STEPS Toward The Reinvention of Programming, 2009 Progress Report Submitted to the National Science Foundation (NSF) October 2009

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

Important Note For Viewing this PDF:

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.

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-2009-016

NSF Award: 0639876
Year 3 Annual Report: October 2009
Steps Toward the Reinvention of Programming

Major Findings in 2009

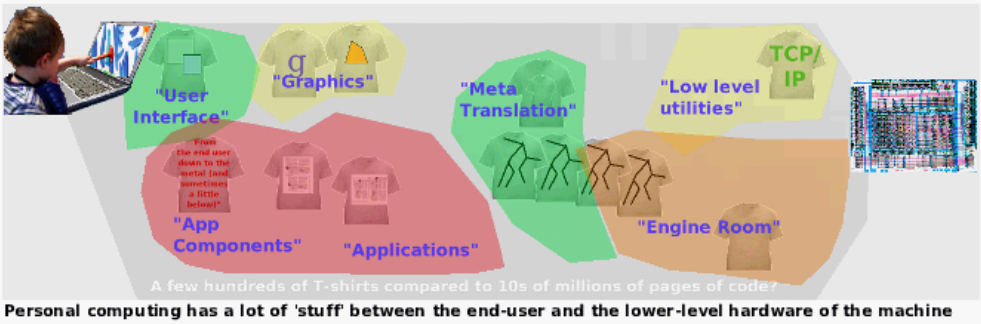
Bounding "Personal Computing" is a bit like trying to find the fractal length of a shoreline. Still, the average user expects a GUI and a "productivity suite", which includes word processing and DTP, high-quality printing, spreadsheet, email, web-browsing, etc.

These require considerable lower level support, including a file system, 2.5 D antialiased alpha blended color graphics, TCP/IP, etc. All of this has to be programmed in one or more programming languages, which in turn have IDEs, ways to make executables, intermodule communication, etc.

This seems like a lot (the current commercial examples and operating systems require tens of millions to hundreds of millions of lines of code).

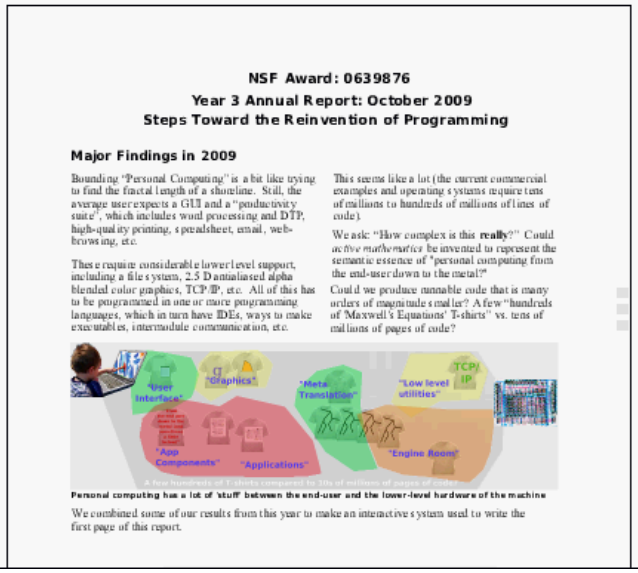
We ask: "How complex is this **really**?" Could *active mathematics* be invented to represent the semantic essence of "personal computing from the end-user down to the metal?"

Could we produce runnable code that is many orders of magnitude smaller? A few "hundreds of 'Maxwell's Equations' T-shirts" vs. tens of millions of pages of code?



Personal computing has a lot of 'stuff' between the end-user and the lower-level hardware of the machine

We combined some of our results from this year to make an interactive system used to write the first page of this report.



Page 1 of this report authored and rendered in our own system

This system is an "omni-application" for general media authoring called "Dynabook Junior". It takes its roots from early DTP architectures at PARC, Hypercard at Apple, and the Squeak Etoys system. It is a layout organizer for views of objects that are derived in turn from an "omni-object" and a rough analogy to the "omnicells" of most biological organisms, which have the same DNA but diversify into hundreds of distinct cell types. In sympathy with our philosophy of generality and self-similarity, and in contrast to typical DTP systems, the document is an object that can be inserted into a document like any other object, even recursively. The first page of the document contains the document itself, scaled by 50%.

We use a "T" metaphor in reference to two of our inspirations: Maxwell's Equations on a T-shirt ("…and then there was light") and the Ford "Model-T", a real automobile made from very few components which could be taken apart and reassembled by most 12-year-olds almost a century ago. Several of the principles that made our report application possible are included in our "T-shirt" findings from this year:

1. The Nile rendering and compositing language.
2. The transformation of syntax and semantics (such as Nile) into an executable form.
3. The transformation of intermediate forms into machine code, bypassing C or other compilers.
4. The report editor, text layout engine for paragraphs, and other "application components".
5. The LWorld graphics and visualization framework.
6. The DynaBook Jr application framework.

"Powerful Principles"

Some of the powerful principles [1] used as architectural building blocks are:

✓Math Wins ✓Meaning Separated From Optimization ✓No Centers ✓Loosely-Coupled Communications ✓Control of Control and Context	✓Model-T-Shirt Programming ✓Particles and Fields ✓Universal Parametric Objects ✓"From Nothing" Bootstrapping

"Oldies but Goodies" + New Ideas

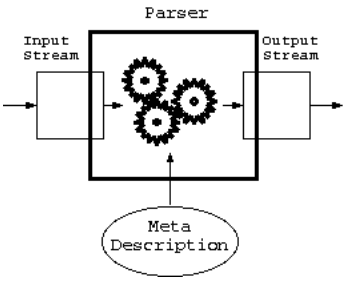
A number of these ideas and techniques have been around for decades, so a number of our tools are new bottles for old wine (or new architectures from old bricks). We have been surprised at how much progress we've been able to make just using reformulations of "oldies but goodies". This leads us to think that if we are successful in achieving the combination of behaviors and brevity that is our goal, we might find that the three to four orders of magnitude code bloat over the last 30 years happened largely in the presence of better ideas that were somehow invisible to the most popular views of computing.

This has made the numerous actual creations of this project (e.g., the geometric approach to graphics, the metatranslators of meaning, how TCP is done, etc.) much easier to invent.

Chains Of Meaning

The "chains of meaning" (CoM) required to make DBjr includes many of the earlier findings of STEPS: high quality antialiased graphics rendering and compositing, languages for expressing these techniques, meta-languages to make the graphic languages, meta-languages to connect them to processing hardware, and so forth. On the structural side, a notion of objects and how they intercommunicate needed to be invented, along with notions of viewing, interaction, layout, etc.

Our approach to making languages is to use pattern-directed translators that operate on and produce a variety of object forms.



A pattern-directed translator

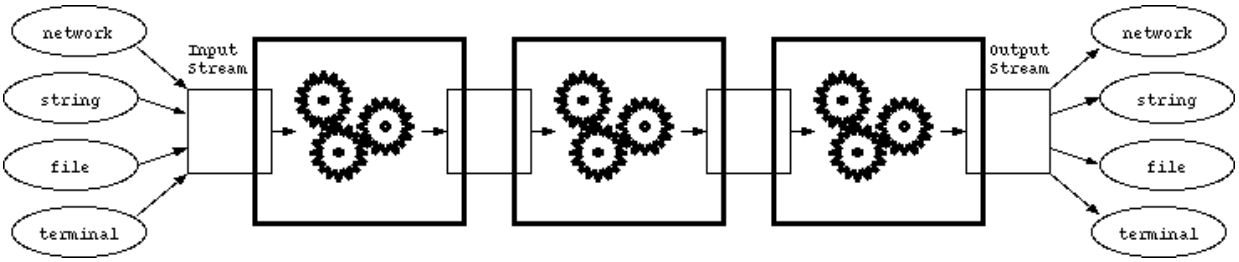
For example

- A stream of characters can be transformed into an abstract syntax tree (AST).
- An AST can be analyzed, optimized and manipulated to produce a "better" AST.
- An AST can be transformed into another language, byte code, or machine code.

The meta descriptions are transformation rules of the form:

```
pattern-to-be-matched → pattern-to-be-generated
```

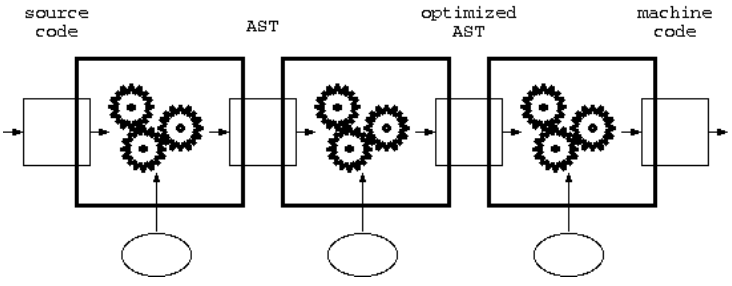
Transformations can be given "nicknames", leading to descriptions that look much like parser grammars.



A Typical Chain of Meaning

Our meta descriptions allow full programming to be done when that is needed. For example, a previous report described a version of TCP/IP written as a non-deterministic parser of input from the Internet that gathers and assembles Internet packets into application messages (all done in about 160 lines of code).

A chain of these metatranslators for one of the several "problem-oriented languages" we've invented might look like this:



A chain of meta-translators, or "chain of meaning"

Meta descriptions must be transformed into a runnable form, just like any other language. A meta description of meta descriptions converts meta descriptions into ASTs and then ASTs into some executable form, making a self-implementing chain of meaning for parsers.

So far, many of our T-shirts wind up carrying about 30 - 200 lines of code (which fits, but is hardly the classic 4 lines of Maxwell's Equations). The tradeoffs between further abstractions and the actual details which have to be given (to represent personal computing in a form readily-understood by human beings and also be automatically executed by computer) suggest that more compaction is possible, but that we are also approaching the "natural size" for understandable expressions of these processes.

We now present a gist of each of our "T-shirts" in order.

1. The Nile Rendering and Compositing Language

Nile is an "active math" programming language designed to express gracefully the rendering and compositing relationships of the Gezira and Jitblt geometries that were findings in previous years. The figure gives a sense of how this "dataflow language which generalizes over collections" is expressed.

This seems like a lot (the current commercial examples and operating systems require 10s of millions to 100s of millions of lines of code).

We ask: "How hard is this **really**?" Could *active mathematics* be invented to represent the semantic essence of "personal computing from the end-user down to the metal?"

Could we make runable code that is many orders of magnitude smaller? A few "100s of 'Maxwell's Equations' T-shirts" vs. 10s of millions of pages of code?

Rendered Characters of text

```
FillBetweenEdges (start : Point) : EdgeContribution >> Real
    x = start.x
    local = 0
    run   = 0
    ∀ [[x', y], w, h]
        n = x' - x
        if n = 0
            local' = local + w × h
            run'   = run   + h
        else
            >>         | local | ⩽ 1
            >(n - 1)> | run   | ⩽ 1
            local' = run + w × h
                                    run'   = run + h
                            if local ≠ 0
                                >> | local | ⩽ 1
```

Excerpt of the main rendering operation in Nile

The basic coverage for these graphical operations is the "extended Postscript" capabilities of (for example) the "Cairo" graphics system, as used in Open Office and the Mozilla browsers.

Gezira rendering handles scalable graphical forms (e.g., Bezier outlines) with pixel and alpha blending contributions (translucency). The "active math" expression in Nile is about **80** lines of code of the kind shown in the code samples above and below.

There are 26 compositing operations in standard use, including those found in Bitblt, Postscript, Porter-Duff, SVG, and Flash. About 100 lines of Nile code are needed to express all of them.

```
CompositeSoftLight : Compositor
    ∀ [a, b]
        c = (1 - b / b.a) × (2 × a - a.a)
        d = b × (a.a - c) + a ⊕ b
        e = b × (a.a - c × (3 - 8 × b / b.a)) + a ⊕ b
        f = b × a.a + (√(b / b.a) × b.a - b) × (2 × a
        >> d ?(2 × A < A.a)? (e ?(B × 8 ≤ B.a)? f)
```

One of the more complex compositing operations

```
CompositeOver : Compositor
    ∀ [a, b]
        >> a + b × (1 - a.a)
```

"Over" - the most used compositing operation

Example of "Radial Gradient"

Composite "Over"

Composite "Invert"

Composite "Subtract"

Composite "Dst-Out"

The complete list of rendering and compositing code includes:

	Lines of Nile Code
• Master rendering operations	81
• The 26 compositing operations used by Flash/SVG, etc.	90
• Sampling	71
• Pen Stroking	76
• Total	318

2. The transformation of syntax and semantics (such as Nile) into an executable form

Now we have to make the Nile programming language in order to run these compact expressions of computer graphics. Nile was designed to be a convenient and compact way to express transformational mathematics, and we now describe what is required to translate the syntactical forms into meanings which can be run efficiently on computers.

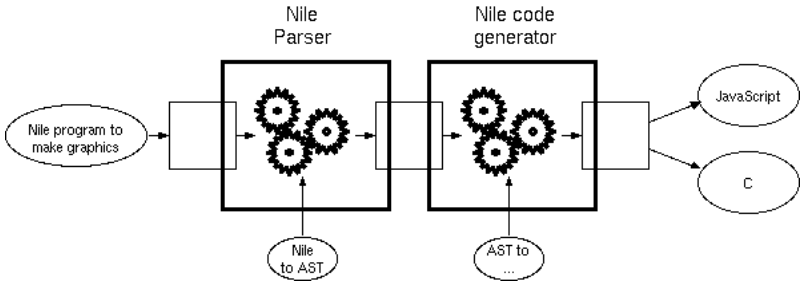
The Nile Parser

This is a translator built from a meta description of the Nile syntax. The first stage builds an abstract syntax tree from the surface forms. The grammar to do this looks like:

```
OffsideRuleParser.level = 0

ometa NileParser <: OffsideRuleParser {
  space    = ^space
           | ``/*'' (~``*/'' :_)* ``*/''
           | ``//'' (~nl :_)* nl,
  ns_ident = letter:x letterOrDigit*:xs exactly("'")*:ps          -> [x].concat(xs).concat(ps).join(''),
  ident    = spaces ns_ident,
  scalar   = spaces digit+:is '.' digit+:fs                       -> parseFloat(is.join('') + '.' + fs.join(''))
           | spaces digit+:is                                     -> parseInt(is.join('')),

  aType    = "[" listOf(#vTypeP, ','):ps "]"                      -> {['vType'].concat(
                                                                       ps.reduce(function (xs, x) { return xs.concat(x) }, [])
                                                                     )
                                                                    }
           | (ident | "â š"):t                                     -> ['nType', t],
  vTypeP   = listOf(#ident, ','):ns ":" ident:t                   -> ns.map(function(n) { return [n, t] }),

  akType   = "(" listOf(#vTypeP, ','):as ")" ":" kType:kt         -> ['akType',
                                                                      as.reduce(function(xs, x) { return xs.concat(x) }, []),
                                                                      kt
                                                                     ],
  kType    = aType:t1 ">>" (aType | "â š"):t2                      -> ['kType', t1, t2]
           | ident:t                                              -> ['nType', t],

  typeDef  = ident:n "<:" (akType | kType | aType):t              -> ['typeDef', n, t],
```

Example of part of the Nile language, expressed in the prototype OMeta meta translator

It takes about 130 lines of meta description to make the Nile parser.



A chain of meaning for Nile, generating JavaScript or C

```
binOp :op trans:x trans:y =
          // op is a typed macro
          ?[typedMacros includesKey: op]
          [typedMacros at: op]:macroType
          [self coerce: x to: macroType top1]:cx
          [self coerce: y to: macroType top2]:cy
          trans(macroType macro value: {#var. cx} value: {#var. cy}):ans
          -> [self coerce: ans to: macroType tans]
|
          // op is not a typed macro
          [currentKernelVarTypes at: x]:tx
          [currentKernelVarTypes at: y]:ty
          [tx max: ty]:tans
          [self coerce: x to: tans]:cx
          [self coerce: y to: tans]:cy
          [self newTmpName]:ans
          [self declVar: ans as: tans]
          [self emitPairwiseOp: (self nileToTargetLangOperator: op) on: cx and: cy type: tans into: ans]
          -> [ans],
```

Excerpt of translating Nile ASTs to a lower level language, expressed in OMeta

The Nile Code Generator

The next stage is to render the higher-level semantics of Nile contained in its AST into a form that can be executed. The common executable intermediate form of the STEPS chains of meaning (see the next section) is almost ready for this, but for now our prototype system uses OMeta to generate JavaScript or C.

It takes about 1110 lines of OMeta to make the Nile AST-to-C translator, bringing the total to 1240 lines of code in the current prototype for transforming "active math for graphical operations" into an indirectly executable form.

Nile is designed for expressing parallel operations on streams of objects. This addresses more than graphics and rendering. One of this year's experiments investigated using Nile to decompress PNG images, and more diverse applications are sure to follow.

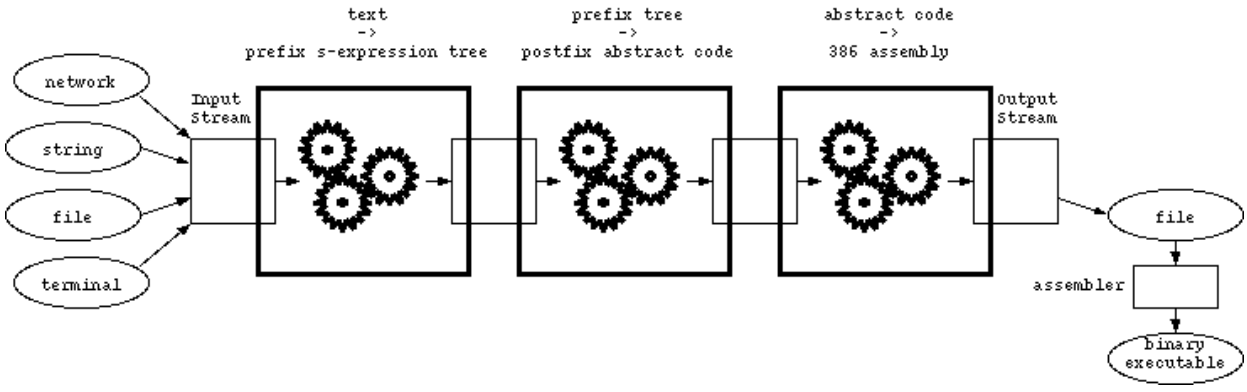
This is an example of a generally true progression: that the building block languages will be reused much more than the top-level specific languages. Our meta translators are used to make many systems that span the chains of meaning, from "input languages" (including the meta translator syntax itself) to "output languages" (both high-level, such as JavaScript and C, and low-level).

The implementation of meta languages in themselves was described in an earlier report [2]. The implementation of low-level output languages is described below.

3. The transformation of intermediate forms into machine code, bypassing C or other compilers

We still somehow have to make code that computer hardware can execute. We currently target Actionscript, Javascript, Squeak, C, s-expressions, and machine code. The first few are attractive for portability and "zero install" deployment. The last two are essential for a self-contained system and are described in some detail in a research memo [3]. The memo contains a small but complete example chain of meaning, from text to machine code, that is gisted here.

The common intermediate executable representation for STEPS translators is an s-expression (tree formed) language that was influenced by Lisp's ability to treat data and programs uniformly. It is our lingua franca target for translation into machine code for a number of CPUs.



A chain of meaning transforming a high-level language into binary executable code

Here is what part of this looks like:

```
start    = sexpr

sexpr    = _ (atom | list)
atom     = symbol | number
list     = "(" sexpr* :l _ ")" -> :l

symbol   = ( letter (letter | digit)* ) $$
number   = digit+ $#10

letter   = [-+!\$\%&*./:<=>?@A-Z\\^_a-z|~]
digit    = [0-9]

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

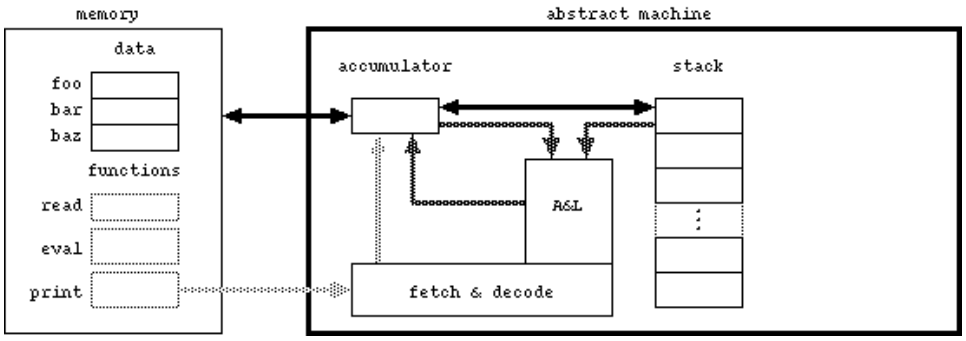
Excerpt from text to s-expressions

```
long    = &`long?   .
name    = &`symbol? .
arity   = .*:x                              ->`(list-length x)

args    = expr:e args:a                     -> (::a ::e save)
        | expr:e                            -> (::e save)
        |                                   -> ()

params  = ( name:h params:t | name:h )      ->`(arg-name h)

expr    =    long:x                         -> (load-long :x)
        |    name:x &`(is-arg x):n           -> (load-arg  :n)
        |    name:x                         -> (load-var  :x)
        | '( '< expr:x expr:y )             -> (::y save ::x less)
        | '( '+ expr:x expr:y )             -> (::y save ::x add)
        | '( '- expr:x expr:y )             -> (::y save ::x sub)
        | '( 'define name:n expr:e )        -> (long :n ::e store-var :n)
        | '( 'lambda '(params) expr*:b )    -> (enter :::b leave):l
                                            ->`(save-lambda l):n
                                            -> (load-label :n)
        | '( 'if expr:t expr:x expr:y )     ->`(new-label):a ->`(new-label):b
                                            -> (          ::t branch-false :a
                                                          ::x branch :b
                                                 label :a ::y
                                                 label :b )
        | '( expr:f &arity:n args:a )       -> (::a ::f call :n)
        |    .:x                            ->`(error "unrecognised expression: " x)

start = expr
```

Excerpt from s-expressions to postfix expressions

We have done considerable experimentation with various code generating schemes, including some of the techniques used in C compilers, but also with rather different schemes that hark back to the B5000. Its philosophy of trying to avoid "trapping state" in registers allows very quick transfers of control as well as models of execution in line with our philosophy of "pervasive simplicity and generality".

Our distilled example takes text describing tree-structured s-expressions, converts it first to the common intermediate AST form and then to a linear, postfix form that is easy to execute on a stack machine. The machine is a "simplified B5000" having an accumulator, a stack, and a memory containing named storage locations (global variables). Numbered labels identify the locations of function entry points and internal branch destinations.



"Simplified B5000" stack-based abstract machine

Literal values can be loaded into the accumulator. Values can be moved between the accumulator and the top of the stack, and moved between the accumulator and a named memory location. Operators use the accumulator as their first operand and take their other (if any) from the top of the stack. Results are left in the accumulator.

Hardware Execution

For execution on real hardware, the abstraction is extended with a stack pointer (identifying the topmost item on the stack) and a frame pointer (identifying the start of the current function activation record).

Function arguments are passed on the stack. Each function activation saves the caller's frame pointer and return address in the stack, loads the frame pointer with the address of the first actual argument (numbered 0) and loads the stack pointer with the address of the saved return address (which is now the topmost item on the stack). The stack grows downward, towards lower memory addresses.

The example code runs at 70% the speed of the same program written in C and compiled with typical optimization. The entire chain of meaning, transforming text into machine code, is implemented in less than 100 lines of meta description.

Several peephole optimizations are possible while converting the sequence of abstract instructions to concrete instructions. Their definitions are straightforward and their impact on performance should be easily measurable.

```
insn  = 'label .:l                  `"L\#1:"
      | 'long .:n                   `"          .data"
                                    `"_V_\$n:   .long 0"
                                    `"          .text"
      | 'load-long  .:l             `"          movl    $\#1, %eax"
      | 'load-label .:n             `"          movl    $L\#n, %eax"
      | 'load-arg   .:n ->`(* 4 n):n `"         movl    (\#n)(%esi), %eax"
      | 'load-var   .:n             `"          movl    _V_\$n, %eax"
      | 'store-var  .:n             `"          movl    %eax, _V_\$n"
      | 'save                       `"          subl    $4, %ebx"
                                    `"          movl    %eax, (%ebx)"
      | 'add                        `"          addl    (%ebx), %eax"
                                    `"          addl    $4, %ebx"
      | 'sub                        `"          subl    (%ebx), %eax"
                                    `"          addl    $4, %ebx"
      | 'less                       `"          cmpl    (%ebx), %eax"
                                    `"          setl    %al"
                                    `"          movzbl  %al, %eax"
                                    `"          addl    $4, %ebx"
      | 'branch .:l                 `"          jmp     L\#1"
      | 'branch-false .:l           `"          cmpl    $0, %eax"
                                    `"          je      L\#1"
      | 'call   .:n ->`(* 4 n):n    `"          call    *%eax"
                                    `"          addl    $\#n, %ebx"
      | 'enter                      `"          popl    %ecx"
                                    `"          movl    %ecx, -4(%ebx)"
                                    `"          movl    %esi, -8(%ebx)"
                                    `"          movl    %ebx, %esi"
                                    `"          subl    $8, %ebx"
      | 'leave                      `"          movl    %esi, %ebx"
                                    `"          movl    -8(%ebx), %esi"
                                    `"          pushl   -4(%ebx)"
                                    `"          ret"
```

Transforming postfix expressions into machine code

4. The report editor, text layout engine for paragraphs, and other "application components"

The most interesting "application component" in our authoring system so far is the text paragraph editor and layout object. We chose a "particles & fields" (or "army of ants") perspective, so that we can think of each text character as a stage actor wearing the grapheme of the character as a costume—these are the "particles/ants". The "fields" are what the "particles" can sense: for example, each other, the margins, other limits of the shape they are embedded in, and so on. This allows a very simple "no-centers" distributed architecture to be created from just a few "situation-action" rules.

Layout of Text

For example: let's imagine that the characters will just wander about (like ants foraging) as their normal mode of existence (this represents the degrees of freedom possible for them). Now let's ask what they need to know and what they need to do to format themselves as a dynamic paragraph of text. They need to know a few things about their size and shape, the character before them, etc. A first order sketch of how they should react to each other and their container would be:

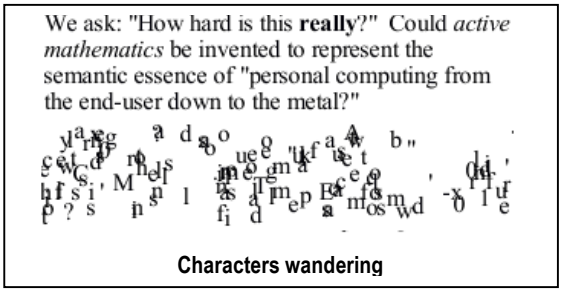
1. **If there is no character before me, then move to the upper left hand corner**
2. **Otherwise, follow the character before me**
3. **If I am over the right hand margin, then tell the character before me**
4. **If I am the first character in a word, and I am told that the word is over the margin, then move to the next line**

Example gloss of Wandering Letters rules for justification

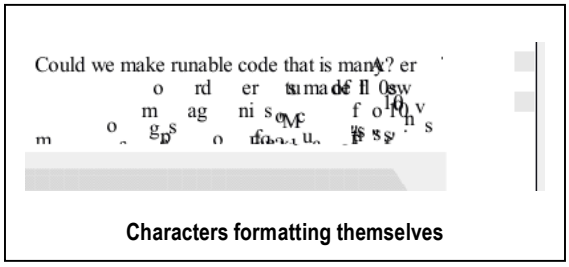
Here is what this language actually looks like as of this year (we are still in the process of designing a good syntax for this semantics (and this gives a sense of an intermediate stage in our design process):

```
WordWrap layout
  When client contents isEmpty not          Do client contents first box pivotBecomes
                                                      ((client shape leftAtY: 0) , maxHeight) + inset.
                                                       rule tell client contents first successor to 'place'.
  When Always                               Do rule tellLater: rule to: 'showSelection'.

WordWrap place
  When I amNil                              Do return me.
  When Always                               Do pred := my predecessor box.
                                               my box pivot: pred right + pred pivotOffset x @ pred pivot y.
  When rule placeIfAfterReturn my box       Do return rule tell my successor to 'place'
  When  rule isClipped my box               Do rule tell my box to 'backToWordStart'.
  When (rule isClipped my box) not          Do rule tell my successor to 'place'.

WordWrap placeIfAfterReturn
  When my predecessor box shape notNil and:
      [my predecessor box shape isNewline]  Do "start of the next line"
                                              my pivotYIncreaseBy: my height.
                                              my pivotLeft: (client shape leftAtY: my pivotPositionY) + inset x.
                                              return true.
  When Always Do return false.

WordWrap isClipped
  When my shape isWhiteSpace                Do return false.
  When Always                               Do return my right + inset x
                                                   > (client shape rightAtY: my pivotPositionY)

WordWrap backToWordStart
  When Always                               Do letterToMove := self startOfWord: me.
  When self isStartOfLine: letterToMove index  Do letterToMove := me.
  When Always                               Do letterToMove pivotYIncreaseBy: letterToMove height.
                                               letterToMove pivotLeft:
                                              (client shape leftAtY: letterToMove pivotPositionY) + inset x.
                                               rule tell: letterToMove successor to: 'place'.

WordWrap startOfWord
  When my predecessor isNil Do return me.
  When my predecessor box shape isWhiteSpace  Do return me.
  When Always                               Do return rule startOfWord: my predecessor box.

WordWrap isStartOfLine
  When Always                               Do return (client contents at: me) box left - inset x <=
                                                   (client shape leftAtY: (client boxAt: me) pivotPositionY)
```

Here are several examples of these rules in action.



Characters wandering



Characters formatting themselves

The entertaining “army of ants” wandering and following is included to help learners understand the logic of this kind of programming. An optimization is for each character to wait until its before character has found its place and to then jump to the next place, etc. This imposes an order that is more efficient and comparable with the more standard loop if the task switching is at the same level of efficiency as message sending (which is the case in STEPS). If the “line-up” actions are done in between frame times, then the effect will be that of an ordinary automatically-formatting paragraph, but done in a very different way.

The layout formatting for the paragraph editor currently takes seven rules.

Editing and Features

Similarly, the interactive editing part of a standard text paragraph can be compactly written in a "no-centers" distributed "particles and fields" style, if we think of the characters "looking" outwards for the mouse cursor and reacting appropriately, etc. This is also efficient if event-driven, with the "field" informing a character whenever it is near something interesting.

The pseudo-code for part of this editor looks like this:

For Copy,
If the selection is not empty, then copy the letters in the selection and put them into the world's clipboard.

For Backspace (Delete),
1. If the selection is an insertion point, then delete the letter before it. Subtract 1 from the index of the insertion point.
2. If the selection is not empty, then delete each letter in the selection. Set an insertion point where the letters were deleted.

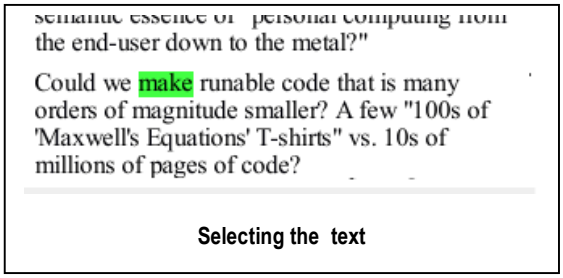
Example pseudocode gloss of Wandering Letters rules for editing

The current actual code for editing looks like:

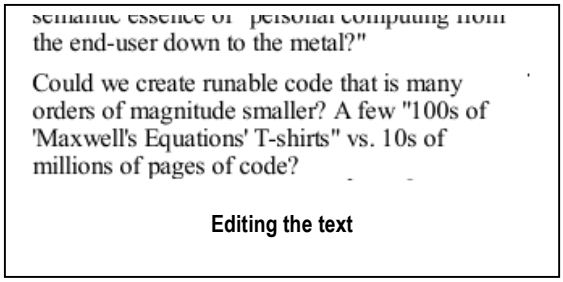
```
WordWrap charTypedCMDc   "Command-C, Copy"
When self selectionEmpty not
        Do client worldState clipboard:
                ((selection start+1 to: selection stop-1)
                        collect: [:ii | (client contents at: ii) box copy])

WordWrap charTyped08      "Delete key"
When self selectionEmpty and: [selection start > 0]
        Do client removeAt: selection start.
        selection := selection start-1 to: selection start.
When self selectionEmpty not
        Do [self selectionEmpty] whileFalse: [
                client removeAt: selection start+1.
                selection := selection start to: selection stop-1].
```

Excerpt of current actual code for Wandering Letters rules for editing

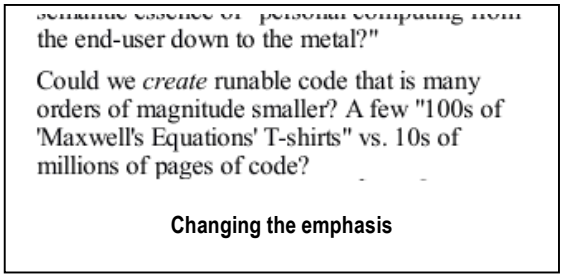
Here are several examples of these rules in action.



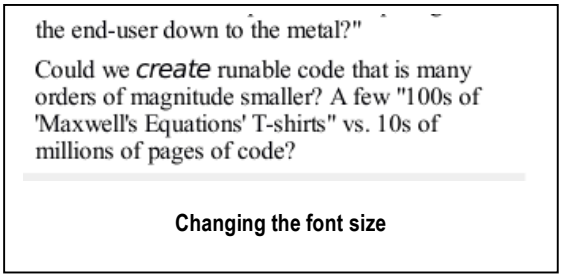
Selecting the text



Editing the text



Changing the emphasis



Changing the font size

The number of rules for each part of the text paragraph object:

Layout

- Layout of text with word wrap: 7

Selection and Typing

- Handle events to Select Text: 5
- Show the selection: 4
- Type in text and Delete: 2

Editing Features

- Arrow keys in text: 6
- Cut / Copy / Paste: 3
- Tabs: 1
- Handle tall letters (font size change within a line of text): 2
- Bold / Italic / Plain: 3
- Font increase/decrease size: 3

Total: 36 rules

5. The LBox Graphics and User Interface Framework Model

This is a viewing and events framework with a number of important twists and experiments. LBox is used to make both the DBjr application's structure and its graphic components (such as the text paragraph editor). But it is also one of a number of experiments investigating:

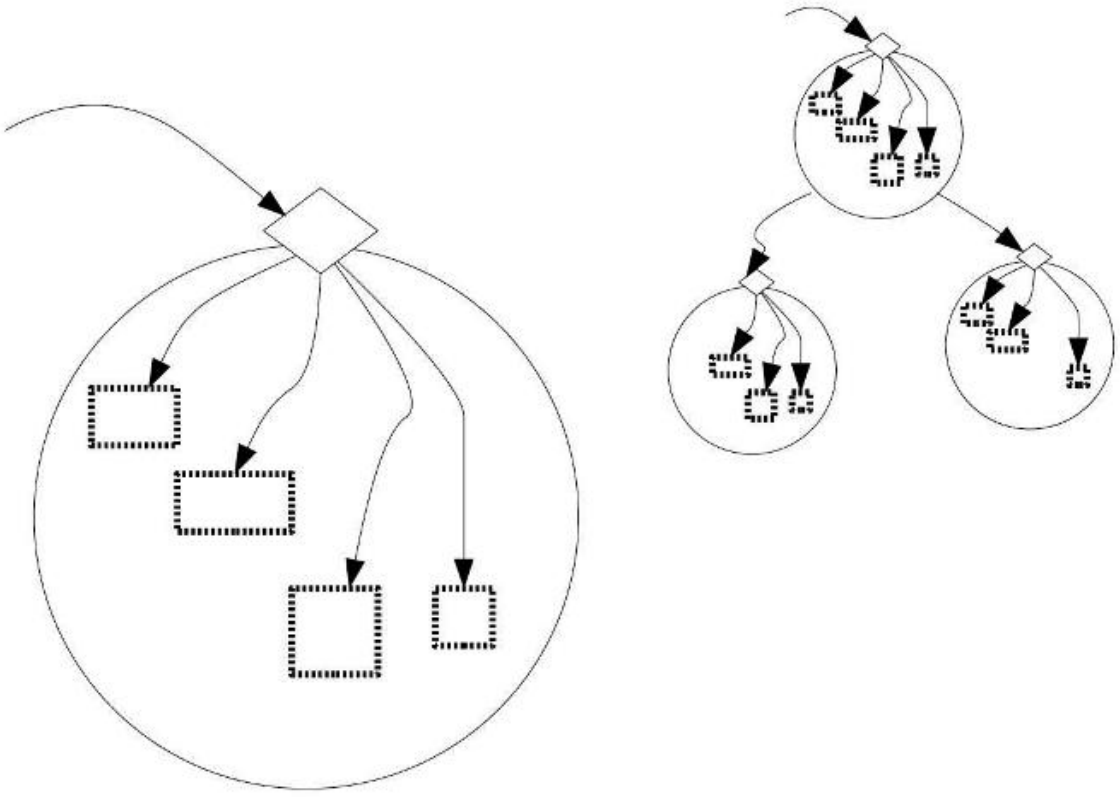
- more general "loose coupling" between massively-parallel objects
- making objects by composition (as opposed to subclassing).

We have felt that one of the barriers to gracefully scaling systems is for "needers" to have to know explicitly which object will supply their needs. A half-way step was attained in the 1970s with the use of dynamic polymorphic dispatch, but the "need" was still expressed with an explicit message selector rather than (say) a semantic description of the need.

An example might be that our "needer" needs the sine function, and in most programming languages, this will require the needer to know the explicit name of this function (e.g., in APL it happens to be "circle 1"!). To scale over large systems (and the Internet itself) it would be much better if we could use some form of brokering in the form of a publish-subscribe mechanism which allows more general ways to connect needs to suppliers of needs.

We can measure the degree of useful loose coupling by determining sweet spots on the tradeoffs between general expressiveness of needs, how fast a supplier for the need can be found and the need satisfied, and to what degree can the objects in the brokerage space be dynamically exchanged and still be able to guarantee that the system will run.

An extreme example might be an analogy to someone in an emergency room only having to yell "Help" and the doctors nearby will see what is needed (maybe a tourniquet on a wound) without the needer having to explain more. In a general loose coupling brokering framework each object would show two kinds of "bulletin boards": one would list its needs and their characteristics, and the other would list the kinds of supplies it makes that could fill needs.



Event Routing in LBox

Announcements (arrows) go through the widgets' *receive:* entry point (widgets are circles and diamonds represent the entry point), and then are re-announced to the components inside. The display tree (above) consists of such widgets and the communication between them is done in the same manner.

One notable simplification made in the framework is that the mechanism unifies the user input event routing with all other typical change notifications. The user input events from the keyboard or pointing device movements are delivered in the display tree by having graphical objects subscribe to their containers on these event types. This simplification provides a uniform mechanism to deal with messages arriving to a graphical object.

Also, with these message objects fully reified, it is straightforward to provide multiple views of an LBox into the display tree. With a minimal and generic mechanism to translate the events going in and out the viewed object, the same LBox can be displayed at multiple places with possibly different transformation. In the screenshot of DBJr in the beginning of this document, the scaled view of the page is embedded into the page. I.e., while the structure of display objects are a tree, there can be loops or shared objects introduced by this projection mechanism.

An LBox can have specific behavior and states by owning “component” objects. For example, a button-like behavior can be installed into an LBox using “sideways composition”. Again, the communication between the LBox and its handlers is done by the announcement mechanism. The default behavior of an LBox when it receives a message is to re-announce it to subscribing objects.

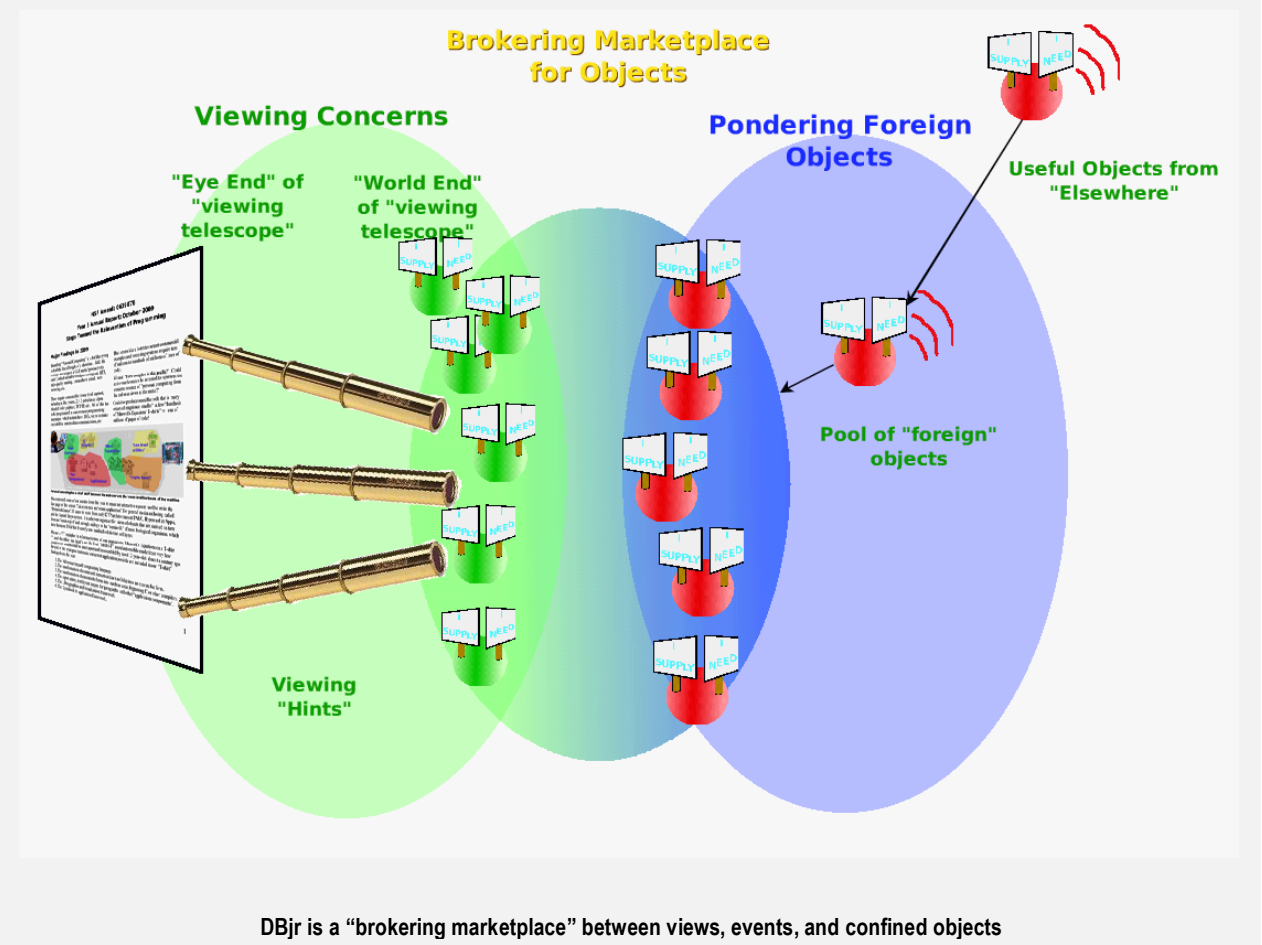
The graphics model is similar to other GUI frameworks done in the projects like BUI and Quiche, but with homogeneous transformations for vector graphics.

6. The DBjr Application itself

"DBjr" is a first pass at an "omni-application" for STEPS. One of the bounds of our "make a working model of personal computing from the end-user down to the metal" is limiting our application aspirations to the general productivity tools supplied in "Office Suites". These traditionally include:

- word processor, and perhaps desktop publishing
- presentations
- email
- web browser
- illustration
- search

We could readily imagine these "applications" (and more) as being "one application" made from a few basic building blocks with an integrated user-interface. There is no reason to have more than one abstract text object, or abstract picture object, or document type, etc.



DBjr is a "brokering marketplace" between views, events, and confined objects

Organizations of Views

Starting from the left, we want the user-interface to be as unified and homogenous as possible, with all visible manifestations provided by the same viewing mechanisms. On the right we have to deal with the extreme heterogeneity of literally millions of "possibly useful objects" which will be made not just by us, but by programmers all over the world. For safety, we will run all of them completely confined.

The interesting part of this scheme is not just the long established idea of loose coupling through some form of brokering, but the extent to which semantic match-ups can be made between our viewing and event mechanisms and the enormous space of functionalities available. In other words, it is not that we can do away with some agreement between the sender and the receiver, but that we want to minimize what has to be agreed on, and especially the size and extent of what has to be agreed on.

As an analogy as well as example, in MIDI, there are literally millions of timbres, and this makes it difficult to play a midi piece on a synthesizer that may not have the exact ones the arranger had in mind (or to display a document that does not travel with its fonts, etc.). In both cases we can make partially do with classes of timbres and fonts (this is what "General MIDI" and "standard fonts" do). We can go much further, making models of our intentions and sending them with the objects in question. Good models allow much better brokerage and match-up between "visiting actors" and the stage they are given to perform on.

This is an ongoing research problem that is central to many critical scaling issues in future programming and we expect to extend what can be done as our research project proceeds.

Other DBjr Features

Much of the rest on the left hand side is simple and straightforward. We draw from the original DTP designs at Xerox PARC, HyperCard at Apple, and the more recent Squeak Etoys approaches to universal user interfaces and end-user objects.

Since Nile graphics are scalable, the views in DBjr carry transformation objects. The views can recur (and there is a parameter which will limit the depth – this was used to make the "page within a page" on the first page of this report).

"Backgrounds" (or "Masters" in DTP parlance) are views which act as superclasses and which serve to make common (or parametrically common) organizations of views which may be shared by many views.

The LWorlds event detections and the efficient publish and subscribe services are used to loosely couple viewers and viewees, including allowing access to the interior of views, buttons, and other effectors.

The "Worlds" mechanism [2] for general provisional execution adds "undo" to DBjr, saving and backtracking to previous states when desired.

References

[1] STEPS Proposal, VPRI Research Note RN-2006002.
http://www.vpri.org/pdf/rn2006002_nsfprop.pdf

[2] STEPS 2008 Progress Report, VPRI Technical Report TR-2008004.
http://www.vpri.org/pdf/tr2008004_steps08.pdf

[3] Chains of Meaning in the STEPS System, VPRI Research Memo RM-2009011.
http://www.vpri.org/pdf/m2009011_chns_mng.pdf