Tips From a Lazy DSL Designer

Andru Luvisi

Sonoma State University

ABSTRACT

This talk will discuss various approaches to implementing Domain Specific Languages (DSLs), with an emphasis on ease of implementation. Some language specific techniques will be discussed, along with some more general approaches that can be applied in many programming languages. A couple of DSLs used within SSU/IT may be described if time permits. The latest version of this handout is available from http://www.sonoma.edu/users/l/luvisi/dsls/

1. Domain Specific Languages: Focusing on the problem domain

A domain specific language is one that focuses more closely on the items and actions within a problem domain than it does on how activities are carried out within the computer.

DSLs are a special case of high level programming languages. A DSL is usually a higher level programming language than the general purpose language that would otherwise be used. As such, DSLs tend to inherit most of the advantages that come with moving to a higher level programming language.

1.1. Shorter Programs

Programs written in DSLs tend to contain less boilerplate code because boilerplate code tends to be the low level stuff that is irrelevant to the problem in question.

1.2. Easier Reasoning

By reducing the number of ideas that the programmer must consider while writing a program, and by increasing the level of abstraction so the programmer does not need to worry about as many low level details, DSLs can make it easier to reason about the meaning of a program.

1.3. Fewer Bugs

Higher level languages facilitate the writing of programs with fewer bugs by giving the programmer less work to do, and hence fewer opportunities for errors ([9] page 135).

1.4. Faster Development

Programmers tend to produce the same number of lines of code per day, no matter what level language is being used ([9] pages 87-94). By making a program shorter, a DSL can decrease the amount of time required to write the program.

1.5. Easier Maintenance

The more verbose a program is, the more difficult it is for a maintenance programmer to determine the original intent of the author. Low level details must be collected and analyzed in order to figure out what goal they are furthering. A well designed DSL will make it possible for the programmer to express the programmer's intent concisely and clearly, in a manner that will later be easily understood by a maintenance programmer, who may even be the original programmer.

1.6. Non-Programmers

Sometimes, if a DSL can be designed in a sufficiently abstract fashion, a subject matter expert who is not a programmer[1] can write or edit programs written in the DSL.

1.7. Managing Complexity

I cannot possibly express this better than Abelson and Sussman[4]:

> Establishing new languages is a powerful strategy for controlling complexity in engineering design; we can often enhance our ability to deal with a complex problem by adopting a new language that enables us to describe (and hence to think about) the problem in a different way, using primitives, means of combination, and means of abstraction that are particularly well suited to the problem at hand.

or Steele and Sussman[32]:

> One decomposition strategy is the packaging of common patterns of the use of a language. For example, in Algol a *for* loop captures a common pattern of *if* and *goto* statements. Packages of common patterns are not necessarily merely abbreviations to save typing. While a simple abbreviation has little abstraction power because a user must know what the abbreviation expands into, a good package encapsulates a higher level concept which has meaning independent of its implementation. Once a package is constructed the programmer can use it directly, without regard for the details it contains, precisely because it corresponds to a single notion he uses in dealing with the programming problem.

[1] By "not a programmer," I mean someone who does not program in a general purpose programming language. Of course, by writing or editing programs written in a DSL, a person becomes a "programmer" in some sense.

2. Types of DSLs

2.1. Standalone/External DSLs

These are languages implemented with conventional interpreters or compilers. A program is contained in a file or a string. Here are some examples that may be familiar to Unix users:

- sed
- awk
- troff
- bc, dc
- make
- lex
- yacc

2.2. Embedded/Internal

These are languages which are embedded within a host language of some sort. They may be implemented by extending the host language to include the desired domain specific features or by passing the source file through a preprocessor that compiles the DSL into the host language. Here are some preprocessor based DSLs that may be familiar to troff users:

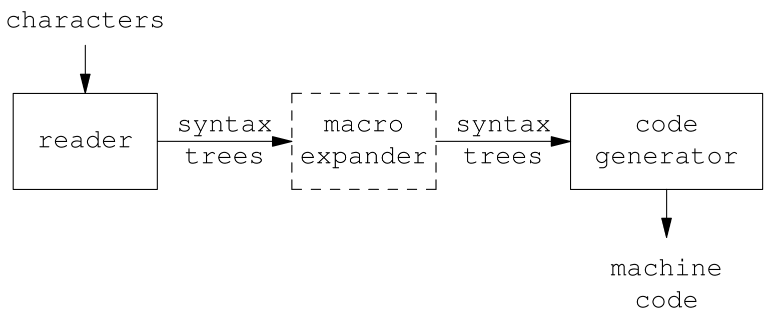
- eqn
- tbl
- pic
- grap

3. Implementation Techniques

3.1. Language specific tricks

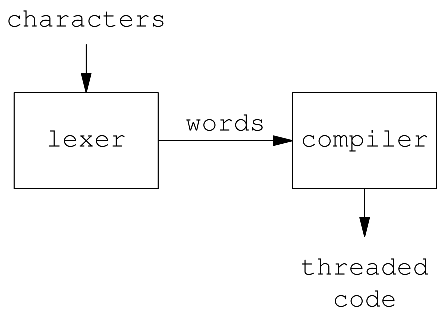
3.1.1. Lisp Macros

Lisp macros are procedures written in Lisp that transform the already parsed representation of part of a Lisp program[17, 16]. This makes it possible to extend the compiler without modifying any of the existing parts of the compiler, and without any knowledge of how they work.



3.1.2. Forth immediate words

In a classic Forth compiler, when a new word is being compiled words are read and the addresses of their definitions are stored one after another into the new threaded code. Words that are marked as "immediate," however, will be executed during compilation. They can read ahead in the input stream and they can compile other words into the code being built. This makes it possible to extend the compiler in arbitrary ways. Several standard builtin words are normally implemented as "immediate" words[8].



3.1.3. C++ Template Metaprogramming

C++ templates provide a Turing-complete language for generating C++ code at compile time[11].

3.1.4. Creating an extended environment

Some languages make it possible to execute a block of code in an environment that has been augmented with domain specific extensions.

3.1.4.1. JavaScript "with" statement

The JavaScript "with" statement executes a block of code in an environment that has been extended with the properties of a given object[13]. In the following example from the *Core JavaScript 1.5 Guide*[1], the free variables PI, cos, and sin are resolved to the properties of the same name within the Math object.

```
var a, x, y;
var r = 10;
with (Math) {
   a = PI * r * r;
   x = r * cos(PI);
   y = r * sin(PI/2);
}
```

3.1.4.2. Ruby instance_eval/instance_exec

In Ruby, the Object#instance_eval method executes a block of code as if it were a member function of the object in question[2]. This can be used by a library author to let the library user write small blocks of code in a highly customized environment. In the following example from *Programming Ruby*[33], there are several identifiers like "title" and "padx" that have a very declarative feel. They are, in reality, member functions of library objects, but to the programmer using the library they seem like primitives in a Domain Specific Language.

```
require 'tk'
root = TkRoot.new { title "Ex1" }
TkLabel.new(root) do
  text 'Hello, World!'
  pack { padx 15; pady 15; side 'left' }
end
Tk.mainloop
```

3.2. Language independent tricks

3.2.1. Create a library

It is possible to write libraries using a philosophy that resembles language design, and offers many similar benefits. Abelson and Sussman have the following to say about this[4]:

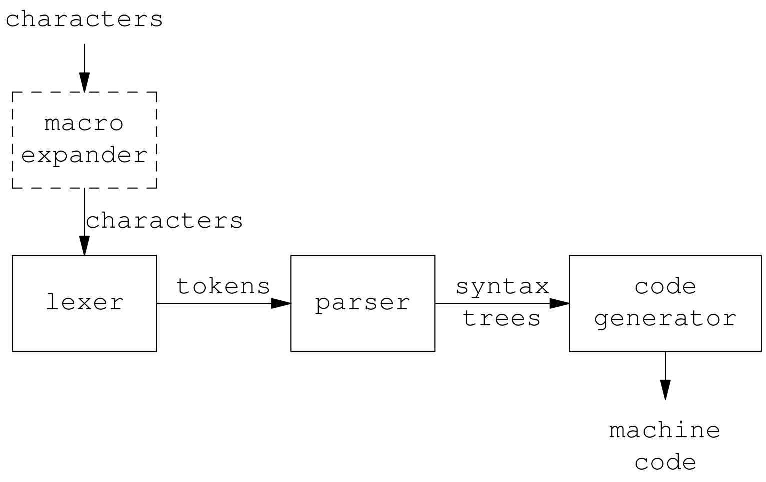
> This is the approach of stratified design, the notion that a complex system should be structured as a sequence of levels that are described using a sequence of languages. Each level is constructed by combining parts that are regarded as primitive at that level, and the parts constructed at each level are used as primitives at the next level. The language used at each level of a stratified design has primitives, means of combination, and means of abstraction appropriate to that level of detail.
> ...
> Stratified design helps make programs robust, that is, it makes it likely that small changes in a specification will require correspondingly small changes in the program.
> ...
> In general, each level of a stratified design provides a different vocabulary for expressing the characteristics of the system, and a different kind of ability to change it.

Dijkstra makes a similar case[12]:

> [W]e have arranged our program in layers. Each program layer is to be understood all by itself, under the assumption of a suitable machine to execute it, while the function of each layer is to simulate the machine that is assumed to be available on the level immediately above it.
> ...
> It is not in vain to hope that many a program modification can now be presented as replacement of one (virtual) machine by a compatible one.

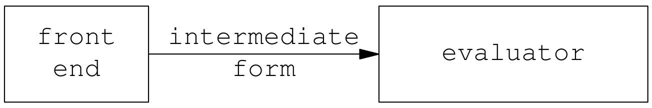
3.2.2. Preprocessor

It is possible to implement a DSL as a preprocessor that compiles DSL instructions into instructions in the host language. Examples include Perl source filters, the before mentioned troff preprocessors, and RATFOR[20].



3.2.3. Classic Interpreter

The following techniques are all ways of implementing a classic blueprint for an interpreter, where a front end creates a data structure representing the program to be executed, and the back end executes it. The choice of the front end is largely independant of the choice of representation and evaluator, but the choice of representation will impact what evaluation strategies are available.



3.2.3.1. Front end

3.2.3.1.1. Full parser

The most general and flexible front end is a parser that converts text into an internal representation. This has the potential to be the most difficult method for creating a front end, but by keeping the language grammar simple and using tools that can create a parser given a grammar description[21, 22, 5], the required effort can still be kept to a minimum.

There are also usability advantages in keeping the grammar simple. The easier it is to write a parser for a language, the easier it tends to be for a human to read the language.

If the language is simple enough, it can be interpreted during parsing, but this choice may limit the ability to extend the language in the future.

3.2.3.1.2. Already written parser

Sometimes an implementer can save effort by borrowing an existing syntax for which a parser has already been written and only implementing the semantics of the new language. For example, Common Lisp and Scheme come with an S-Expression parser built in. If a DSL is based on S-Expressions, then a Common Lisp or Scheme program can parse the DSL into a syntax tree with no effort on the part of the implementer.

Libraries are also available for many programming languages that parse other common data representations, such as YAML and JSON.

It is possible to use an already written XML parser and base the DSL on XML, but that is a very cruel thing to do to the user of the DSL.

3.2.3.1.3. Data structure literals

If the host language has sufficiently convenient syntax for data structure literals, then programs written in the DSL can be embedded directly within the code of the host program.

3.2.3.1.4. Function calls

It is possible to construct the intermediate form directly using nested procedure calls[23]. This gives the DSL a lisp like look, with "procedure(arg, arg)" replacing "(procedure arg arg)". Using this approach, the DSL piggybacks on the lexer and parser of the host language. Partial programs can be stored in variables and used in the construction of more complex programs.

3.2.3.2. Intermediate forms

There are many ways of representing a program. Which representation is best for a particular implementation depends on what will be done with the program. Some representations are easier to analyze than others, some are easier to execute than others, and some are easier to serialize and store in a file or pass over the network than others. This is an example of a general data processing principle described by Peter Naur ([26] page 28):

> The data representation must be chosen with due regard to the transformation to be achieved and the data processing tools available.

In this talk I will only discuss three representations which are convenient with respect to executing the program.

3.2.3.2.1. Syntax Trees

This is one of the most flexible ways to represent a program. A node in the tree is a tagged union capable of representing any type of expression in the language. Expressions containing other expressions are represented using pointers.

Syntax trees are easy to analyze or change and they are relatively easy to serialize for storage or transmission over a network.

Interpreting a syntax tree is usually done with a recursive procedure that takes a node and an environment as arguments and chooses its behavior based on the type of node passed. This is a well trodden path and there are many excellent descriptions in print of how to do this[4, 32, 29, 28, 24, 30, 27, 14, 6, 18, 10, 25].

Here is an example in C of a syntax tree interpreter for a very simple expression language that contains only constants, variables, comparisons, and conditionals, and only returns integers.

```
struct expression {
  enum { CONSTANT, VARIABLE, COMPARISON, CONDITIONAL } type;
  union {
    int constant;
    char *variable;
    struct {
      struct expression *expression1;
      struct expression *expression2;
    } comparison;
    struct {
      struct expression *condition;
      struct expression *true_branch;
      struct expression *false_branch;
    } conditional;
  } u;
};

int evaluate(struct expression *exp, struct environment *enp) {
  switch(exp->type) {
    case CONSTANT:
      return exp->u.constant;
    case VARIABLE:
      return lookup(exp->u.variable, enp);
    case COMPARISON:
      return evaluate(exp->u.comparison.expression1, enp) ==
             evaluate(exp->u.comparison.expression2, enp);
    case CONDITIONAL:
      if(evaluate(exp->u.conditional.condition, enp))
        return evaluate(exp->u.conditional.true_branch, enp);
      else
        return evaluate(exp->u.conditional.false_branch, enp);
  }
}
```

Code is generated from a syntax tree using an approach very similar to that of interpretation. A recursive procedure walks the tree and returns or outputs the generated code. Again, this is a well trodden path[4, 27, 28, 6, 5, 7].

3.2.3.2.2. Objects (The "Interpreter" Design Pattern)

In an object oriented host language, the nodes of the syntax tree can be objects which implement an "evaluate" member function that takes an environment as an argument, performs the actions associated with the node, and returns the result of the computation[15].

An object oriented syntax tree is still relatively easy to analyze or modify, but is more difficult to serialize for storage or transmission.

Here is an example C++ implementation of object oriented syntax trees.

```
class expression {
  public:
    virtual int evaluate(environment *e) = 0;
};

class constant : public expression {
  private:
    int value;
  public:
    constant(int i) : value(i) {}
    int evaluate(environment *e) {
      return value;
    }
};

class variable : public expression {
  private:
    char *var;
  public:
    variable(char *s) : var(s) {}
    int evaluate(environment *e) {
      return lookup(var, e);
    }
};

class equals : public expression {
  private:
    expression *expression1;
    expression *expression2;
  public:
    equals(expression *exp1, expression *exp2) :
      expression1(exp1), expression2(exp2) {}
    int evaluate(environment *e) {
      return expression1->evaluate(e) ==
             expression2->evaluate(e);
    }
};

class ifelse : public expression {
  private:
    expression *condition;
    expression *true_branch;
    expression *false_branch;
  public:
    ifelse(expression *condition,
           expression *true_branch,
           expression *false_branch) :
           condition(condition),
           true_branch(true_branch),
           false_branch(false_branch) {}
    int evaluate(environment *e) {
      if(condition->evaluate(e))
        return true_branch->evaluate(e);
      else
        return false_branch->evaluate(e);
    }
};
```

3.2.3.2.3. Closures

In languages that support lexical closures, it is possible to construct a procedure that performs the actions of the DSL program. The procedure can then be executed[4, 28]. This is a lightweight approach that can be implemented with a fairly small amount of code. The procedure created is easy to execute, but difficult to analyze, change, or serialize. In functional programming languages this is sometimes implemented using combinator libraries[3].

A lexical closure is a procedure that can access parts of the environment in which it was created. Here is an example of lexical closures in Perl:

```
sub make_adder {
  my($augend) = @_;
  return sub {
    my($addend) = @_;
    return $addend + $augend;
  };
}

my $add2 = make_adder(2);
my $add5 = make_adder(5);

print $add2->(10), "\n"; # Prints "12"
print $add5->(10), "\n"; # Prints "15"

sub compose {
  my($procedure1, $procedure2) = @_;
  return sub {
    my($argument) = @_;
    return $procedure1->($procedure2->($argument));
  };
}

my $add7 = compose($add2, $add5);
print $add7->(10), "\n"; # Prints "17"
```

When implementing an interpreter in this fashion, it is common to define a common interface to procedures implementing programs. For example, they may all accept a single argument that specifies an environment and return a single value that represents the result of the computation. All created procedures must be of this type. For simplicity, I represent environments as hash refs in the following example.

```
# A constant will always evaluate to the same value, regardless of
# the contents of the environment.
sub constant {
  my($value) = @_;
  return sub {
    my($environment) = @_;
    return $value;
  };
}

# A variable will be looked up in the environment.
sub variable {
  my($name) = @_;
  return sub {
    my($environment) = @_;
    return $environment->{$name};
  };
}

# Equality test.  This evaluates two expressions and compares their values
# for equality.
sub equals {
  my($expression1, $expression2) = @_;
  return sub {
    my($environment) = @_;
    return $expression1->($environment) ==
           $expression2->($environment);
  };
}

# Conditional.  Evaluate the condition, and depending on whether it
# is true or not, evaluate the true or false branch.
sub ifelse {
  my($condition, $true_branch, $false_branch) = @_;
  return sub {
    my($environment) = @_;
    if($condition->($environment)) {
      return $true_branch->($environment);
    } else {
      return $false_branch->($environment);
    }
  };
}

my $environment = {
  x => 10,
  y => 20,
};

my $expr = ifelse(equals(variable("x"), constant(100)),
                  constant(1),
                  ifelse(equals(variable("y"), constant(20)),
                         constant(2),
                         constant(3)));

print $expr->($environment), "\n"; # Prints "2"
```

3.2.4. Generating Code

Any of the approaches above can be used to generate new code rather than perform the actions described by the DSL. It takes some effort to generate efficient assembly code like a "real" compiler, but there are some ways to make things much easier.

Targeting virtual machines or threaded code can save some effort, but the result is still too difficult to qualify as a "lazy" approach.

Generating C or C++ code lets the implementer piggyback on already written code for low level details like instruction selection, register allocation, and optimization. It also provides some platform independence, and does all of this without sacrificing much speed.

Moving up the stack, if the speed hit is acceptable, an implementer could target Perl, Python, Ruby, JavaScript, or even sh. Any language that can perform the necessary operations, and for which a working compiler or interpreter is available, can be made to work. It is often convenient to target the same language in which the rest of the project is written.

Using code templates that are filled in with specific values during code generation can help to decouple the code generation logic from the logic within the generated code[31, 19].

If the host language has an "eval" feature (like Perl, Python, Ruby, Tcl, Common Lisp, Scheme, some BASICs, etc.), then the generated code can be in the host language, and it can be evaluated[31].

Generated code should never be edited by hand. Changes should always be made to the original source, in the DSL.

4. Examples

4.1. Internet Services Language

LIBS was a terminal based menu driven program in the early 1990s that was written by Mark Resmer and maintained briefly by myself. It listed libraries that were connected to the Internet. Here is the menu showing libraries from California:

```
  California Libraries:

     1 Cal Poly State U.       2 CSU Chico              3 CSU Dominguez Hills
     4 CSU Fresno              5 CSU Hayward            6 CSU Humboldt
     7 CSU Long Beach          8 CSU Los Angeles        9 CSU Sacramento
    10 CSU San Francisco      11 CSU San Luis Obispo   12 CSU San Marcos
    13 CSU Stanislaus         14 Lawrence Berkeley     15 Loyola Marymount U.
    16 Mills College          17 Point Loma Nazarene   18 Saint Mary’s College
    19 San Diego State U.     20 San Francisco Public  21 SJSU Library
    22 Santa Clara U.         23 Simpson College       24 Sonoma State U.
    25 Stanford University    26 UC Berkeley - GLADIS  27 UC Irvine
    28 UC Northern Regional   29 UC San Diego          30 UC Santa Barbara
    31 UC System - MELVYL     32 Univ. of San Diego    33 U of San Francisco
    34 U of the Pacific

              Press RETURN alone to see previous menu

              Press Control-] Q {return} to exit at any time

              Enter the number of your choice:
```

When the user selects a library, the program attempts to connect to the library’s online system, first giving the user instructions on how to login. For example, here are the (long outdated) instructions for loging into the SSU public access catalog.

```
      Sonoma State University

      Note the following instructions carefully

      Once you are connected:

      Username: OPAC {return}

      Press Control-] Q {return} to exit at any time

      Do you want to connect now? (Y or N):
```

The program was distributed in two forms, .COM files for VMS (COM is short for COMMAND. These are scripts written in Digital Command Language (DCL).) and shell scripts for Unix like systems. The information about the menus and libraries was maintained in a big file written in a Domain Specific Language called ISL (Internet Services Language). A Pascal program converted the ISL into a .COM file, and another Pascal program converted the ISL into around 500 shell scripts.

Here is the entry defining the "California Libraries" menu. The ".slabel" command indicates that an entry called "California" is to be entered into the menu called "LMENU", the internal label for the menu for the United States. The ".select" command indicates that when selected this item will display another menu with with 34 entries and "California Libraries" as the header.

```
.slabel LMENU California
.select 1-34 LIBCA California Libraries
```

Here is the entry defining the "Sonoma State U." entry in the California Libraries menu. The ".slabel" command enters this as an option in the "LIBCA" (California Libraries) menu. All lines that don't start with a command are print commands. This is an example of optimizing for the common case. The ".instruct" and ".quitinstr" commands print some standard strings. The ".telnet" command asks the user whether the user wants to connect, and connects if the answer is "Y".

```
.slabel LIBCA Sonoma State U.

Sonoma State University {b}
.instruct
Username: OPAC <return>
.quitinstr
.telnet vax.sonoma.edu
```

4.2. Tsunami query language

Tsunami is the inventory database used by Computer Operations, the server group in SSU/IT. The data is stored in an Oracle database and the business logic is implemented in Perl. One of the key features of Tsunami is extensibility. If the system administrators decide that they need to store a new attribute for objects of a particular type (such as "Server"), then they can add the new attribute to the object type without our programmers needing to modify the database schema or any code.

The API for interacting with Tsunami includes a query language for searching the database. Here is some code that creates a couple of queries. The use of these queries within the API is beyond the scope of this talk.

```
my $isActive =
     qor(qand(qis_null("Status"),
              qor(qequal("System Lifecycle Stage", "Has OS"),
                  qequal("System Lifecycle Stage", "In Use"))),
         qnot(qor(qequal("Status","Storage"),
                  qlike("Status","Surveyed%"))));

my $isServer = qtype_name("Server");

my $solarisboxes = qand($isServer,
                        $isActive,
                        qlike("OS Version", "Solaris %"));

my $dellboxes = qand($isServer,
                     $isActive,
                     qlike("Server Make", "Dell%"));
```

Notice how the $isActive and $isServer queries can be used in constructing more elaborate queries.

4.3. Permissions for IT Services Manager

The IT Services manager is a web application for administering access to IT services. Different application users have different levels of access to different properties of different community members. Access levels are ACCESS_NONE, ACCESS_READ, ACCESS_WRITE, and ACCESS_BOTH. Access is determined using a list of rules. Each rule contains a condition and an access level. The condition can test the source of the community member being accessed (how they got into the system), what groups the user doing the accessing is in, and what attribute is being accessed.

Here are some sample rules with a few details simplified to protect the guilty.

```
my $rules = [
        # For accounts from PeopleSoft, let certain attributes be edited,
        # make the rest read only.
        [ pand(psources("PeopleSoft"),
               pgroup("IT"),
               pattrs(qw(ssuMailForwardingAddress
                         ssuMailCopyTo
                         ssuMailLocalAddress
                         description
                         mail
                         # ...
                         ))),
          ACCESS_BOTH ],
        [ pand(psources("PeopleSoft"),
               pgroup("IT")),
          ACCESS_READ ],

        # For locally created accounts, protect certain attributes
        # and allow write access to the rest since they will be filled
        # out locally.
        [ pand(psources("Information Technology"),
               pgroup("IT"),
               pattrs(qw(dn sonomaedupersonauthoritativesource
                         sonomaedupersonaccountexpiredate
                         sonomaedupersonpasswordexpiredate
                         # ...
                         ))),
          ACCESS_READ ],
        [ pand(psources("Information Technology"),
               pgroup("IT")),
          ACCESS_BOTH ],
];
```

4.4. Wireless internet access role assignment

Shibboleth[2] is a technology that allows a user to be authenticated by a different web server than the one providing a service. The CSU is investigating the possibility of using Shibboleth to authenticate wireless network access. This would make it possible for a user from one CSU campus to use the wireless network at another CSU campus while authenticating using the credentials from the user's home campus.

The campus providing wireless service may wish to provide different levels of access to the wireless network depending on what campus the user is from and what affiliation(s) the user has. During a pilot program we used a role system similar to the one above.

```
my $role_rules = [
  # Roles for local users.
  [ r_and(r_issuer("https://paddle.sonoma.edu/shibboleth/idp"),
          r_attribute("eduPersonAffiliation", "Employee")),
    "SSU-Employee" ],
  [ r_and(r_issuer("https://paddle.sonoma.edu/shibboleth/idp"),
          r_attribute("eduPersonAffiliation", "Student")),
    "SSU-Student" ],
  # Role for employees from other CSU campuses.
  [ r_attribute("eduPersonAffiliation", "Employee"),
    "External-Employee" ],
  # Everyone else gets an error message
  [ r_true(),
    undef ],
];
```

[2] http://shibboleth.internet2.edu/

5. Final Thoughts

- If possible, use Lisp and Lisp macros.
- Use procedure calls for the front end, closures for the representation, and call them to interpret the language. This makes it easy to store parts of a program in variables and reuse them.
- Prefer a declarative style when possible. Try to create a language in which you can state "what is", rather than "how to."
- Prefer a functional style when possible. If you can, avoid having procedures in your language that perform Input/Output, or change the values of variables. Try to have your language only compute return values.
- Avoid statements. If you need imperative features like modification of variable values or Input/Output, include expressions with side effects rather than complicating the syntax and semantics of your language with statements.
- Avoid infix operators. Keep everything prefix or postfix.
- If you must have infix operators, give them all the same precedence level and associativity, like Smalltalk (left associative) or APL (right associative).
- If you generate code, fully parenthesize the expressions that you generate. That way you can just forget about precedence.

References

1. *Core JavaScript 1.5 Guide.* http://developer.mozilla.org/en/docs/Core_JavaScript_1.5_Guide.
2. *Ruby Core Reference.* http://www.ruby-doc.org/core/classes/Object.html#M000607.
3. http://en.wikipedia.org/wiki/Combinator_library.
4. Abelson, Hal and Sussman, Gerald, *Structure and Interpretation of Computer Programs,* MIT Press (1996). http://mitpress.mit.edu/sicp/full-text/book/book.html.
5. Aho, Alfred V., Sethi, Ravi, and Ullman, Jeffrey D., *Compilers: Principles, Techniques, and Tools,* Addison-Wesley, Reading, MA (1986).
6. Allen, John, *Anatomy of Lisp,* McGraw-Hill (1978).
7. Appel, Andrew W., *Modern Compiler Implementation in ML,* Cambridge University Press (1998).
8. Brodie, Leo, *Starting Forth.* http://www.forth.com/starting-forth/.
9. Brooks, Frederick P., *The Mythical Man-Month,* Addison-Wesley, Reading, MA (1975).
10. Burge, William H., *Recursive Programming Techniques,* Addison-Wesley, Reading, MA (1975).
11. Czarnecki, Krzysztof and Eisenecker, Ulrich W., *Generative Programming,* Addison-Wesley, Reading, MA (2000).
12. Dajl, O.-J., Dijkstra, E. W., and Hoare, C. A. R., *Structured Programming,* Academic Press Inc. (London) Ltd. (1972). http://portal.acm.org/citation.cfm?id=SERIES11430.1243380.
13. Flanagan, David, *JavaScript: The Definitive Guide (5th Edition),* O'Reilly & Associates, Inc., Sebastopol, CA (2006).
14. Friedman, Daniel P., Wand, Mitchell, and Haynes, Christopher T., *Essentials of Programming Languages,* MIT Press, Cambridge, Massachusetts (1992).
15. Gamma, Erich, Helm, Richard, Johnson, Ralph, and Vlissides, John, *Design Patterns: Elements of Reusable Object-Oriented Software,* Addison-Wesley, Reading, MA (1995).
16. Paul Graham, *On Lisp,* Prentice Hall (1993). http://www.paulgraham.com/onlisp.html.
17. Hart, Timothy P., *MACRO Definitions for LISP,* Massachusetts Instute of Technology Artificial Intelligence Laboratory (October 22, 1963). ftp://publications.ai.mit.edu/ai-publications/pdf/AIM-057.pdf.
18. Henderson, Peter, *Functional Programming: Application and Implementation,* Prentice-Hall, Englewood Cliffs, New Jersey (1980).
19. Herrington, Jack, *Code Generation In Action,* Manning Publications Co., Greenwich, CT (2003).
20. Kernighan, Brian W. and Plauger, P. J., *Software Tools,* Addison-Wesley, Reading, MA (1976).
21. Kernighan, Brian W. and Pike, Rob, *The Unix Programming Environment* (1984).

22. Levine, John R., Mason, Tony, and Brown, Doug, *lex & yacc (2nd Edition),* O'Reilly & Associates, Inc., Sebastopol, CA (October 1992).

23. McCarthy, John, *Towards a Mathematical Science of Computation* (1962). http://www-formal.stanford.edu/jmc/towards.html.

24. McCarthy, John, Abraham, Paul W., Edwards, Daniel J., Hart, Timothy P., and Levin, Michael I., *Lisp 1.5 Programmer's manual,* MIT Press, Cambridge, Massachusetts (1962). http://www.softwarepreservation.org/projects/LISP/book/LISP%201.5%20Programmers%20Manual.pdf.

25. McCarthy, John, *Recursive Functions of Symbolic Expressions and Their Computation by Machine,* Massachusetts Instute of Technology Artificial Intelligence Laboratory (March 13, 1959). ftp://publications.ai.mit.edu/ai-publications/pdf/AIM-008.pdf.

26. Naur, Peter, *Concise Survey of Computer Methods,* Petrocelli Books (1974).

27. Norvig, Peter, *Paradigms of Artificial Intelligence Programming: Case Studies in Common Lisp,* Morgan Kaufmann Publishers, Inc., San Mateo, California (1992).

28. Christian Queinnec, *Lisp in Small Pieces,* Cambridge University Press, New York (1996).

29. Reynolds, John, "Definitional Interpreters for Higher-Order Programming Languages," *Higher-Order and Symbolic Computation,* 11, 4, pp. 363-397 (1998). http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.110.5892.

30. Siklossy, Laurent, *Let's Talk Lisp,* Prentice-Hall, Englewood Cliffs, New Jersey (1976).

31. Srinivasan, Sriram, *Advanced Perl Programming,* O'Reilly & Associates, Inc., Sebastopol, CA (1997).

32. Steele, Guy and Sussman, Gerald, *The Art Of The Interpreter or, The Modularity Complex,* Massachusetts Instute of Technology Artificial Intelligence Laboratory (May 1978). ftp://publications.ai.mit.edu/ai-publications/pdf/AIM-453.pdf.

33. Thomas, Dave, *Programming Ruby (2nd Edition),* The Pragmatic Bookshelf (2005).