Standard ML as a Meta-Programming Language

Samuel Kamin [*]
Computer Science Dept.
University of Illinois
Urbana, Illinois
s-kamin@uiuc.edu

September 20, 1996

Abstract

Meta-programming languages, or program generators, are languages whose programs produce programs in other languages. We show how Standard ML makes an excellent meta-programming language, by adding appropriate program-valued — by which we mean string-valued — operations for each domain. We do so by giving four examples of meta-programming languages: a top-down parser generator; a "geometric region server" language modelled on one developed at Yale; a version of the "Message Specification Language," developed at Oregon Graduate Institute; and a pretty-printing specification language. Embedding meta-programming languages in ML in this way is easy to do, and the result is a language that, unlike most meta-programming languages, is higher-order.

1 Introduction

It is a kind of folklore in the programming language community that

$$\text{programming language} = \lambda\text{-calculus} + \text{constants}$$

In this paper, we explore this claim by developing several languages by adding constants to Standard ML.

All of our examples are meta-programming languages, intended to aid in the production of C++ programs. The examples are:

- *Top-down parser generator.* This is a well known example of "combinator-style" programming [8]. The combinators used in developing a parser in a functional language can be redefined to generate a parser in C++.
- *Geometric region server.* The Haskell code presented by Carlson et al. [4] to solve this problem directly can be modified to produce C++ code that solves it. The metaprogram is remarkably similar to the original.

[*]Supported by NSF Contract CCR 93–03043

- *Message Specification Language.* This language was developed at Oregon Graduate Institute [3, 13] as an experiment in using domain-specific languages to promote code reuse in a specialized class of programs.
- *Pretty-printer.* Pretty-printing specifications are translated to C++ code to perform the pretty-printing.

This work is a contribution to the program laid out in the workshop report [9]: to aid in the design and implementation of special-purpose languages. The workshop's conclusions included the finding that these languages represent the greatest potential for application of research in programming languages. Our view is that many special-purpose languages are far more special than their purpose requires. They are often designed with just the right primitive operations and "first-order" syntax, but with an overarching language structure that is feeble and *ad hoc.*

Meta-programming languages are a useful class of languages, because they can be used immediately in on-going software projects without the need for the project to change its main programming language. They are also a good example of a class of languages that tend to lack structure. Languages like `yacc` and `MSL` can fairly be described as "zero-order" in the sense that no operations *on programs* are provided.

The contribution of this paper is to show that a functional language — we use Standard ML, but could as easily use Scheme or Haskell — can be used to develop a meta-programming language very simply. Moreover, the resulting language has all the language structure of ML, because it *is* ML. We illustrate how this language structure can be useful in meta-programming.

1.1 Related work

Meta-programming is a very common activity, but meta-programming languages do not always seem to be considered as full-fledged languages, with data types, control structures, and so on. Examples include Gelernter's program builders [1], and Waters's KBEmacs [14]. Each provides a fixed set of program-building operations, but there is no direct manipulation of programs, nor is it easy for users to define their own operations.

Work at OGI [3] (on which section 5 of this paper is based) and at ISI [2] is specifically aimed at the development of meta-programming languages. In both cases, the meta-program is translated to a very-high-level language, from which an imperative language program is extracted. Our approach is more direct: the meta-programming language provides operations that manipulate *programs*, not *specifications.* An important contribution of this paper is to show that such a non-abstract approach can yield quite satisfactory results in terms of usability of the resulting language. Furthermore, that language has powerful features inherited from its base language (ML), and the translation is not dependent upon the success of a compiler in compiling a very-high-level language down to imperative code.

Sheard and Nelson have specifically addressed the use of ML as a meta-language for itself [11]. Their concern is to guarantee the type correctness of generated ML programs.

In earlier work [5], we advocated the use of "higher-order macros," in the context of a tree editor, as a way to allow for a very flexible presentation of programs. In our Emacs "`glue-mode`" one could present a program with the inner loop at the top of the tree, providing macros that would unmangle the tree structure into a linear representation. This could be regarded as a generalized "literate programming" mode.

Paul Hudak and his associates, in [6, 7], emphasize the notion that a set of definitions in a functional language can be regarded as a new language. This is our basic position as well; we are simply applying it in the domain of meta-programming.

2 A note on the meta-language

Our meta-language is Standard ML. We have chosen to represent syntax fragments in the object language as strings; though abstract syntax trees would offer greater power and efficiency, for this proof-of-concept demonstration, strings are adequate.

We use the Standard ML of New Jersey dialect, with no additions. However, we use one feature of that implementation that is not widely known, namely the "anti-quotation" feature [12, page BASE–20]. An explanation of that feature will aid the reader in understanding the code we present later.

Suppose we want to produce statements of the form `x = x+1;`, where *x* is an integer variable to be named later. We can write the ML function

```
bump varname = varname ^ " = " ^ varname ^ "+1;" ;
```

using the ML string concatenation operator "^". Then, `bump "i"` returns the string `i = i+1;`, and `bump "A[0]"` returns the string `A[0] = A[0]+1;`.

Standard ML's anti-quotation features allows us to write the same function like this:

```
bump varname = %`^varname = ^varname+1;` ;
```

The `` %` `` acts as an opening double quote and the ending `` ` `` as a closing double quote. However, within these quotes, items beginning with the caret (^) character are "anti-quoted," that is, evaluated (to strings, otherwise there is a type error) and their values spliced into the string.[1]

The item after the caret can be any ML expression that evaluates to a string. Thus, `bumpAndPrint` can be defined in either of these two ways:

```
bumpAndPrint varname = (bump varname) ^ "cout << " ^ varname ^ ";" ;

bumpAndPrint varname = %`^(bump varname) cout << ^varname;` ;
```

The anti-quotation notation can be nested. If `f` is a string-valued function with a string argument, we may write

```
%`... ^(f %`...`) ...`
```

The argument to `f`—which may, of course, contain anti-quoted expressions—is evaluated to a string, then passed to `f`, and then the value of the call to `f` is spliced into the result.

[1]The anti-quotation mechanism is actually more general than what we are describing here; consult the manual for a fuller description.

3 Top-down parser

Top-down parsing is a well-known example of combinator-style programming in functional languages. It is also a simple example of metaprogramming.

In the usual presentation of top-down parsing in a functional language, one starts by defining the type of “parser”:

```
datatype 'a Option = Yes of 'a | No;
type Parser = token list -> (token list) Option;
```

A parser looks at a token list and either matches a prefix of it, returning the unmatched part, or fails to do so. Given this definition, we can define the parser associated with a single token, and combinators to combine parsers by sequence or alternation:

```
term: token -> Parser
  ++: Parser * Parser -> Parser
  ||: Parser * Parser -> Parser
```

Now, simply writing down a grammar, with a few syntactic decorations, yields a parser. For example, the grammar

```
A ::= aBB | B
B ::= b | cA
```

is programmed by writing

```
val rec A = (term "a") ++ B ++ B || B
    and B = (term "b") || (term "c") ++ A
```

(This definition does not work in ML, due to eager evaluation, but it would work in Haskell. In ML, eta-abstraction must be applied, but otherwise the definition is the same.)

However, the disadvantage of this approach is that it only gives us a parser in the functional language. This is fine if that is our main language, but if we are doing our programming in another language — say, a more conventional language like C++ — it will be difficult to overcome the lack of interoperability between the two languages. For a programmer seeking to develop a special-purpose parser-generator language to be incorporated in an on-going software-development project, this is likely to be a fatal objection.

It turns out that combinators above can be defined in such a way that combinator expressions produce C++ programs. A “parser” is a C++ function. A “right-hand side” is, basically, the body of a parser: it is a piece of C++ code that attempts to match the right-hand side of a grammar rule and, if it fails, jumps to a given label. Since a non-terminal can have several rules, we cannot choose one label but must instead supply the label when the piece of code is included in a parsing function. Thus

```
type Parser = CFunction;

type RHS = Label -> CCommand
```

where CFunction, Label, and CCommand (as well as Token, Label, and Nonterm, used below) are synonyms for string.

A right-hand side corresponding to a terminal compares the next character to that terminal and, if there is no match, jumps to the label; if there is a match, the token pointer is incremented. (For our parser, we assume the input is contained in an array, and the "token pointer" is a global integer variable called current.)

```
(* term: Token -> RHS *)
fun term (t:Token) = fn (lab:Label) =>
   %`if (tokens[current] == '^(t)')
       current++;
     else
       goto ^lab;` ;
```

The sequencing combinator takes two right-hand sides and attempts to match one right after the other. If either one fails, they will jump to the *same* failure label.

```
(* ++: RHS -> RHS -> RHS *)
infix 3 ++;
fun ((rhs1:RHS) ++ (rhs2:RHS))  = fn (lab:Label) =>
  %`^(rhs1 lab)
    ^(rhs2 lab)` ;
```

The alternation combinator combines two right-hands sides in such a way that if the first fails, it will jump to the second; if the second fails, then the right-hand side as a whole fails. Thus, we need to generate an intermediate label, for which we define a function genLabel.

```
val label_counter = ref 0;
fun genLabel () =
  let val l = makestring(!label_counter)
  in label_counter := !label_counter + 1;
     "L"^l
  end;

(* ||: RHS -> RHS -> RHS *)
infix 2 ||;
fun ((rhs1:RHS) || (rhs2:RHS))  = fn (lab:Label) =>
  let val l = genLabel()
  in %`^(rhs1 l)
       return true;
    ^l: current = pos;
       ^(rhs2 lab)`
  end;
```

We can't quite generate a parser yet, because right-hand sides need to be tied to non-terminals, and non-terminals need to be turned into right-hand sides. This is one major difference from the functional language parser described above: a parser cannot simply be defined recursively as being equal to its right-hand side.

We will instead assume that each non-terminal A has an associated parsing function `parse`A, and non-terminals will be coded as a call to that function:

```
(* nonterm: Variable -> RHS *)
fun nonterm (v:Variable) = fn (lab:Label) =>
  %`if (!parse^v()) goto ^lab;` ;
```

and non-terminals will be tied to right-hand sides by the function that generates that parsing function:

```
(* ::= : Variable -> RHS -> CFunction *)
infix 1 ::= ;
fun (v:Variable) ::= (rhs:RHS) =
   let val lab = %`L1000`
   in %`int parse^v () {
        int pos = current;
        ^(rhs lab)
        return true;
     ^lab:
        current = pos;
        return false;
   }
`
   end;
```

Thus, each rule produces a parsing function. The rules above are written slightly differently:

```
"A" ::= (term "a") ++ (nonterm "B") ++ (nonterm "B")
     || (nonterm "B") ;

"B" ::= (term "b")
     || (term "c") ++ (nonterm "A") ;
```

Each produces a C++ function:[2]

```
int parseA () {
     int pos = current;
     if (tokens[current] == 'a')
         current++;
     else
         goto L0;
     if (!parseB()) goto L0;
```

[2]As in the other examples in this paper of code produced by our meta-programming languages, we have taken the liberty of "beautifying" it for better readability, to the extent of changing indentation and line breaks.

```
     if (!parseB()) goto L0;
         return true;
 L0:
     current = pos;
     if (!parseB()) goto L1000;
         return true;
 L1000:
     current = pos;
     return false;
}

int parseB () {
     int pos = current;
     if (tokens[current] == 'b')
         current++;
     else
         goto L1;
     return true;
 L1:
     current = pos;
     if (tokens[current] == 'c')
         current++;
     else
         goto L1000;
     if (!parseA())
         goto L1000;
     return true;
 L1000:
     current = pos;
     return false;
}
```

We gain many of the same benefits as we did in the non-meta-programming parser. For example, the function that produces a collection of parsing rules by mapping over a list of binary operators applies with very little change:

```
   "Exp" ::= fold (op ||)
             (map (fn bop => (nonterm "Exp") ++ (term bop)
                                             ++ (nonterm "Exp"))
                  ["+", "-", "*", "/"])
             (fn lab => %`return false;`);
```

4 Geometric region server

Carlson, Hudak, and Jones [4] report an experiment in which a Haskell program was developed as a prototype "geometric region server" for Naval command and control. A geometric region server is a program to determine whether certain objects in a battle scenario fall within regions which make them vulnerable to certain weapons, make weapons unsafe to fire, etc. For purposes of this prototype, each object is represented as a point; the problem, then, is to determine whether some given points fall within some given geometric regions.

The notion of "point" is defined in the Haskell solution in a straightforward way, as an abstract data type. Addition and subtraction of points is also defined; we use infix symbols `++` and `--`.

The heart of the solution is the definition of a kind of language for specifying regions. We quote parts of the solution here, translated to ML.

Since the only question being asked about a region is whether or not a point falls inside it, the natural definition of a region is as a function:

```
type Region = Point -> Bool;

infix inRegion;
fun (p inRegion r) = r p;
```

We can now define a collection of operations to construct and combine regions. We give a sampling of the dozen or so such functions defined in [4].

- Circular region with given radius:

```
    (* circle: real -> Region *)
    fun circle r = fn p => sqrdist p < r*r;
```

- The halfplane to the left of the line joining two points:

```
    (* halfplane: Point -> Point -> Region *)
    fun halfplane a b =
       let fun zcross (Pt(x,y)) (Pt(u,v)) = x*v - y*u
       in fn p => (zcross (a -- p) (b -- a)) > 0.0
       end;
```

- Intersections and unions:

```
    (* /\: Region -> Region -> Region *)
    infix /\;
    fun (r1 /\ r2) =
       fn p => (r1 p) andalso (r2 p);

    (* \/: Region -> Region -> Region *)
    infix \/;
    fun (r1 \/ r2) =
       fn p => (r1 p) orelse (r2 p);
```

- Translation by a point:

```
    (* at: Region -> Point -> Region *)
    infix at;
    fun r at p0 = fn p => (r (p -- p0));
```

- Convex polygons:

```
    (* convexPoly: Point list -> Region *)
    fun convexPoly (p :: ps) =
        intersect (zipWith halfplane ([p] @ ps) (ps @ [p]));
```

We can define the same functions, mostly in very similar ways, to create a meta-programming language of regions, which produces C++ code.

We assume there is a C++ class `Point`, so that we can define `CPoint` to be a C++ expression (types `CPoint`, `CExpr`, and `CPred` are all synonyms for `string`), and a region to be a function from `CPoint` to `CPred`. That is, given an expression denoting a point in C++, a region produces a predicate in C++ which tests whether the given point is in the region.

```
type CPoint = CExpr;
type Region = CPoint -> CPred;
```

There are a number of utility functions on points which we won't show here, such as the definitions of `++` and `--`. Here are the definitions of the region operations mentioned above:

```
(* circle: CFloat -> Region *)
fun circle f = fn p => %`(^(sqrdist p) < ^f*^f)`;

(* halfplane: CPoint -> CPoint -> Region *)
fun halfplane a b =
   let fun zcross e1 e2 =
                %`(^(fst e1)*^(snd e2) - ^(snd e1)*^(fst e2))`
   in fn p => %`(^(zcross (a -- p) (b -- a)) > 0.0)`
   end;

(* /\: Region -> Region -> Region *)
infix /\;
fun (r1 /\ r2) = fn p => %`(^(r1 p) && ^(r2 p))`;

(* \/: Region -> Region -> Region *)
infix \/;
fun (r1 \/ r2) = fn p => %`(^(r1 p) || ^(r2 p))`;

(* at: Region -> CPoint -> Region *)
infix at;
fun (r at p0) = fn p => (r (p -- p0));
```

```
(* convexPoly: CPoint list -> Region *)
fun convexPoly (p :: ps) =
    intersect (zipWith halfplane ([p] @ ps) (ps @ [p]));
```

In many cases, the meta-programming versions of the functions are very similar to their Haskell version; note in particular the definition of convexPoly, which is exactly the same, though it produces a very different result.

As an example of these operations, one of the regions defined in [4] is the "tight zone":

```
(convexPoly [Pt(0.0,5.0), Pt(118.0,32.0),
             Pt(118.0,62.0), Pt(0.0,25.0)])
 \/
(convexPoly [Pt(118.0,32.0), Pt(259.0,5.0),
             Pt(259.0, 25.0), Pt(118.0,62.0)])
```

It produces a region which, when applied to the C++ expression p, produces this C++ predicate:

```
(((((0.0-p.x)* ( 32.0- 5.0) -  ( 5.0-p.y)*(118.0-0.0)) > 0.0)
 && ((((118.0-p.x)* ( 62.0- 32.0) -  ( 32.0-p.y)*(118.0-118.0)) > 0.0)
 && ((((118.0-p.x)* ( 25.0- 62.0) -  ( 62.0-p.y)*(0.0-118.0)) > 0.0)
 && ((((0.0-p.x)* ( 5.0- 25.0) -  ( 25.0-p.y)*(0.0-0.0)) > 0.0)
 && 1))))
 ||
 ((((118.0-p.x)* ( 5.0- 32.0) -  ( 32.0-p.y)*(259.0-118.0)) > 0.0)
 && ((((259.0-p.x)* ( 25.0- 5.0) -  ( 5.0-p.y)*(259.0-259.0)) > 0.0)
 && ((((259.0-p.x)* ( 62.0- 25.0) -  ( 25.0-p.y)*(118.0-259.0)) > 0.0)
 && ((((118.0-p.x)* ( 32.0- 62.0) -  ( 62.0-p.y)*(118.0-118.0)) > 0.0)
 && 1)))))
```

5 MSL-in-SML

The Message Specification Language, or MSL, was designed as an approach to engineering reusable code in the domain of "message translation and validation." This is a domain of programming which has been studied by software engineers [3, 10] precisely as a model area for the study of software re-use. The problem is to translate and validate incoming "messages"—bit streams—from a variety of sources in a variety of formats. The formats of these messages are described in semi-formal "interface control documents." An example is given in Figure 2 of [3] and reproduced here as Figure 1. The translation process involves pulling bits off the incoming stream and storing appropriate values in records; validation is simply checking that the messages have the required form. In past work, the language in which translation and validation has been performed is ADA. In this paper, we use C++.

The problem is that there are many messages, and the various message formats are described only informally, as shown in Figure 1. As a result, many programmers have written similar programs for these translations, resulting in wasted time and needless bugs. For a software engineer, the question is how to take advantage of the commonality of these messages to obtain

No.	Field Name	Size	Range	Amplifying Data
1	Course	3	001–360	In degrees.
			000	No value reported.
	Field Separator	1	/	Slash.
2	Speed	4	0000–5110	In knots.
	Field Separator	1	/	Slash.
3	Altitude or Track Confidence	0 or 2	01–99	In thousands of feet.
			HH	High Confidence.
			MM	Medium Confidence.
			LL	Low Confidence.
			NN	No Confidence.
			Blank	No altitude value reported or altitude less than 1000 feet.
	Field Separator	1	/	Slash.
4	Time			Time Group
		2	00–23	Hour
		2	00–59	Minute
	End of line	1	CR	Carriage return.

Figure 1: Reproduction of Figure 2 from [3] — Sample Interface Control Document

reusable code. The approach taken by the group at OGI [3]—which is the point of departure for the current work—is to define the MSL language, in which message formats can be described *formally.* MSL specifications can be translated automatically to ADA code. All that remains to be done by hand is to translate the interface control documents into MSL.

Figure 3 in [3], which we reproduce in Figure 2, is the MSL specification for the interface control document in Figure 1. This description has two parts. The *type declarations* are an abstraction of ADA record definitions, describing the "logical" structure of this type of message. The *action declarations* describe how the incoming bit stream should be "parsed," so as to read the appropriate portions of the bit stream, translate them to appropriate values, and store them into appropriate fields in the logical structure.

For our demonstration, we will show that with an appropriate set of functions defined in ML, the action declarations of this MSL specification can largely be reproduced intact. The resulting ML expression will denote a string which, interpreted as a C++ program, reads data from a bit stream and writes it into a structure. In effect, we are defining a new language simply by defining ML functions. The rest of this section is a description of the ML code.

5.1 MSL-in-SML and its definition

The following discussion is an explanation of three figures given in this paper. The "language definition" is given by the SML definitions shown in Figure 3; this is the code that most requires explanation. Given these definitions, our version of the MSL message specification of Figure 2 is the set of SML definitions in Figure 4. Note that this corresponds only to the action declarations

```
(* Type declarations *)
type Confidence_type = [High, Medium, Low, No];

type Alt_or_TC_type = [Altitude: integer(1..99),
                       Track_confidence: Confidence_type,
                       No_value_or_Alt_less_than_1000];

type Time_type = {Hour: integer(0..23),
                  Minute: integer(0..59)};

message_type MType = {Course: integer(0..360),
                      Speed: integer(0..5110),
                      Alt_or_TC: Alt_or_TC_type,
                      Time: Time_type};

(* Action declarations *)
EXRaction to_Confidence = [High: Asc 2 | "HH",
                           Medium: Asc 2 | "MM",
                           Low: Asc 2 | "LL",
                           No: Asc 2 | "NN"];

EXRaction to_Alt_or_TC = [Altitude: Asc2int 2,
                          Track_confidence: to_Confidence,
                          No_value_or_Alt_less_than_1000: Skip 0
                         ] @ Delim "/"; (* field separator "/" *)

EXRaction to_Time = [Hour: Asc2Int 2,
                     Minute: Asc2Int 2
                    ] @ Delim "\n"; (* CR as field separator *)

EXRmessage_action to_MType = {Course: Asc2Int 3 @ Delim "/",
                              Speed: Asc2Int 4 @ Delim "/",
                              Alt_or_TC: to_Alt_or_TC,
                              Time: to_Time};
```

Figure 2: Reproduction of Figure 3 from [3]

in the MSL specification, except that the subranges in the type declarations, which are needed for message validation, are incorporated. MSL-in-SML has no type declaration part. Given the definitions in Figures 3 and 4, the expression

```
 print (MType (newbitsource "A" "bit") (recordptr "target") "");
```

evaluates to the string shown in Figure 5.[3] (That string has been "beautified" to the extent of changing the indentations of lines and changing some line breaks.)

Our main point in this paper, and the key to understanding Figure 3, is that defining a new language is largely a matter of defining the "primitive" values in the new language as values of appropriate type in the host language. Here, the most important type of value is a `Message`, which we define as follows:

```
type Message = bitsource -> recordfield -> statement -> statement;
```

A message—this includes both an entire message and any component of a message—is a function. Its arguments are an incoming bit stream, a field within a record, and a statement to be executed in the event that the message is ill-formed. Its result is a statement that extracts the appropriate number of bits from the bit stream (updating a pointer into the bit stream) and stores the appropriate value in the given record field, assuming the input stream is well-formed; if not, it executes the given alternative statement.

We have, as discussed earlier, taken the view that C++ expressions and statements are simply strings. It would be both more efficient and, potentially, more powerful to define a type of C++ abstract syntax trees, but it is easier, and adequate for our current purposes, to use strings.

The code in Figure 3—that is, the MSL-in-SML language—has built-in assumptions about how the various operations are to be implemented in C++. It assumes that the incoming bit stream is represented as an array of bytes together with an integer variable; the latter contains the index of the next location in the bit stream to be translated. However, that index is given as if the array of bytes were actually an array of bits; thus, to get at a specific byte, this index must be divided by 8. (An alternative approach would be to keep two indices, a byte index and a bit index, the latter always being between 0 and 7.) The names of the byte array and the index comprise a value of the type `bitsource`. Operations on `bitsource` values include dereferencing (obtaining the current byte) and incrementing (adding 8 to the index). Although the index represents a specific bit in the bit stream, bit-accessing operations are not needed in this example, so are not included in the `bitsource` data type.

A `recordfield` consists of a record pointer and a list of nested field names. The record into which a message is to be stored consists of a set of fields, which may in turn contain records, hence the nested field structure.

The code consists of a number of functions that produce `Messages`. Among these, `asc2int`, `asc`, `skip`, and `delim` are "primitive"—not constructed from other messages—while `seq`, `alt`, and `infield` construct new message from existing messages. We will describe only `delim`, `asc2int`, `seq`, and `alt`.

Given an expression e denoting a character in the C++ programming (usually a character constant, though it could be a variable), `delim` e produce C++ code to check if the current byte

[3]There is no corresponding figure given in [3] with which to compare our Figure 5.

```
System.Control.quotation := true;
val % = implode o (map (fn QUOTE x => x|ANTIQUOTE x=>x));

type expr = string;
type predicate = expr;
type statement = string;
type fieldname = string;

abstype bitsource =
   source of expr * expr  (* byte array and index, in bits *)
with

   fun (* newbitsource: expr -> expr -> bitsource *)
       newbitsource a i = source(a, i);

   fun (* init: bitsource -> statement *)
       init (source (_, i)) = %`^i = 0;`;

   fun (* getByte: bitsource -> expression *)
       getByte (source (a, i)) = %`^a[^i / 8]`;

   fun (* getNthByte: bitsource -> int -> expression *)
       getNthByte (source (a, i)) (n:int) =
          if (n = 0)
          then %`^a[^i / 8]`
          else %`^a[^i / 8 + ^(makestring n)]`;

   fun (* advanceByte: bitsource -> statement *)
       advanceByte (source (a, i)) = %`^i = ^i - (^i % 8) + 8;`;

   fun (* advanceNBytes: bitsource -> int -> statement *)
       advanceNBytes (source (a, i)) (n:int) =
          if (n = 0)
          then %``
          else %`^i = ^i - (^i % 8) + (8 * ^(makestring n));`;
end;

abstype recordfield =
   field of expr * (fieldname list)  (* record ptr and fields *)
with
   fun (* recordptr: expr -> recordfield *)
       recordptr e = field(e, []);

   fun (* subfield: recordfield -> fieldname -> recordfield *)
       subfield (field(e, fl)) f = field(e, f::fl);

   fun (* deref: recordfield -> expr *)
       deref (field(e, fl)) =
          %`(*^e)^(implode (map (fn f => "."^f) (rev fl)))`
end;
```

Figure 3: The MSL-in-SML language definition (part 1)

```
type Message = bitsource -> recordfield -> statement -> statement;

fun (* infield: fieldname -> Message -> Message *)
    infield f m (src : bitsource) (tgt : recordfield)
           = m src (subfield tgt f);

fun (* C_if : expression * statement * statement -> statement *)
    C_if (e, s1, s2) =
       if (e = "1" orelse e = "(1)")
       then s1
       else %`if (^e) {
             ^s1
            } ^(if (s2 <> "") then %`else {
            ^s2
            }` else %``)`;

fun (* seq: Message list -> Message *)
    seq (m::ml) (src : bitsource) (tgt : recordfield) S =
         %`^(m src tgt (%`error_action();`)) ^(seq ml src tgt S)`
  | seq [] src tgt S = %``;

fun (* asc2int : int -> (int * int) -> Message *)
    asc2int (w:int) (lo:int, hi:int)
            (src : bitsource) (tgt : recordfield) S =
  let fun ms (n:int) = makestring n
  in C_if(%`inrange(^(getByte src), ^(ms w), ^(ms lo), ^(ms hi))`,
          %`^(deref tgt) = getint(^(getByte src), ^(ms w));
            ^(advanceNBytes src w)`,
          S)
  end;

fun (* alt: Message list -> Message *)
    alt (m::ml) (src : bitsource) (tgt : recordfield) S =
       m src tgt (alt ml src tgt S)
  | alt [] src tgt S = S
;

fun (* delim: expr -> Message *)
    delim e (src : bitsource) (tgt : recordfield) S =
           %`if (^(getByte src) == '^(e)')
             ^(advanceByte src)
 else ^S` ;
```

Figure 3: The MSL-in-SML language definition (part 2)

```
fun zip [] [] = []
  | zip (a::x) (b::y) = (a,b) :: (zip x y);

fun range (i:int) (j:int) =
  if (i > j) then [] else (i :: (range (i+1) j));

fun C_and [] = %``
  | C_and [pred] = %`(^pred)`
  | C_and (pred1::pred2::preds) = %`(^pred1) && ^(C_and (pred2::preds))`;

fun (* asc: string -> string -> Message *)
    asc chars value (src : bitsource) (tgt : recordfield) S =
      C_if(C_and (map (fn (i, c) => %`^(getNthByte src i) == '^(c)'`)
                  (zip (range 0 ((size chars)-1)) (explode chars))),
           %`^(deref tgt) = ^value;`,
           S);

fun (* skip : int -> Message *)
    skip n (src : bitsource) (tgt : recordfield) S =
      %`^(deref tgt) = 1;
        ^(advanceNBytes src n)` ;
```

Figure 3: The MSL-in-SML language definition (part 3)

in the bit source equals e; if so, it increments the bit source (without storing any value into the record field); if not, it invokes the alternative statement.

```
fun (* delim: expr -> Message *)
    delim e (src : bitsource) (tgt : recordfield) S =
           %`if (^(getByte src) == '^(e)')
              ^(advanceByte src)
              else ^S` ;
```

Thus, if the bit source `bs` is the array `A` and index `bit`, and if the record field is called `f`, then the expression

```
delim "/" bs f "abort();";
```

has as its value the string

```
if (A[bit / 8] == '/')
  bit = bit - (bit % 8) + 8;
else abort();
```

The function `asc2int` takes an integer, giving the length of the field in the incoming message, in bytes, and a pair of integers, giving the minimum and maximum numerical values of the field, and produces a Message. We have assumed the existence of C++ functions `inrange`, which checks the validity of the incoming message, and `getint`, which returns the integer value of the message.

```
val to_Confidence =
   alt[asc "HH" "High",
       asc "MM" "Medium",
       asc "LL" "Low",
       asc "NN" "None"
      ];

val to_Alt_or_TC =
   alt[infield "Altitude" (asc2int 2 (1, 99)),
       infield "Track_confidence" to_Confidence,
       infield "No_Value_or_Alt_less_than_1000" (skip 0)];

val to_Time =
   seq[infield "Hour" (asc2int 2 (0, 23)),
       infield "Minute" (asc2int 2 (0, 59))
      ];

val MType =
   seq[infield "Course" (asc2int 3 (0, 360)),
       delim "/",
       infield "Speed" (asc2int 4 (0, 5110)),
       delim "/",
       infield "Alt_or_TC" to_Alt_or_TC,
       infield "Time" to_Time
      ];
```

Figure 4: The MSL-in-SML specification corresponding to the MSL specification in Figure 2

```
if (inrange(A[bit / 8], 3, 0, 360)) {
  (*target).Course = getint(A[bit / 8], 3);
  bit = bit - (bit % 8) + (8 * 3);
} else { error_action(); }
if (A[bit / 8] == '/')
  bit = bit - (bit % 8) + 8;
else error_action();
if (inrange(A[bit / 8], 4, 0, 5110)) {
  (*target).Speed = getint(A[bit / 8], 4);
  bit = bit - (bit % 8) + (8 * 4);
} else { error_action(); }
if (A[bit / 8] == '/')
  bit = bit - (bit % 8) + 8;
else error_action();
if (inrange(A[bit / 8], 2, 1, 99)) {
  (*target).Alt_or_TC.Altitude = getint(A[bit / 8], 2);
  bit = bit - (bit % 8) + (8 * 2);
} else {
  if ((A[bit / 8] == 'H') && (A[bit / 8 + 1] == 'H')) {
    (*target).Alt_or_TC.Track_confidence = High;
  } else {
    if ((A[bit / 8] == 'M') && (A[bit / 8 + 1] == 'M')) {
      (*target).Alt_or_TC.Track_confidence = Medium;
    } else {
      if ((A[bit / 8] == 'L') && (A[bit / 8 + 1] == 'L')) {
        (*target).Alt_or_TC.Track_confidence = Low;
      } else {
        if ((A[bit / 8] == 'N') && (A[bit / 8 + 1] == 'N')) {
          (*target).Alt_or_TC.Track_confidence = None;
        } else {
          (*target).Alt_or_TC.No_Value_or_Alt_less_than_1000 = 1;
        }
      }
    }
  }
}
if (inrange(A[bit / 8], 2, 0, 23)) {
  (*target).Time.Hour = getint(A[bit / 8], 2);
  bit = bit - (bit % 8) + (8 * 2);
} else { error_action(); }
if (inrange(A[bit / 8], 2, 0, 59)) {
  (*target).Time.Minute = getint(A[bit / 8], 2);
  bit = bit - (bit % 8) + (8 * 2);
} else { error_action(); }
```

Figure 5: Code produced by evaluating MSL-in-SML specification in Figure 4

```
fun (* asc2int : int -> (int * int) -> Message *)
    asc2int (w:int) (lo:int, hi:int)
            (src : bitsource) (tgt : recordfield) S =
  let val ms = makestring
  in C_if(%`inrange(^(getByte src), ^(ms w), ^(ms lo), ^(ms hi))`,
          %`^(deref tgt) = getint(^(getByte src), ^(ms w));
            ^(advanceNBytes src w)`,
          S)
  end;
```

For example, given bs as above, and given that the field f is the field Hour in the record p, the call

```
asc2int 2 (0, 23) bs f "abort();";
```

produces the statement

```
if (inrange(A[bit / 8], 2, 0, 23)) {
  (*p).Hour = getint(A[bit / 8], 2);
  bit = bit - (bit % 8) + (8 * 2);
} else {
  abort();
}
```

The most interesting functions are the "combinators" seq and alt. seq is used to extract a sequence of message components from the bit source, all of which must be present, and store them (except delimiters) into the fields of a structure. Specifically, its argument is a list of Messages and its result is a Message. Since each component of the message must be present, the alternative action for each message is "error." Since the entire message can fail only if one of its components fails—and since backtracking is not permitted—the message as a whole can just ignore its alternative action.

```
fun (* seq: Message list -> Message *)
    seq (m::ml) (src : bitsource) (tgt : recordfield) S =
         %`^(m src tgt (%`error_action();`)) ^(seq ml src tgt S)`
  | seq [] src tgt S = %``;
```

alt is used when any one of a list of messages can be found in the bit source. If none of them is found, then alternative action must be taken.[4]

```
fun (* alt: Message list -> Message *)
    alt (m::ml) (src : bitsource) (tgt : recordfield) S =
       m src tgt (alt ml src tgt S)
  | alt [] src tgt S = S ;
```

[4]The treatment of alternatives needs work. They should correspond to variant records, or *tagged* unions. In this version of MSL-in-SML, they are treated as untagged unions.

We have attempted to show that much of the value of MSL, as defined in [3], can be obtained directly in a functional language like SML. There remains some syntactic awkwardness, but in general the language illustrated in Figure 4 (MSL-in-SML) is not very much different from the one illustrated in Figure 2 (MSL).

As compared to that of [3], this approach has several advantages:

- It is much simpler to produce a new language this way. The MSL language of Figure 2 is specified with the usual context-free syntax, and sophisticated language-development tools.
- The language designer has complete control over the code that is generated. The languages designed using the methods of the OGI group depend upon a sophisticated optimizing compiler to produce an acceptable program in the target language.
- The languages we produce preserves the underlying language, namely Standard ML. The sophisticated client could extend the language or use the underlying features of SML.

To elaborate on this last point, suppose a new type of message is to be constructed as follows: It consists of a sequence of ASCII digits, of fixed length, which are to be summed before being stored. Since processing this message involves repetition, there is no way to do so directly using the message-forming operations we have provided. However, we have all the power of ML at our disposal:

```
fun digitsequence n bs f S =
   let val L1 = newlabel()
       and L2 = newlabel()
   in
      let fun aux 0 = %``
            | aux n = %`if (isdigit(^(getByte bs))) {
                        ^(deref f) += ^(getByte bs) - '0';
                        ^(advanceByte bs)
                        ^(aux (n-1))}
                        else goto ^L1;`
      in %`^(deref f) = 0; ^(aux n) goto ^L2; ^L1: ^S ^L2:`
      end
   end;
```

Then `digitsequence 2 bs f (%`error();`)` is

```
(*p).Hour = 0;
if (isdigit(A[bit / 8])) {
  (*p).Hour += A[bit / 8] - '0';
  bit = bit - (bit % 8) + 8;
  if (isdigit(A[bit / 8])) {
    (*p).Hour += A[bit / 8] - '0';
    bit = bit - (bit % 8) + 8;
  }
  else goto L11;
}
else goto L11;
goto L12;
L11: error();
L12:
```

The point is that this definition use the features of ML itself—local definitions, recursion—which are not available in MSL.

6 Pretty printer

The pretty-printing languages allows the programmer to say how programs should be formatted, using rules of the form

pattern ==> *layout*

where the *pattern* names a language construct and the layout says how to display that type of construct. An example is this rule to format `if-then-else` statements:

```
IF _c _S1 _S2 ==>
      emit "if (" oo !c oo emit ")" oo
          indent (newline oo !S1) oo newline oo
          emit "else" oo
          indent (newline oo !S2)
```

(For this language, we have taken the liberty of employing a simple preprocessing step: a Perl script changes occurrences of _*name* to "*name*" and occurrences of !*name* to (pp "*name*").)

The oo operator should be read as "followed by." This rule says: print the word `if`, then the condition in parentheses, skip to a newline to print `S1` (indented), print `else` on the following line and finally print `S2` (indented).

Each rule

pattern ==> *layout*

as a whole generates a C++ `if` statement that checks if an (abstract) syntax tree node has the type indicated in the pattern and, if so, prints it according to the layout. More accurately, the rule generates a function that, given the name of the syntax tree node and an "environment," generates the C++ `if` statement. The *pattern* can contain variables (`c`, `S1`, and `S2` in the `IF` rule above) which are to denote subtrees of the syntax tree node being pretty-printed. These are (recursively) pretty-printed in the *layout*.

Thus, the overall types for patterns and layouts are:

```
type Pattern = CNode -> Env -> (CPred * Env)
 and Layout = CNode -> Env -> CCommand;
```

We have failed to explain one thing: why does a pattern take an environment argument? This is because rules can be nested, with an entire rule occurring in the layout portion of an enclosing rule. The nested rule is permitted to see the names defined in the enclosing rule; hence, the environment argument.

For our example, we use a simple structure for abstract syntax trees:

```
enum STMTTYPE {ASSIGN, BEGIN, IF, WHILE} ;

struct STMTLIST {
  STMT *hd;
  STMTLIST *tl;
};

typedef int EXPR;

struct STMT {
  STMTTYPE stype;
  EXPR lhs;
  EXPR rhs;
  STMTLIST *stmts;
};
```

(In assignment statements, the expressions `lhs` and `rhs` have the obvious meanings; for `IF` and `WHILE` statements, `lhs` is used to hold the condition and `rhs` is unused.)

Now suppose we apply the `IF` rule given above to C++ name `nd` and the empty environment $\{\}$. The pattern produces the pair

```
(nd->stype == IF, { "cond" → "nd->lhs",
                    "S1" → "nd->stmts->hd",
                    "S2" → "nd->stmts->tl->hd"})
```

The `==>` operator applies the pattern and then passes the node and environment to the layout, producing the statement

```
if (nd->stype == IF) {
   cout << "if (";
   ppEXPR(nd->lhs);
   cout << ")";
   indent += 2; cout << endl; skipspaces(indent);
   ppSTMT(nd->stmts->hd, indent); indent -= 2;
   cout << endl; skipspaces(indent);
   cout << "else";
   indent += 2; cout << endl; skipspaces(indent);
   ppSTMT(nd->stmts->tl->hd, indent); indent -= 2;
}
```

```
"STMT" :::
    ASSIGN _v _e ==> !v oo emit " := " oo !e oo emit ";"
 || BEGIN _stmts ==> !stmts
 || IF _c _S1 _S2 ==>
       emit "if (" oo !c oo emit ")" oo
           indent (newline oo !S1) oo newline oo
           emit "else" oo
           indent (newline oo !S2)
 || WHILE _c _S ==>
       emit "while (" oo !c oo emit ")" oo
         using "S" (BEGIN "stmts" ==>
                      newline oo emit "{" oo
                      indent (newline oo !stmts) oo
                      newline oo emit "}"
                    ||
                      indent (newline oo !S)
                   ) ;

"STMTLIST" :::
   STMTLIST _h _t ==>
      (empty ==> skip
       || using _t
            (  empty ==> !h
            || !h oo newline oo !t)
       ) ;
```

Figure 6: Sample pretty-printing rules

There is an operator `:::` not shown in the above example that generates function definitions. *name*`:::`*rule* generates the definition of function pp*name*, with its body given by *rule*. Each pretty-printing function generated in this way has two arguments: the syntax tree node `nd` and the current indent.

Figure 6 gives a set of rules for statements and statement lists, which translate to the C++ functions shown in Figure 7.

The complete listing of the language, as it applies to statements and statement lists, is given in Figure 8. We explain selected functions.

As described above, the operator `==>` applies its pattern argument to a node n and an environment ρ, then passes n and the resulting environment ρ' (combined with ρ) to its layout argument.

```
(* ==>: Pattern * Layout -> Layout *)
fun ((pat:Pattern) ==> (lo:Layout)) (nd:CNode) (rho:Env) =
     let val (pred, rho1) = pat nd rho
     in C_if(pred,
void ppSTMT (STMT *nd, int indent)
{
   if (nd->stype == ASSIGN) {
      ppEXPR(nd->lhs);
      cout << " := ";
      ppEXPR(nd->rhs);
      cout << ";";
   }
   else {
      if (nd->stype == BEGIN) {
         ppSTMTLIST(nd->stmts, indent);
      }
      else {
         if (nd->stype == IF) {
            cout << "if (";
            ppEXPR(nd->lhs);
            cout << ")";
            indent += 2; cout << endl; skipspaces(indent);
            ppSTMT(nd->stmts->hd, indent); indent -= 2;
            cout << endl; skipspaces(indent);
            cout << "else";
            indent += 2; cout << endl; skipspaces(indent);
            ppSTMT(nd->stmts->tl->hd, indent); indent -= 2;
         }
         else {
            if (nd->stype == WHILE) {
               cout << "while (";
               ppEXPR(nd->lhs);
               cout << ")";
               if (nd->stmts->hd->stype == BEGIN) {
                  cout << endl; skipspaces(indent);
                  cout << "{";
                  indent += 2; cout << endl; skipspaces(indent);
                  ppSTMTLIST(nd->stmts->hd->stmts, indent); indent -= 2;
                  cout << endl; skipspaces(indent);
                  cout << "}";
               }
               else {
                  indent += 2; cout << endl; skipspaces(indent);
                  ppSTMT(nd->stmts->hd, indent); indent -= 2;
               }
            }
         }
      }
   }
}
```

Figure 7: Output from pretty-printing rules of Figure 6 (part 1)

```
void ppSTMTLIST (STMTLIST *nd, int indent)
{
   if (nd == NULL) {
   }
   else {
      if (nd->tl == NULL) {
         ppSTMT(nd->hd, indent);
      }
      else {
         ppSTMT(nd->hd, indent);
         cout << endl; skipspaces(indent);
         ppSTMTLIST(nd->tl, indent);
      }
   }
}
```

Figure 7: Output from pretty-printing rules of Figure 6 (part 2)

```
              (lo nd (rho1 ++ rho)),
              %``)
      end;
```

The operator ::: creates a C++ function definition with two arguments, as described earlier.

```
(* ::: : Name -> Layout -> CFunction *)
fun ((nm:Name) ::: (cmdfun:Layout)) =
    %`void pp^nm (^nm *nd, int indent)
     {
     ^(cmdfun (%`nd`) emptyenv)
     }
    ` ;
```

Operations occurring in layouts take a node and an environment and produce C++ code. The simplest of these do not look at the node or the environment. For example, this is `newline`:

```
(* newline: Layout *)
fun newline (nd:CNode) (rho:Env) =
      %`cout << endl; skipspaces(indent);` ;
```

`indent` takes an entire layout and returns code that will give the identical layout but with an additional two spaces of indent.

```
(* indent: Layout -> Layout *)
fun indent (lo: Layout) (nd: CNode) (rho: Env) =
      %`indent += 2; ^(lo nd rho) indent -= 2;`;
```

Operator oo simply sequences commands:

```
System.Control.quotation := true;
val % = implode o (map (fn QUOTE x => x|ANTIQUOTE x=>x));

(* C_if : expression * statement * statement -> statement *)
fun C_if (e, s1, s2) =
   if (e = "1" orelse e = "(1)")
   then s1
   else %`if (^e) {
          ^s1
        } ^(if (s2 <> "") then %`else {
        ^s2
        }` else %``)`;

fun fst (x,y) = x;
fun snd (x,y) = y;

type SyntaxFragment = string;

type CNode = SyntaxFragment
 and CPred = SyntaxFragment
 and CCommand = SyntaxFragment
 and CFunction = SyntaxFragment;

type Name = string;

type Env = Name -> CNode * CCommand;

exception UndefinedName of string;
fun emptyenv (s:Name) = raise UndefinedName s;
infix 4 |->;
fun ((id:Name) |-> (e:CNode * CCommand)) (s:Name) =
    if id = s then e else raise UndefinedName s;
(*
fun unitenv (id:Name) (e:CNode * CCommand) (s:Name) =
    if id = s then e else raise UndefinedName s;
*)
infix 3 ++;
fun ((rho1:Env) ++ (rho2:Env)) (id:Name) =
  (rho2 id) handle UndefinedName id => rho1 id;


type Pattern = CNode -> Env -> (CPred * Env)
 and Layout = CNode -> Env -> CCommand;

infix 2 ==> ;
(* ==>: Pattern * Layout -> Layout *)
fun ((pat:Pattern) ==> (lo:Layout)) (nd:CNode) (rho:Env) =
     let val (pred, rho1) = pat nd rho
     in C_if(pred,
             (lo nd (rho1 ++ rho)),
             %``)
     end;
```

Figure 8: The pretty-printing language definition (part 1)

```
infix 0 ::: ;
(* ::: : Name -> Layout -> CFunction *)
fun ((nm:Name) ::: (cmdfun:Layout)) =
    %`void pp^nm (^nm *nd, int indent)
     {
     ^(cmdfun (%`nd`) emptyenv)
     }
    ` ;

(* Operators appearing in Layout's *)

(* emit: string -> Layout *)
fun emit (s:string) (nd:CNode) (rho:Env) =
   %`cout << "^(s)";` ;

(* ||: Layout * Layout -> Layout *)
infixr 1 ||;
fun ((lo1:Layout) || (lo2:Layout)) (nd:CNode) (rho:Env)
  = %`^(lo1 nd rho)
    else {
    ^(lo2 nd rho)
    }` ;

(* oo: Layout * Layout -> Layout *)
infix 3 oo;
fun ((lo1:Layout) oo (lo2:Layout)) (nd:CNode) (rho:Env)
  = %`^(lo1 nd rho)
      ^(lo2 nd rho)` ;

(* using: Name -> Layout -> Layout *)
fun using (nm:Name) (lo:Layout) (nd:CNode) (rho:Env)
  = lo (fst (rho nm)) rho;

(* pp: Name -> Layout *)
fun pp (nm:Name) (nd:CNode) (rho:Env) = snd (rho nm);

(* newline: Layout *)
fun newline (nd:CNode) (rho:Env) =
       %`cout << endl; skipspaces(indent);` ;

(* indent: Layout -> Layout *)
fun indent (lo: Layout) (nd: CNode) (rho: Env) =
       %`indent += 2; ^(lo nd rho) indent -= 2;`;

(* space: Layout *)
val space = emit " ";

(* skip: Layout *)
fun skip (nd:CNode) (rho:Env) = %``;
```

Figure 8: The pretty-printing language definition (part 2)

```
(* Pattern constructors *)

(* non_list: Pattern *)
fun non_list (nd:CNode) (rho:Env) =
    (%`(^nd->sxptype != LISTSXP) && (^nd->sxptype != NILSXP)`,
     emptyenv);

(* empty: Pattern *)
fun empty (nd:CNode) (rho:Env) = (%`^nd == NULL`, emptyenv) ;

(* ASSIGN: Name -> Name -> Pattern *)
fun ASSIGN (var:Name) (exp:Name) (nd:CNode) (rho:Env) =
     (%`^nd->stype == ASSIGN`,
      var |-> (%`^nd->lhs`, %`ppEXPR(^nd->lhs);`)
      ++ exp |-> (%`^nd->rhs`, %`ppEXPR(^nd->rhs);`)
      );

(* BEGIN: Name -> Pattern *)
fun BEGIN (stmts:Name) (nd:CNode) (rho:Env) =
     (%`^nd->stype == BEGIN`,
       stmts |-> (%`^nd->stmts`, %`ppSTMTLIST(^nd->stmts, indent);`));

fun WHILE (cond:Name) (stmt:Name) (nd:CNode) (rho:Env) =
     (%`^nd->stype == WHILE`,
       cond |-> (%`^nd->lhs`, %`ppEXPR(^nd->lhs);`)
      ++ stmt |-> (%`^nd->stmts->hd`, %`ppSTMT(^nd->stmts->hd, indent);`)
      );

fun IF (cond:Name) (stmt1:Name) (stmt2:Name) (nd:CNode) (rho:Env) =
     (%`^nd->stype == IF`,
       cond |-> (%`^nd->lhs`, %`ppEXPR(^nd->lhs);`)
      ++
       stmt1 |-> (%`^nd->stmts->hd`, %`ppSTMT(^nd->stmts->hd, indent);`)
      ++
       stmt2 |-> (%`^nd->stmts->tl->hd`,
                  %`ppSTMT(^nd->stmts->tl->hd, indent);`)
      );

(* STMTLIST: Name -> Name -> Pattern *)
fun STMTLIST (hd:Name) (tl:Name) (nd:CNode) (rho:Env)
  = (%`1`,
      hd |-> (%`^nd->hd`, %`ppSTMT(^nd->hd, indent);`)
     ++
      tl |-> (%`^nd->tl`, %`ppSTMTLIST(^nd->tl, indent);`)
    );
```

Figure 8: The pretty-printing language definition (part 3)

```
(* oo: Layout * Layout -> Layout *)
fun ((lo1:Layout) oo (lo2:Layout)) (nd:CNode) (rho:Env)
  = %`^(lo1 nd rho)
      ^(lo2 nd rho)` ;
```

When a node is matched by a pattern, the pattern places in the environment function calls to pretty print the subtrees. The function pp just retrieves these saved commands.

```
(* pp: Name -> Layout *)
fun pp (nm:Name) (nd:CNode) (rho:Env) = snd (rho nm);
```

As an example of a pattern-forming function, consider `ASSIGN`:

```
(* ASSIGN: Name -> Name -> Pattern *)
fun ASSIGN (var:Name) (exp:Name) (nd:CNode) (rho:Env) =
     (%`^nd->stype == ASSIGN`,
      var |-> (%`^nd->lhs`, %`ppEXPR(^nd->lhs);`)
      ++ exp |-> (%`^nd->rhs`, %`ppEXPR(^nd->rhs);`)
      );
```

It expects two pattern variables, to be assigned to the syntax tree nodes representing, respectively, the left and right-hand sides of the assignment. Given these, it returns a C++ predicate testing whether the given node is an assignment node, and an environment in which the two names are bound to the actual addresses of the nodes they name and to function calls that can be used to pretty print them.

7 Conclusions

We have presented a simple and elegant method of developing powerful meta-programming languages, by defining program-generating functions in a functional language. With this approach, languages can be developed with relatively modest effort. Moreover, the result is a "higher-order" meta-programming language; this means the language will scale up well, allowing for well-structured and maintainable meta-programs, rather than being extended with a set of *ad hoc* data types and control structures, as so often happens to special-purpose languages.

However, the method is not without its drawbacks, some apparent and some subtle, which are the subject of current research. One obvious problem is that the languages are not as clean syntactically as they would be if developed in a more conventional way; for example, a conventional parser generator does not require that you put the word `term` before every occurrence of a terminal symbol.

The deeper problems have to do with the precise choice of operators. This problem is not specific to meta-programming languages, but arises whenever a language is defined by adding new data types to an existing language, as we suggested in the first paragraph of this paper. The problem is that the set of operators may not be powerful enough. This can have two consequences:

1. Users will simply be unable to use the language to solve all of their problems. In effect, this is the situation with current special-purpose languages.

2. Users will need to deal with the underlying representation of the new values. This is the unfortunate situation in the languages we have defined. Since the values are defined in terms of existing ML types, users may be confronted with error message given in terms they cannot understand, and may need to go down to data representation level to add new capabilities to the language.

Both problems can only be solved by choosing a sufficiently powerful set of operations on the new data. This simply indicates that a great deal of effort needs to go into that choice. There seems to be little known about how to do this.

References

[1] S. Ahmed, D. Gelernter, *Program builders as alternatives to high-level languages,* Yale Univ. C.S. Dept. TR 887, November 1991.

[2] B. Balzer, N. Goldman, D. Wile, *Rationale and Support for Domain Specific Languages*, USC/Information Sciences Institute, available at `http://www.isi.edu/software-sciences/dssa/dssls/dssls.html`.

[3] J. Bell, F. Bellegarde, J. Hook, R.B. Kieburtz, A. Kotov, J. Lewis, L. McKinney, D.P. Oliva, T. Shear, L. Tong, L. Walton, and T. Zhou, *Software design for reliabiity and reuse: A proof-of-concept demonstration*, TRI-Ada '94.

[4] W. E. Carlson, P. Hudak, M. P. Jones, *An experiment using Haskell to prototype "Geometric Region Servers" for navy command and control*, Research Report YALEU/DCS/RR–1031, Yale Univ. C. S. Dept., May 1994.

[5] D. Carr, *Glue: A tree-based program development and maintenance system which uses explicit, typed, higher order cliches,* M.S. Thesis, Univ. of Illinois C.S. Dept. Report UIUCDCS–R–89–1495, Feb. 1989.

[6] P. Hudak, *Building domain-specific embedded languages,* position paper for Workshop on Software Engineering and Programming Languages, Cambridge, MA, June 1996.

[7] P. Hudak, T. Makucevich, S. Gadde, B. Whong, *Haskore music notation: An algebra of music*, J. Func. Prog., to appear.

[8] G. Hutton, *Higher-order functions for parsing*, J. Func. Prog. 2(3), 323–343, July 1992.

[9] S. Kamin, *Report of a workshop on future directions in programming languages and compilers*, ACM SIGPLAN Notices 30 (7), July 1995, 9–28.

[10] C. Plinta, K. Lee, and M. Rissman, *A model solution for C^3I message translation and validation*, Technical Report CMU/SEI–89–TR–12, Software Engineering Institute, Carnegie Mellon University, December 1989.

[11] T. Sheard, N. Nelson, *Type safe abstractions using program generators*, OGI Tech. Rpt. 95–013.

[12] . *Standard ML of New Jersey User's Guide*, February 1993.

[13] L. Walton and J. Hook, *Message Specification Language (MSL): Reference Manual, Revision: 1.8*, Oregon Graduate Institute, Oct. 6, 1994.

[14] R. C. Waters, *The Programmer's Apprentic: A session with KBEmacs*, IEEE Trans. Software Eng. SE–11(11), 1296–1320, Nov. 1985.