more general (albeit more primitive) than transactions

Tamacola - A Meta Language Kit for the Web [9]

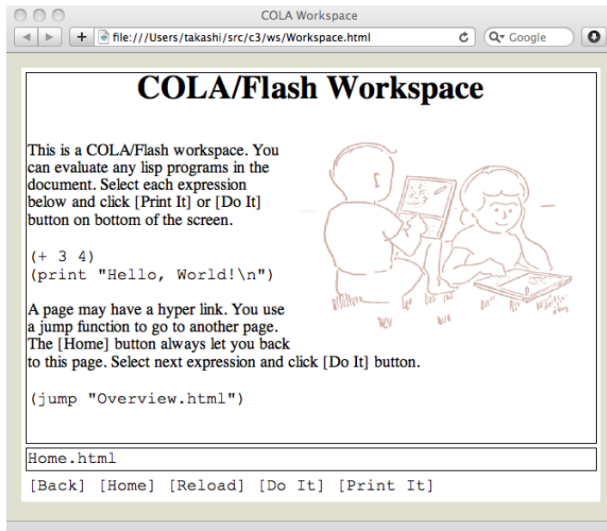
Takashi Yamamiya, Yoshiki Ohshima

Presented at the S3 Conference, Tokyo, Japan, September 2010

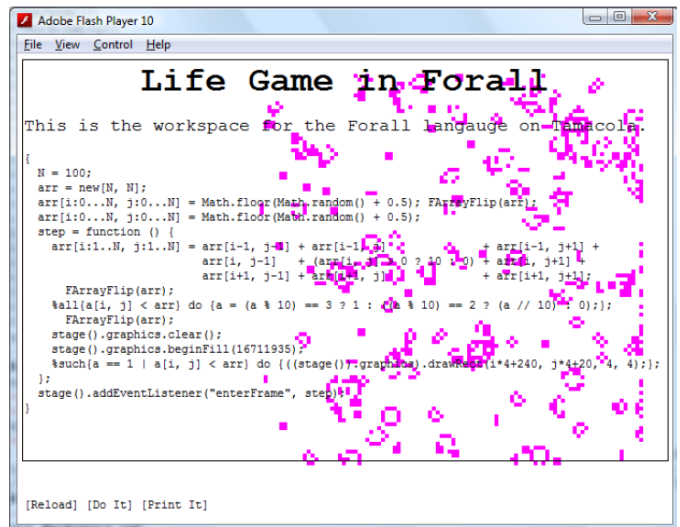
Tamacola is a dynamic, self-sustaining meta-language system grounded upon the Tamarin VM.1 Tamacola compiles a Scheme-like S-expression language into ActionScript bytecodes, and contains meta-linguistic features, such as a PEG parser generator and macro system, which make it useful for defining new languages. In fact, Tamacola is written in itself, using its meta-linguistic features.

Since the Tamarin VM can load ActionScript bytecode files to extend and replace running programs, Tamacola can extend itself and define new languages while it is running. Furthermore, since the Tamarin VM is part of the ubiquitous Adobe Flash player, this self-modification can be accomplished while running in a web browser, with no extra installation requirement

Objects in Tamacola are intimately tied to their ActionScript counterparts, providing good interoperability between Tamacola and the Flash Player. To show that the system is ready for practical use, we used Tamacola to implement both an interactive programming environment (“Workspace”) and a simple particle language



Interactive workspace for presentations and programming using the Actionscript bytecode engine



The Forall language is made in Tamicola, and then is used to program the game of “Life”

PEG-based Transformer Provides Front-, Middle- and Back-end Stages in a Simple Compiler [10]

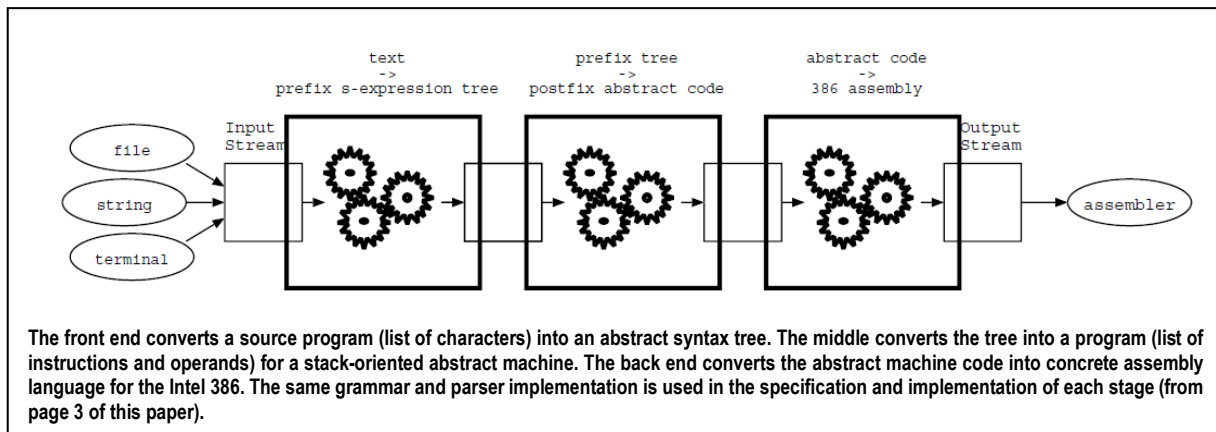
Ian Piumarta

Presented at the S3 Conference, Tokyo, Japan, September 2010

This paper describes an experiment in compiler construction that uses a single parser to implement each of the three stages of compilation: source parsing to create an AST, intermediate code generation from the AST, and machine code generation from the intermediate representation. Compilers are often broken into three (or more) stages: front-end parsing, one or more middle-end analysis/optimization steps, and back-end code generation. This separation helps assure simple and understandable transformations between adjacent stages, that are easy to construct, debug and maintain

A parser is typically generated automatically from a grammar using tools like Yacc or ANTLR. Analysis and optimization is often performed by tree rewriting—pattern matching to identify a particular subtree and replacing it with a “better” subtree. The tree rewriter can also be generated from a grammar that describes output structure (trees) generated for particular statements of an input language (patterns in a tree). Code generators can be implemented with grammar-driven bottom-up tree rewriting, a well-known example being the BURG family of code generators. Each stage usually uses a specialized grammar and parser/generator generator.

Our experiment applies a single parsing mechanism, driven by a uniform grammar to all three compilation stages.



A parser has an input stream, a set of rules (generated from an extended PEG) that recognize input structure and generate output structures, an output stream to collect generated output, and a current result (semantic value from the most recent expression) that can be read and written within rules describes the architecture of the compilation chain.

Training And Development

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

Outreach Activities

November (4 and 6) 2009, at Keio and Kyoto Universities, Japan

Alan Kay gave a keynote address to Microsoft Research Asia's "Computing in the 21st Century" Conference. "Three Screens and One Cloud: Rethinking Computing," brought together researchers and scientists to discuss how businesses and consumers take advantage of technologies with different form factors.

March 2010

Alan spoke and participated at the "Computational Thinking Workshop" in Arlington, VA with National Library of Medicine and DARPA

April 2010

Alan visited CMU and shared STEPS with Ken Koedinger (CMU) Steve Ritter from Carnegie Learning, and others at CMU.

May 2010

Alan spoke at NYU's "Games for Change" conference hosted and organized by Dr. Ken Perlin and Microsoft Corp.

June 2010

Alan visited Dr. David Patterson's lab at UC Berkeley and shared progress on STEPS.

September 2010

Kim Rose co-chaired "S3 - A Workshop on Self-Sustaining Systems" held at University of Tokyo on September 27-28.

Takashi Yamamiya, Yoshiki Ohshima and Ian Piumarta presented papers at the S3 workshop (see in this report the section "2010 STEPS Experiments and Papers" for more details).

October 2010

Alan Kay gave a lecture at UCLA undergrad computer science students describing the ideas and progress of STEPS research.

During this program year, Co-PI Daniel Ingalls moved from Sun MicoSystems to SAP Labs. As such SAP is a new "collaborator or contact" in regard to this research.

References and Notes

- [1] Herb Simon, "The sciences of the artificial", MIT Press, 3rd Edition, 1996
- [2] Alan Kay, "Computer Software", Scientific American, Sept 1984
- [3] Jay Fenton & Kent Beck, "Playground, an object-oriented simulation system with agent rules for children of all ages", ACM OOPSLA Conf 1989
- [4] Mike Travers, "Agar, an animal construction kit", MIT PhD Thesis, 1988
- [5] Gerry Popek, et al, "The LOCUS distributed systems architecture", MIT Press, 1986
- [6] Martin Richards, "[BCPL: a tool for compiler writing and system programming](#)", AFIPS SJCC, 1969
- [7] Niklaus Wirth, "Euler: a generalization of algol", Parts 1 and 2. CACM Jan, Feb 1966
- [8] Alex Warth, Yoshiki Ohshima, Ted Kaehler, Alan Kay, "[Worlds: Controlling The Scope Of Side Effects](#)"
- [9] Takashi Yamamiya, Yoshiki Ohshima, [Tamacola - A Meta Language Kit for the Web](#), Workshop on Self-Sustaining Systems (S3), Tokyo, Japan, Sept 2010. ACM Digital Library
- [10] Ian Piumarta, [PEG-based Transformer Provides Front-, Middle- and Back-end Stages in a Simple Compiler](#), Workshop on Self-Sustaining Systems (S3), Tokyo, Japan, Sept 2010. ACM Digital Library

Appendix 1: Nothing Syntax in OMeta

```
addExpr = addExpr:x "+" mulExpr:y -> [{{#binOp. '+' . x. y}}
| addExpr:x "-" mulExpr:y -> [{{#binOp. '-' . x. y}}
| mulExpr

andExpr = andExpr:x "&&" bitOrExpr:y -> [{{#binOp. '&&' . x. y}}
| bitOrExpr

bitAndExpr = bitAndExpr:x "&" eqExpr:y -> [{{#binOp. '&' . x. y}}
| eqExpr

bitOrExpr = bitOrExpr:x "|" bitXorExpr:y -> [{{#binOp. '|' . x. y}}
| bitXorExpr

bitXorExpr = bitXorExpr:x "^" bitAndExpr:y -> [{{#binOp. '^' . x. y}}
| bitAndExpr

charLit = '$' char:c '$' -> [#charLit -> c asciiValue]

decl = "name":n "=" expr:v -> [{{#decl. n. v}}
| "name":n -> [{{#decl. n. {#number. 0}}}]

eqExpr = relExpr:x "==" relExpr:y -> [{{#binOp. '==' . x. y}}
| relExpr:x "!=" relExpr:y -> [{{#binOp. '!=' . x. y}}
| relExpr

expr = orExpr

field :s :offset = "name":n (":" "number" | [32]):l
[structs at: s]:d
[(d includesKey: n) ifTrue: [self error: 'duplicate field ', n, ' in struct ', s]]
[d at: n put: offset -> l] -> [offset + 1]

hexDigit = char:d ( ?[d >= $0] ?[d <= $9]
| ?[d >= $a] ?[d <= $f]
| ?[d >= $A] ?[d <= $F] ) -> [d]

ident = "name"

mulExpr = mulExpr:x "*" primExpr:y -> [{{#binOp. '*' . x. y}}
| mulExpr:x "/" primExpr:y -> [{{#binOp. '/' . x. y}}
| mulExpr:x "%" primExpr:y -> [{{#binOp. '%' . x. y}}
| selExpr

name = <letter letterOrDigit*>:n
-> [((Keywords includes: n)
ifTrue: [n asSymbol]
ifFalse: [n first isUppercase ifTrue: [#structName] ifFalse: [#name]]) -> n]

number = ``0b`` <($0 | $1) ($_ | $0 | $1)*>:n -> [#number -> (self numberFrom: n base: 2)]
| $0 <octDigit ($_ | octDigit)*>:n -> [#number -> (self numberFrom: n base: 8)]
| ``0x`` <hexDigit ($_ | hexDigit)*>:n -> [#number -> (self numberFrom: n base: 16)]
| <digit ($_ | digit)*>:n -> [#number -> (self numberFrom: n base: 10)]

octDigit = digit:d ?[d >= $0] ?[d < $8] -> [d]

orExpr = orExpr:x "||" andExpr:y -> [{{#binOp. '||' . x. y}}
| andExpr

primExpr = primExpr:arr "[" expr:idx "]" -> [{{#wordAt. arr. idx}}
| "args" "[" expr:idx "]" -> [{{#wordAt. {#args}. idx}}]
| primExpr:arr "@" expr:idx -> [{{#byteAt. arr. idx}}]
| primExpr:f "(" listOf(#expr. ','):args ")" -> [{{#call. f}, args]
| "memory" "." "name":n ?[n = 'wordSize'] -> [{{#memoryWordSize}}]
| "memory" -> [{{#memory}}]
| "numArgs" -> [{{#numArgs}}]
| "name":n -> [{{#name. n}}]
| "number":n -> [{{#number. n}}]
| "charLit":c -> [{{#number. c}}]
| "!" primExpr:e -> [{{#unOp. '!' . e}}]
| "~" primExpr:e -> [{{#unOp. '~' . e}}]
| "string":s "->" primExpr:e -> [{{#string. s. e}}]
| "(" expr:e ")" -> [e]
| "structName":s "." "name":n ?[n = 'wordSize'] -> [{{#structSize. s}}]
| "structName":s "(" expr:e ")" "." "name":n -> [{{#structAt. s. e. n}}]

prog = topLevelStmts:ss spaces end -> [{{#prog. {#compound. {#label. '$$'} . ss}}]

relExpr = shiftExpr:x "<" shiftExpr:y -> [{{#binOp. '<' . x. y}}]
```

```

| shiftExpr:x "<=" shiftExpr:y -> [{{binOp. '<='. x. y}}]
| shiftExpr:x ">" shiftExpr:y -> [{{binOp. '>'. x. y}}]
| shiftExpr:x ">=" shiftExpr:y -> [{{binOp. '>='. x. y}}]
| shiftExpr

sc = spacesNoNl (exactly(Character cr) | &$) | end)
| ";"

selExpr = primExpr:addr ":" expr:offset "#" expr:length -> [{{#select. addr. offset. length}}]
| primExpr

shiftExpr = addExpr:x "<<" addExpr:y -> [{{binOp. '<<'. x. y}}]
| addExpr:x ">>" addExpr:y -> [{{binOp. '>>'. x. y}}]
| addExpr

space = ^space | fromTo('/', '. String cr) empty(Character cr) | fromTo('/', '*'. '*/')

spacesNoNl = (~exactly(Character cr) space)*

special = <
$( | $) | $[ | $] | $@ | $, | $; | $+ | ``->' | $- | $* | $/ | $% | ${ | $} |
``=='' | $= | ``!='' | $! | ``<<' | ``<=' | $< | ``>>' | ``>=' | $> |
$~ | ``&&' | $& | ``||'' | $| | ``=' | $: | $# | ``...'' | $.
>:s -> [s -> s]

stmt = "name":n ":" -> [{{#label. n}}]
| stmtNoLabel

stmtNoLabel = "{" stmts:ss "}" -> [{{#scope. ss}}]
| "if" expr:c "goto" "name":n -> [{{#jnz. c. {#name. n}}}]
| "unless" expr:c "goto" "name":n -> [{{#jz. c. {#name. n}}}]
| "goto" "name":n -> [{{#jmp. {#name. n}}}]
| "if" expr:c stmtNoLabel:t ("else" stmtNoLabel | [{{#skip}}]):e -> [{{#if. c. t. e}}]
| "while" expr:c stmtNoLabel:s -> [{{#while. c. s}}]
| "break" -> [{{#break}}]
| expr:l "!=" expr:r sc -> [{{#assign. l. r}}]
| "var" listOf(#decl. ','):ds sc -> [{{#compound}, ds}]
| "print" expr:e sc -> [{{#print. e}}]
| "printb" expr:e sc -> [{{#printb. e}}]
| "printc" expr:e sc -> [{{#printc. e}}]
| "printo" expr:e sc -> [{{#printo. e}}]
| "printh" expr:e sc -> [{{#printh. e}}]
| "prints" expr:e sc -> [{{#prints. e}}]
| "newline" sc -> [{{#newline}}]
| "halt" sc -> [{{#halt}}]
| "clear" sc -> [{{#clear}}]
| structDecl
| "return" expr:x sc -> [{{#return. x}}]
| "return" sc -> [{{#return. {#number. 0}}}]
| "eval" expr:e sc -> [{{#eval. e}}]
| expr:e sc -> [{{#compound. e. {#pop}}}]
| ";" -> [{{#skip}}]

stmts = stmt*:ss -> [{{#compound}, ss}]

string = $" <(~$" anything)*:s> $" -> [#string -> s]

structDecl = "struct" "structName":s
[{{structs includesKey: s} ifTrue: [self error: 'duplicate decl for struct ', s]}]
[{{structs at: s put: Dictionary new}}:d
[0]:offset
"{" field(s. offset):offset ("," field(s. offset):offset)* "}"
[d at: '$bitSize' put: offset]
-> [{{#structDecl. s. d}}]

tok = spaces (name | number | special | string | charLit)

token :tt = tok:t ?[tt = t key] -> [t value]

topLevelStmt = "func" "name":n "(" listOf(#ident. ','):formals (","? "...")? ")" stmtNoLabel:body
-> [{{#func. n. formals. body}}]
| stmt

topLevelStmts = topLevelStmt*:ss -> [{{#compound}, ss}]

```

Appendix 2: A Copying GC written in Nothing

```
struct OTE { ptr, numSlots } // struct for Object Table (OT) Entries

var ot      = memory, // OT starts @ lowest valid address
    otSize  = 32 * OTE.wordSize, // max. number of objects
    heap1   = ot + otSize, // base addr. of 1st heap
    heapSize = (memory.wordSize - heap1) / 2, // num. words in each heap
    heap2   = heap1 + heapSize, // base addr. of 2nd heap (also addr of 1st word after 2nd heap)
    heap2Lim = heap2 + heapSize // addr of 1st word after 2nd heap

func withOtdo(f) { // This function maps f over all OT entries.
    var ptr = ot
    while ptr < heap1 {
        f(ptr)
        ptr := ptr + OTE.wordSize
    }
}

func initOTE(ote) { // initialize OT
    OTE(ote).ptr := 0
    OTE(ote).numSlots := 0
}

withOtdo(initOTE)

func oop2ote(oop) { return (oop >> 1) * OTE.wordSize + ot } // These functions translate between oops (indices into
func ote2oop(ote) { return ((ote - ot) / OTE.wordSize) << 1 } // the OT) and ptrs to their OTEs.

func mkInt(x) { return x << 1 | 1 } // Ints are tagged. These functions create an int object
func intVal(i) { return i >> 1 } // and extract value of an int object, respectively.

var freeList, heap = heap2, heapLim, heapPtr, otherHeapPtr = heap1
func fixFreeListHelper(ote) {
    if OTE(ote).ptr < heap || OTE(ote).ptr >= heapLim {
        OTE(ote).ptr := freeList
        OTE(ote).numSlots := 0
        freeList := ote
    }
}

func fixFreeList() { // Organize the free OTEs into a linked list so that alloc
    freeList := 0 // doesn't have to scan the OT for free OTEs.
    withOtdo(fixFreeListHelper)
}

func switchActiveHeap() {
    heapPtr := otherHeapPtr
    if heap == heap1 { heap := heap2; heapLim := heap2Lim; otherHeapPtr := heap1 }
    else { heap := heap1; heapLim := heap2; otherHeapPtr := heap2 }
    fixFreeList()
}

switchActiveHeap()

func moveToOtherHeap(oop) { // This function recursively moves an object to the other heap
    if oop & 1 // and updates its OTE to point to the new body.
        return // (No need to do anything for ints.)
    var ote = oop2ote(oop)
    if OTE(ote).ptr < heap || OTE(ote).ptr >= heapLim // If this object's body is not in this heap, it's already been
        return // moved, so we're done.
    var ptr = otherHeapPtr // The new address of the object's body.
    otherHeapPtr := otherHeapPtr + OTE(ote).numSlots
    var idx = 0 // Copy the object's slots to the new body.
    while idx < OTE(ote).numSlots {
        ptr[idx] := OTE(ote).ptr[idx]
        idx := idx + 1
    }
    OTE(ote).ptr := ptr // Update the object's OTE to point to the new body.
    idx := idx - 1 // Recursively move this object's slots to the other heap. We start
    while idx >= 0 { // at the bottom b/c "next" ptrs are usually the last inst var, and
        moveToOtherHeap(OTE(ote).ptr[idx]) // we want locality of reference.
        idx := idx - 1
    }
}

var root = mkInt(0) // An object is live iff it is reachable from this variable.

func alloc(n) { // This function allocates a new object with n slots and returns
    var triedBothHeaps = 0 // its oop.
ll:
    if freeList == 0 || heapPtr + n > heapLim { // If the object doesn't fit, either b/c there are no free OTEs or
        if triedBothHeaps // not enough memory for the object in this heap, GC and try again.
            return 0
        }
    return heapPtr
}
```

```

        halt
        triedBothHeaps := 1
        moveToOtherHeap(root)
        switchActiveHeap()
        goto ll
    }
    var ans = freeList
    freeList := OTE(freeList).ptr
    OTE(ans).numSlots := n
    OTE(ans).ptr := heapPtr
    heapPtr := heapPtr + n
    var idx = 0
    while idx < n {
        OTE(ans).ptr[idx] := mkInt(0)
        idx := idx + 1
    }
    return ote2oop(ans)
}
// (If we've already tried this, then the object doesn't fit -- ERROR!)

// Take the new object's OTE from the free list.

// Make room for the object's body in the heap.

// Initialize all of its slots to zero.

// Return the oop of the new object.

```