
# Fostering Little Languages

## *Picking up where language designers leave off*

John Clements, Matthias Felleisen, Robert Bruce Findler, Matthew Flatt, and Shriram Krishnamurthi

Programmers constantly design and implement little programming languages. Some of those languages disappear under many layers of software. Others help with repetitive tasks, may thus spread to colleagues, and even evolve into general-purpose languages over time. Languages such as AWK, Make, Perl, bash, autoconf, and Tcl come to mind.

Programmers who wish to create a little language face a choice. One possibility is to build the little language from scratch — an option that involves building a lexer, parser, and interpreter. The other possibility is to build the little language on top of an existing general-purpose language. In this case, the little language shares the host language's syntax (variables, data, loops, functions) where possible, its typechecker, interpreters and compilers, and perhaps other tools. This kind of extension is often called "language embedding."

Table 1 summarizes the salient differences between implementing a language "from scratch" in a language A, and the strategy of embedding a little language into an existing language B. The implement-from-scratch strategy *uses* technology, while an embedding strategy *reuses* technology. Table 1 also underscores a glaring problem for implementors of little languages. If a programmer chooses to implement the new language from scratch, there is no programming environment. If the new language is embedded, the existing IDE for the language may also work for the little language, but it probably won't understand the language in its own right.

We have worked with embedding little languages into host languages for several years. Our effort consists of two related projects. The first project is to develop a host language into which programmers can easily embed other little languages and where they might even compose such little languages. Our inspirations, in this case, are Lisp's and Scheme's macro mechanisms, which have been used for decades to create small languages for specific problem domains.

The second project involves creating a programming environment that easily adapts to embedded little languages. Emacs is a primitive example of what we have in mind, but modern IDEs offer far more than Emacs. In addition to an editor, an IDE nowadays offers tools that help programmers understand a program's properties. For example, an environment may provide syntax coloring, an integrated test coverage checker, a debugger, and a stepper. Ideally, the tools of the host environment should seamlessly work for programs in the embedded little language and for programs that contain and compose little language programs.

In this article, we show how such an IDE might work, what it means for a programming environment to adapt itself to a little language, and how this works for the specific example of a small XML processing language.

*John is a graduate student at Northeastern University; Matthias is a professor at the College of Computer Science at Northeastern University; Robert is an assistant professor of computer science at the University of Chicago; Matthew is an assistant professor in the School of Computing at the University of Utah; and Shriram is an assistant professor of computer science at Brown University. They can be contacted at clements@ccs.neu.edu, matthias@ccs.neu.edu, robby@cs.uchicago.edu, mflatt@cs.utah.edu, and sk@cs.brown.edu, respectively.*

### Processing XML

Almost everyone in the IT world is at least vaguely familiar with XML. Entire industries, organizations, and individual programmers are already using XML in many applications.

*(continued from page 16)*

Moreover, a fair number of competing committees are working on languages for processing XML. Naturally, programmers want to integrate XML and XML-processing tools directly into their programs, and their programming environments should support this integration.

At first glance, XML documents are similar to HTML documents. Elements may be either character data or tags (optionally annotated with an attribute list) enclosing a list of zero or more XML elements. On a deeper level, XML consists of two related parts — a concrete syntax and an abstract syntax. Listing One is an example of the concrete syntax and Figure 1 is its corresponding abstract syntax tree.

Sublanguages of XML are specified using schemas (or other means). A schema defines the set of valid tags, their possible attributes, and the constraints on the XML elements appearing between a pair of tags. A schema for the newspaper article language from Listing One appears in Listing Two. This schema specifies, among other things, that the header field must contain a title and author. The ability to specify XML languages explicitly using schemas most clearly separates XML and HTML.

XML documents are data. To use this data, programmers write programs that accept and manipulate the data. Such programs may just search XML documents or may render them in a different format. For instance, a newspaper may wish to render an article stored in an XML-structured database (as in Listing One) as a web page (see Listing Three) or in a typesetting system.

On the surface, processing XML data appears to be a tedious process, involving the design and implementation of lexers and parsers. But below their surface syntax, many XML expressions are basically just trees. Each node is either character data or a tagged node containing a set of attributes and a set of subtrees. Once you strip away the concrete syntax and focus on the essence of XML's structure, the tree becomes obvious, and processing trees becomes the essence of XML processing. One way to help programmers write programs for processing XML is to provide them with notations for writing down XML transformations as tree transformations.

*The core of every little language is a library of functions and data structures*

| Implementing "From Scratch" | Embedding a Little Language |
|---|---|
| Variables, loops, etc. are new | Variables, loops, etc. are those of B |
| Safety/type-soundness may not exist | Safety/type-soundness is that of B |
| Lexer is implemented in A | Lexer is an extension of B's |
| Parser is implemented in A | Parser is an extension of B's |
| Interpreter is implemented in A | Interpreter is B's |
| IDE doesn't exist | IDE is that of B |

***Table 1:*** *Use/reuse of language technology.*

```
<article>
  <header>
    <title>Not an Article</title>
    <author> John Clements </author>
  </header>
  <text>
    This is not a newspaper article.
    But if it was, this is where the
    article's text would be.
  </text>
</article>
```

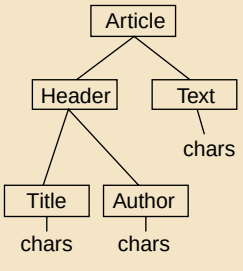



***Figure 1:*** *Abstract syntax tree.*

## S-XML

We believe that XML processing can benefit from a "little" language for tree transformations. Furthermore, if you embed this language rather than build it from scratch, programmers can use the little language to develop small programs and can compose "little programs" using the host language. Indeed, you can escape from the little language and use the host language to process XML if the little language proves too inefficient or too cumbersome for a specific problem.

We've created such a little language in Scheme called "S-XML." Scheme is well suited for this purpose because its data language makes it easy to create and process XML-like trees. Specifically, S-XML represents XML elements with S-expressions. Otherwise, S-XML is like every other little embedded language. It consists of a (small) number of special forms (syntax) and some auxiliary functions (the runtime). Scheme provides the rest of S-XML's functionality, including function definition, function application, iteration, loops, and the like.

S-XML supports three special forms: *xml* and *lmx* for creating XML elements, and *xml-match* for writing down pattern-based tree transformations. In addition, the language also provides a notation for schemas so that you can express XML language specifications.

An XML document may specify a footer for use in an HTML rendering. A naive translation would represent such information as a string, like this:

```
"<center>page number <em>3</em></center>"
```

Naturally, such a string fails to capture the tree structure of the document. Every procedure that operates on this data must parse the string all over again, which makes it difficult to abstract over XML transformations. S-XML uses trees instead, so that the footer information is represented as:

```
(xml (center "page number " (em "3")))
```

Within the form *(xml ...)*, each nested subexpression is taken to describe an XML element. Just as double-quotes are used in many languages to denote literal data, *(xml ...)* is used to denote XML literals. XML elements may also contain attributes. The *xml* form permits the addition of attributes to elements. These attributes appear as an optional (parenthesized) list immediately following the tag name. Thus, an element such as:

```
<body bgcolor="BLUE"> ... </body>
```

would be written as:

```
(xml (body ((bgcolor "BLUE")) ...))
```

With *xml*, you can construct large constants, but what you really need are mechanisms for constructing constants with holes

that are filled with computed values. S-XML, therefore, supports the *lmx* construct, which lets you compute a portion of an XML tree. For example, you may wish to specify the footer of a page relative to a page number:

```
;; Number -> XML
(define (make-footer page-number)
  (xml (center "page number: "
    (em
     (lmx
      (number->string page-number))))))
```

The *lmx* form evaluates its subexpression and splices the result into the XML tree in place of the entire *lmx*-expression. Here, it converts the given *page-number* into a string and places this string into an ** element.

Now that you know constructs for building XML trees, you can switch your attention to tree processing. Following a long-standing tradition, S-XML supports pattern-oriented tree processing. Specifically, it provides *xml-match* with which S-XML programmers can easily specify a conditional that matches XML patterns and returns XML data.

Take a look at the function definition in Listing Four. This function consumes an article element and produces an *<html>* element. The transformation is specified with *xml-match*, which matches the function's sole argument against a pattern that looks just like an *xml* data element in S-XML. The difference is that the pattern may also contain *lmx*-designated pattern variables—that is, *title-string*. As in other pattern-matching notations, a pattern variable matches everything and represents what it matches. A pattern such as *(text (lmx-splice body-text))* matches a *<text>* element that contains a sequence of elements, and the entire sequence is bound to *body-text*. When *lmx-splice* is used for the output, a sequence bound to a pattern variable is spliced into the output.

Each pattern-matching clause in *xml-match* contains a result in addition to the pattern. The result is another *xml* data element that contains pattern variables. In the result part of a clause, the pattern variables represent what they matched if the match succeeded. For example, if *render* is applied to an S-XML representation of the XML element in Listing One, then *title-string* stands for "Not an Article" in the result expression of the first clause. Similarly, *body-text* stands for the sequence of words "This," "is," "not," and so on.

## Building S-XML

The core of every little language is a library of functions and data structures. In fact, for some tasks such a domain-specific library is a complete solution to the language-design problem. For many problem statements, however, a library-based language is not enough. There are just too many important language forms that cannot be implemented as ordinary functions. Among these are shortcuts for creating structured data (for example, *xml* and *lmx*), language forms that introduce variable bindings (such as *xml-match*), and language forms that affect the flow of control (*xml-match* again).

Creating new language forms is outside the scope of most programming languages. At a minimum, it requires the ability to translate new notation into the core of the language. But as C macros demonstrate, this is not enough. It simply doesn't suffice to think of new notations as strings; the translator must gracefully die on syntax (and other S-XML) errors and report them in an informative manner. This, in turn, requires some integration with the parser and a notation for rewriting parse trees. LISP introduced a compromise solution, which Scheme adapted in several steps over the past 20 years.

Consider a form such as *(xml (center "page"))*. If you wish to represent a * * element as a record with three fields—one

for a tag, one for the attributes, and one for the text sequence—then the correct translation into Scheme is:

```
(list 'center (list) (list "page"))
```

Roughly speaking, macros are specified with just such rules, by (abstract) example.

Naturally, translating *xml* isn't quite that simple. The translator must also recognize embedded *lmx* expressions, as in this term:

```
(xml (center "page " (lmx the-page) " of 8)"))
```

Here, we expect to find this translation:

```
(make-center (list) (list "page " the-page " of 8"))
```

That is, when *xml* finds an embedded *lmx*, it splices *lmx*'s subexpression into the proper expression context. Listing Five is an S-XML module for *xml, lmx,* and *xml-match* as presented in this article. To use it, enter *(require (file "....xml-lmx.ss"))* where *"...."* is the full path to the file. Alternatively, you can put the file in the directory where you start DrScheme, and just use *(require "xml-lmx.ss").*


***Figure 2:*** *DrScheme: The core environment.*

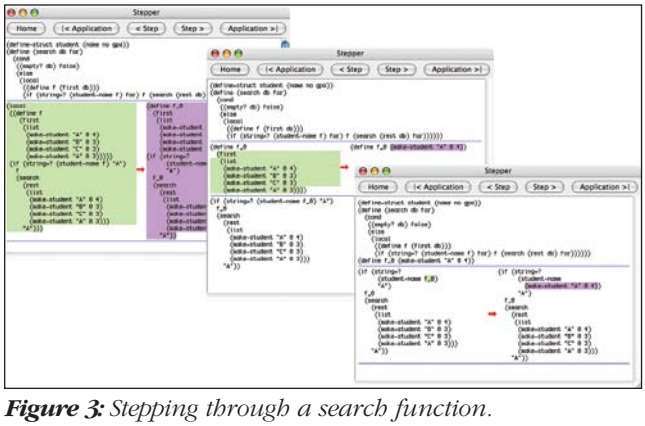


***Figure 3:*** *Stepping through a search function.*

## Little Environments

Once you have an embedded implementation for a little language, you should think about what kind of support programmers may desire from the existing programming environment. For example, if the environment performs some syntax coloring for the host language (say, distinguishing variables from keywords), then the embedded language should also benefit from this tool. Specifically, variables in the embedded language should be colored like variables in the host language, and so on.

Similarly, if the host language supports systematic variable naming or variable binding diagrams, programs in the embedded language should be able to use variable renaming or variable binding diagrams, too. Better still, if the program in the embedded language refers to some surrounding host program and vice versa, then the environment should be able to trace variable bindings back and forth between the two programs.

Ideally, most of the tool support should come from the existing environment without any additional support from the language implementor. But this is too much to ask for, given the current status of IDEs. The best we can hope for is that an embedded language benefits from most tools in the surrounding IDE and that the extensions to the IDE can be kept to a minimum. Using DrScheme—our home-grown environment—this is now almost a reality.

## DrScheme

DrScheme (http://www.drscheme.org/) is a graphical IDE for Scheme that runs on most major platforms (UNIX, Linux, Mac OS X, and Windows). Originally targeting beginning students, our goal with DrScheme was to provide a simple, easy-to-use IDE—without the plethora of buttons, menus, tools, and other accessories of professional IDEs. Along the way, the environment has grown up and has become a useful tool for many Scheme programmers without losing its simple interface.

The core environment (Figure 2) consists of two panes and a simple toolbar with four buttons. One pane is an editor; the other is an interactions window. As in most modern IDEs, the editor is graphical and syntax aware. "Graphical" means that pictures are plain values, just like numbers or strings. "Syntax aware" means that the editor indents properly on return and visually matches parentheses, moving from a closing to the corresponding opening parentheses and gray-shading the code between them. As Figure 2 shows, the editor also colors keywords, variables, literal constants, and so on in different colors.

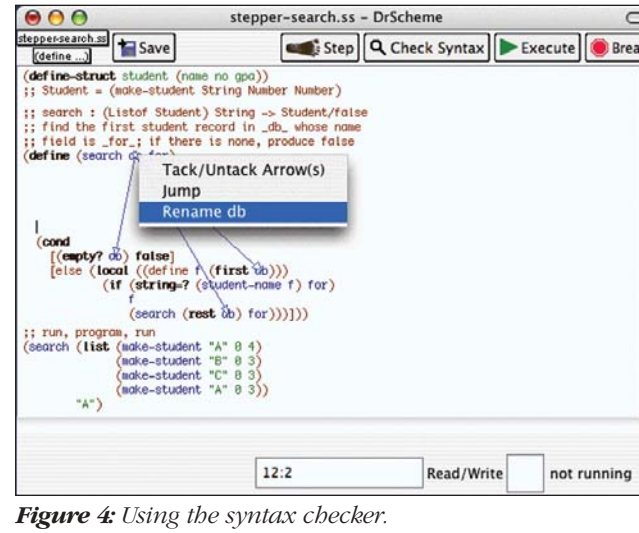
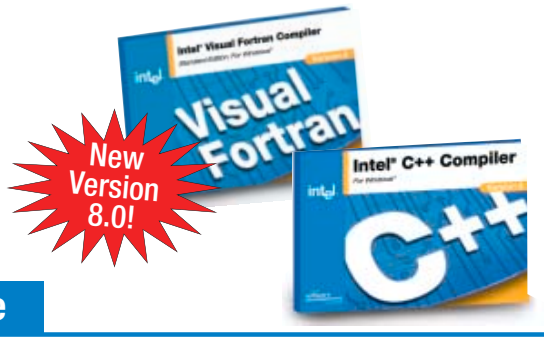


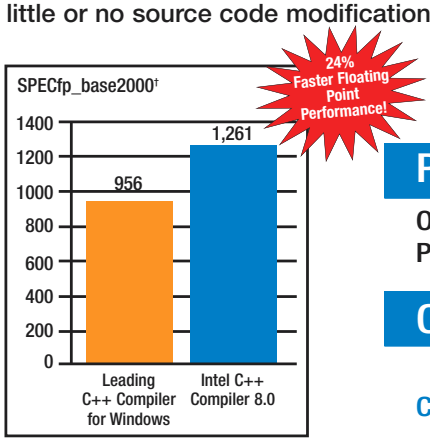

***Figure 4:*** *Using the syntax checker.*

# Maximize Your Application Performance

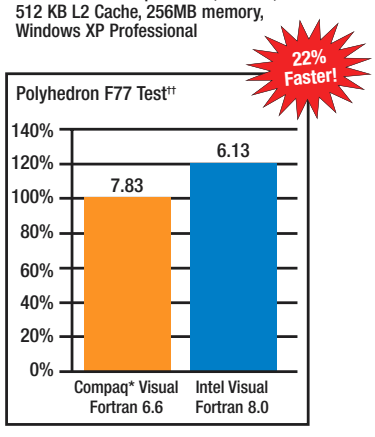
## Intel® C++ and Visual Fortran 8.0

The latest version of Intel's fast compilers is here! Give your application a boost in performance with little or no source code modifications.




† Intel® Pentium® 4 processor, 3.2GHz, 512 KB L2 Cache, 256MB memory, Windows XP Professional




†† Intel® Pentium® 4, 1.8 GHz, 256 MB memory Microsoft Windows 2000 See www.polyhedron.com

### Performance

Outstanding performance on Intel architecture including Intel® Pentium® 4, Intel® Xeon™ and Intel® Itanium® 2 processors.

### Compatibility

- Both integrate into Microsoft Visual Studio .NET

C++ →

- Source & binary compatibility with Microsoft Visual C++ .NET

Fortran →

- Strong compatibility with Compaq* Visual Fortran

### Support

1 year of product upgrades and Intel Premier Support included

*"We tried the Intel® C++ Compiler for Windows and...(it) generated impressive code, worked amazingly well with our Integrated Development Environment, and delivered substantial performance improvements... We recommend this to any C++ developer building applications for delivery on Windows-based systems using Intel processors."*

*—Mike Marchywka*
*Chief Scientist; EyeWonder, Inc.*


### Intel® Integrated Performance Primitives Library 4.0

Highly optimized code for graphics, multimedia, math and signal processing.


### Intel® Thread Checker 2.0

Detects Win32 and OpenMP threading bugs. Race conditions, deadlocks and more.

| YOU SAVE UP TO $60! | Paradise # | Retail | Discount |
|---|---|---|---|
| Intel® C++ Compiler 8.0 for Windows | I23 0B40 | $399.00 | $378.99 |
| Intel® Visual Fortran 8.0 | I23 0B42 | $499.00 | $473.99 |
| Intel® Visual Fortran 8.0 Upgrade for CVF Users | I23 0B43 | $200.00 | $189.99 |
| Intel® IPP 4.0 for Windows | I23 0B47 | $199.00 | $189.99 |
| Intel® Thread Checker 2.0 for Windows with VTune Analyzer 7.1 | I23 0C3D | $1,198.00 | $1,137.99 |

Programmer's Paradise®

To order or request additional information call:
800-423-9990
Email: intel@programmers.com

*FREE trials available at:*
programmersparadise.com/intel

*(continued from page 20)*

The interactions pane (or window) is a Lisp-style listener. It waits for programmers to type in complete expressions (or statements), then evaluates them. If the evaluation terminates and has a visible result, the listener prints this result and waits for the next input.

Programmers can evaluate the definitions and expressions from the editor with a click on the Execute button in the toolbar. The other three buttons in the toolbar provide additional functionality:

- Break terminates the evaluation in the interactions window.
- Step invokes the stepper on the definitions and expressions in the editor. In contrast to conventional debuggers or steppers, DrScheme's stepper displays the steps of a program execution as if it were algebra homework for an eighth grader. Figure 3 displays some of the evaluation steps for a function that searches a list of records. The stepper is the most popular tool among high-school teachers who use DrScheme for introductory programming courses.
- Check Syntax analyzes the code in the definitions window syntactically, colors it properly, and allows programmers to explore the lexical regions (scope) of the program. Figure 4 illustrates how programmers can use the information from the syntax checker to create arrows that show all bound occurrences of a function parameter or to rename a function parameter systematically.

In addition to these basic tools, DrScheme provides a module browser for navigating the modules and libraries of a program, a contour outline for navigating the content of an individual module, a test-suite manager, an expression coverage checker that highlights those parts of the code in the editor that are not executed by the test suite (this tool is always turned on for students), a performance profiler that colors expressions according to their execution intensity, a static debugger for analyzing potential violations of basic invariants, and a conventional debugger. The conventional debugger and some other experimental tools are still under development.

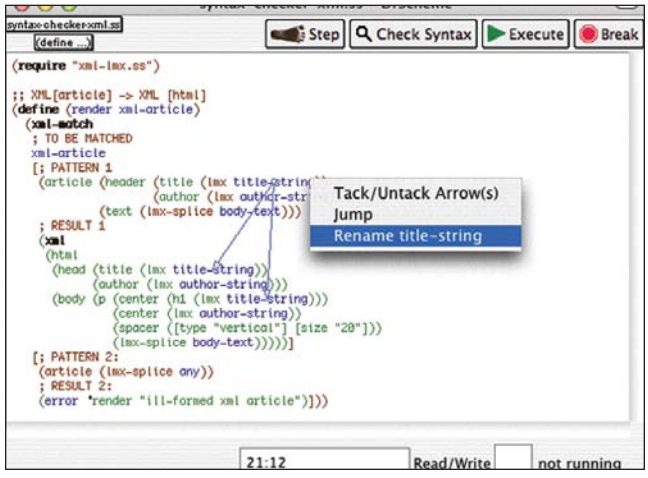



***Figure 5:*** *Using the syntax checker with XML.*

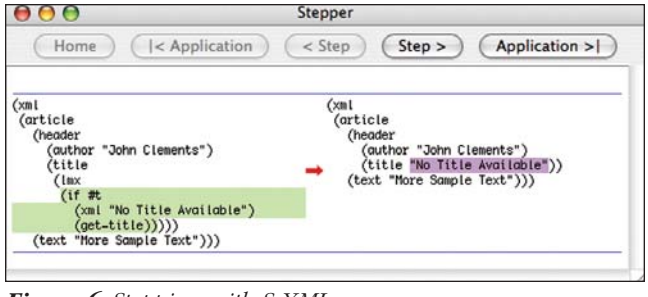



***Figure 6:*** *Stepping with S-XML.*

## An Environment for S-XML

Most of DrScheme's tools work with embedded little languages without any modification. Since an embedded little language in Scheme is just another parenthesized language, the core editor almost immediately copes with programs in the new language. To get the indentation depth correct, you must tell DrScheme about the new keywords and their indentation depth in a preference dialog. The syntax coloring (at the moment) doesn't recognize the new keywords, though this is, in principle, possible and is a work in progress.

Similarly, other tools work if they don't need to understand the full meaning of the constructs in the embedded language. Consider *xml-match*, which introduces pattern variables and binds them to values in patterns and result expressions. Check Syntax, which lets you browse such variable bindings and rename them systematically, deals with these new constructs in a completely transparent manner. For example, Figure 5 shows how the syntax checker can draw arrows from the pattern variables to their uses and how you can rename one of them.

DrScheme tools that need to understand the full meaning of S-XML, however, must be adapted manually. The stepper is a primary example for such a tool. Figure 6 shows the stepper's actions for an application of render to an <*article*> element. While the S-XML programs are translated to plain Scheme programs, a symbolic stepper must display the execution steps as if they had taken place at the source level. Since the stepper works on plain Scheme programs, it must uncompile intermediate execution stages into S-XML programs, which the stepper as-is (naturally) cannot do. Put differently, the stepper needs additional hints so that it can uncompile intermediate execution stages into source code.

At the moment, hints for the stepper (and other semantic tools, such as the static debugger and the symbolic, dynamic debugger) must come from the S-XML designer. In this particular case, the stepper must become aware of *xml* and *lmx* because they build values in the little language. Conversely, a stepper must be able to display intermediate steps of the processes that construct XML values piece by piece. To add this knowledge, we

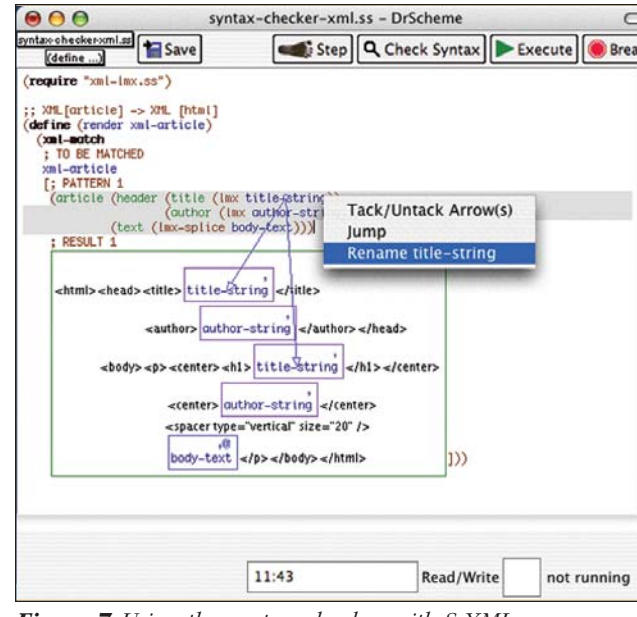



***Figure 7:*** *Using the syntax checker with S-XML.*

# Programmer's Paradise®

***Your best source for software development tools!***

## GUARANTEED BEST PRICES*

Should you see one of these products listed at a lower price in another ad in this magazine, CALL US! **We'll beat the price**, and still offer our same quality service and support!

*Terms of the offer:

- Offer good through March 31, 2003
- Applicable to pricing on current versions of software listed
- March issue prices only
- Offer does not apply towards obvious errors in competitors' ads
- Subject to same terms and conditions

PROGRAMMER'S GUARANTEED BEST PRICES PARADISE

*Prices subject to change. Not responsible for typographical errors.*

### LEADTOOLS Document Imaging v.14

***by LEAD Technologies***

New version 14 offers enhanced OCR support with Handprint (ICR) and OMR recognition. Features include new annotation objects with added output formats, image processing functions, document file formats, image registration functions, document image display, Twain support and more.

**Paradise #**
**L05 0520**
**$1662.99**

**www.programmersparadise.com/lead**


**Programmer's Paradise #1 Best-Selling Help Authoring Tool for 7 Years Running!**

### RoboHelp Office

**The Industry Standard in Help Authoring**

Create professional Help systems for desktop and Web-based applications, including .NET.

- Create all popular Help formats
- Create standard and advanced Help-specific features
- Work in WYSIWYG or true code
- Easily create context-sensitive Help
- Generate printed documentation
- Winner of 55 industry awards

** Price after mfr's mail-in rebate. New US/Can licenses only. Expires 3/31/04.*

**Paradise #**
**E75 0311**
**$879.99***

**www.programmersparadise.com/ehelp**

### IP*Works! Red Carpet Subscriptions

***by /n software***

IP*Works! Red Carpet™ Subscriptions give you everything in one package: communications components for every major Internet protocol, SSL and SSH security, S/MIME encryption, Digital Certificates, Credit Card Processing, ZIP compression, Instant Messaging, and even e-business (EDI) transactions. .NET, Java, COM, C++, Delphi, everything is included, together with per developer licensing, free quarterly update CDs and free upgrades during the subscription term.

**Paradise #**
**D77 0148**
**$1264.99**

**www.programmersparadise.com/nsoftware**

### dtSearch Web with Spider


Quickly publish a large amount of data to a Web site, with "blazing speeds" *(CRN Test Center)* searching.

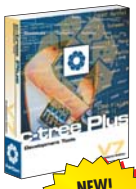
- Features over a dozen indexed and fielded data search options.
- Highlights hits in XML, HTML and PDF, while displaying links and images; converts other files ("Office," ZIP, etc.) to HTML with highlighted hits.
- Spider adds a third-party site to a site's own searchable database.
- Optional API supports SQL, C++, Java, and all .NET languages.

**Single Server**
**Paradise #**
**D29 070F**
**$888.99**

**Download dtSearch Desktop with Spider for immediate evaluation**

"The most powerful document search tool on the market"—*Wired Magazine*

**www.programmersparadise.com/dtsearch**



NEW! SQL Support!

### c-tree Plus®

***by FairCom***

With unparalleled performance and sophistication, c-tree Plus gives developers absolute control over their data management needs. Commercial developers use c-tree Plus for a wide variety of embedded, vertical market, and enterprise-wide database applications. Use any one or a combination of our flexible APIs including low-level and ISAM C APIs, simplified C and C++ database APIs, SQL, ODBC, or JDBC. c-tree Plus can be used to develop single-user and multi-user non-server applications or client-side application for FairCom's robust database server—the c-tree Server. Windows to Mac to Unix all in one package.

**Paradise #**
**F01 0131**
**$850.99**

**www.programmersparadise.com/faircom**

### TX Text Control ActiveX 10.0

***by The Imaging Source***

**Add RTF, DOC, HTML, CSS and PDF Support to Your Application**

NEW .NET VERSION!


TX Text Control is royalty-free, robust and powerful word processing software in reusable component form. The new Enterprise/XML version features a rich set of properties for the manipulation of XML and CSS. Developers can now offer end-users the ability to separate their textual content from their formatting rules.

**Download a demo today.**

DEMO AVAILABLE
programmersparadise.com

**Enterprise Edition**
**Paradise #**
**T79 0214**
**$1,495.99**

**Professional Edition**
**Paradise #**
**T79 0215**
**$729.99**

**www.programmersparadise.com/theimagingsource**

### Teamstudio For Java Bundle Edition 4

***by Teamstudio***

- Eliminate the frustration of using trial-and-error to locate performance bottlenecks and memory leaks in your Java applications.
- Stop getting aggravated reading other people's badly written code.

Teamstudio for Java brings you powerful tools for auditing code, speeding application performance, and reducing memory footprints. Seamlessly integrates with your IDE.

**Paradise #**
**T2M 0103**
**$1,072.99**

**www.programmersparadise.com/teamstudio**

### PR-Tracker™ v5.1

***by Softwise Company***

Affordable scalable enterprise level bug tracking system featuring classification, assignment, sorting, searching, reporting, access control, user permissions, attachments and email notification. Integrates with PR-Tracker Web Client (included) and ProblemReport.asp (included for your betatest or customer support interface). Supports Access and SQL Server.

Download Today!

DOWNLOAD NOW!
programmersparadise.com

**Paradise #**
**S3R 0147**
**$117.99**

**www.programmersparadise.com/softwise**

## Paradise Picks


### DevTrack 5.6

**Powerful Defect and Project Tracking**

***by TechExcel***

DevTrack, the market-leading defect and project tracking solution, comprehensively manages and automates your software development processes. DevTrack 5.6 features sophisticated workflow and process automation, seamless source code control integration with VSS, Perforce and ClearCase, robust searching, and built-in reports and analysis. Intuitive administration and integration reduces the cost of deployment and maintenance.

**programmersparadise.com/techexcel**

**Paradise #**
**T34 0199**
**$482.99**

New X86 Platform Version!

### Sun™ ONE Studio 8, Compiler Collection

***by Sun Microsystems***

Now with improved performance and better portability to boost programmer productivity! The Sun ONE Studio 8, Compiler Collection delivers complete language systems and tools designed to speed software development for you and your team.

- Improved compile time—Language parsers, optimizer, and code generator have been modified to reduce compilation time, allowing you to deploy your application sooner.
- Continued support for de facto (gcc, Visual C++) and de jure standards (C99, OpenMP)
- Licensing structure based on a serial number model

**www.programmersparadise.com/sunone**

**Paradise #**
**S69 0U66**
**$960.99**

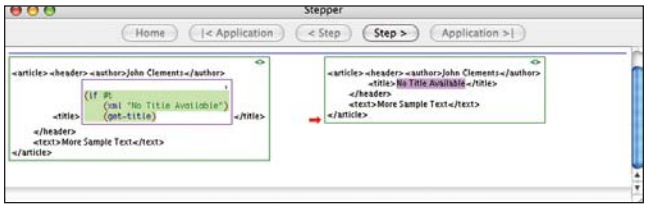



***Figure 8:*** *Stepping with S-XML.*

*(continued from page 22)*
currently extend semantics-based tools by hand. One of our research objectives is to find out whether these extensions can be specified with little languages, too.

## Visual Support for S-XML

On occasion, a little language such as S-XML spreads and many programmers start to use it. At that point, it often makes sense to extend the IDE for the host language with tools that are targeted to the little language. In this particular case, we added XML text boxes for visualizing *xml* and Scheme text boxes for visualizing *lmx*.

Figure 7 shows an S-XML program that uses XML and Scheme boxes. The figure also shows how such boxes are (almost automatically) integrated with other DrScheme tools, such as Check Syntax and the program contour browser (on the right). The initial implementation of visual support for S-XML took one day. Although the majority of the functionality was added on that day, minor refinements occurred over the following months, perhaps totaling another day or two of concentrated effort. It currently consists of about 800 lines of code. This is the largest extension for S-XML besides the stepper. Figure 8 illustrates how the visual support for S-XML is integrated with the stepper. This preliminary screenshot shows a step in the evaluation of an *xml* article whose title is supplied at runtime.

## Conclusion

A programming language's environment affects how useful the language is to programmers. This is true for mainstream languages as well as little languages. Indeed, for the latter, providing a good development environment may be a major factor to its success.

To understand what it takes to turn the environment for a host language into an environment for a little language, we have begun a multiyear research effort to expand DrScheme to DrX—where X is any little language. This article shows how much can already be done automatically for a little XML language. We are now testing DrX for other little languages— one for dealing with plots, another for dealing with timed expressions— and we're hoping to prove that building and offering adaptable IDEs is not just a dream.

**DDJ**

## Listing One

```
<article>
   <header>
      <title>Not an Article</title>
      <author> John Clements </author>
   </header>
   <text>
   This is not a newspaper article.
   But if it was, this is where the
   article's text would be.
   </text>
</article>
```

## Listing Two

```
<schema>
   <element name="header">
      <sequence> <element-ref name="title"/>
                 <element-ref name="author"/>
      </sequence>
   </element>
   <element name="body">
      <mixed> <pcdata/> <mixed/>
   </element>
   <element name="article">
      <sequence> <element-ref name="header"/>
                 <element-ref name="body"/>
      </sequence>
   </element>
</schema>
```

## Listing Three

```
<html>
   <head><title>Not an Article</title></head>
   <body>
      <center><h1>Not an Article</h1>
      by John Clements</center>
      <spacer type="vertical" size="20">
      <p>This is not a newspaper article. But if it was,
      this is where the article's text would be.</p>
   </body>
</html>
```

## Listing Four

```
;; XML[article] -> XML[html]
(define (render xml-article)
   (xml-match
      ; TO BE MATCHED:
      xml-article
      [; PATTERN 1:
         (article
            (header
              (title (lmx title-string))
              (author (lmx author-string)))
            (text (lmx-splice body-text)))
         ; RESULT 1:
         (xml
           (html
             (head (title (lmx title-string)))
             (body
               (p (center (h1 (lmx title-string)))
                  (center (lmx author-string))
                  (spacer [(type "vertical") (size "20")])
                  (lmx-splice body-text)))))]
      [; PATTERN 2:
       (article (lmx-splice any))
       ; RESULT 2:
       (error 'render "ill-formed xml-article")]))
```

## Listing Five

```
(module xml-lmx mzscheme
   (require (lib "match.ss"))
   (provide xml xml-match)

   (define-syntax (xml stx)
     (letrec ([process-xexpr
               (lambda (xexpr)
                 (syntax-case xexpr (lmx lmx-splice)
                   [(lmx-splice unquoted) #`(unquote-splicing unquoted)]
                   [(lmx unquoted) #`(unquote unquoted)]
                   [(tag ([attr val] ...) . sub-xexprs)
                    (identifier? #`tag)
                    #`(tag ([attr val] ...)
                           #,@(map process-xexpr (syntax->list #`sub-xexprs)))]
                   [(tag . sub-xexprs)
                    (identifier? #`tag)
                    #`(tag ()
                           #,@(map process-xexpr (syntax->list #`sub-xexprs)))]
                   [str
                    (string? (syntax-e #`str))
                    xexpr]))])
       (syntax-case stx ()
         [(_ xexpr) #`(quasiquote #,(process-xexpr #`xexpr))])))

   (define-syntax (xml-match stx)
     (letrec ([process-xexpr
               (lambda (xexpr)
                 (syntax-case xexpr (lmx lmx-splice)
                   [(lmx-splice unquoted)
                                #`(unquote-splicing (unquoted (... ...)))]
                   [(lmx unquoted) #`(unquote unquoted)]
                   [(tag ([attr val] ...) . sub-xexprs)
                    (identifier? #`tag)
                    #`(tag ([attr val] ...)
                           #,@(map process-xexpr  (syntax->list #`sub-xexprs)))]
                   [(tag . sub-xexprs)
                    (identifier? #`tag)
                    #`(tag ()
                           #,@(map process-xexpr (syntax->list #`sub-xexprs)))]
                   [str
                    (string? (syntax-e #`str))
                    xexpr]))])
       (syntax-case stx ()
         [(_ matched (pat rhs) ...)
          (with-syntax ([(pattern ...) (map process-xexpr (syntax->list
                                                #`(pat ...)))])
            #`(match matched ((quasiquote pattern) rhs) ...))])))
```

**DDJ**