Subtext: Uncovering the Simplicity of Programming

Jonathan Edwards
MIT CSAIL
32 Vassar St.
Cambridge, MA 02139

edwards@csail.mit.edu http://subtextual.org

ABSTRACT

Representing programs as text strings makes programming harder then it has to be. The source text of a program is far removed from its behavior. Bridging this conceptual gulf is what makes programming so inhumanly difficult – we are not compilers. *Subtext* is a new medium in which the representation of a program is the same thing as its execution. Like a spreadsheet, a program is visible and alive, constantly executing even as it is edited. Program edits are coherent semantic transformations.

The essence of this new medium is copying. Programs are constructed by copying and executed by *copy flow*: the projection of changes through copies. The simple idea of copying develops into a rich theory of *higher-order continual copying of trees*. Notably absent are symbolic names, the workhorse of textual notation, replaced by immediately-bound explicit relationships. Subtext unifies traditionally distinct programming tools and concepts, and enables some novel ones. *Ancestral structures* are a new primitive data type that combines the features of lists and records, along with unproblematic multiple inheritance. *Adaptive conditionals* use first-class program edits to dynamically adapt behavior.

A prototype implementation shows promise, but calls for much further research. Subtext suggests that we can make programming radically easier, if we are willing to be radical.

Categories and Subject Descriptors

D.1.7 [**Programming Techniques**]: Visual Programming; D.1.1 [**Programming Techniques**]: Functional Programming; D.2.6 [**Software Engineering**]: Programming Environments – *interactive environments, graphical environments*; D.2.3 [**Software Engineering**]: Coding Tools and Techniques – *program editors*; H.5.2 [**Information Interfaces and Presentation**]: User Interfaces – *interaction styles*.

General Terms

Human Factors, Languages

Keywords

Non-textual programming, visual programming, prototypes, copying.

OOPSLA'05, October 16–20, 2005, San Diego, California, USA.

1. INTRODUCTION

Programming is inhumanly hard. It stretches our mental abilities past their natural limits. The extraordinary difficulty of programming causes or aggravates all the chronic ills of software development. **Programming does not need to be so hard.**

Making things easy for people is the study of *usability*. Donald Norman [29] identified two basic usability problems: the *Gulfs of Execution and Evaluation*. The Gulf of Execution is the difficulty of translating a desired goal into an action to be executed. The Gulf of Evaluation is the difficulty of determining whether an observable state meets the desired goals. These gulfs loom vast for programming languages, because programs are represented as text strings.

The Gulf of Evaluation arises when we try to understand a program by readings its source text, a task so complex that only computers can do it reliably. Compilation is an intricate global analysis, and execution requires huge stores of memory. Testing and debugging tools give us only brief glimpses across the gulf. Preceding work on Example Centric Programming [11] proposed using examples to help comprehend program execution, but was severely constrained by the abstract nature of text.

Matters are no better when we turn to the Gulf of Execution. The affordances offered by text – inserting and deleting characters – are meaningless on their own. Most of the possible editing changes we can make leave the program invalid. Most interesting changes in semantics require delicately coordinated edits in widespread locations. The increasingly popular practice of unit testing [3] asserts that we cannot trust even simple changes without testing them automatically.

Norman's two gulfs arise when there is a mismatch between physical representation and conceptual meaning. **A major reason that programming is so hard is that text strings are a poor representation for programs.**

Text is paper-centric: pen and paper are a complete implementation. Modern software technology allows us to create arbitrary computer-based media, free of the limits of paper. A program can be represented in an abstract data model, and the programmer can use a GUI to directly manipulate [33] that model: WYSIWYG programming. This has long been done with other complex information artifacts, such as spreadsheets, documents, and diagrams. In all these cases, we no longer expect a paper printout to be a complete representation. It is time to transcend paper-centric programming.

There have been a number of attempts to escape the limitations of textual programming, notably visual programming languages and syntax-directed editing. These efforts are discussed in §5 (Related

Work), where it is argued that they stayed largely within the margins of paper.

Subtext is an experiment to develop a paper-free medium of programming, one designed for usability. In this medium the representation of a program is the same thing as its execution. Aligning syntax and semantics narrows the conceptual gulfs of programming. The experience of programming becomes more akin to using a spreadsheet than a keypunch. This medium is based upon a single unifying concept: copying; which develops into a rich substrate for the entire process of programming.

2. A BRIEF TOUR OF SUBTEXT

To discuss the design of Subtext, and the theory that underlies it, we will first introduce the basic features of the research prototype. This prototype is implemented in Java and SWT. It is only a proof of concept, lacking many niceties.

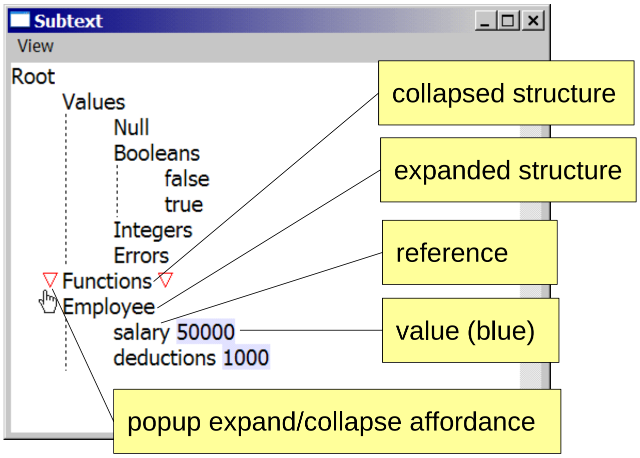
The graphical and interactive features of the user interface (UI) are essential to the experience of Subtext, so it is difficult to convey an accurate impression with only prose and a few screenshots. The interested reader is encouraged to view the 18 minute video at http://subtextual.org/demo1.html instead of reading this section. An online version of this paper, including full-color screenshots, is at http://subtextual.org/OOPSLA05.pdf.

All code and data in Subtext is organized into a single tree of *nodes*. There are two types of nodes: *structures* and *references*. The structures form the tree: each structure is said to *contain* the nodes below it in the tree, which are called its *subnodes*. The subnodes of a structure are ordered. Every node has exactly one *container* structure (except the *root* node, which has no container). At the leaves of the tree are empty structures and references. Empty structures are often used as “atomic values”. A reference is a pointer to another node, called its *value*.

Every node has a *label*, which is a string. The labels on the subnodes of a structure make it look like a traditional record, but as will be explained in §4.3, labels are purely comments, not identifiers. Subtext supplies primitive data types like the Booleans and integers, each of which is a structure containing all values of the type. Each primitive value is an empty structure whose label is an appropriate print string. For example, the integers are labeled with decimal strings, and are contained in the correct order in the Integers structure. The integers are properly infinite.

The Subtext user interface is primarily based on views of the tree of nodes, using an outline metaphor (often called a *tree widget*). A window can be opened on any sub-tree, and within that window, structures can be hierarchically expanded or collapsed. An expanded structure shows its subnodes indented on the following lines.

Figure 1 shows a window based at the root of the tree, expanded to show the contents of Booleans and Employee. The Functions structure shows an expansion affordance activated by the proximity of the mouse. The Employee structure contains two subnodes, salary and deductions, which are references. Their values are displayed to their right, shaded in blue.

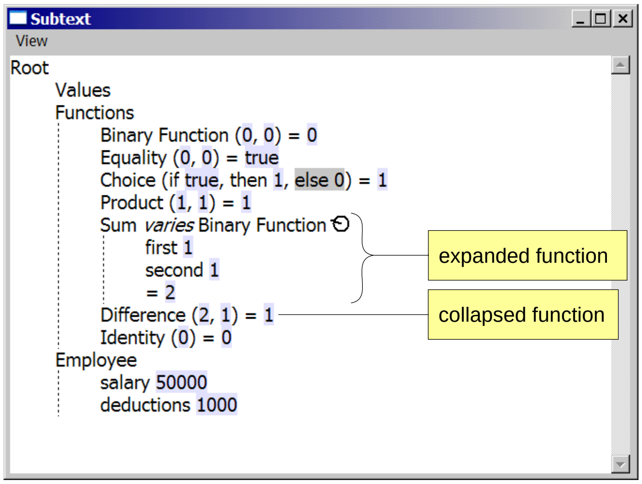


```
Root
    Values
        Null
        Booleans
            false
            true
        Integers
        Errors
    ▽ Functions ▽
    Employee
        salary 50000
        deductions 1000
```

Figure 1. Tree outline

2.1 Functions

Functions are structures that react to change. In Figure 2, the Sum function contains 3 subnodes, labeled first, second, and =, which are respectively its two arguments and result. Changing either of the two arguments will cause the result to automatically change to contain their sum. This happens essentially by magic, because Sum is a primitive built-in function.



```
Root
    Values
    Functions
        Binary Function (0, 0) = 0
        Equality (0, 0) = true
        Choice (if true, then 1, else 0) = 1
        Product (1, 1) = 1
        Sum varies Binary Function
            first 1
            second 1
            = 2
        Difference (2, 1) = 1
        Identity (0) = 0
    Employee
        salary 50000
        deductions 1000
```

Figure 2. Functions

The arguments and result(s) of a function are labeled nodes like in a record. This is similar to keyword arguments in some languages. But note the collapsed form of the Difference function: the labels of the arguments are hidden, and the result node is moved to the right of the parentheses. There is a convention that the first, second, … = nodes should be presented in this more familiar mathematical style when the structure is collapsed. This convention is a user-selectable option that can be altered globally or locally.

2.2 Creation by Copying

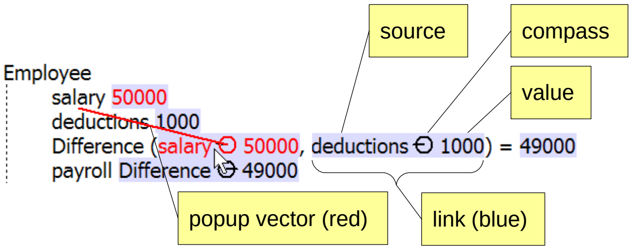
New nodes are created in only one way: by copying. An existing node is copied to some position within an existing structure. If the original node is a structure, its entire sub-tree of nodes is copied along with it. Copying is initiated by the programmer with drag-and-drop operations (or copy-and-paste, which has not yet been implemented). Copying is used to both instantiate data structures and call functions, which are actually the same thing in Subtext. The original definition of a function or data structure that serves as a template for copying is referred to as its *prototype*.

To see how functions are called, we will add some behavior to the Employee structure. We will calculate payroll as the difference between salary and deductions. We create the payroll node by dragging a copy of one of the other nodes and editing its label. We call the Difference function by also dragging a copy of it into the data structure.

2.3 Links

The final step of this example is to *link* the arguments and result of the function to the nodes of the data structure. Recall that a reference is a pointer to another node, called its value. Links control the values of references. A reference is always linked to exactly one other node, called its *source*. If it is linked to a structure, then that structure is its value, and the reference is called a *constant*. All the primitive values are empty structures, so any reference to them is a constant. If a reference is instead linked to another reference, it is called a *variable*. The value of a variable is the same as the value of its source. In other words, variable links are chased until a constant is found. Links are reactive: if the value of a reference changes, all of the references linked to it change their values along with it.

Continuing with the example, we need to make three links, connecting the arguments and result of the Difference function to the data nodes of Employee. Linking is initiated like copying, through drag-and-drop or cut-and-paste. Drag-and-drop operations draw a rubber band to visualize the link being established. Primitive values can be also be linked with in-place keyboard editing. Figure 3 shows the result after having made the needed links. The Employee structure now automatically calculates the value of the payroll node as the difference of salary and deductions, and recalculates automatically whenever they change. This data structure now acts like a function: you change it and it changes in response. Note how every intermediate value of the computation is visible, and the internal execution of the call of Difference can be made visible by expanding it.



```
Employee
    salary 50000
    deductions 1000
    Difference (salary ⊖ 50000, deductions ⊖ 1000) = 49000
    payroll Difference ⊖ 49000
```

Figure 3. Linking

Figure 3 shows some of the options for presenting links in Subtext. Every reference node displays its value. If the reference is a variable, the source is also displayed, controlled by a number of options. For example, a link to an = node can substitute the label of the containing function, as shown in the payroll node, whose source presents as Difference.

Following the source label is a circular widget called a *compass*. The compass has an *indicator* tick oriented in the direction of the source node. Compasses avoid the visual confusion that would result if all links were displayed as vectors drawn between nodes, as some visual languages have attempted. Instead, vectors are drawn selectively and interactively. When the mouse is over a link, the compass indicator extends into a vector reaching all the way to the source node, as shown for the link from salary. This interactive revelation of links is more effective than can be conveyed in text or video; you need to be driving the mouse to fully appreciate it.

There are many further options for representing links. For example, groups of related links can be vectorized together. The presentation of links is crucial to the usability of Subtext, as discussed further below.

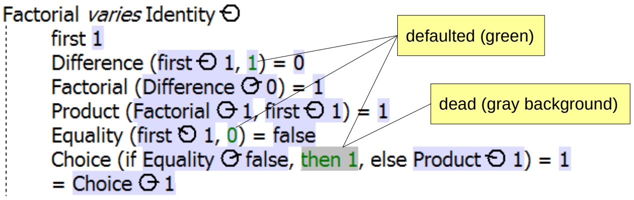
2.4 Calling by Copying

The Difference function was called by making a copy of its prototype. The internal structure and behavior of the function was duplicated in the copy. Likewise, if a new instance of the Employee structure is created, it will duplicate all the internal structure and linkages, including another copy of Difference, and thus replicating the automatic calculation behavior of payroll. Copying a node copies the entire subtree of contained nodes and preserves the internal structure of the links between them.

What if the payroll calculation changes after Employee instances have been created? Or what if the internal implementation of Difference changes? These changes will be propagated automatically to all the affected copies.

2.5 Conditionals and Recursion

Figure 4 shows a recursive factorial function. Recursion – a function calling itself – is done by just copying the function into itself. This creates an infinitely deep tree, which is tolerated because copying is materialized lazily. Projection of values down links triggers copying on demand. Infinite recursion is stopped by putting a maximal depth on structures, analogous to an execution stack limit, and returning an error value on links that traverse this boundary.



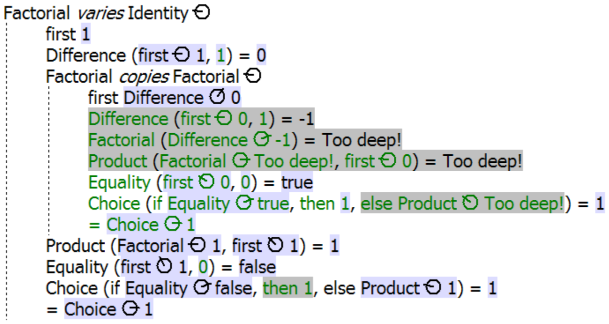
```
Factorial varies Identity ⊖
    first 1
    Difference (first ⊖ 1, 1) = 0
    Factorial (Difference ⊖ 0) = 1
    Product (Factorial ⊖ 1, first ⊖ 1) = 1
    Equality (first ⊖ 1, 0) = false
    Choice (if Equality ⊖ false, then 1, else Product ⊖ 1) = 1
    = Choice ⊖ 1
```

Figure 4. Factorial

Note how the second arguments of Difference and Equality are green. This indicates that they have been defaulted from the prototypes of those functions. Calling as copying provides defaulting on all arguments.

Conditionals are built with the Choice primitive function, which has 4 subnodes: if, then, else, =. In the conventional manner, the if argument is a Boolean, which selects either the then or else argument, returning it in =. Conditionals help visualize their operation by graying-out either the then or else argument, whichever was not chosen, as seen in the then node in Figure 4. The gray background indicates the node is *dead*: its value does not contribute to the result of the function.

Death is infectious – it propagates into the linked sources of a dead node, unless they are resuscitated by a link from a live node. This is shown in Figure 5, where the first recursive call has been expanded. It does not need to recurse, and so the else argument is dead. That else is linked to further recursion, which is shown executing, unboundedly, and returning the Too deep! error. But this error is irrelevant because the conditional is ignoring it, and the looping code is shown as dead. This behavior is similar to a non-strict lazy functional language.



```
Factorial varies Identity ⊖
    first 1
    Difference (first ⊖ 1, 1) = 0
    Factorial copies Factorial ⊖
        first Difference ⊘ 0
        Difference (first ⊖ 0, 1) = -1
        Factorial (Difference ⊘ -1) = Too deep!
        Product (Factorial ⊖ Too deep!, first ⊖ 0) = Too deep!
        Equality (first ⊘ 0, 0) = true
        Choice (if Equality ⊘ true, then 1, else Product ⊘ Too deep!) = 1
        = Choice ⊖ 1
    Product (Factorial ⊖ 1, first ⊘ 1) = 1
    Equality (first ⊘ 1, 0) = false
    Choice (if Equality ⊘ false, then 1, else Product ⊖ 1) = 1
    = Choice ⊖ 1
```

Figure 5. Factorial recursion

3. PRINCIPLES OF SUBTEXT

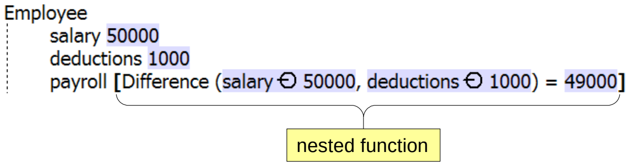
Having introduced the basic features of Subtext, we can now discuss the principles behind its design, which are all oriented towards making programming easier.

3.1 Language Extension through Presentation

Subtext introduces a new way to extend programming languages: *presentations*, which offer alternative ways to view and edit aspects of the program. We have already seen an example of this, in the way that the first, second, … = nodes of a collapsed function can be laid out mathematically. Presentations can be controlled from property sheets attached to nodes. Only the user interface is extended to support presentations, not the underlying semantics of Subtext, nor the programs themselves. **Presentations are like syntactic stylesheets for programs.**

A more significant example is *nesting*. Traditional syntax-based languages have two kinds of data flow. One kind uses expression nesting to encode the flow of return values up a tree of expressions. The other kind of data flow cross-cuts the expression tree via variable assignment and reference. It is often necessary to translate between these two different forms. When an expression value is needed in more than one place, it must be de-nested and assigned to a variable. Variables are also introduced when expressions become nested too deeply to be understood. Nesting further requires that expressions have only one value. This constraint becomes awkward in many situations, leading to multi-value wrappers, or communication through side-effects. Subtext avoids these problems with a single kind of data flow that supports an arbitrary graph structure. New edges (links) can be added to the graph without introducing local variables, refactoring code, or bundling values.

However a tree-structured data flow is a very common pattern, and the mathematically inspired convention of nested expressions is both deeply entrenched and highly expressive. To exploit these benefits, Subtext offers expression nesting as a presentation option. Any reference to the = node of a function can be nested by clicking on the link. The referenced function is then embedded in place of the link, surrounded by square brackets. Figure 6 shows the result of nesting the function call in Figure 3. Clicking on one of the square brackets de-nests the function. Nesting does not require a strictly tree-structured data flow, it merely allows the programmer to designate a spanning tree within the graph to be represented with bracketing.



```
Employee
    salary 50000
    deductions 1000
    payroll [Difference (salary ⊖ 50000, deductions ⊖ 1000) = 49000]
```

Figure 6. Nesting

Nesting provides the best of both worlds: the generality and flexibility of graph-structured data flow with the expressiveness and familiarity of tree-structured data flow. It does this unobtrusively in the user interface. In textual languages, such matters are typically permanent commitments made in the basic design of a language, and thus fraught with dilemmas.

Textual languages support extension through macros and layered translation. These techniques are highly disruptive, because the entire development tool-chain is affected, as are all the programmers, like it or not. Subtext gracefully sidesteps these problems because of the clean separation of concerns between the internal model of a program and its external user interface. Textual representations conflate these issues, forcing the compiler and the programmer to work with exactly the same representation.

Presentations avoid dilemmas of textual language design and extension by offering checkboxes on a stylesheet.

3.2 What's in a Name?

Symbolic names are the workhorses of programming languages. They carry the burden of everything not implied by grammatical structure. **Names serve many masters, but none of them well.** Names are used to:

1. Establish relationships between points in the program, by repeating the same spelling. Constantly inventing pithy unique names is burdensome. Misspellings and homonyms easily disrupt name-based relationships. Renaming is undecidable in the presence of reflection.
2. Implement abstractions, by delaying the binding of same-spelled names until compile-time or run-time. Much language semantics is smuggled in through arcane binding rules, for example method dispatch in OO. Delayed binding makes relationships implicit and contingent, obscuring them from the programmer.
3. Serve as comments and mnemonic aids. `The otherUsesOfNames interfereWith this English.Noun.Purpose`.

Subtext separates the uses of names into distinct mechanisms tailored to their purpose. The first purpose of names is to establish relationships. In Subtext, relationships are explicit, immediately-bound links within programs. These relationships are directly captured during editing in Subtext's internal database, without recourse to names. Every label could be `foo`, confusing the programmer no end, but not the computer. Textual programming requires the constant invention of unique names just to create structure, a burden that is lifted in Subtext.

3.2.1 *Abstraction without Indirection*

The second purpose of names is to support abstraction through delayed binding. Consider the primordial form of abstraction in programming languages: function calls. A function is represented by a symbolic name, which is resolved at compile-time or later into the function's definition. The definition can change at any time prior to binding. The arguments of the function are bound at call-time, so that they are free to change up till then.

In Subtext, a function call is immediately in-lined at edit-time, so that its definition is explicit and visible. If the definition of the function changes, those changes are globally replicated to all the inlined copies. The arguments of the function are also bound during editing, so that its execution is explicit and visible. If the values of the arguments change, the function recomputes as needed.

Functional abstraction is thus achieved without hiding meaning behind the delayed binding of an indirect reference. This principle is called *abstraction without indirection*. It is made possible by the ability of the program representation to automatically react to change. **Abstraction does not need to be obscured by indirection and deferral – that is only necessary in static notations that can not react to change.** Other examples of this principle in Subtext are the elimination of symbolic node names (§4.3), and higher-order functions (§4.5).

3.2.2 *The Efficiency of Ambiguity*

The third and final purpose of names is to serve as comments and mnemonic aids. These are matters of human communication and understanding, quite different from the needs of compilers and interpreters. Subtext frees names of their other burdens so as to optimize them for this purpose, and amplifies them through user interface techniques. **Names are too rich in meaning to waste on talking to compilers.**

The use of names in natural languages is quite different from that in programming languages. The vocabulary of natural languages is relatively fixed, and ambiguous overloading is common. Anaphoric abbreviation (e.g., a pronoun) is routine. Humans are highly skilled at disambiguating from context. There is a good reason for this: ambiguity increases the bandwidth of communication. Any information that can be inferred contextually by the listener is redundant.

Subtext exploits the human ability for contextual disambiguation to increase the bandwidth of programming. Links display the label of their source node to help the programmer understand or remember the link. These labels will often not be unique, but still perfectly clear from context, and more succinct than globally unique names would be. Presentation options allow the programmer to tune for the desired level of ambiguity, ranging from a fully-qualified containment tree path (with subscripting of homonyms), all the way down to elision into pronouns like `that` (referring to the prior = node).

Ambiguity is most effective in human communication during a conversation, when questions can be asked. Subtext offers interactive disambiguation through the mouse. Mousing over a link causes its compass to extend into a vector to the source node. Hovering the mouse over a link can open a "tool tip" popup with the full containment path of the source and a small display of its container context. A mouse gesture can open another window on the source node.

The planned keyboard interface for link editing also exploits the efficiency of ambiguity. Drag-and-drop is a convenient way to make a link when the desired source is visible, or can easily be browsed to. In other situations, typing a name on the keyboard may be more efficient. There is a useful analogy with web browsing. A textual language expects you to type in the one true unique name, much like a URL. Subtext will be more like using Google – names will be used as search keys, with the hits ranked and contextually summarized for easy recognition and selection.

Subtext provides an interactive medium that establishes meaning conversationally.

3.3 Overt Semantics

The Gulf of Evaluation is the difficulty of understanding what a program does from its source representation. The standard textual representation of a program is far removed from its run-time behavior. Subtext seeks to narrow this gap by using a different medium of representation. Every node always has a value, and every function is a living example of its execution. Static and dynamic aspects are intertwined, and there is no difference between edit-time and run-time. This is reminiscent of the way spreadsheets work, except that even spreadsheets hide the internal workings of their formulas, whereas Subtext is transparent all the way down.

The full meaning of a program is the set of all its possible executions. In Subtext, every execution of a program is a structurally equivalent projection of it, in which specific values change but the structure remains intact. The single example demonstrated by the program's definition is thus a revealing exemplar of its full meaning. This design principle is called *overt semantics*.

Overt semantics dispels the mystery of debugging. There is no need to guess at what happened inside the black box of run-time: debugging becomes merely browsing the erroneous execution, which is a copy of the program.

Overt semantics is an application of the proven power of examples to elucidate abstractions, as called for in the prior work on Example Centric Programming [11]. The Gulf of Evaluation is so wide because programming is so abstraction-intensive. Examples have proven to be the best way to learn and understand abstractions of all kinds. Subtext takes this lesson to heart by integrating examples into the very fabric of programming. It is not even possible to write code without simultaneously supplying an example. It is not possible to expose an API without simultaneously supplying at least one example of its use. Every execution of a program is another example, taking the same form as its definition.

Overt semantics narrows the Gulf of Evaluation because every definition is an example, and every execution is like the definition; syntax and semantics are aligned.

3.4 Semantic Editing

The flip side of the Gulf of Evaluation is the Gulf of Execution: the difficulty of determining how to change a program to achieve a desired change in behavior. This is inherently difficult in textual languages. The basic editing operations on text strings are character insertion and deletion, which mean nothing on their own, and are far removed from the semantic transformations we want to make. **Subtext narrows the Gulf of Execution by making editing operations be meaningful semantic transformations.**

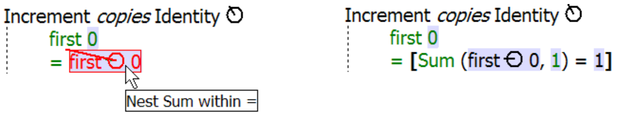
A *refactoring* [12] is a semantics-preserving change to a program. Subtext trivializes a number of these refactorings. A simple example is renaming. The spelling of a label is semantically irrelevant in Subtext, and is left as an uninterpreted comment. Editing a label is guaranteed to leave the program's semantics unchanged. Any links to that node will automatically display the new label, but will not be affected otherwise. Making this change in a textual program is referred to as the "rename" refactoring, and requires a global program analysis and transformation (and is undecidable in the presence of reflection). Subtext eliminates the need for this refactoring because it represents the underlying semantics of naming directly. Likewise refactorings such as *introduce local variable*, and *inline expression* become irrelevant.

Refactoring is symptomatic of poor notation. The hallmark of a good notation is that equivalent situations are equivalently described. The need for complex code transformation tools just to move between obviously equivalent descriptions indicates an ill-suited representation. Subtext dissolves certain refactorings, like renaming, by aligning syntax and semantics properly. When this is not possible, Subtext attempts to provide refactorings as direct-manipulation edits, rather than black-box "wizards".

The power of editing operations in Subtext is that they preserve important semantic properties. Automatic projection of changes ensures copy consistency. Many editing operations are based on copying, and thus preserve internal structure. It is particularly useful to preserve or transform link topology: this is called *link conservation*.

Links are never broken by editing operations; instead they are meaningfully transformed. The simplest example of this is when a structure is moved, all of its external links are preserved, both incoming and outgoing. Moving laterally within the same container preserves semantics, while moving up or down converts its role between that of parameter, closure, or call (see §4.1). Many refactorings are chiefly concerned with automating the delicate surgery needed to conserve symbolic links; they degenerate into move operations in Subtext because of link conservation.

Another example of link conservation is *splicing*. A function can be spliced into a link, executed by dragging it onto the link. Splicing results in a call to the function being inserted, and the original link being split into two links: one connecting the first argument of the function to the original source of the link; the other linking the original node to the result of the function. Splicing works well with nesting, so there is a presentation option to automatically nest when splicing. Figure 7 shows what a splice operation looks like during and afterwards. Splicing can be quite useful – it allows the factorial function in Figure 4 to be built with 10 mouse gestures.



```
Increment copies Identity
    first 0
    = first ⊖ 0
```

a) dragging to splice

```
Increment copies Identity
    first 0
    = [Sum (first ⊖ 0, 1) = 1]
```

b) nested splice

Figure 7. Splicing

Further experience with Subtext will likely reveal other semantic invariants beyond link conservation, and other editing operations that conserve them. **Subtext narrows the Gulf of Execution by providing high-level editing operations that coherently transform the semantics of the program while preserving relevant invariants.**

4. TOWARDS A THEORY OF COPYING

Subtext is only possible because the underlying mechanism of copying ties it together consistently. Copying is how programs are constructed, and how they execute. Copying has the inherent advantage of being a concrete concept, and the way programmers often work in practice. **The simple idea of copying generates a rich theory that includes an abstract model of computation.** A theory of copying is informally developed in this section.

Recall that the basic setting of Subtext is a tree of *nodes*. Nodes are either *structures* or *references*. All non-leaf nodes of the tree are structures, and the nodes immediately beneath them in the tree are called their subnodes. Empty structures can be at the leaves of the tree, and can be thought of as "atoms". References are only at the leaves of the tree, and are *linked* to a *source* node. The *value* of a reference is found by chasing links through references until a structure is found.

Some nodes are built-in originally, but new nodes are created only by copying old ones. A copy operation takes a *parent* node and creates a *child* node, which is inserted at a specified position within some existing structure. Copying duplicates the entire subtree below the parent into the child, causing *nested copies* of all the subnodes. Copies of primitive functions also inherit their built-in behavior.

Having reviewed this terminology, we can now state the primary properties of copying: it is isomorphic, continual, and higher-order. We will defer discussion of higher-order copying until §4.5.

Copying is *isomorphic*: it preserves internal link structure. If a reference in the subtree of the parent has a source that is also in the subtree, then the copy of the reference will be linked to the copy of the source. References with sources outside the parent subtree will be linked to the same source. Link isomorphism is simulated in textual languages with hierarchical name scoping. Subtext eliminates the need to declare scopes: links establish their own scopes implicitly as the least upper bound in the tree of their target and source.

Copying is *continual*: changes project bi-directionally between copies, keeping them isomorphic. Change projection is selectively blocked to allow copies to diverge from each other. There are three basic kinds of change that are projected: inserting a copy, deleting a node, and modifying a reference.

One way to block change projection is to declare that a child is a *variant* of its parent. Changes made within the subtree of the child will not project to the parent, and are called *divergences*. Changes made within the parent will continue to project into the child, unless they are *overridden* by a divergence. In particular, modifying a reference link overrides modifications to the parent reference. This is similar to modification in prototype-based languages (§5.1).

Divergence also occurs outside variants. References can be designated as *inputs*, which means that changing their link is considered a divergence even if they are not in a variant. Automatic divergence of inputs allows each call of a function to have different input links.

Divergence can be revised. It is possible to *revert* a node so that all its contained divergences are undone (sort of a structurally local undo). It is also possible to *equalize* divergences, propagating the changes up to the parent.

4.1 Reactive Computation

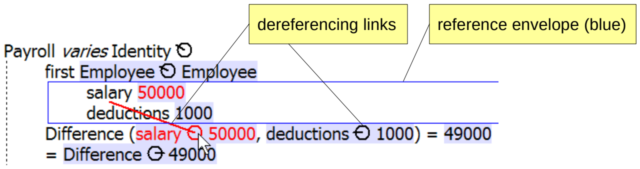
Execution in Subtext is driven by reaction to change. We have just described how structural changes (insertion, deletion, and linking) project through copy relationships between parents and children. However there is another kind of change: changing the link of a reference can change its value. Changes to values cascade down links and through primitive functions. Reaction to value change in this way is traditional data flow computation. **The difference with traditional data flow is the extra dimension of "copy flow", particularly because copying replaces calling.**

Traditional programming languages have three syntactically distinct ways of passing data: literals, variables, and function returns. Subtext combines all three of these into the single mechanism of a link. The role a link plays is determined by its direction in the tree structure. A link that refers downward is a function return[1]. A link that refers upward within a function is a variable reference. A link that refers upward outside the function, but within a containing function, is a free variable captured in a closure[2]. A link that refers upward through all containing functions is a global parameter (if to a reference) or a literal constant (if to a structure). A link can shift between these roles simply by moving its source to different locations. The same shifts require a coordinated series of edits in a textual notation.

4.2 Dereferencing

An important feature of linking has been ignored up to this point. Links can pass "through" references, navigating into the subtree of the referenced structure. Such links are called *dereferencing* links. They are often represented in textual languages with "dotted paths" of names.

The Subtext UI makes dereferences seem like regular links. Any reference node can be expanded in the same way that structures are. The subnodes of the referenced value are displayed on following indented lines, like the subnodes of a structure would be, except with a rectangular envelope drawn around them. The subnodes displayed in this way are called *dereferenced* nodes. The reference envelope is in a sense an embedded window on the referenced value. Naturally, dereference expansions can be nested, producing nested indented envelopes. Figure 8 shows an example of a function that calculates the payroll of an Employee structure which is passed by reference. The reference is expanded, and dereferencing links are made directly to the dereferenced nodes.



```
Payroll varies Identity ⟲
    first Employee ⟲ Employee
        salary 50000
        deductions 1000
    Difference (salary ⟲ 50000, deductions ⟲ 1000) = 49000
    = Difference ⟳ 49000
```

Figure 8. Dereferencing

When the value of a reference changes, any links that pass through that reference must be changed to follow the "same path". Textual languages handle this by using symbolic node lookup at run-time to navigate the path. But Subtext eliminates delayed binding, so another mechanism must be used. The key idea is to define what it is that makes paths the "same", based on the concept of *node identity* introduced below.

4.3 Ancestral Node Identity

The traditional method for determining whether two elements of a structure are the same is that their names are spelled the same. **Subtext captures a deeper notion of identity, based on *ancestry*.**

Let us start with a structure containing some nodes of interest. Instances of that structure are created by copying it, which automatically creates nested copies of all the subnodes. These copied subnodes are considered to be identical to the originals. But only nested copies are identical – top-level copies create new node identities. Nested copying generates an equivalence relation on nodes which defines identity.

This notion of identity is not affected by changing the spelling of node labels, nor by inserting or deleting nodes. A variant can thus rename, extend or contract the nodes defined in its parent.

We can now state how dereferencing links are affected by changes to references: they follow the same path of nodes, as determined by node identity. This maintains the principle of isomorphism of links, only modulo node identity. There is a problem, however, if one of the nodes along the path is missing. In this case, a special *ghost* node with the correct identity is inserted to preserve the path of the link. A ghost node is

---

[1] Multiple return values are allowed without the usual constraint of conventional languages that they all be bundled together at a "point of return".

[2] Closures may be so puzzling because of the way they straddle static lexical structure and dynamic call structure. These structures become one in Subtext.

highlighted in the UI as an error (usefully pinpointing it), and the link itself will carry an error value. Ghost nodes preserve the overt model of links even when they are broken.

Dereferencing links are another example of the principle of abstraction without indirection. Rather than deferring resolution of meaning with run-time node look-ups, the meaning is established immediately and visibly, and then changed as needed contextually.

4.4 Merging

So far we have seen how a single parent node can spawn a tree of children. **It is more powerful to allow nodes to be *merged* from multiple parents.**

Every node has a list of parents. Parents can be inserted, deleted, and replaced in this list. A node has a single parent when it is first created by copying. A structure will contain a copy of every subnode of every parent, with identical subnodes recursively merged together. The order of subnodes within a child structure preserves the order within each of the parents, extended where needed by the merge-order of the parents. References with multiple parents are linked based on the *overriding* rules described below.

Historically, there have been two kinds of primitive data structure in programming languages: lists and records. Subtext provides a novel alternative: *ancestral structures*, which blend features of both lists and records. Like lists, structures in Subtext provide a traversable order on their subnodes, and support insertion and deletion. Concatenation (between disjoint lists) is provided by merging. Unlike lists, Subtext provides an insertion-invariant notion of position based on ancestral identity.

Like records (and their inheritors, classes) Subtext allows position-insensitive random access to nodes, and node extensibility. Unlike records, no confusion is possible due to misspellings or homonyms (different names with the same spelling). The identity of nodes is determined definitively by their ancestry, irrespective of spelling. **Merging allows structures to be combined as with multiple inheritance, but without the riddle of homonyms.** [32]

References can only have one link. When they are merged from multiple parents, overriding rules determine which parent dominates, or if there is a *conflict*. An example of merging is shown in Figure 9. A diamond-shaped parent graph is constructed with Employee at the top, Manager and Part-time Employee as its children, and Part-time Manager as the merged grandchild. The parent of a node is displayed when it is expanded, similarly to a reference's link, with a label and a compass, and additionally indicating whether it is a copy or variant[3]. Non-divergent (unmodified) nodes are displayed in green, indicating that they are "inherited" or "defaulted" from their parent. The salary node of both Manager and Part-time Employee was modified, causing a conflict error when they are merged in Part-time Manager. The deductions node was modified in Part-time Employee, but inherited in Manager, so Part-time Manager inherits the overriding modification.

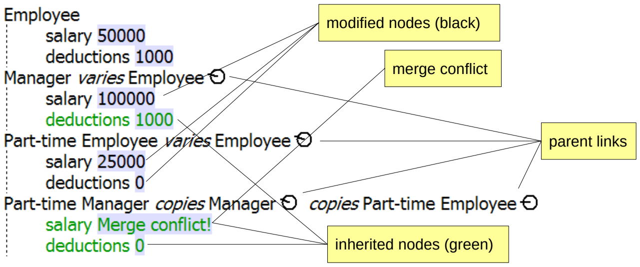


Figure 9. Merging

The rules for overriding are similar to that of traditional software revision control systems. Merging provides built-in version control for free[4]. **The same mechanism behind multiple inheritance also serves to merge versions of code.** Better, Subtext provides *exact* version control. Version control based on textual comparison can only correlate versions heuristically, whereas Subtext knows the precise history and ancestral relationships, down to the node level.

Note that with the addition of merging, we have also introduced the ability to change the parents of a node: new parents can be added, and old parents can be deleted or replaced. Change to parentage is governed by the principle of *conservation of divergence*: divergences in the child are preserved. For example, if a parent is deleted, subnodes copied from it are also deleted, unless they are divergent. A divergent subnode will not be deleted, but instead will turn into a node insertion (maintaining the same node identity). Another way of describing this principle is that when a node's parents change, it maintains a constant delta relative to them, with this delta being defined by the divergences.

4.5 Higher-order Copying

There are two kinds of relationships in Subtext: copying and linking. Linking can be seen as a special kind of copying, and copying in turn can be seen as a relationship subject to copying itself. This generalization is called *higher-order copying*. **Higher-order copying provides a simpler and more powerful foundation for Subtext.**

A hint can be seen in the way that references are presented visually. A reference to a value is expanded into an indented structure (in the reference envelope) that looks much like inserting a copy of the linked value. In fact a reference to a value can be seen as a copy of the value, except for the semantics of equality. The Equality function must consider two references to the same value to be equal, while normal copies would be considered distinct. Such "referential copies" are not allowed to diverge. Thus references can be seen as a special case of copying.

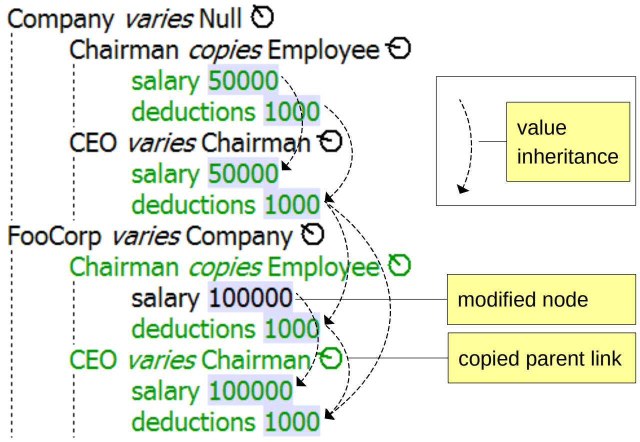
Copies map the structure of links isomorphically, so if a link is a copy, then the structure of copies must also be mapped isomorphically. A simplistic example is shown in Figure 10. The CEO node of a Company is by default a copy of the Chairman

---

[3] An alternative tabular presentation for parental relationships is proposed in §6.2.

[4] Currently only variants are tracked, but revisions are include in the support for mutable state proposed in §6.3.

node. The FooCorp instance of Company inherits this internal copy relationship. Changing the salary of FooCorp's Chairman changes the default CEO salary. Dotted arcs have been added to the screenshot to show the inheritance of node values. Note how the FooCorp CEO deductions node inherits from both FooCorp Chairman and Company CEO, with modifications to the former overriding the latter. Higher-order copying results in such multi-dimensional relationships.



```
Company varies Null
    Chairman copies Employee
        salary 50000
        deductions 1000
    CEO varies Chairman
        salary 50000
        deductions 1000
FooCorp varies Company
    Chairman copies Employee
        salary 100000
        deductions 1000
    CEO varies Chairman
        salary 100000
        deductions 1000
```

Figure 10. Higher-order Copying

Higher-order copying allows the parent of a structure to be a reference, making that structure a copy of the value of the reference. If the value of the reference is a function, the child becomes a higher-order function call. **Higher-order copies yield higher-order functions.**

Subtext provides higher-order functions while maintaining the principle of abstraction without indirection. In the definition of the higher-order code, the function-valued reference will have some default value, and the higher-order call will be an inlined example of calling that default function. This maintains overt semantics while permitting higher-order abstraction.

Higher-order functions in Subtext have a novel property called *ancestral signatures*. Untyped higher-order languages require only that function values match in the number of arguments with the caller. Typed higher-order languages further require a match of the function's type signature. In Subtext, ancestral node identity is used to establish compatible function signatures. Compatible functions are like compatible classes: derived by extension or merging from a common prototype function, not just accidental alignment of the number and types of the arguments.

4.6 Copying as the Essence of Programming

The unification of linking with higher-order copying boils Subtext down to one essential ingredient: copying. Subtext is *higher-order continual copying of trees*. **The simple idea of copying turns out to be rich and subtle enough to generate an abstract model of computation as well as a model of programming itself**.

Data flow and copy flow (§4.1) become one: the driving force of computation in Subtext becomes solely projection of changes through copies. It is interesting to contrast the resulting model of computation with the classical theory of Lambda Calculus [2]. The driving force of computation in Lambda Calculus is reduction, which is implemented by name-sensitive substitution. Substitution can be seen as a form of copying (with the name-sensitivity ensuring isomorphism). The difference between Subtext and Lambda Calculus is in when the copying happens. Lambda Calculus programs execute by copying, consuming the program in the process, until it is reduced to a result. Subtext programs are built by copying, but execute reactively by change projection, leaving the program intact and continuously executing, and thus making its semantics overt.

Copying provides not just a model of computation, but a model of the entire programming process. What a program does and what a programmer does are the same: manipulate copying. It is significant that copying is actually the way programmers tend to work in practice – a good omen for the goal of usability. Further, the unification of computation and programming provides novel synergies. For example, merging implements both multiple inheritance and version-control. A number of the proposed future research directions (§6) explore such synergies.

The unification offered by Subtext has a major usability benefit: simplification. Conventional programming involves a formidable array of specialized tools and languages and formalisms. Every one of these seems to have its own style of IF statement. **Subtext reduces the baroque complexity of textual programming into a seamless environment with a single conceptual framework.** There is no longer a need for a distinct compiler, debugger, interactive shell, unit-tester, program builder, or version-control system: programming becomes mode-less. There is no conceptual difference between edit-time and run-time, code and data, syntax and semantics. Calling, referring, instantiating, sharing, refining, modularizing, and versioning all become forms of copying. **The ultimate usability feature is coherence.**

5. RELATED WORK

Subtext builds upon many related efforts throughout the history of programming languages. There is only enough space here to discuss the most prominent figures in this heritage. Foremost is Self [40][41], which first proposed that copying (in the guise of prototypes) could provide a unifying basis of both a programming language and its interactive environment. Self promulgated the principles of "concreteness, uniformity, and flexibility" [34] and immediacy [39]. Subtext is in large part an attempt to carry forward the pioneering vision of Self.

5.1 Prototypes

Subtext's essential mechanism of copying is a generalization of *prototypes* [22][27]. Self [34][40][41] developed prototypes the furthest into a full fledged programming language and environment. Many flavors of prototypes were proposed, with different mixtures of sharing, modification, and delegation. Subtext tries to hide such implementation issues behind a simple model of distinct copies linked through change projection. Subtext generalizes prototypical copying to include function calling and even variable valuation. Although prototypes served as living data instances, code was still relegated to the netherworld of dead text awaiting execution. Subtext extends prototypes "all the way down" into the fabric of the code itself, making it is alive as data. The prototype languages were all object-oriented, whereas Subtext is functional, modeling computation as reactive structures. Subtext could be described as *functional prototypes*.

5.2 Visual Programming

There is a long history of research in Visual Programming Languages [6][25], so-called because they used diagrammatic rather than textual representations of programs. The early results were disappointing [13][30]. A common criticism was that diagrams did not scale well to large programs, resulting in incomprehensible mazes of boxes and lines, and laborious manual layout. Diagrams are good at compactly summarizing information, but are not well suited to highly detailed and large scale descriptions. Text is in fact quite densely detailed and has highly evolved conventions for large scale organization. The jury is still out as to whether Subtext can avoid the scaling limitations of visual languages. There is some reason for hope, because the Subtext UI is largely textual with only graphical embellishment, and so it can fall back on proven techniques for scaling text.

From the point of view of Subtext, diagrams and text are equally limited by being paper-centric, conflating the issues of instructing compilers with human communication. Subtext uses both text and graphics, but only as user-interface techniques, which is what they are good for; not as a semantic model, for which they are poorly suited. This separation of concerns frees Subtext from the constraints of paper, for example allowing execution details to be melded with the static representation of a program.

Some visual languages also revealed live execution details, but not while also supporting iteration or recursion. An exception was Pictorial Janus [17]. It had a unified representation of programs and their execution, supported recursion as infinite containment, and replaced names with topological properties. However its imperative semantics meant that while you could animate program execution, you could not see a program and its execution at once, an important usability goal of Subtext.

Vital [14] is a visual environment for Haskell. Function results (but not their internal execution details) are presented continuously. Lazy execution is triggered when results are scrolled into view within an unbounded workspace. Type-driven stylesheets alter the presentation. Vital is most similar to Subtext when dealing with data structures, which can be edited through copy-and-paste operations that correspondingly alter their definitions.

Subtext is related in many ways to Forms/3 [5], one of the most advanced visual languages. Forms/3 extends the familiar spreadsheet into a first-order functional programming language. In this way programming obtains the usability benefits of spreadsheets, such as *liveness* [15][36]. General purpose programming concepts are cleverly, but intricately, simulated within the spreadsheet metaphor. For example, abstract functions involve deductive inference to properly link call-sites and call-frames. Subtext shares with Forms/3 the principle that human factors should guide programming language design. The basic difference is that Forms/3 tries to coerce a spreadsheet into a programming language, while Subtext tries to invent a programming language that is like a spreadsheet.

The modularity mechanism of Forms/3 is called *similarity inheritance* [10]. It replaced the symbolic relationship of inheritance with continual copying relationships, established through cut-and-paste operations, resulting in “self-sufficiency”. This may have been the first proposal of Subtext’s principle of abstraction without indirection. Similarity inheritance was limited to static sharing of formulas between spreadsheet forms. Subtext’s more general copying mechanism replaces all forms of sharing and reference, and implements computation as well.

5.3 Ergonomic Programming

There is increasing focus on human factors as the critical issue in the design of programming languages and tools [21]. Subtext has been guided by the principles of the Cognitive Dimensions framework [13] and the Attention Investment model [4]. An application of these principles to the design of abstract functions in a spreadsheet [16] led to some similarities with the way functions are called by copying in Subtext.

Copy & paste operations are pervasive in actual programming practice [19]. *Linked editing* [37] proposed text-editor support for continual copying of text regions, and studied its use as a surrogate for functional abstraction in the language. Subtext uses continual copying within a richer structure than text to entirely replace symbolic abstraction.

5.4 Programming by Demonstration

Programming by Demonstration [8][24] seeks to let the programmer live in a world of concrete examples, with the computer intelligently abstracting these examples into general purpose programs. The most closely related work in this node is Tinker [23], which allowed both its recorded examples and generated Lisp code to be incrementally edited. Subtext also taps into the power of examples to tame abstraction, but not so much to make the computer seem smarter, as to help the programmer work smarter. Nevertheless, Subtext should provide a good platform for such research, allowing examples and their abstractions to be recorded and represented commensurably.

5.5 Syntax-directed Editing

Subtext bears some resemblance to syntax-directed editors [31][35]. These editors had built-in knowledge of the language syntax, so that instead of editing a string of characters, the programmer was directly editing the abstract syntax tree (AST). New code would be added via templates with blank nodes corresponding to the production rules of the grammar. The program was kept syntactically valid at all times, and syntactically-specific editing assistance was provided. Syntax-directed editors met resistance from practicing programmers [26]. A common complaint was that forcing the program to be syntactically valid at all times blocked well-worn shortcuts through invalid states. The compromise that has emerged in modern programming editors is to maintain a textual representation, but to add syntactic and semantic assistance “on the side”. This assistance is provided on a best-effort basis through such mechanisms as coloring, completion, and pop-ups.

The goal of Subtext is not to make syntax easier to use, but to avoid having to use it in the first place. In fact humans have a well-developed and largely subconscious ability to parse language. Syntax-directed editing trades this subconscious facility for the conscious manipulation of explicit structure, structure which is mostly about signaling the compiler, not expressing the meaning of the program. This is a loss in usability: if it is necessary to use syntax it should be left implicit as in natural languages. But even better is to do away with the middle-man of grammar altogether. Grammar is fundamentally about encoding meaning into a serial channel, which is no longer needed once we have evolved to direct manipulation. Subtext is *semantics-directed editing*.

Intentional Programming [9] appears to be related to Subtext, although it is hard to tell precisely since only partial descriptions of it have been published. It seems to be a form of syntax-directed editing that does not block character-based editing shortcuts. Compatibility with (and extension of) mainstream languages is a primary goal. The underlying model appears to be a generalization of an abstract syntax tree. Names are abstracted into binding relationships. Specialized notations can be embedded, much like embedding a diagram in a WYSIWYG word processor document. This allows extension through presentation, but all the notations are still paper-centric. Subtext shares the goal of WYSIWYG programming, but rejects the constraints of backward compatibility with old languages and hard copy.

5.6 Functional Programming

Subtext is in spirit a functional programming language, harkening back to the original call of Backus [1] to liberate programming from its hardware roots. One of Backus' goals was to lessen the dependency on names, a goal shared by Subtext, but not carried forward in the subsequent development of functional languages. Modern functional programming languages have demonstrated the power of sophisticated high-level abstractions. Subtext is trying to find ways to make such abstraction easier to use.

6. FUTURE WORK

Subtext is a young idea, perhaps no more mature than the first experiments with compilers in the 1950's. **Subtext is like starting over from the beginning with an alternative to punched cards.** The initial prototype merely demonstrates that programming in this alternative medium is possible, and holds promise. Exploring the potential of the approach will require much further research and development.

Here are just some of the challenges and opportunities:

1. Performance. The design of Subtext has so far fearlessly ignored performance issues in order to optimize for usability. Scalable implementation will offer interesting challenges.
2. User interface design and usability testing. Subtext must compete with the highly evolved and deeply entrenched user interface of text editing. Quite a few programmers have assured the author that they will give up Emacs when it is pried from their cold dead hands.
3. Programming in the large. The disappointing track-record of visual languages justifies skepticism that any non-textual language can scale to real-world programs. Subtext must convincingly address this issue to be taken seriously. Because the Subtext UI is largely textual, scaling techniques proven for textual languages can be applied. In particular, traditional hierarchical decomposition ought to fit well with the tree structure of Subtext.
4. Formalization. The theory of copying needs to be formalized in order to better understand its properties and expressive power.
5. Types. What is the role of types in a language without a compile-time, where self-executing definitions automatically flush out many common type errors?
6. Modularity. Modularity [28] is a precept of software design, while undisciplined copying is often considered its antithesis. Subtext offers a third way: ad-hoc copies which are recorded, and which can propagate changes. The chaos created by covert copying can be replaced with tools that manage copying and supervise change propagation. Modularity, rather than the antithesis of copying, can be seen as a special pattern of copying, one which can be refactored out of ad-hoc patterns.
7. Refactoring and code transformations. Subtext allows some major refactorings to be replaced by direct-manipulation operations such as dragging nodes to change their location. Code transformations like splicing can also become direct manipulations. A taxonomy of useful refactorings and transformations needs to be developed and correlated with UI affordances.
8. Databases. With the addition of declarative queries, Subtext would become a database. This offers a new take on the infamous "impedance mismatch" problem [7]: the clash between the data models of the language and the database; which become identical in Subtext.
9. Meta-programming. Subtext can be made fully reflective, and implemented meta-circularly. Reactive computation plus declarative queries may enable novel meta-programming capabilities. Do meta-queries support Aspect Oriented Programming [18]? What is the potential for Domain Specific Languages in Subtext?

The second version of Subtext is currently under development, and is the source of the screenshots in this paper. This version implements the model of higher-order copying, and focuses on the ideas discussed in the following three subsections.

6.1 Functional Iteration

There is a long-standing dispute between recursion and iteration. Recursion is more elegant formally, but many programmers find iteration to be simpler. **Subtext reconciles iteration and recursion by offering iteration as a presentation of recursion.**

This presentation is called *functional iteration*. The basic idea is to flatten a singly-recursive function into a linear scrolling-window display, with each recursive call occurring vertically below, rather than nested within, its caller. Links that drill in and out of recursive calls would now hop between these "steps". Links passing between the same node in adjacent steps (called *chained* links) would display as vertical vectors, and be labeled next or previous. This presentation would be most effective when at least three steps were visible, so that all the links in and out of the middle one were fully visible.

Functional iteration promises the best of both worlds: the simple semantics of functional recursion with the simple conceptual model of imperative iteration. It can support capabilities of stream-processing languages such as Lucid [42] and Lisp Series [44]. While these languages manipulated streams of data, Subtext will manipulate streams of code. Chained links propagate variable values up and down these streams, providing the convenience of side-effecting assignment as in imperative iteration, but without violating single-assignment semantics. Functional iteration can be implemented largely as a presentation on top of the existing model of recursion.

6.2 Adaptive Conditionals

Adaptive conditionals are a new kind of conditional construct that exploits the higher-order features of Subtext. The intuition is this: when we informally describe a complex process, we tend to first explain a basic case of the entire process. Then we come back and explain how special cases differ from the basic case: "except when *A* do *B* instead of *C*". To implement such an informal description using normal conditionals, we must disentangle the interrelated exceptions into a linear logical flow, producing a recipe a computer can follow. An adaptive conditional is different: it lets you write the program just like the informal description, and delegates the job of producing a recipe to the compiler.

You start by implementing the simple base case as a function. A special case is implemented by making a variant of the base case. This variant is edited to implement the exceptions in the informal description, literally replacing the code for *C* with code for *B*. There are now two functions: the base case, and a variant case. What we need to do is somehow blend these two functions together into a single function that does the right thing in all cases. To do this, we introduce *predicates*.

A predicate is a special Boolean-valued function. The simplest kind of predicate is to test whether two nodes are equal. Predicates conditionalize merging. When a child structure is merged from multiple parents, any of the parents containing a false predicate will be masked. Returning to the example, we must insert a predicate into the variant case that is true when *A*. Now merging the base case and the variant case produces a function with the right behavior. If *A* is false, the merge is equal to the base case. If *A* is true, the merge is equal to the variant, since all its edits override the base case in the merge. **Adaptive conditionals merge code variants conditionally – like "run-time version-control".**

Adaptive conditionals will be supported by a special presentation called a *case table*, somewhat reminiscent of decision tables [20]. Figure 11 shows a mock-up. The rows of the table are nodes and the columns are cases. Differences between the cases are indicated by background coloring (this will also be used as a general difference visualization). A factorial function is shown implemented by two cases. The basic case contains the recursion. That case is overridden by a zero case triggered by a 0 input, setting the result node to 1. The zero case is false in this call.

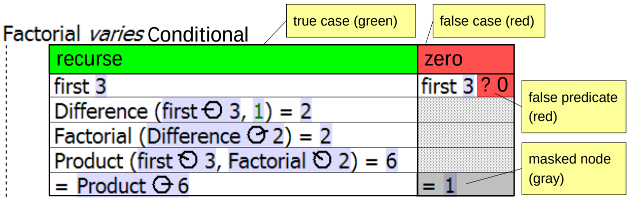


Figure 11. Adaptive Conditional

Adaptive conditionals are an extreme experiment, but with interesting potential benefits:

1. Normal conditionals must often be duplicated in multiple places, coordinated by a shared Boolean flag. Adaptive conditionals combine variations at scattered locations into a single variant case, capturing the meaning without redundant notation.

2. Predicates are similar to guarded expressions and pattern matching in functional languages [38], which can be a succinct form of expression.

3. It is easier to see the conditional logic of a program because it is moved into an orthogonal dimension from the rest of the program's structure. Conditional expressions, like choice, are intertwined with data flows. Conditional statements are more distinctive, with keywords and bracketed blocks, but block structure is overloaded for many other purposes. Adaptive conditionals use the "3rd dimension" of case table columns.

4. Cases can easily cause a conflict error by modifying the same node. This is not a bug, it is a feature. It indicates a combination of cases the programmer has ignored. The situation can be resolved by adding another case merging the conflicting ones and resolving the dispute. Normal conditionals are orthogonal: in every possible situation they will produce some result, perhaps incorrectly. The programmer must consider all these possibilities up front. Adaptive conditionals allow design decisions to be divided and conquered.

5. Adaptation supports intercession into code somewhat like Aspect Oriented Programming [18].

6. Conditional cases directly map to specific examples, and can be created directly from such examples [11].

7. Informal specifications are naturally expressed as tangled exceptions that can be directly modeled by adaptive conditionals. The conceptual gulf between the language of specification and the language of implementation is narrowed.

8. It is simpler and more concrete to create variants of code with different behavior than to abstract a single version of the code with conditional logic. Proper abstraction, if it is necessary, can be deferred to later refactoring. **Adaptive conditionals reduce the cognitive burden of abstracting dynamic behavior.**

6.3 Mutable State

Mutable state is a major dilemma of programming language design. Functional languages have avoided it for good reason (the chaos of side-effects), but at great cost in complexity (for example monads [43]). **Usability mandates support of mutable state simply because it is so deeply entrenched in common sense.**

The approach being explored involves recording the history of all changes so that complete copies of past states of the system appear to have been recorded. Copying is Subtext's forte, and it already has the ability to lazily instantiate copies with differences – exactly what an implementation of history recording requires. A scalable implementation would also need the ability to "forget" the details of history, replacing them with summarizations. Recording and forgetting history may seem a hopelessly inefficient scheme, but perhaps no more so than garbage collection seemed 30 years ago.

Subtext would thus reduce mutable state to copying (as with just about everything else). The challenge is to support a common sense notion of mutable state while maintaining the benefits of the complete static visibility of program dynamics. How can a program mutate state when there is no such thing as run-time? Subtext proposes *actions*: functions that take a reference to any subtree (called the *input state*), and produce an *output state* that is a modified copy of the input. The input state of an action is by default the current global state. The output of such an action is thus a *potential future* state, which visibly reveals what the action's effects would be, were it to be *executed*. Executing an action turns its potential future into the actual present. Note that actions are themselves part of the global state, and thus create recursive copies of the global state. Time is modeled as global recursion. Time is partially ordered: actions can be wired up in "state-flow" graphs to perform parallel computation free of implicit side-effects. **This approach combines the clear semantics of functional programming with a common sense notion of mutable state**. It will be up to specialized presentations to display history, futures, and state-flow in a simple and concrete way.

7. CONCLUSION

The exceptional difficulty of programming is in large part due to encoding programs as text strings, a design cemented in the very first programming languages. We have gotten as far as text will take us. **The metaphor of programming as writing is no longer helpful.**

Subtext offers an alternative medium to text, one designed from scratch to make programming easier by shortening mental leaps. The representation of a program is the same thing as its execution: syntax overtly aligns with semantics. Relationships are direct, not intermediated by delayed binding of symbols. Editing is coherent transformation of semantics. The essence of this new medium is copying: higher-order continual copying of trees generates a unified theory of both computation and programming. The traditional assortment of programming tools and formalisms collapses into one seamless workspace, with a simple and consistent conceptual model. Programming becomes more akin to using a spreadsheet than a keypunch.

Subtext is a fresh start. The initial prototype shows promise, but is still nascent. Much work, and risk, is left: Subtext opens up a whole new territory to explore. **There is hope that we can make programming fundamentally easier.**

8. ACKNOWLEDGEMENTS

Daniel Jackson, Rob Miller, Derek Rayside, Rob Seater, and Emina Torlak provided helpful discussions, and the Software Design Group at MIT provided a creative environment. The anonymous referees provided constructive criticism.

9. REFERENCES

[1] Backus, J. Can Programming be Liberated From the von Neumann Style?: A Functional Style and its Algebra of Programs. *Commun. ACM 21, 8* (1978) 613-641.

[2] Barendregt, H. P. *The Lambda Calculus*. Elsevier, 1984.

[3] Beck, K. *Test-Driven Development: By Example*. Addison-Wesley 2002.

[4] Blackwell, A. First Steps in Programming: A Rationale for Attention Investment Models. In *Proc. of the IEEE 2002 Symposia on Human Centric Computing Languages and Environments (HCC'02)* (Arlington, VA, Sep. 2002) 2-10.

[5] Burnett, M., Atwood, J., Djang, R., Gottfried, H., Reichwein, J., Yang, S., Forms/3: A First-Order Visual Language to Explore the Boundaries of the Spreadsheet Paradigm. In *Journal of Functional Programming 11, 2* (March 2001).

[6] Burnett, M., Goldberg, A., Lewis, T. *Visual Object-Oriented Programming: Concepts and Environments*. Manning, Greenwich, CT, 1995.

[7] Copeland, G., Maier, D. Making smalltalk a database system. In *Proceedings of the 1984 ACM SIGMOD international Conference on Management of Data* (Boston, Massachusetts, June 18 - 21, 1984).

[8] Cypher, A. *Watch What I Do: Programming by Demonstration*. MIT Press, Cambridge, MA. 1993

[9] Czarnecki, K., Eisenecker, U. W. Intentional Programming. Chapter 11 in *Generative Programming: Methods, Tools, and Applications*. Addison-Wesley, 2000.

[10] Djang, R., Burnett, M. Similarity Inheritance: A New Model of Inheritance for Spreadsheet VPLs. In 1998 IEEE Symp. Visual Languages (Halifax, Canada, Sep. 1998) 134-141.

[11] Edwards, J. Example Centric Programming. In *Companion to the 19th annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications (OOPSLA '04 Onward)* (Vancouver, BC, CANADA) 124. http://subtextual.org/OOPSLA04.pdf

[12] Fowler, M., Beck, K., Brant, J., Opdyke, W., and Roberts, D. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, 1999.

[13] Green, T. R. G., Petre, M. Usability Analysis of Visual Programming Environments: a 'cognitive dimensions' framework. *Journal of Visual Languages and Computing 7, 2*, 131-174.

[14] Hanna, K. Interactive Visual Functional Programming. In *Proceedings of the Seventh ACM SIGPLAN international*

Conference on Functional Programming (Pittsburgh, PA, USA, October, 2002).

[15] Ingalls, D., Wallace, S., Chow, Y., Ludolph, F., Doyle, K. Fabrik: A Visual Programming Environment. In *Conference proceedings on Object-oriented programming systems, languages and applications(OOPSLA '88)* (San Diego, California, United States, Sep. 1988) 176 – 190.

[16] Jones, S. P., Burnett, M., Blackwell, A. A user-centred approach to functions in Excel. In *Proc International Conference on Functional Programming (ICFP'03),* (Uppsala, Sweden, Sept 2003) 165-176.

[17] Kahn, K. M., Saraswat, V. A. Complete Visualization of Concurrent Programs and Their Executions. In *Proceedings of the ICLP 1990 Workshop on Logic Programming Environments,* (Eilat, Israel, June 1990).

[18] Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Videira Lopes, C., Loingtier, J.-M., and Irwin, J. Aspect-Oriented Programming. In *Proc. of European Conference on Object-Oriented Programming (ECOOP 1997)* (Jyväskylä, Finland, 1997).

[19] Kim, M., Bergman, L., Lau, T., Notkin, D. An Ethnographic Study of Copy and Paste Programming Practices in OOPL,. In *Proc. of the 2004 ACM-IEEE International Symposium on Empirical Software Engineering* (Redondo Beach, CA, Aug. 2004).

[20] King, P. J. H. Decision Tables. *Computer Journal 10, 2* (Aug. 1967) 135-142.

[21] Lewis, C., Olson, G. M. Can Principles of Cognition Lower the Barriers to Programming? In *Empirical Studies of Programmers: Second Workshop* (Washington D.C. 1987) 248-263.

[22] Lieberman, H. Using Prototypical Objects to Implement Shared Behavior in Object-Oriented Systems. In *Conference proceedings on Object-oriented programming systems, languages and applications (OOPSLA'86)* (Portland, Oregon, United States, 1986), 214-223.

[23] Lieberman, H. Tinker: A Programming by Demonstration System for Beginning Programmers. In [8].

[24] Lieberman, H.(editor) *Your Wish is my Command: Programming by Example*. Morgan Kaufmann, San Fransisco, CA, 2001.

[25] Myers, B. Taxonomies of Visual Programming and Program Visualization. *Journal of Visual Languages and Computing 1, 1* (March 1990), 97-123.

[26] Neal, L. R. Cognition-sensitive design and user modeling for syntax-directed editors. In *Proc. of the SIGCHI/GI conference on Human factors in computing systems and graphics interface* (Toronto, Ontario, Canada, 1987) 99-102

[27] Noble, J., Taivalsaari, A., Moore, I. *Prototype-Based Programming: Concepts, Languages and Applications*. Springer-Verlag Telos, January, 1999.

[28] Parnas, D. L. On the criteria to be used in decomposing systems into modules *Commun. ACM 15, 12* (Dec. 1972) 1053-1058.

[29] Norman, D. A. *The Design of Everyday Things.* Basic Books, New York, 1988.

[30] Petre, M. Why Looking Isn't Always Seeing: Readership Skills and Graphical Programming. *Comm. ACM 38, 6* (June 1995) 33-44.

[31] Reps, T., Teitelbaum, T. *The Synthesizer Generator: A System for Constructing Language-based Editors*. Springer-Verlag, New York, 1988.

[32] Schärli, N., Ducasse, S., Nierstrasz, O., Black, A. Traits: Composable Units of Behavior. In *Proc. of European Conference on Object-Oriented Programming (ECOOP 2003)*, (July 2003), 248-274.

[33] Shneiderman, B. Direct Manipulation: A Step Beyond Programming Languages. *IEEE Computer 16, 8* (August, 1983) 57-69.

[34] Smith, R. B., Maloney, J., Ungar, D. The Self-4.0 user interface: manifesting a system-wide vision of concreteness, uniformity, and flexibility. In *Proc. of the tenth annual conference on Object-oriented programming systems, languages, and applications (OOPSLA '95)* (Austin, Texas, United States, 1995), 47-60.

[35] Szwillus, G., Neal, L. (editors) *Structure-based Editors and Environments.* Academic Press, San Diego, CA, 1996.

[36] Tanimoto, S.L. Towards a theory of progressive operators for live visual programming environments. In *Proc. of the 1990 IEEE Workshop on Visual Languages* (Skokie, IL, USA), 80-85.

[37] Toomim, M. Begel, A., Graham, S. L. Managing Duplicated Code with Linked Editing. In *2004 IEEE Symposium on Visual Languages - Human Centric Computing (VLHCC'04)*(Rome, Italy, Sep. 2004) 173-180.

[38] Turner, D. A. Miranda: A non-strict functional language with ploymorphic types. In *Proceedings IFIP International Conference on Functional Programming Languages and Computer Architecture,* Nancy France, September 1985.

[39] Ungar, D., Lieberman, H., Fry, C. Debugging and the experience of immediacy *Commun. ACM, 40, 4,* (April 1997), 38-43.

[40] Ungar, D., Smith, R. B. Self: The power of simplicity. In *Conference proceedings on Object-oriented programming systems, languages and applications (OOPSLA '87)* (Orlando, Florida, United States, Oct. 1987), 227-242.

[41] Ungar, D., Smith, R. B. Programming as an Experience: The Inspiration for Self. In *ECOOP '95 Conference Proceedings,* LNCS 952. Springer Verlag, 1995

[42] Wadge, W. W., Ashcroft, E. A. *Lucid, the Dataflow Programming Language.* Academic Press 1985.

[43] Wadler, P. The essence of functional programming. In *19th Symposium on Principles of Programming Languages* (Albuquerque, NM, January, 1992).

[44] Waters, R. C. Series. In *(Steele, G. L.) Common Lisp the Language*. Digital Press 1984