Fun with Embedded Domain-Specific Languages: Embedding an XHTML-template language into Perl

Thomas G. Moertel

Moertel Consulting

8 March 2006

What's an EDSL?

Two great tastes that taste great together!

- Embed a special-purpose, domain-specific language ...
- Into a general-purpose programming language ...
- So that you can use both languages together

Example: XHTML-construction language embedded in Perl

```
html {
    head {
        title { "My grand document!" }
    };
    body {
        h1 { "Heading" };
        p {
            class_ "first";        # attribute class="first"
            text "This is the first paragraph!";
            style_ "font: bold";  # another attr
        };
        # it's just Perl, so we can mix in other code
        for (2..5) {
            p { "Plus paragraph number $_." }
        }
    };
};
```

Example: Sample output

```
<html>
<head>
<title>My grand document!</title>
</head>
<body>
<h1>Heading</h1>
<p class="first" style="font: bold">This is the first
paragraph!</p>
<p>Plus paragraph number 2.</p>
<p>Plus paragraph number 3.</p>
<p>Plus paragraph number 4.</p>
<p>Plus paragraph number 5.</p>
</body>
</html>
```

Game plan

1. Capture the domain's magic in an *embedding syntax*

   ```
   p { "I'm a paragraph" }
   ```

2. Translate the syntax into a sensible *internal representation*

   ```
   [ "p", undef, [ "I'm a paragraph" ] ]
   ```

3. Render the IR into the final output

   ```
   <p>I'm a paragraph</p>
   ```

Embedding syntax—XHTML-y Perl

Let Perl do the parsing for us

```
element   =   name { children }

children  =   string
          |   mixed*

mixed     =   element
          |   attribute
          |   text

attribute =   name_ value

text      =   text string
```

Internal representation—Perl LoL-style tree

Capture the domain's magic in a simple Perl data structure

```
element    =  [ "name", attributes, children ]

attributes =  undef
           |  [ attribute+ ]

attribute  =  [ "name", value ]

children   =  undef
           |  [ child+ ]

child      =  string        ← text node
           |  element
```

Implementation in detail

In other words, *The Code*

Translating the embedding syntax into IR

The goal:

Embedding syntax $\longrightarrow$ Internal representation

Translating the embedding syntax into IR

The goal:

Embedding syntax $\longrightarrow$ Internal representation

```
p { "I'm a paragraph" }
```

$\downarrow$

```
[ "p", undef, [ "I'm a paragraph" ] ]
```

Parsing the syntax

Define:

```
sub p(&) { make_elem( "p", @_ ) }
```

Parsing the syntax

Define:

```
sub p(&) { make_elem( "p", @_ ) }
```

```
p { "I'm a paragraph" }
```

↓

```
make_elem( "p", sub { "I'm a paragraph" } )
```

Parsing the syntax (2): what `make_elem` does

```
# start with empty tree root

our $__frag = [
    undef, undef,  # name, attrs
    undef          # children
];

make_elem( "p", sub { "I'm a paragraph" } );

# now tree looks like this:

$__frag = [
    undef, undef,
    [ [ "p", undef, [ "I'm a pragraph" ] ] ]
];
```

Parsing the syntax (3): appending an element

```
sub make_elem {
    my ($elem_name, $content_fn) = @_;
    # an element = [name, attrs, children]
    my $elem = [$elem_name, undef, undef];
    my $t = do { local $__frag = $elem; $content_fn->() };
    $elem->[2] ||= [ $t ] if defined $t;
    push @{$__frag->[2]}, $elem;
    undef;
}
```

Parsing the syntax (4): appending attrs and text

```
sub make_attr {
    my ($attr_name, $val) = @_;
    push @{$__frag->[1]}, [$attr_name, $val];
    undef;
}

sub text($) {
    push @{$__frag->[2]}, @_;
    undef;
}
```

Parsing the syntax (5)

```
sub define_vocabulary {
    my ($elems, $attrs) = @_;
    eval "sub $_(&) { make_elem('$_',\@_) }"
        for @$elems;
    eval "sub ${_}_(\$) { make_attr('$_',\@_) }"
        for @$attrs;
}

BEGIN {
    define_vocabulary(
        [qw( html head title body div p img br h1 ... )],
        [qw( src href class style id ... )]
    );
}
```

Rendering the IR into XML

The goal:

```
[ "p", undef, [ "I'm a paragraph" ] ]
```

↓

```
<p>I'm a paragraph</p>
```

An XML::Writer-based renderer

```
use XML::Writer;
sub render_via_xml_writer {
    my $doc = shift;
    my $writer = XML::Writer->new(@_);  # extra args -> new()
    my $render_fn;
    $render_fn = sub {
        my $frag = shift;
        my ($elem, $attrs, $children) = @$frag;
        $writer->startTag( $elem, map {@$_} @$attrs );
        for (@$children) {
            ref() ? $render_fn->($_)
                  : $writer->characters($_);
        }
        $writer->endTag($elem);
    };
    $render_fn->($doc);
    $writer->end();
}
```

An XML::Writer-based renderer (2)

```
sub render_doc(&) {
    my $docfn = shift;
    render_via_xml_writer(
        doc( \&$docfn ),
        DATA_MODE => 1,
        UNSAFE => 1
    );
}

sub doc(&) {
    my ($content_fn) = @_;
    local $__frag = [undef,undef,undef];
    $content_fn->();
    $__frag->[2][0];
}
```

Finally, it all comes together

```
render_doc {
    p { "I'm a paragraph" }
};
```

```
<p>I'm a paragraph</p>
```

Finally, it all comes together

```
render_doc {
    div {
        p { id_ "p$_"; "I'm paragraph $_." } for 1..3;
        a { href_ "#p1"; "Link to paragraph 1" };
    };
};
```

↓

```
<div>
<p id="p1">I'm paragraph 1.</p>
<p id="p2">I'm paragraph 2.</p>
<p id="p3">I'm paragraph 3.</p>
<a href="#p1">Link to paragraph 1</a>
</div>
```

For more information

Thomas G. Moertel
Moertel Consulting
tgm@moertel.com
http://blog.moertel.com/