Curl: A Language for Web Content

Steve Ward

Massachusetts Institute of Technology, Cambridge, Massachusetts

Mat Hostetter

Curl Corporation, Cambridge, Massachusetts

Abstract: We describe a language designed for the representation of a broad spectrum of web content, including formatted text, graphics, and programmed application-level function. The approach described maps conventional markup tags to underlying, more general programming constructs, and provides local extensibility of the markup language by addition of programmed objects, procedures, and macros to the underlying object-oriented programming language. An implementation strategy based on a mix of static and dynamic just-in-time compilation techniques is described. The discussion focuses on a number of specific technical challenges raised by the language's breadth and performance goals, and the impact of these issues on Curl's architecture.

Key words and phrases: web language, programming language design, content delivery, incremental compilation, extensible language design, markup language

Biographical Notes:

Steve Ward is a Professor of Computer Science and Engineering at the Massachusetts Institute of Technology, where his current research interests focus on component software architectures and assembly for web applications. Past research foci have been in computer architecture, operating systems, and languages. Steve has been in charge of a core course in digital systems architecture since 1980. He holds SB, SM, and Ph.D. degrees from MIT.

Mat Hostetter is the compiler architect at Curl Corporation. His research interests include compilers, emulators and language design, as well as algorithms of all kinds. Mat holds SB and M.Eng degrees from MIT.

1.0 Introduction

Since the advent of high-level programming languages, there has been an ongoing tension between the goal of unifying mechanism into a single general-purpose language and the natural proliferation of independent languages specialized to particular application domains. Compelling arguments can be made for both sides of this issue. On one hand, much of the power of modern computer technology stems from its basis on universality: the fact that a toolkit consisting of application-independent algorithmic primitives can be reconfigured to address arbitrary new application domains. On the other hand, within every domain, progress is reflected in the specialization of the algorithmic toolkit to the particular application area, typically by the addition of layers of primitives specific to that application domain.

In 1995, a group including the authors undertook to explore this tension within the emerging universe of web content. Then, as now, pressures to enrich web content were stimulating a growing array of independently conceived, and generally incompatible, language technologies specialized to each content type. HTML and its derivatives dominate the markup sector, scripting languages on both client and server have proliferated for simple programmed behavior, Java[1] has become a de facto standard for applet and servelet programming, Flash and RealMedia deliver high-performance graphics and streaming A/V data, and C++ (accompanied by various delivery technologies) remains prevalent for high-performance application-level functionality. The general research question addressed by the authors during the mid-90s was this: can a single, semantically coherent language framework address the needs of this broad range of content, unifying

their common aspects via universal language primitives while providing for the unique requirements of each content type?

Several characteristics were identified as critical to the success of the desired universal content language. These included:

Extensibility. The specialization of language constructs to the needs of specific application domains remains the powerful attraction of domain-specific languages, while the cost of gratuitous linguistic differences among independently conceived languages motivates our search for general-purpose solutions. This tension suggests a compromise involving a flexible core language framework whose extensibility, both syntactic and semantic, supports domain-specific constructs within the context of a coherent basis for common requirements.

Approachability. The explosive growth of Web content is due in part to the simplicity of its principal representation technology, HTML; however, limitations inherent in that technology are responsible for the proliferation of alternative, incompatible content representation technologies such as Javascript and Java. In an effort to offer HTML's approachability to neophyte users while affording advanced mechanisms to the programming sophisticate, the authors embraced the gentle slope philosophy espoused by a 1992 DARPA study[2].

Gentle slope refers to the minimization of discontinuities in the curve relating function to sophistication required from its creator. In the context of web content technologies, the gentle slope argument is illustrated in figure 1, where the discontinuities between independent implementation technologies are eliminated by use of a single implementation language. The goal of the gentle slope approach is not necessarily to reduce the sophistication required to implement a given function; rather, it is to assure that incremental functional improvements are attainable through incremental additional sophistication on the part of the creator. By elimination of the overhead of mastering new languages at each discontinuity, however, a unified language approach tends to lower the total intellectual investment necessary to implement advanced functions. In addition to the advantage of lowered conceptual burden on the content author, the use of a single coherent language to address the entire content spectrum offers potential functional improvements by minimizing communication barriers between component technologies. Scripting, programming, and markup may, for example, share a single namespace for variables and other content elements.

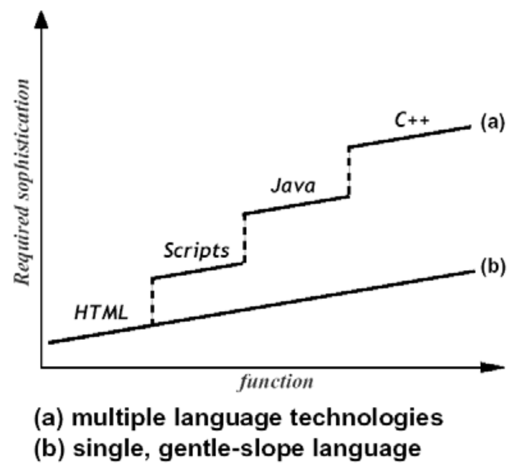

Figure 1: The Gentle Slope

Incrementalism. Even in the mid-1990s it was apparent to many, including the authors, that the Web could eventually provide simple, incremental access to application-level function traditionally supported by heavyweight software installations fraught with configuration management and other complexities with which the typical user is ill-equipped to cope. A key to following this evolutionary path was seen to be the adoption of the browser model for Curl: a user browses a universe of Curl content, each “page” of which dictates the presentation or experience presented to the user. There are many practical differences between user expectations from the browser model and those associated with conventional applications; among the most challenging is the typical intolerance of page-load delays in a browser that have become acceptable, at least grudgingly, when sustained in the invocation of a conventional application.

The technical challenges faced by this approach revolve about two related issues:

- whether the requirements over the content spectrum are so disparate as to defy integration of their representations, leading to an agglomeration of orthogonal mechanism into an unwieldy and incoherent patchwork of language elements; and
- whether a single solution of the required breadth can be competitive at each point on the spectrum with conventional technologies of narrower scope.

Sections 2 and 3 of this paper provides a telescopic glimpse of the language design decisions that resulted from the above desiderata, barely sufficient to support subsequent discussion here; for more detail, the interested reader is referred other sources[3, 4, 5]. Subsequent sections detail several relevant implementation technologies and provide a retrospective evaluation of Curl's architectural choices in the context of contemporary alternatives.

2.0 Curl: first glimpse

The design of Curl is based on the observation that a parenthesized leading polish notation serves as an effective syntax for markup, but can serve as well as the syntactic basis for an extensible language for serious programming and structured data. The approach taken in Curl bases markup-level tags and syntax on contemporary programming constructs, supporting markup as simple examples of a rich and deeply layered contemporary programming language.

Following the HTML model, top-level Curl source is natural language text with interspersed markup and other non-textual content enclosed in syntactically distinct punctuation. In order to minimize the burden on text authors to escape common punctuation, Curl attaches special significance to only four infrequently-occurring characters at this lexical level: `{`, `}`, `|`, and `\`. The curly brackets identify markup and programmatic content; vertical bar is used for comments and lexical quoting; and backslash is a character-level escape.

HTML-style formatted text is represented in Curl as text embellished with markup enclosed in curly brackets, e.g.

```
More information on {bold Curl} is available
at {link href={url "http://www.curl.com"}, www.curl.com}.
```

whose `{bold ...}` and `{link ...}` forms have much the same effect as the corresponding `<B>...</B>` and `<A>...</A>` forms in HTML. Unlike HTML elements, however, the curly-enclosed markup forms of Curl are names bound in an importation environment to a rich and extensible universe of programmed objects. Curl's uniqueness as a language stems in large part from this nexus between markup and programming constructs, aspects of which are explored further in subsequent sections.

Beyond these specific "content language" issues, Curl has a number of features which generally support its intended uses, ranging from interactive browsing experiences to the delivery of application-level function as platform-independent web content. These features include:

- Platform independence. The semantics of Curl and its primitives are designed to be independent of operating-system specifics. Linux and Windows implementations of Curl are in current use.
- An optimizing JIT compiler that dynamically generates native code.
- A code packaging model incorporating *versioning*, allowing applets and library code to document their dependencies on versions of the Curl runtime system and allowing multiple versions to run simultaneously.
- Language support for units and dimensional checking and conversion, along lines suggested in [6,7,8]. A value of `2(m/s)` denotes velocity in meters per second; it can be multiplied by a *time* such as `22min` to yield a *distance* quantity, but generates an error if it is added to, say, a *mass* quantity like `3kg`.

- Internationalization. Text is represent in unicode, and support is provided for customization of local syntax preferences for such data as dates and times.
- First-class types and procedures, with efficient compiled support for parameterized types and dynamic procedure creation (via closures).
- Reflection and introspection: support for run-time inspection, modification, and construction of arbitrary data, including run-time compiler invocation.
- Efficient (compiled) support for passing multi-component values (such vectors representing points in 3-space), including multiple-valued returns from procedures and methods.

The intent of Curl's feature set is to support the a mix of sophisticated programmers and technically naive authors, allowing the former to contribute domain-specific extensions easily accessed by the latter.

3.0 The markup/programming nexus

The promotion of markup constructs to a special case of the application of programmed objects within a more general programming model is central to Curl's integration of programmed and textual content, and the provision of a rich variety of applicable object types within that model is the basis for its unusual extensibility. This section explores the mechanism underlying these properties of the language.

3.1 The Curl extension property

The fundamental syntactic property of Curl, and the source of its name, stems from the processing of forms enclosed in curly bracket characters. Curl content is always presented to the Curl processing "engine" in a form semantically equivalent to source, although various alternative forms offer compression and obfuscation advantages. The engine JIT-compiles and executes curly forms in a compilation environment containing bindings for all accessible primitives. When a form like

```
{keyword ...}
```

is processed, the object bound to **`keyword`** in the compilation environment dictates both the semantics and the syntax of the remainder of the form. In general, the curly form is processed by invoking that object, passing it arguments which represent the remaining content of the curly form after some level of preprocessing depending on the type of the invoked object. Typically, this processing yields as a value an object to be interpolated in the stream of output being presented to the user via his browser window; thus simple markup, like the **`{bold Curl}`** in the above example, produces a graphic object that displays as bold-faced text to the user. More sophisticated return values, such as the value of the **`{link ...}`** form above, can convey graphics, tables, animations, or objects exhibiting arbitrary behavior and user interfaces.

Since markup tags are names which evaluate to arbitrary Curl objects, simple extensions to the markup language can be effected by the definition of procedures and classes to be used as new tags as sketched in subsequent sections.

3.2 Programming Syntax

The most straightforward embedding of programming in a parenthesized polish notation leads to a LISP-like programming syntax, and this approach was taken in early Curl implementations[9,10]. Although diehard LISP programmers found this acceptable, the vast majority of programmers find conventional infix notation such as **`a[i]+3`** to be much more readable than a pure prefix version like **`{+ {aref a i} 3}`**. Unfortunately, the infix syntax of conventional programming languages does not extend well to text and markup, leading to a tension between Curl's programming and markup goals.

This issue is resolved in modern Curl by a syntactic compromise that preserves the {...} semantics essential to the extension property of the previous section, but which allows conventional infix expressions in contexts where program constructs are expected. Moreover, curly brackets may be omitted from certain common constructs to enhance readability. With this relaxation of its syntax from the Spartan LISP extreme, complaints among Curl initiates from the mainstream programming community diminished substantially. Of course, this improvement comes at the cost of additional complexity, both in the syntax of the programming sublanguage and in its implementation.

3.3 Procedures as markup

When **`{keyword ...}`** is encountered and **`keyword`** is bound to a procedure object, the remainder "**`...`**" of the form is parsed as some combination of positional and named arguments and the run-time value of that form becomes the value returned by a compiled call to the indicated procedure. Procedures may be defined by forms like

```
{define-proc {ancestors gen}
   {if gen == 0 then
      {return 1}
    else
      {return 2 * {ancestors gen - 1}}
   }
}

{define-proc {ancestor-name gen}
   {if gen == 1 then
      {return {text parents}}
    elseif gen == 2 then
      {return {text grandparents}}
    else
      {return {text great-{ancestor-name gen - 1}}}
   }
}
```

and used in subsequent content, such as

```
The number of {italic {ancestor-name 6}} you have is {ancestors 6}.
```

which would yield the displayed text "The number of *great-great-great-great-grandparents* you have is 64."

The default parsing of procedure arguments parses the **`6`** in **`{ancestors 6}`** as an integer data value, rather than as text. A more general mechanism is required to define new markup tags designed to take arbitrary source content as input, exemplified by

```
{define-text-proc {sale ...}
   let color = "green"
   let flash = {TextFlowBox color = color, ...}
   {flash.animate
       interval = 1s,
       {on TimerEvent do
           {if color == "red" then
               set color = "green"
            else
               set color = "red"
           }
           set flash.color = color
       }
   }
   {return flash}
}
```

The ellipsis (“`...`”) in the formal parameters of the above definition matches arbitrary content, which gets embedded in a graphic object (a **`TextFlowBox`**) configured to flash red and green at 1-second intervals. Given this definition, the content

```
{big {bold Chrome-plated Trailer Hitches -- just {sale $99.95}}}
```

would yield a garish advertisement with a blinking price tag.

Simple format extensions may be specified by a shorthand allowing a new format object to be specialized from an existing one. For example, the definition

```
{define-text-format emphasis as text with
    font-style = "italic",
    font-family = "serif"
}
```

defines a variant of the **`text`** expression with alternative font parameters, whence

```
{paragraph You are in {emphasis big} trouble!}
```

becomes a concise equivalent to

```
    {paragraph You are in
               {text font-style = "italic",
                     font-family = "serif",
                     big}
               trouble!

}
```

3.4 Objects as markup

Classes are themselves objects, and may be instantiated by their application to actual parameters in much the same way as procedures. As a **`{keyword ...}`** form is processed where **`keyword`** is bound to a class object, a call is compiled (again using actual parameters parsed from the remainder of the form) to a method that instantiates a new member of the class. Thus although the neophyte user may view a form like

```
{Table columns=2,
       {bold IBM},  {get-quote “IBM”},
       {bold MSFT}, {get-quote “MSFT”}
}
```

as HTML-style markup, in fact it represents code that is compiled and executed to instantiate a **`Table`** object containing ticker symbols and real-time stock quotes.

3.5 Curl Types

One of the tensions that makes the gentle slope goal interesting pits the conceptual simplicity of “typeless” or dynamically-typed variables of scripting languages against the robustness and efficiency advantages of the strong type systems favored by professional programmers. Curl hedges this choice by providing a system of strong types with attendant compile-time type checking, but including ambiguous types whose run-time representation includes tags to carry type information in addition to the storage of values.

Type declarations are generally optional; variables and other values whose types are undeclared default to type **`any`** whose run-time representation includes complete type information as well as a run-time value consistent with that type. Variables and parameters without specified types, or those explicitly declared as **`any`**, offer scripting-language flexibility at the cost of run-time efficiency and absence of compile-time type checking; such code can be freely intermixed with fully-declared types. Thus the declarations

```
let s1:String = “Hello, World”
```

```
let s2 = "Hello, World"
```

instantiate a pair of variables with identical initial values but different compile-time types. Accesses to `s2` will involve run-time type checking and hence be slower; assignment of (say) an integer value to `s1` (but not to `s2`) will yield a compile-time error.

3.5.1 Class definitions and objects

Curl's implementation of objects follows a fairly conventional approach among compiled languages. The model supports full multiple inheritance, and offers protection attributes for both classes and their members (methods, fields, etc.). Specially-declared *accessor* methods can be defined to mimic the behavior of passive fields, allowing an apparently simple assignment like

```
set g.width = 4cm
```

to cause execution of arbitrary code, perhaps reformatting contents of the graphic g in some class-specific way.

After considerable debate in the early history of Curl about alternatives (such as *generic functions*), mechanism for adding methods to pre-existing classes was deliberately omitted from the language. Reasons for this decision include our lack of good answers to fundamental semantic questions (Is there a representation for the generic *length*? Is it bound in a global namespace?) and nervousness about its impact on our gentle slope goals (Will the neophyte understand which *length* is being applied? Need he?). The authors periodically reconsider this debate, and have thus far reaffirmed its conclusion.

3.5.2 First-class types

The decision to support dynamic (run-time) types necessitates a run-time representation for type information, leading to the status of types in Curl as first-class data objects. Types, including classes, can be passed as arguments and returned as values; they can be created dynamically, assigned to variables, used for type discrimination via Curl's `isa` predicate, and used to instantiate new objects.

These semantics provide a natural syntax for parameterized types, allowing declarations like

```
let a:{Array-of {Stream-of char}}
```

to declare an array of character streams thereby restricting the type of each stream and array element to a compile-time constant (`char` and `{Stream-of char}`, respectively). This contrasts with syntactic add-ons in proposals for parameterized types in Java[11] and C#[12], languages in which the role of types is more confined.

In order to provide guarantees of reasonable compile-time behavior (*e.g.*, termination of type checking), Curl does not currently allow arbitrary type-valued expressions (involving, say, user-defined procedures) to appear in declarations despite the absence of syntactic obstacles to such generality. Certain useful special cases are, however, provided for; these include the user definition of *parameterized classes* whose definition involves parameters that may be bound to compile-time constants. Thus a sophisticated Curl programmer developing a generic approach to inventory management might define a parameterized class `{Inventory-of <type>}` so that less sophisticated users writing application code might instantiate data structures of type `{Inventory-of Milk}` or `{Inventory-of MicrowaveOvens}`.

The objects representing parameterized types are compiled lazily on demand. During compilation, the specified parameters appear as compile-time constants, enabling the exploitation of parameter-specific optimizations. In our static compiler, parameterized type method instantiations that compile to identical machine code are coalesced, and our dynamic linker automatically associates newly instantiated methods with pre-existing parameterized types to reduce redundant compilation.

3.6 General extension mechanism

In its most general application, the syntactic extension property of Curl allows sublanguages of nearly arbitrary syntax and semantics to be embedded in Curl source. One could, for example, embed C, Java, or HTML code via a form like

```
{C for (i=0; i<5; i++) j += i; }
```

Such a form is used to within a curl-to-C translator to embed literal C code in a Curl program. It parses special escapes to determine where in the C code to insert accesses to Curl variables and expressions.

More interestingly, this mechanism can be used to embed within Curl compiled sublanguages specific to particular application domains. Library code for a particular application domain might provide a primitive like **`{chemical-flow-plant ...}`** whose remaining contents (the "`...`" in the above form) constitute a description *in language natural to the domain* of a chemical flow plant and whose resulting value produces an appropriate animated diagram. Such extensions have been written to support a wide variety of simple graphical applications (like equations and organization charts).

More ambitious extensions have involved substantial excursions from the default Curl semantics, exploiting the compile-time processing afforded by the extension mechanism described in this section. These include a Prolog-like planning subsystem based on goals, and a generic function definition mechanism.

3.6.1 Macros

Curl's extensibility is not limited to markup, but offers general macro facilities influenced by a model presented by Bachrach and Playford[13]. Using **`define-macro`**, a sophisticated user can use Curl's powerful procedural macros to extend the language in nearly arbitrary ways.

Well-chosen macros can add clarity to source code by reducing a complex operation to a simple form, making the code easier to read, write, and maintain. For example, Curl provides a simple **`with`** macro that temporarily sets one or more values, executes some code, and then restores the original values, even if an exception is thrown:

```
{with self.busy? = true do
    {self.do-something}
}
```

Of course, a programmer could always implement this idiom by manually declaring temporary variables and writing out a **`try/finally`** block longhand, but the result would be ugly and harder to maintain. Writing a macro for this idiom makes the code elegant.

Curl's macros are *procedural*, meaning that they are compile-time procedures that take source code as input and return new source code as output. Unlike many languages, Curl's macros are not limited to a simple rewrite language executed by a preprocessor; the full power of the language is at their disposal. They can create objects, call subroutines, and analyze their inputs however they like.

Curl's **`format`** macro, similar to C's **`printf`**, uses these analysis capabilities to make sure that the number of arguments matches the format string. It also substantially optimizes certain cases at compile time, such as transforming a call where the format string contains no percent signs to a faster call that simply prints the characters without further parsing at runtime.

3.6.2 Pattern matching

Curl provides a backtracking pattern-matcher to assist in writing macros. The pattern-matching language supports regular expression concepts like **`one-of`**, **`optional`**, and **`sequence`**, as well as shorthand for var-

ious Curl language constructs like **identifier**, **expression** and **literal**. Here is a simple example, using the actual implementation of Curl's **unless** feature:

```
{define-macro public {unless ?predicate:expression do ?body:statements}
      {return {expand-template {if not ?predicate then ?body}}}
}
```

Macros typically parse their input into pieces using patterns, and **expand-template** gives them a way to put them back together again. It produces new source code from a template, with **?** escapes substituting in other source code objects. Interestingly, **expand-template** is also just a macro that transforms a simple template syntax into calls to lower-level parser routines.

Pattern matching is not restricted to the arguments to a **define-macro**; it is available via **syntax-switch** (which is just a macro). **syntax-switch** takes a source code object as input, matches it against a sequence of patterns in turn until it finds one that matches, and then executes the code for that case.

For example, this **syntax-switch** distinguishes between two different ways to invoke a **for** loop:

```
{define-macro public {for ?args:tokens}
    {syntax-switch args, must-match? = true
     case {pattern for ?x:identifier = ?low:expression
                                       to ?high:expression
           do
              ?body:statements
          }
     do
         || Handle looping over a range of values.

     case {pattern for ?x:identifier in ?container:expression do
              ?body:statements
          }
     do
         || Handle looping over the elements of a container.
    }
}
```

3.6.3 Macro hygiene

Curl macros are *hygienic*, meaning that they protect the macro author from accidental name clashes in a way not possible in simpler macro systems like that of C/C++. Each macro expansion creates a new namespace for any identifiers defined and used by that macro. Because they are in a new namespace, temporary variables defined by the macro are, by default, invisible to the caller, and undefined identifiers in the macro definition are looked up in the package where the macro was defined, rather than where the macro was called.

For example, in the **unless** example above, suppose the caller of the macro had defined a procedure called **if** (not a good idea!) One would naturally wonder whether that user calling **unless** would fail, because **unless** simply expands into a call to **if**. Fortunately, it works. The hygiene rules tell the compiler that **if** should mean what it meant in the package where **unless** was defined, not what it means to the caller.

Macro hygiene provides a means by which naive or untrusted users can be given restricted access to inherently dangerous primitives. A package written by a trusted system programmer might exclude a resource (such as a lock) from external access, but publicly export a macro that uses the lock in a manner carefully engineered to guarantee safety. While the untrusted user of the macro cannot access and abuse the lock, the exported macro can afford access to the higher-level function that requires use of the lock.

3.6.4 Arbitrary syntax

The arguments to a macro need not be Curl programming constructs. While mechanism for parsing Curl expressions is provided, its use is optional; the macro can choose to parse its input as any combination of Curl expressions, raw text, or Klingon prose if desired. The only constraints on the syntax of the extension are those necessary to assure that the lexical parser can identify the endpoint of that form. Thus, for example, internal curly brackets must be either balanced or escaped.

To efficiently support the low-level processing of source content at the level required for this generality, the language features the **`CurlSource`** class capable of representing source content at various levels of parsing. CurlSource and its subclasses offer a variety of off-the-shelf parsing alternatives, as well as tracking source file and line/character position information to support error reporting in language extensions.

3.6.5 Macros querying the compiler

Even arbitrary syntactic analysis sometimes isn't enough, so Curl provides even more powerful macros using **`define-syntax`**. Such macros, which can only be called in contexts where code is being generated, can actually query the compiler to determine the compile-time types of their arguments and generate code accordingly. For example, Curl's **`for`** loop construct (which can iterate over either a range of numbers or the contents of a container, using the flexible parsing mechanism to distinguish which case it is) examines the type of the container it's being asked to loop over and generates code specialized for that type.

3.7 Options

Early in the development of Curl we encountered an interesting mismatch between the conventional lexical binding applicative semantics provided by Curl and the needs of a markup language. Consider a simple nested hierarchy describing formatted text:

```
{text font-style=”italic”,
      This text is italic; some of it
      {text color=”red”, is red.}
}
```

In this example, our natural expectation is that the “italic” property applies to the entire sentence while the last two words are colored red as well. This is awkward to implement, since the inner **`{text ...}`** form is processed before the outer one, and does not by default inherit values from the environment corresponding to the processing of the outer form. The processing by the outer **`{text ...}`** could, of course, crawl over its arguments tediously changing non-italic text to italics; but we sought a more natural and efficient means to this end. The problem is that the above code generates a hierarchy of graphic objects, and we’d like the binding of graphic properties like italic to follow the structure of the *graphic* hierarchy rather than the lexical structure of the code constructing that hierarchy.

To this end, Curl class definitions may declare *nonlocal options*, typed data members which may be set at runtime, but which inherit default values from a specified *parent* object. Curl graphical objects, for example, are based on a class **`Graphic`** which declares a number of nonlocal options including **`font-style`** and **`color`**.

Graphical objects are organized into hierarchies wherein each object other than the root has a parent object from which it can inherit nonlocal option values. A nonlocal option, e.g. **`font-style`**, may be bound to an explicit value at any node in the hierarchy; if not bound, its value at that node is inherited from its specified parent. Thus, in the tiny hierarchy of the above example, the inner **`{text ..}`** will inherit the italic property from the outer one as expected.

An option (local or nonlocal) is accessed like other class members; for example, the color of a graphic object **`g`** can be referenced by **`g.color`**. However, setting or unsetting a nonlocal option like **`g.color`** invokes mechanism to assure that the inheritance semantics of the option are maintained. This may involve propa-

gating the new value to children of `g` in the case of a set, or finding a new inherited value from an ancestor of `g` in an unset. An object may intercept changes in option values via *change handlers*, allowing objects to react to changes in option values in arbitrary ways.

4.0 Supporting the incremental execution model

The goal of delivering application-level function and performance as platform-independent web content raises several important implementation issues not confronted by conventional languages and development systems. These include

- Dynamic delivery vs. code base size. It is practically infeasible to deliver a runtime environment as rich as that offered by Curl entirely as source to be JIT-compiled on the client, if only from the standpoint of real-time performance overhead.
- Incremental rendering. Modern browsers render content incrementally as a page is loaded, allowing the user to view early page content while subsequent content is being read and thereby reducing apparent page-load delays.
- Modularity. The initial loading of a substantial runtime system, even from precompiled binaries cached locally at the client, entails delays (as well as memory overhead) that are inconsistent with tolerable page-load times for a browser.

The current implementation of Curl takes the form of the *Surge Runtime Environment*[5], which includes a client-side JIT compiler and support library. It is packaged in several forms, including a plug-in for contemporary browsers, and is written in Curl. In this section, several aspects of its implementation strategy that relate to the above issues are discussed.

4.1 Static vs. Dynamic compilation

An early architectural decision in Curl's history dictates that Curl content will be distributed in source form -- like HTML and XML, but in contrast to the precompiled byte code approach taken by Java and other languages. This choice dictates that a Curl-equipped client must include a full JIT compiler, as opposed to a potentially smaller virtual machine execution environment. There are a number of compensating advantages, including Curl's freedom to assume source language compilation as a feature of its runtime -- supporting a dynamic programming environment as well as the "lazy" postponement of certain compilation steps to runtime.

Since the code base is itself written in Curl, a systematic route to the bootstrapping of the system to the native code of specific platforms is required. Curl's compiler is structured as a single platform-independent front end producing an internal, portable intermediate representation of code. A platform-specific code generator translates the intermediate representation to native code; code generators exist for each supported processor type. The fact that the intermediate representation is not published as an external interface standard allows some flexibility in its evolution; work in progress, for example, is replacing an older intermediate representation by a contemporary *static single assignment* (SSA) form[14].

An additional code generator translates the intermediate code to stylized C, which can be compiled to native code by a platform-specific C compiler. This path is used to batch-compile the bulk of the Curl runtime code base to platform-specific native code, a time-consuming process involving the state-of-the-art optimizations available via contemporary C compilers. This *static compilation* process allows the automatic construction of several platform-specific Curl runtimes from a common Curl code base, assuming availability of a C compiler for each platform.

The alternative *dynamic* compilation path shares the same front end, but applies an incremental *just in time* (JIT) backend to generate platform-specific code in real time. This compilation route is used in the processing of Curl code embedded in pages being browsed, and entails a lower level of optimization due to the real-

time sensitivity of the browser execution model. Since the two compilation paths share the same front end code, they are semantically nearly indistinguishable (with several minor exceptions, such as the ability to escape to inlined C code in the static compilation path). JIT-compiled code is transparently cached on the client disk, eliminating redundant compilation of frequently encountered programming.

Although this dual-path compilation strategy has served us well, translation to C is unfortunately incompatible with use of a precise garbage collector. We are therefore considering using the JIT backend as a static compiler, to gain complete control over the generated code. Given the caching of JIT-compiled code, the basic mechanics of this simplification are straightforward; complicating issues include the difference in optimization levels between the paths, and support for in-lined C which appears occasionally in statically-compiled code to serve as glue between Curl and the host operating system.

4.2 Incremental Rendering

Browsers render passive HTML content incrementally, displaying early portions of a page while the remainder of the page is still loading. This technique is both effective and reasonably straightforward with HTML content, since the rendering of each HTML element is independent of subsequent elements on the same page. HTML has a number of *reverse dependencies*, in which an early element (such as the specification of a base URL) can effect the interpretation of subsequent content on the page, but *forward dependencies* are generally absent.

Unfortunately, forward dependencies are both common in, and essential to, modern programming paradigms. The definition of mutually-recursive procedures or cross-referencing object classes, for example, require early references to values subsequently defined. Given that Curl's aspirations include its status as a state-of-the-art programming language, a general taboo on forward dependencies seems out of the question. Moreover, the C approach (detaching early declarations, often in header files, from subsequent definitions) was rejected as a tedious burden on the content author. On the other hand, the goal of performance-competitive browsing of arbitrary content -- including HTML-like passive text -- requires incremental rendering at least where feasible.

Curl's compromise in this dimension involves a distinction between top-level content, often consisting largely of browsable text, and imported packages containing code. Top-level content is incrementally rendered to yield browser-like performance, while the importation of a Curl source package involves loading its source in its entirety and multiple-pass processing to resolve forward dependencies. This approach leads to minor constraints on top-level content; the definition of mutually-recursive procedures, for example, may cause a compile-time error. This problem is partially mitigated by batch-compiling consecutive blocks of code not separated by textual content, so that consecutive definitions of mutually-recursive procedures will work; but arbitrary forward dependencies, interspersed with text, can cause errors unless they are moved to an imported package rather than appearing at the top level. This approach also may entail a modicum of redundant rendering, as in the top-level content

```
{let scrap = {text color="blue", scrap of blue text}}
Here is a {value scrap}.  Subsequently, it will
change to red...
{set scrap.color = "red"}
```

in which a scrap of text will be rendered in blue and red in rapid succession. In extreme cases, the user may witness real-time revisions of objects rendered early on a page as forward dependencies are resolved by later content. Of course, an author who understands the execution model can easily write the content in such a way as to avoid such problems, e.g. by ensuring that each value to be displayed is completely finalized before it is encountered in a context that will cause it to be rendered. In the above example, a simple cure is to move the change of color before the reference to `{value scrap}`.

4.3 Atomization

The Surge Runtime Environment is distributed using a fine-grained backend modularization technique we call *atomization*. Each procedure, method, string constant, class, etc. in the runtime is statically compiled to machine code or binary data and stored as a separate *atom*. These atoms form the fundamental building block for our downloading, execution, and patching operations. Atomization yields dramatically improved startup times, reduced memory usage and smaller patches.

An atom is a block of bytes plus references to other atoms. For example, suppose a tiny program has a procedure **`main`** that calls procedure **`printf`**, passing it string constant **`"hello, world"`**. The program will have three atoms. The atom for **`main`** will have bytes composed of its raw machine code, and will have references to the atoms for **`printf`** and the string constant.

The *atom loader* is a small C program that manages a set of atom databases. It loads and decompresses atoms from disk into memory buffers as those atoms are first needed. Storing atoms on disk in exactly the order they are typically accessed and loading them only as they are needed and improves startup time, memory usage and cache locality. Sophisticated random access code decompression algorithms are described by Lucco[15] and Lekatsas and Wolf [16], but we implemented a simpler approach because our needs were not as demanding. Our decompressor does, however, handle lazy loading of static data in addition to code.

The Curl runtime is versioned at the level of individual atoms. Patches need only update information about atoms that changed, rather than replace entire libraries or executables as many traditional patchers do. Since a "critical update" might affect only a handful of important procedures, this can result in a dramatic reduction in patch size. This is true even when compared against "binary diff" algorithms such as *xdelta*[17] applied to dynamically linked libraries (*DLL*s), because changes in the size of one procedure in a DLL can modify addresses and offsets stored throughout the rest of the file, resulting in an unnecessarily large patch file.

Atom patch files contain a sequence of graph transformations that render an atom graph already on the user's machine isomorphic to the atom graph of the desired program. The more similar a new program is to programs already installed, the smaller the patch file. Atom patches do not actually modify atoms, since atoms are immutable; rather, they update an indirection table that "replaces" old atoms with new ones in the patched program. These indirection tables allow us to simultaneously support very similar versions of the runtime (e.g. 2.0.1 and 2.0.2) with minimal overhead.

Distributing the Surge Runtime Environment as DLLs requires 87 separate DLLs totaling 18MB in size. Distributing it atomized requires one atom database taking 8.5MB. We achieved this space reduction by coalescing all isomorphic subgraphs in the atom database, removing inter-file linking information and applying simple compression to the remaining atoms. Our installer further compresses the atom database down to 3.6MB for downloading.

By improving disk locality, switching from DLLs to atomization sped up our cold-start times by a factor of between four and eight, depending on the machine. It's possible that applying a modern working-set tuning algorithm to our DLLs[18] could have provided a similar speedup, but we rejected this approach because it did not give us atomization's patching and compression advantages, and because most working set tuners only reorder code but not data.

5.0 Comparative Evaluation

Certain of the architectural choices reflected in the design of Curl contrast markedly with alternative approaches. Several of the most conspicuous of these are discussed in the following paragraphs.

5.1 Server vs. client

While the trend toward provision of application-level services via the web is clear, the preponderance of web-accessed services are supplied by application servers rather than client-side code. Server-side approaches simplify such issues as administration and security, since their mechanism is housed completely on a localized server and hence within a single administrative and security domain. While the administrative and security issues can be addressed by versioning and security provisions like those offered by Curl, the attraction of eliminating these problems and the need for such provisions is compelling.

However, there are fundamental limits to server-side implementation of web applications, with consequent pressure to move at least portions of the computation load to the client. The most serious of these is the extent to which the user's interactive experience is compromised by the overhead of network communications in application responses. Highly interactive user environments, such as those of real-time games, graphic editors, or music synthesizers cannot be implemented effectively without client-side application-specific code.

Secondly, restriction to server-resident applications confronts several fundamental scaling problems. As the number of users of an application server grows, the computational capacity of the server, as well as the network capacity to its users, must grow in proportion. In contrast, users bring their own client-side computational capacity to the web in proportion to their number, so that client-resident applications mitigate the computation scaling problem. Moreover, since the network service required in the latter case is a one-time distribution of the application code (rather than continuous communication while the application is used), the network sees load which is typically both lower and amenable to further reduction via content caching approaches.

5.2 Scripting vs. Programming

Interpreted scripting languages such as Python[19], JavaScript[20], Tcl[21], and Perl[22] on both client and server abound, and are widely used in the implementation of web-based services. They share with Curl certain top-level goals, such as the approachability and incremental execution overhead mentioned in the introduction of this paper. In general, they differ from more serious system programming languages in a number of dimensions: they are interpreted rather than compiled, their execution model is dynamic, and they use runtime tags rather than compile-time declarations for data types.

While Curl aspires to the ease of use and lightweight execution characteristics of these languages, its design reflects an unwillingness on the part of the authors to give up such features as compile-time type checking in the web services arena. Our experience implementing server-side systems in Python -- a carefully designed interpreted language with dynamic types -- is riddled with examples of users tripping across simple errors in rarely-executed code branches. Without strong compile-time analysis it is difficult for a developer to avoid having his users share in the debugging experience, an experience to which most web users are occasionally treated.

A major difference between Curl and scripting languages is their respective performance goals. Unlike scripting languages, typically unsuitable for performance-sensitive applications, Curl compiles to optimized native code for runtime performance similar to that of other compiled languages like Java.

5.3 Virtual Machines vs. source distribution

The major difference between Curl and conventional programming languages, such as Java and C#, is Curl's goal of *content language* breadth. While detailed comparison between the programming language subset of Curl and these extant languages reveals a number of technical differences, most of these details are orthogonal to Curl's top-level goals and hence beyond the scope of this paper.

One noteworthy architectural difference between Curl and both Java and the Common Language Runtime that supports C# and other languages is Curl's distribution of content via source rather than the compiled

byte code consumed by a *virtual machine*[23,24]. The implementation advantages of portable byte code as a distribution standard are well known, and exploited to advantage in Java and CLR language implementations. Curl, in comparison, requires a full just-in-time compiler at the client, along with the source code (possibly compressed and/or obfuscated). There is not, at least as yet, a portable standard Curl intermediate code form.

One compensating advantage of Curl's approach is the potential freedom it offers for the generation of optimized code from language extensions. A `{for-pixel ...}` construct dictating pixel-level iteration over an image can be defined so as to generate code exploiting platform-specific optimizations, unconstrained by the need to compile to a fixed virtual machine instruction set and subsequently expanded, in a different host environment, to native code. Source-level information is lost in the VM representation, and may be arbitrarily hard to reconstruct in a decoupled code generator.

The use of source code as the canonical content representation assures, in addition, direct support for a simple, dynamic execution model. Edit the source code, click "reload", and view the revised content -- with no visible compilation step. This transparency of the compilation process is, in theory, attainable using a virtual machine model; but in practice, the compilation step is an added complication.

Microsoft's CLR is designed to be a universal virtual machine that can act as a target for numerous programming languages. This is laudable, but it takes a compiler expert to target a language to the VM, making it impractical for many domain-specific languages. With its extensible syntax, Curl provides universality at the level of source code, not arcane byte code, so Curl is easily extended into new problem domains by a much wider audience.

Finally, the authors share an admitted attraction to the openness of the source distribution model. The early explosion of HTML, in our view, was due in considerable part to this feature: one could click *view source* to study the implementation of an admired web page, and replicate its features in new content.

5.4 Why not XML?

The language that is perhaps closest to Curl in its top-level goals is Water, designed by Plusch and Fry[25,26]. Like Curl, Water aspires to integrate programmed content, markup, and data within a coherent semantic and syntactic framework; also like Curl, it does so in the context of a contemporary object-oriented language. Unlike Curl, however, Water is based on XML[27] rather than an alternative, nonstandard syntax. This approach has a number of strong attractions. It rides on the growing popularity of XML, it and offers superset compatibility with HTML, whose practical momentum as a standard is vastly greater than that of any programming language.

We know of two fundamental reasons to prefer the Curl approach to Water's XML basis. The first is simply the limited suitability of XML as a syntactic substrate for a modern programming language. While the basic universality properties of XML allow it to represent arbitrary structure, including programs, the resulting embedding will be far from optimal from the standpoint of human readability or ease of maintenance. This factor may be unimportant in situations where the programmed content is machine generated, but the authors' bias as programmers makes us skeptical that we would ever choose an XML-based language to code in.

The second drawback we see in Water's XML approach is its inconsistency with our goal of syntactic as well as semantic extensibility. While XML is an explicitly *extensible* markup language, it anticipates only extensions whose syntax is constrained to the rigid rules of XML.

5.5 Second thoughts and plausible futures

There are a number of aspects of the Curl architecture that could be improved considerably, and which represent candidates for future work. One of these is its lack of an exported interface at the level of C programming, a feature that would (a) give external users access to the static compilation mechanism used internally

to build platform-specific runtime systems, and (b) encourage external packages with C interfaces to be imported by ordinary users. The absence of this importation provision reflects an early bias on the part of Curl's developers to build the Curl universe entirely in Curl, including graphics, GUI, XML parsing, compiler, and network support. While this Spartan self-sufficiency forced the evolving language and implementation to meet demanding functional and performance standards, it led to an immense engineering task. We observe, in contrast, the success of Python[19] whose capacity to encapsulate existing packages from the C universe has enhanced its lightweight interpretive core language technology with an exceedingly rich array of runtime facilities. Fortunately, we see no serious obstacle to providing similar encapsulation capabilities within Curl.

A second arena for possible improvement represents a more serious research challenge: unification of mechanisms behind procedure application and macro expansion into a single semantically coherent construct. The conventional model underlying macro expansion, including that of Curl, is a source-to-source transformation effected by user-tailored code; this contrasts markedly with the model for procedure (or method) definition, in which an abstract template is provided whose instances vary in the values of certain parameters. While the source translation model certainly provides the capability for arbitrary syntactic and semantic extensions, it does so at the cost and complexity of a severe semantic discontinuity which forces its user to view constructs as data rather than as program and raises a number of subtle issues associated with interpretation, environments, captured variables, and the like. We suspect that this disruption is unnecessary, at least in the vast majority of cases where macros are used. The generalization of the procedure definition syntax to include pattern matching and control over argument evaluation order might support these cases as a simple generalization of conventional procedural semantics.

6.0 Conclusions

From the subjective viewpoint of the authors, and that of the biased but technically sophisticated community of Curl content developers, the approach to integration of programmed and textual content described here has been a clear success. The most substantial evidence to date supporting this conclusion is the use of Curl as the vehicle for implementing both the Surge Runtime Environment and its integrated documentation, although a growing number of less incestuous Curl-based applications provide substantial reinforcement.

Curl's competence at the general-purpose programming language end of the gentle slope spectrum is attested to by its effectiveness as a vehicle for coding the runtime system, compiler, and runtime support. These include many performance-critical aspects, such as pixel-level manipulation of image data, which compile to native-code implementations offering performance competitive with that of conventional languages. The language implementation itself makes substantial use of the extension mechanisms described here (and available to users to create their own language-level customizations), diminishing the usual barrier between language designer and language user.

The use of Curl for its own documentation provides even stronger support for the emancipation of the user from the constraints imposed by a language designer. While markup tags available in HTML are restricted to a fixed set blessed by a standards body, Curl content can exploit markup mechanism whose definition and implementation are provided by the author as unprivileged content. Extension of the content language can be done locally, individually, and without negotiation or appeal to global authorities.

The Curl documentation exploits this freedom in a number of dimensions. API-level documentation is integrated with the code as text-laden language constructs, obviating the common redundancy between information repeated between a program's comments and its documentation. The documentation consists of arbitrary Curl source (not simply stylized comments) and includes a number of powerful features, including an **`example`** form which presents a code sample and the result of its execution to the user as an interactive "sandbox" which can be modified and re-executed at the reader's whims.

Guy Steele has argued that the most important goal of the design of a contemporary language should be its plan for growth[28], a view that succinctly captures the extensibility, approachability, and incrementalism

characteristics mentioned in the introduction as early aspirations of Curl. In this light, Curl represents more than simply a language: it is the basis for an evolutionary trajectory that may yield new dialects for the indefinite future.

7.0 Acknowledgements

Development of Curl during the 1995-8 period was supported by DARPA under contract number DABT63-95-C-0009. Major contributors to that effort include David Kranz, who implemented the majority of the MIT prototype system[10], and Chris Terman who was a principal contributor to its architectural decisions.

Subsequent Curl development has been supported by Curl Corpration, and has been influenced by literally dozens of contributors. Prominent members of this list include Christopher Barber, Boleslaw Ciesielski, Bert Halstead, Ben Harrison, David Hollingsworth, David Kranz, Morgan McGuire, Dan Nussbaum, Aaron Orenstein, Chris Terman, and many others. Both Curl and this paper owe their existence to the efforts of the uniquely talented team at this unusual company. The fruits of this continuing work is available at [5].

The authors gratefully acknowledge the thoughtful reviews and material contributions to this paper by Bert Halstead, Chris Terman, and Dan Nussbaum, and the drafting of Figure 1 by Helen Cargill.

8.0 References

[1] Gosling J., Joy B., & Steele G., *The Java Language Specification*, Addison Wesley, 1997.

[2] Dertouzos *et al*, “Gentle Slope Systems - Making Computers Easier to Use”, DARPA ISAT summer study, Woods Hole, MA, August 16, 1992.

[3] Bruce Mount, Gary Gray, and Nikhil Damle, *Curl Programming Bible*, Wiley, 2002.

[4] Chris Ulman et al., *Early Adopter Curl*, Wrox press, 2001.

[5] Curl Corporation release of Surge. As of this writing, available for download (with documentation) at **http://www.curl.com**.

[6] Gordon S. Novak, Jr., “Conversion of Units of Measurement”, IEEE Transactions on Software Engineering, vol. 21, no. 8 (August 1995), pp. 651-661.

[7] M. Karr and D. B. Loveman, “Incorporation of Units into Programming Languages”", Communications of the ACM, vol. 21, no. 5, pp. 385-391, May 1978.

[8] N. Gehani, ``Units of Measure as a Data Attribute'', Computing Languages vol. 2, no. 3, pp. 93-111, 1977.

[9] M. Hostetter et al., “Curl: A Gentle Slope Language for the Web”, WWW Journal v2 issue 2, Spring 1997.

[10] MIT Curl prototype (no longer maintained); available at **http://cag-www.lcs.mit.edu/curl.**

[11] O. Agesen, S. N. Freund, and J. C. Mitchell. *Adding type parameterization to the Java language*. In and Applications (OOPSLA), October 1997.

[12] A. Kennedy and D. Syme. Design and Implementation of Generics for the *.NET Common Language Runtime*. In ACM-SIGPLAN 2001 Conference on Programming Language Design and Implementation. ACM, June 2001.

[13] Jonathan Bachrach and Keith Playford, "The Java Syntactic Extender (JSE)," *ACM SIGPLAN Notices (Proceedings of the 2001 ACM SIGPLAN Conference on Object Oriented Programming, Systems, Languages and Applications (OOPSLA'01))*, **36** no. 11 (November 2001): 31-42.

[14] Ron Cytron, et al., Efficiently computing static single assignment form and the control dependence graph, ACM Transactions on Programming Languages and Systems, 13(4):451-490, October 1991.

[15] Steven Lucco, "Split-stream dictionary program compression",ACM SIGPLAN Notices, Volume 35 , Issue 5 (May 2000).

[16] Haris Lekatsas and Wayne Wolf, *Random access decompression using binary arithmetic coding*, Data Compression Conference, 1999, pp. 306-315.

[17] Josh MacDonald. "Versioned File Archiving, Compression, and Distribution". UC Berkeley. Available via **http://www.cs.berkeley.edu/~jmacd/.**

[18] N. Gloy, T. Blackwell, M. D. Smith, and B. Calder. Procedure placement using temporal ordering information. Proceedings of the 30th Anual ACM/IEEE Intl. Symposium on Microarchitecture, pages 303--313, Dec. 1997.

[19] The Python programming language. See **http://www.python.org**.

[20] D. Flanagan, *JavaScript: The Definitive Guide*, Second Edition, O'Reilly and Associates, 1997.

[21] J. K. Ousterhout, *Tcl and the Tk Toolkit*, Reading : Addison-Wesley, 1994.

[22] L. Wall, T. Christiansen, R. Schwartz, *Programming Perl*, O'Reilly and Associates (1996).

[23] T. Lindholm and F. Yellin, *The Java Virtual Machine Specification,* Second Edition, Addison Wesley, 1999

[24] The .NET Common Language Runtime. See **http://msdn.microsoft.com/net**.

[25] Mike Plusch and Christopher Fry, *Water Programming*, Clear Methods, 2002. Also available at **http://www.waterlang.org**.

[26] Water Rationale, May 2002; **http://www.waterlang.org/doc/water_rationale.htm**

[27] Tim Bray, Jean Paoli, C. M. Sperberg-McQueen: Extensible Markup Language (XML). World Wide Web Journal 2(4): 27-66 (1997).

[28] Guy Lewis Steele Jr. Growing a language. *Lisp and Symbolic Computation*, 1998.