Little Languages and their Programming Environments
Rice University Technical Report TR99-350

John Clements, Shriram Krishnamurthi, and Matthias Felleisen
Department of Computer Science
Rice University
Houston, TX 77005-1892
Contact: <clements@cs.rice.edu>

December 1, 1999

Summary

Programmers constantly design, implement, and program in little languages. Two different approaches to the implementation of little languages have evolved. One emphasizes the design of little languages from scratch, using conventional technology to implement interpreters and compilers. The other advances the idea of extending a general-purpose host language; that is, the little language shares the host language's features (variables, data, loops, functions) where possible; its interpreters and compilers; and even its type soundness theorem. The second approach is often called a language embedding.

This paper directs the attention of little language designers to a badly neglected area: the programming environments of little languages. We argue that an embedded little language should inherit not only the host language's syntactic and semantic structure, but also its programming environment.

We illustrate the idea with our DrScheme programming environment and S-XML, a little transformation language for XML trees. DrScheme provides a host of tools for Scheme: a syntax analysis tool, a static debugger, an algebraic stepper, and an interactive evaluator. S-XML supports the definition of XML languages using schemas, the convenient creation of XML data, and the definition of XML transformations.

The S-XML embedding consists of two parts: a library of functions and a set of syntactic extensions. The elaboration of a syntactic extension into core Scheme preserves the information necessary to report the results of an analysis or of a program evaluation at the source level. As a result, all of DrScheme's tools are naturally extended to the embedded language. The process of embedding the S-XML language into Scheme directly creates a full-fledged S-XML environment.

We believe that this method of language implementation may be generalized to other languages and other environments, and represents a substantial improvement upon current practice.

1 Reusing Language Technology

Programmers constantly design little programming languages. Many of these languages die a quick death or disappear under many layers of software; network protocols, GUI layout declarations, and scripting tools are examples. Others evolve and survive to fill a niche; `AWK`, `make`, `Perl`, and `Tcl` come to mind.

Once a programmer understands that some problem is best solved by designing a new little language, he must make an implementation choice. One possibility is to build the little language from scratch. This option involves the tasks of specifying a (typically formal) syntax, a (semi-formal) system of context-sensitive constraints, and an (informal) semantics; and of implementing the required software: a lexer, a parser, a type checker, a code generator and/or an evaluator.

The other option is to extend an existing general-purpose language with just those constructs that the task requires. In this case, the little language shares the host language's syntax (variables, data, loops, functions) where possible; its interpreters and compilers; and even its type soundness theorem. The second approach is often called a *language embedding*.

The following table summarizes the strong differences between strategy of implementing a language "from scratch" in a language *A* and the strategy of embedding a little language into a language *B*.

designing a little language "from scratch"	embedding a little language
variables, loops, etc. are designed explicitly	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**	the lexer is an extension of **B**'s
parser is implemented in **A**	the parser is an extension of **B**'s
validity checker is implemented in **A**	the validity checker is **B**'s
interpreter is implemented in **A**	the interpreter is **B**'s

Succinctly put, the "implement from scratch" strategy *uses* technologies; an embedding shares, and thus truly *reuses*, technology for the construction of a little language.

This paper argues that, and illustrates how, a language embedding can reuse more of the host's technology than just the evaluator. Specifically, we argue that if a programming environment for a host language is properly constructed and if we use a well-designed embedding technology, the mere act of constructing the embedding also creates a full-fledged programming environment for the little languages.

In support of our argument we construct an embedded little language, called S-XML, and derive its environment from DrScheme, our Scheme programming environment [6]. S-XML permits programmers to create and manipulate XML-like data. More precisely, they can use a set of constructs to specify XML trees in a natural manner, and they can define tree transformations on the data with an easy-to-use pattern-matching construct. DrScheme provides a host of tools for Scheme: a syntax analysis tool that includes a variable binding display and a variable renaming mechanism; a static debugger; an algebraic stepper; and an evaluator that correlates run-time exceptions with the program source.

```
<article>
  <header>
    <title>Pokemon Rekindle WWII Sentiments</title>
    <author> Bobbi Bland </author>
  </header>
  <text>
    Responding to an armed incursion of the
    feared Japanese "Pocket Monsters," President
    Clinton today declared a state of war in our
    nation's "schools, playgrounds and young minds."
  </text>
</article>
```

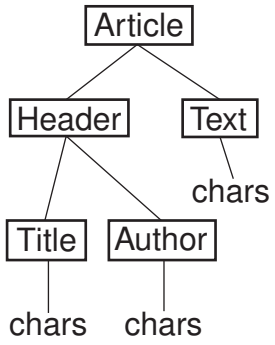

Figure 1: Correspondence between concrete and abstract syntaxes

The S-XML embedding consists of two parts: a function library and a set of constructs that cannot be defined as functions. The implementation of the latter exploits DrScheme's syntax definition mechanism, which, in turn, is based on Scheme's macro technology. DrScheme's syntax extensions are completely transparent to DrScheme's tools. The elaboration of a syntactic extension into core Scheme preserves all necessary information to report the results of an analysis or of a program evaluation at the source level. By adding two small extensions that undo the elaboration at certain strategic places, we thus ensure that DrScheme's syntax checker checks the syntax and context-sensitive properties of S-XML transformations; the static debugger turns into an XML validity checker; the stepper shows how the transformations rewrite XML trees at the level of XML data constructors; and the interpreter prints XML results and reports errors in terms of S-XML transformations. In short, the process of embedding the S-XML language into Scheme directly creates a full-fledged S-XML environment.

The following section introduces XML and S-XML; the third section discusses the S-XML embedding in Scheme. The fourth and fifth section present DrScheme and the little language environment created with the embedding. The underlying technology is explained in the sixth section. The seventh section relates our work to the relevant areas. The last section summarizes our ideas and suggest topics for future extensions.

2 A Running Example: S-XML

To Illustrate our ideas, we develop a little language—and an accompanying programming environment—for operating on XML documents.

2.1 XML

XML (for "eXtensible Markup Language"), is a proposed standard for a family of languages. It was designed to provide a middle ground between the universally accepted but inconsistent and semantically rigid HTML language and the extensible but overly complex

```
<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>
```

Figure 2: A simple schema for newspaper articles

SGML family of languages. An XML element may be either character data or a tag pair annotated with an optional attribute association list and enclosing a list of zero or more XML elements[2]. In this regard, HTML and XML are similar.

On a deeper level, XML consists of two related parts: a concrete syntax and an abstract syntax. Figure 1 shows an example of the concrete syntax and a corresponding abstract syntax tree.[1]

Specific languages within the XML domain are specified using "schemas". A schema defines the set of valid tags, their possible attributes, and constraints upon the XML elements appearing between a pair of tags. A schema for the newspaper article language from figure 1 appears in figure 2.[2] This schema specifies, among other things, that the *header* field must contain a *title* and an *author*. The ability to specify XML languages explicitly using schemas is what most clearly separates XML and HTML.

XML documents are data; in order to use this data, programmers must write programs that accept and manipulate this data. Walsh[23], a member of the XML design team, states:

> ...[I]t ought to take about two weeks for a competent computer science graduate student to build a program that can process XML documents.

The implication is that processing XML data is a tedious and time-consuming process, involving the design and implementation of a project-specific package of I/O routines.

Below the surface syntax, XML expressions are purely trees. Each node is either character data or a tagged node containing a set of attributes and a set of subtrees. A program that processes XML data will be a tree-processing program. Given the complexity of the defined syntax, it makes sense to abstract away from that concrete syntax into a purely tree-based paradigm.

[1] This example is taken from an assignment in a Rice University undergraduate class.

[2] The W3C has not yet settled on a schema standard. The schema shown here is written in a simple illustrative schema language designed to be read easily. Also, the trivial schemas for *author* and *title* are omitted.

```
<html>
  <head><title>Pokemon Rekindle WWII Sentiments</title></head>
  <body>
    <center>
      <h1>Pokemon Rekindle WWII Sentiments</h1>
      by Bobbi Bland
    </center>
    <spacer type="vertical" size="20">
    <p>
      Responding to an armed incursion of the
      feared Japanese "Pocket Monsters," President
      Clinton today declared a state of war in our
      nation's "schools, playgrounds and young minds."
    </p>
  </body>
</html>
```

Figure 3: The result of a simple XML transformation

Once the work of parsing concrete syntax is moved out of the programmer's domain, processing XML trees becomes a more manageable task. Many if not most XML programs will consist of a small set of tree transformations, taking the data from one XML language into another. For instance, a newspaper's web site might be designed to transform an article stored in an XML-structured database (as shown in figure 1) into a web page shown to a reader. The result of this transformation is illustrated in figure 3.

2.2 S-XML

The simple and specialized nature of XML transformations makes them an ideal candidate for an embedded language solution. The language should include special forms for creating and validating XML elements, and a mechanism for expressing tree transformations easily. On the other hand, a language for XML processing should not preclude the production of more complex programs. Rather, it should allow programmers to work with the full power of the general-purpose host language, if they so choose.

We call this language S-XML. It uses S-expressions to match the tree-based structure of XML elements. It provides the **xml** and **lmx** forms for creating XML elements and embedding computation; the **xml-match** form to state pattern-based transformations on these elements; and a language of schemas to express language restrictions. We explain these constructs below.

2.2.1 xml

The little language must provide language forms for constructing XML elements conveniently, because any program that transforms XML data needs to construct XML elements. To take a simple example, a HTML footer might contain a horizontal line and a page number. In XML's concrete syntax, such a footer might be expressed as a string:

```
(define (format-article xml-article)
  (xml-match xml-article (title-string author-string body-text T)      ; keywords
    [(article (header (title title-string) (author author-string))     ; pattern
              (text body-text ...))
     (xml (html (head (title title-string))                            ; result
                (body (center (h1 title-string) "by " author-string)
                      (spacer ((type "vertical") (size "20")))
                      body-text ...)))]
    [(page T)
     (error 'format-page "badly formatted xml-article")]))
```

Figure 4: A simple transformer

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

The obvious shortcoming of the string representation is its lack of structure; every procedure which operates on this data must parse the string all over again. This is wasteful and time-consuming. A better way is to transform this text into structured data. Our language should provide a straightforward way to create such "parsed" structures, independent of the representation of these data. Ideally, the program text that creates an XML element should closely resemble the XML text itself, less the end tag.[3] In the S-XML language, this data is therefore represented with the following program text:

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

Within the form (**xml** ...), each nested subexpression is taken to describe an XML element. Just as double-quotes and backslashes 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 list immediately following the tag name. So, an HTML *body* tag with the *bgcolor* attribute might be written as:

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

2.2.2 lmx

With the **xml** construct, programmers can conveniently specify large XML constants. But programmers may also wish to abstract such tree constructions over certain parameters. For example, a programmer may wish to specify the footer of a page relative to a page number. To allow these parameterizations, we add the **lmx** construct to S-XML:

(**lmx** *expression*)

An **lmx** expression is always a sub-expression of some **xml** expression. It evaluates its subexpression; the result is spliced into the XML tree in place of the **lmx**-expression. Using a combination of **lmx** and **xml** forms, a programmer can now easily define a function that produces a page footer:

[3] In an S-expression, there is no need for an end tag; such a tag is unnecessary and may be mistyped.

```
(schema
 (element ((name "header"))
          (sequence (element-ref ((name "title")))
                    (element-ref ((name "author")))))
 (element ((name "body"))
          (mixed (pcdata)))
 (element ((name "article"))
          (sequence (element-ref ((name "header")))
                    (element-ref ((name "body"))))))
```

Figure 5: A S-XML Schema for an Article Language

```
(define (make-footer page-number)
  (xml (center "this is page number: " (em (lmx page-number)))))
```

2.2.3 xml-match

The programmer now has the tools needed to build elements of the desired XML language. Next, he needs a mechanism to manipulate these elements in a simple way. The most convenient method is to use pattern-matching; our S-XML language provides the **xml-match** form, to perform pattern-matching and tree-processing on xml elements.

To evaluate an **xml-match** expression, each pattern is matched against the input. Once a match is found, the result expression is evaluated, with the bindings introduced by the pattern-match.

Figure 4 shows the definition of the transformer illustrated earlier. Note that both input and output patterns are specified in the same way that **xml** elements are.

2.2.4 schema

One of the most important features of xml is the ability to restrict xml languages, using *schemas*. A schema describes the set of valid XML elements for a specific XML language. A schema is also itself an xml element, and may therefore be described using the same S-XML conventions. Figure 5 shows the S-XML representation of the schema shown earlier. A comparison with the XML specification of this schema (in figure 2) reveals the trivial similarity between the two.

3 Building a Little Language

Much of the functionality of a little language may be established by building a library of functions and constants. In fact, for some tasks a domain-specific library serves as a complete solution to the embedding problem.

There are, however, several kinds of language form that cannot be implemented as ordinary functions. Among these are shortcuts for creating structured data (e.g. **xml** and **lmx**), language forms that introduce variable bindings (e.g. **xml-match**), and language forms that affect the flow of control in non-standard ways (**xml-match** again).

These new language forms may be added using macros. Macros are tree-rewriting rules that are applied to syntax trees during compilation. They elaborate the language forms of the little language into the forms of the host language. In our case, this language is Scheme.

3.1 Scheme Macros

The notion of syntactic abstraction is not a new one. Nearly every general-purpose programming language has some facility for declaring and invoking macros. However, the vast majority of these are deeply flawed. Macro systems like C's gained a well-deserved reputation as dangerous and inelegant. Their ill-considered use often leads to problems for novices and experts alike. Embedding a little language in C using these macros would be difficult at best.

Fortunately, languages like Scheme offer more controlled and useful macro mechanisms. These systems operate on expressions, rather than tokens, and they have a well-defined semantics as tree rewriting systems. As a simple example, consider the **let** form of Scheme. The **let** form binds values to variable names. In many languages, this type of operation is built into the language. In Scheme, it need not be. Instead, Scheme may implement **let** with a macro that elaborates each use of the form into the application of a procedure.

Here is the rewriting rule for **let**:

(**let** ((<*var*> <*exp*>) ...) <*body*> ...)) $\mapsto$ ((**lambda** (<*var*> ...) <*body*> ...) <*exp*> ...)

The ellipses are not a notational shorthand but are an integral part of the macro language described in the Revised^5 Report on Scheme[11]. On the left-hand-side of the macro, they indicate that the previous pattern is to be repeated zero or more times, as in a BNF grammar. This input pattern is matched against the input, and where ellipses occur, bindings of lists are created. The right-hand-side pattern uses ellipses to generate sequences of output patterns drawn from these bindings. The components of the matched patterns may be split from each other, as illustrated by the **let** macro shown here.

3.2 Building S-XML

S-XML is implemented as an embedding within Scheme. The embedding (comprising the forms enumerated in section 2.2) is constructed as a combination of a small functional library and a set of macros.

The **xml** form is implemented as a single macro. This macro transforms uses of the **xml** form into expressions that construct Scheme data. The form also permits the omission of empty attribute fields; it is this kind of syntactic shorthand that gives the little language one of its true advantages over the unmodified general-purpose language. The action of the **xml** macro is shown in this example, where an **xml** form is translated into Scheme code which creates a structure:

(**xml** (*center* "Text: " (**lmx** (*get-text*)))) $\mapsto$ (*make-center* (*list*) (*list* "Text: " (*get-text*)))

Each use of the **schema** form elaborates into a structure declaration and a MrSpidey type declaration. An example of this macro's translation is shown here:

```
(schema                                          (begin
  (element ((name "elt"))                  ↦       (define-struct elt (attrs elements))
    (sequence                                      (define-type elt (cons other null)))
      (element-ref ((name "other"))))))
```

Note that adopting a richer schema language is simply a matter of building a single macro; no other code needs to change.

The **xml-match** form is implemented using a macro in conjunction with a library function. The macro delays the evaluation of the patterns and their matching expressions. It also provides bindings for any pattern variables that occur in the expressions. The function accepts a value and these pattern-expression pairs, and evaluates the first expression whose pattern matches the input value.

A transformer which takes centered text to italicized text is elaborated like this:

```
                                                 (xml-match-fn (xml (center 3))
(xml-match (xml (center 3))                        (list 'text)
  (text)                                           (list
  ((xml (center text))                     ↦         (list '(center text)
   (xml (italic text))))                                    (lambda (text)
                                                              (xml (italic text))))))
```

The *xml-match-fn* procedure is a part of S-XML's runtime library.

Through these three forms, Scheme becomes S-XML, a little language ideal for constructing and manipulating XML-like data, along with the full gamut of Scheme values. Variables and functions are inherited from Scheme. As a result, first-semester undergraduates can program using XML in a matter of days, rather than the weeks of work that might otherwise be required.

4 DrScheme

Building an S-XML evaluator using macros and functions is not enough. This is the lesson that programmers have learned in the course of implementing many little languages. In fact, for many little languages, the execution framework is a small fraction of the total work required to make the language usable. To use a language productively, programmers need a host of related tools: editors, checkers (syntax and semantic), debuggers, and the like.

We demonstrate these ideas with DrScheme. This section describes the tools it provides.

DrScheme is a programming environment for the Scheme language. It is a graphical, cross-platform environment for developing programs. It includes a syntax-sensitive editor, a read-eval-print loop or "REPL", a syntax checker, a stepper, and a static type checker. The challenge is to reuse these tools in the design and execution of an embedded language.

Scheme programs are composed entirely of S-expressions, and DrScheme's editor takes advantage of this in many ways. It provides a set of S-expression-directed movement and

editing functions. It supports dynamic parenthesis-matching, as well as static highlighting of S-expressions adjacent to the cursor. DrScheme automatically indents lines, and unmatched parentheses are highlighted in red.

Another of the tools DrScheme provides is a syntax-checker. This tool performs a number of tasks:

1. it identifies and highlights syntax errors;
2. it highlights unbound identifiers;
3. it draws arrows from bound identifiers to their binding occurrences; and
4. it permits alpha-renaming, whereby all occurrences of an identifier in a given declaration scope may be renamed.

The syntax checker is useful for beginners, as it helps them to understand the syntax of the source language. The checker is also useful for experts, who write large programs with nested scopes and re-used variable names.

Next, DrScheme features among its tools a symbolic algebraic stepper, which can display a program's execution as an algebraic calculation, according to a standard reduction semantics for Scheme. The stepper displays the program in three pieces:

1. the evaluated definitions;
2. the expression prior to the current step; and
3. the expression following the current step.

The stepper is useful both in debugging and in understanding the details of the language semantics.

Finally, DrScheme provides static type-checking through MrSpidey analyzer[7]. MrSpidey seeks to prove for a given program that no type errors occur, where a type error is defined as the calling of a primitive procedure with an improper type. When MrSpidey cannot offer such a guarantee, it flags the location where the mis-application of a primitive may occur.

MrSpidey also has an explicit assertion mechanism, of the form (*: expression type*). Using this form, the user may force MrSpidey to check whether an expression is guaranteed to evaluate to a given type. So, for instance, the assertion (*:* (+ 3 5) *str*) fails, because the result of evaluating (+ 3 5) is a number rather than a string.

Furthermore, MrSpidey provides useful information to the user in the form of graphical inference chains. If an inappropriate argument might reach a primitive, MrSpidey visually depicts the execution path whereby this argument arrives at the erroneous application.

5 Building a Little Language Environment

In order to deliver a useful programming environment to the little-language programmer, DrScheme's tools must work seamlessly with the new forms of S-XML. In the following sections, we examine several of DrScheme's tools and how their behavior must change to accommodate the embedded language.

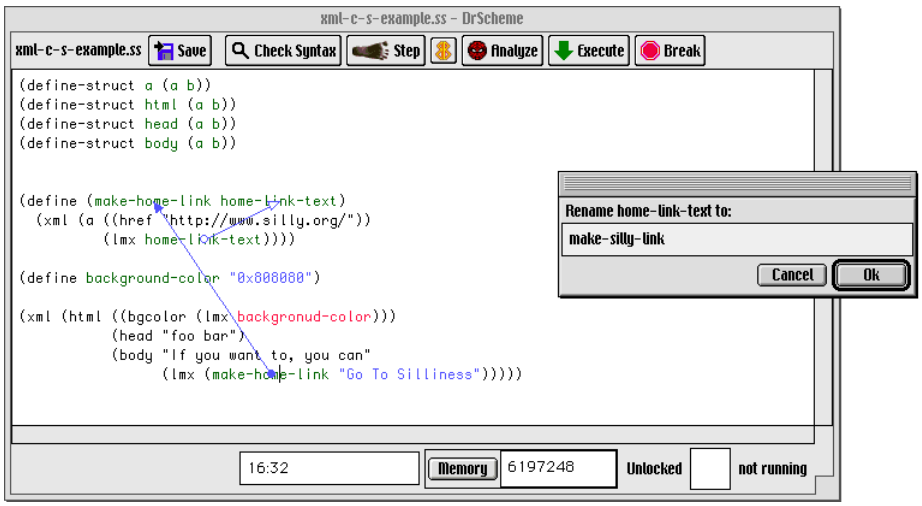

Figure 6: Check Syntax works through macros

5.1 Editing

Designing the little language as a tree-structured expression guarantees that these features are inherited immediately; editing programs in the little language is as convenient as editing Scheme. The only modification required to the programming environment is the addition of the **xml-match** keyword to the list of specially indented keywords in DrScheme's preference panel.

5.2 Check Syntax

The Check Syntax tool is designed to work transparently through macros. No modification whatsoever is required to extend the syntax checker for an embedded language.

The syntax checker is particularly useful for embedded languages, where the language's syntax is often described informally. For instance, even an experienced programmer might be surprised when using an embedded language to discover that certain identifiers are unbound, or are bound to locations other than expected.

For an example of this, see figure 6, an example using the S-XML language. The **xml** form declares an XML element. Within this element, the **lmx** form allows the user to insert evaluated Scheme code — an "escape" into the parent language. This example shows the definition of a simple web page. The binding arrows show how *make-home-link* and *home-link-text* are bound, and the red highlighting on *backgronud-color* indicate that this identifier is unbound (in this case, because of a simple typo). Finally, the 'rename ... to' box shows how users can rename all occurrences of a specific binding in an S-XML transformation.

5.3 Stepper Example

When a programmer embeds a little language within Scheme, the Foot should be transparent with respect to the macros and libraries introduced by the embedded language. In

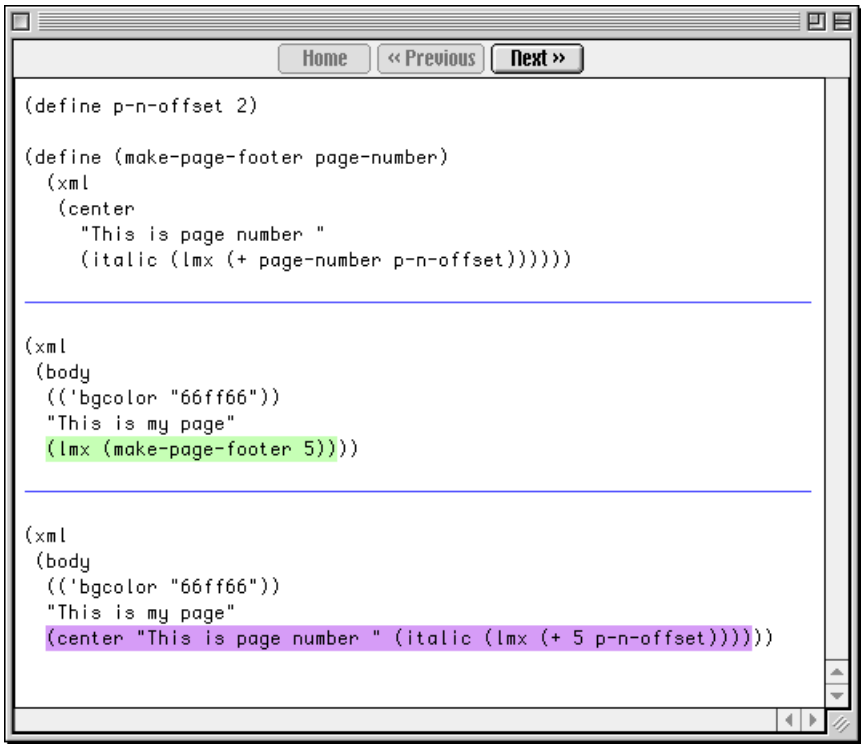

Figure 7: The stepper works through macros

other words, it must "step" in a manner that corresponds to the reductions of the embedded language, rather than the host language.

S-XML embeds several forms within Scheme; each has a natural reduction sequence. The **xml** form must simply be transparent; **xml** values are displayed as such, and computation within these terms (using the **lmx** form) are properly embedded. The **schema** form is trivial, as it contains no runtime computation. The **xml-match** form shows steps corresponding to the location of the proper pattern, and those within the corresponding pattern.

Figure 7, shows a step in the evaluation of a simple HTML construction. The stepper highlights the reducible subexpression in green, and the resulting subexpression in purple. The call to *make-page-footer* is replaced by the body of the procedure, and the value of the argument is substituted in the bound location in the body.

5.4 Validity Checking

MrSpidey provides an assertion mechanism to enable programmers to check statically that certain variables may only be bound to values of a given type. The natural extension of this assertion ability in the S-XML language is to use the assertion operator for validity checking. In S-XML, a schema expands into a MrSpidey type definition.

This type definition may then be used to implement S-XML validity checking, as shown

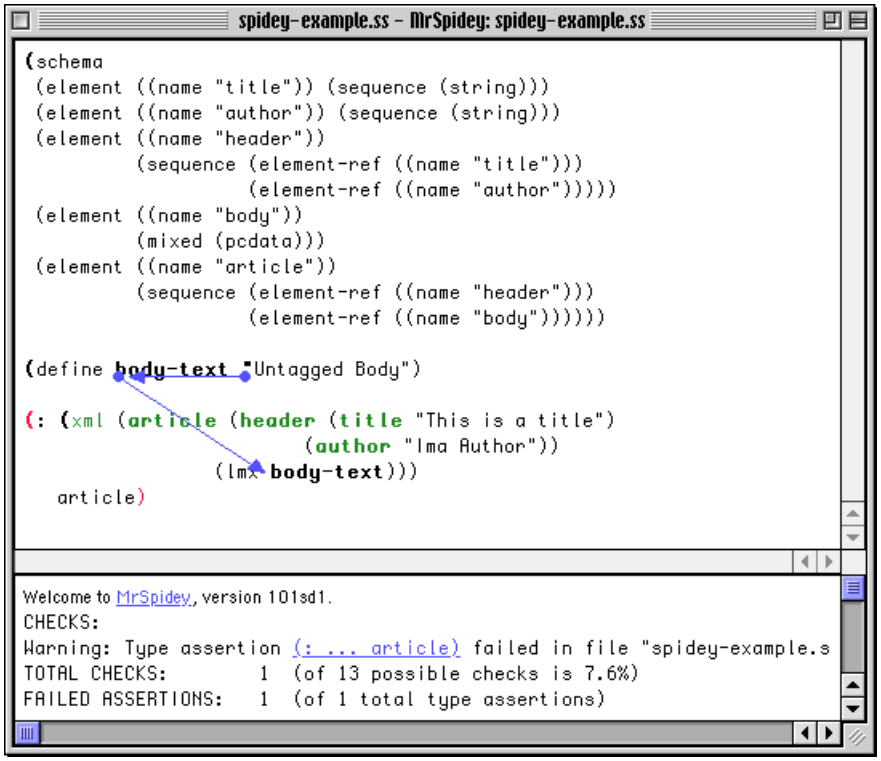

Figure 8: MrSpidey catches validity errors

in figure 8. Rather than a *body*, this *article* has simply a string. This is illegal, by the schema that appears above. Therefore, MrSpidey highlights the offending assertion in red. The path from the string to its use in the **xml** form is indicated by a series of arrows.

6 How It All Works

The extension of DrScheme's programming tools to S-XML is largely automatic. The key technologies required are source correlation and *rectifiers*.

In DrScheme, source elaboration of macros is performed by McMicMac[15]. McMicMac transforms a source file (a character stream) into an abstract syntax tree. Each term in the tree has a reference to some position of the source file. These references are preserved by McMicMac's subsequent macro elaboration, so that each term in the fully elaborated program has a direct reference to a source location. This elaborated program goes to the evaluator for execution.

As a consequence, the static tools (including the syntax-checker and MrSpidey) operate transparently with respect to macros. These tools draw conclusions about the elaborated program, and display the results using source-correlation indirection. Hence, they require no modification whatsoever to accommodate the embedded language.

The interpreter and the stepper draw heavily on source correlation as well. However,

since these tools are not static, they must also display the runtime values and expressions of the embedded language. DrScheme employs Rectifiers to perform these back-translations. There are two types of rectifiers; value rectifiers, and expression rectifiers.

A little language that enriches the value set of the host language must include a way to display its values to the user. Value rectifiers perform this translation. That is, if the little language introduces new language forms for the creation of data, the programming tools should display the resulting values using the same forms that the programmer employed to create the data. In S-XML, the following REPL interaction illustrates this:

> (**xml** (*center* "this is page number " (*em* (**lmx** (+ 1 2)))))
(**xml** (*center* "this is page number " (*em* 3)))

Since value rectifiers deal exclusively with runtime values, they have no need of source correlation. A value rectifier provides a mapping from values to displayed information.

The second category of rectifier comprises the expression rectifiers. These arise in the operation of the stepper, which must reconstruct each step within the host language's evaluator as a step within the embedded language. In some cases, the elaborated forms may have been partially evaluated. For instance, the evaluation of the **xml-match** form may proceed through many reductions. Each of these must be displayed as an **xml-match** term. Expression rectifiers make heavy use of source correlation information, as they must reconstruct source terms based upon the history of macro elaboration imposed upon the source.

For the S-XML language, we have constructed these rectifiers explicitly. Future work includes generating them automatically from the macros and libraries that make up the language embedding.

7 Related Work

Our work relates to four distinct areas. They are, in descending order of relevance: the construction of programming environments; the embedding of little languages in host languages; the problem of debugging optimized code; and transformation languages for XML. EMACS is by far the most prominent effort to produce an extensible and customizable programming environment [19]. With a few hundred lines of EMACS code, a programmer can create an EMACS mode that assists with some syntactic problems (indentation, syntax coloring) or with a read-eval-print loop (source correlation of run-time environment). But, the EMACS extensions have to be produced manually; they are not connected or derived from the little language embedding.

Most other work on the construction of programming environments focuses on the creation of tools from language specifications. For example, Teitelbaum, Reps, and others have created the Cornell Synthesizer Generator [17], which permits programmers to use attribute grammar technology to define syntax-directed editors. The ASF+SDF research effort [12] has similar, but more comprehensive goals. A programmer who specifies an algebraic-denotational semantics for a little language can create several interesting tools in this framework. In contrast, our work concentrates on the pragmatic problem of creating or prototyping language tools rapidly. In particular, we accommodate an existing implementation without any modifications. Given that most implementations are not derived formally, our work has greater potential to be applied to other environments.

Second, our most interesting technical problem concerns the relationship between the execution of elaborated code and the source text. At first glance, this suggests a commonality between our work and the work on debugging optimized object code. More specifically, code optimizations are problematic for debuggers and our algebraic stepper. Both need to cope with code transformations when they interrupt the execution of a program. Hennessy[8], Adl-Tabatabai and Gross[1], and Cooper, Kennedy and Torczon[3] describe solutions to the problem of debugging optimized code. We believe, however, that the two communities apply different techniques for the backwards translations due to the radically different levels of languages. We are currently studying whether the techniques carry over from the debugging to the stepping problem and whether the adaptation of these techniques has any advantages.

Third, although our paper is not about techniques for language embeddings, it heavily draws on ideas in that area. The history of language embeddings starts with LISP [20] and McIlroy, who introduced the notion of macro transformations in 1962 [16]. Over the past decade, the Scheme programming language introduced three important innovations in macro systems. First, Kohlbecker, et al.[13] showed how to render macro expanders *hygienic*, that is, make them compatible with the lexical structure of a host language. Second, Kohlbecker and Wand introduced the macros-by-example specification method[14]. Last, but not least, Dybvig, Hieb and Bruggeman [4] implemented the first source-correlating macro system; our work is based on the more powerful McMicMac program elaborator[15].

More recently, other language communities have rediscovered the idea of embedding languages for reuse. Fairbairn[5], Hudak[9], Wallace and Runciman[22] use Haskell's infix operators and higher-order functions to embed little languages,[4] including a little language for XML; Kamin and Harrison [10] are working along similar lines, using SML. All of these efforts focus on embedding techniques; none has paid attention to the programming environments of little languages.

Fourth, our paper, like that of Wallace and Runciman [22] and Thiemann[21] address the problem of transforming XML elements. Our solution solves a problem from which both of the other approaches suffer. Specifically, using S-XML programmers can specify XML trees in a generic manner yet they still get the benefits of XML validity checking.

8 Conclusion

We must learn to re-use all levels of language technology in the construction of little languages. The potential benefits are enormous. Shivers[18] reports that his version of `AWK`, which is more powerful than the original, is one tenth of the original's size. A small implementation is also easy to manage and to change. Hence, an embedded language is easier to extend than a stand-alone language. An improvement to the host language generally improves the embedded language(s) immediately. Finally, if one language plays host to several embedded languages, programs in the latter can easily exchange structured forms

[4] These efforts use higher-order functions to express little language programs because the chosen host languages do not provide facilities for defining new language constructs that declare variables. A detailed discussion of this distinction is irrelevant to the topic of our paper.

of data, e.g., lists, trees, arrays. In contrast, stand-alone implementations must employ the operating system's tool box, which often means that "little language programmers" must write parsers and unparsers.

With this paper we wish to contribute to the argument for language embeddings, and we hope to direct the attention of researchers to the programming environments of little languages. More centrally, we illustrate how an embedding also creates a powerful programming environment for little languages. The construction hinges on three properties of the host language and environment. First, the host language must have a mechanism for defining new language constructs. Otherwise the user of a little language must immediately know everything about the host language. Second, the mechanism must translate instances of the new constructs in such a manner that the tools can report results in terms of the surface syntax. Finally, the tools must not contain hard-wired assumptions about the source language.

For our example, we had to add two small functions to two environment tools: one for translating Scheme values back into S-XML syntax, and another one for reconstructing an S-XML construct that has a multi-step algebraic reduction semantics. Based on our experience, we conjecture that this effort can be automated and we plan to tackle the problem in the future.

References

[1] Adl-Tabatabai, A.-R. and T. Gross. Source-level debugging of scalar optimized code. In *Programming Language Design and Implementation*, May 1996.

[2] Bray, T., J. Paoli and C. Sperberg-McQueen. Extensible markup language XML. Technical report, World Wide Web Consortium, Feburary 1998. Version 1.0.

[3] Cooper, K. D., K. Kennedy, L. Torczon, A. Weingarten and M. Wolcott. Editing and compiling whole programs. In *Software Engineering Symposium on Practical Software Development Environments*, December 1986.

[4] Dybvig, R. K., R. Hieb and C. Bruggeman. Syntactic abstraction in Scheme. *Lisp and Symbolic Computation*, 5(4):295–326, December 1993.

[5] Fairbairn, J. Making form follow function: An exercise in functional programming style. *Software—Practice and Experience*, 17(6):379–386, June 1987.

[6] Findler, R. B., C. Flanagan, M. Flatt, S. Krishnamurthi and M. Felleisen. DrScheme: A pedagogic programming environment for Scheme. In *International Symposium on Programming Languages: Implementations, Logics, and Programs*, pages 369–388, 1997.

[7] Flanagan, C., M. Flatt, S. Krishnamurthi, S. Weirich and M. Felleisen. Catching bugs in the web of program invariants. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 23–32, May 1996.

[8] Hennessy, J. Symbolic debugging of optimized code. *Transactions on Programming Languages and Systems*, 4(3):323–344, 1982.

[9] Hudak, P. Modular domain specific languages and tools. In *International Conference on Software Reuse*, 1998.

[10] Kamin, S. and D. Hyatt. A special-purpose language for picture-drawing. In *USENIX Conference on Domain-Specific Languages*, 1997.

[11] Kelsey, R., W. Clinger and J. Rees. Revised^5 report on the algorithmic language Scheme. *ACM SIGPLAN Notices*, 33(9), October 1998.

[12] Klint, P. A meta-environment for generating programming environments. *ACM Transactions on Software Engineering and Methodology*, 2(2):176–201, 1993.

[13] Kohlbecker, E. E., D. P. Friedman, M. Felleisen and B. F. Duba. Hygienic macro expansion. In *ACM Symposium on Lisp and Functional Programming*, pages 151–161, 1986.

[14] Kohlbecker, E. E. and M. Wand. Macros-by-example: Deriving syntactic transformations from their specifications. In *Symposium on Principles of Programming Languages*, pages 77–84, 1987.

[15] Krishnamurthi, S., M. Felleisen and B. F. Duba. From macros to reusable generative programming. In *Generative and Component-Based Software Engineering*, September 1999.

[16] McIlroy, M. D. Macro instruction extensions of compiler languages. *Communications of the ACM*, 3(4):214–220, 1960.

[17] Reps, T. W. and T. Teitelbaum. *The Synthesizer Generator*. Springer-Verlag, 1989.

[18] Shivers, O. A universal scripting framework or, Lambda: the ultimate "little language". In Jaffar, J. and R. H. C. Yap, editors, *Concurrency and Parallelism: Programming, Networking and Security*, pages 254–265. Springer-Verlag, 1996. LNCS 1179.

[19] Stallman, R. EMACS: the extensible, customizable, self-documenting display editor. In *Symposium on Text Manipulation*, pages 147–156, 1981.

[20] Steele, G. L., Jr. and R. P. Gabriel. The evolution of Lisp. In Bergin, T. J., Jr. and R. G. Gibson, Jr., editors, *History of Programming Languages—II*, pages 233–308, 1996.

[21] Thiemann, P. Modeling HTML in Haskell. In *Practical Applications of Declarative Languages*, January 2000.

[22] Wallace, M. and C. Runciman. Haskell and XML: Generic document processing combinators vs. type-based translation. In *International Conference on Functional Programming*, September 1999.

[23] Walsh, N. A technical introduction to XML. *World Wide Web Journal*, Winter 1997.