Electronic Notes in Theoretical Computer Science **14** (1998)
URL: `http://www.elsevier.nl/locate/entcs/volume14.html` 20 pages

Research on Domain-specific Embedded Languages and Program Generators

Samuel N. Kamin [1]

Computer Science Department
University of Illinois at Urbana-Champaign
Champaign, IL, USA

Abstract

Embedding is the process of implementing a language by defining functions in an existing "host" language; the host language with these added functions *is* the new language. As a consequence, the new language comes equipped with all the features of the host language, with no additional work on the part of the language designer. Embedding works particularly well when the host language is a functional language.

We describe several examples of embedded languages. The first is a language for specifying simple pictures. The others are program generators, that is, languages used to specify programs in other languages. In all of these examples, the host language is Standard ML; in the program generating languages, the target language is C++. The power obtained from the host language is the main emphasis of our presentation.

1 Introduction

The goal of research in programming languages is to develop concepts and tools to facilitate language design and implementation. These tools should be of help not only for the design of traditional general-purpose languages, but also — in fact, especially — for the design of *special-purpose*, or *domain-specific*, languages. Furthermore, they should not only simplify the construction of language processors, but should aid in the design of *high-quality* languages.

To many programming language researchers, the highest quality languages are the functional languages, such as Haskell [5] and Standard ML [16]. As it happens, there is a simple way to construct languages for specific application areas so that these languages will, without fail, be well-designed functional languages: *embedding* [2,4,7,10,12]. Embedding is the process of implementing a language by defining functions in an existing "host" language; the host

[1] Partial support received from NSF under Grant CCR 96–19655.

language with these added functions *is* the new language, so that the new language has all the power of the host language. Though this method could be used with any language, certain features of functional languages — such as higher-order functions — tend to make the results of embedding in a functional language particularly satisfactory.

The embedding approach is particularly useful for the implementation of domain-specific languages, languages that incorporate operations peculiar to a narrow area of computation. They tend to be used for comparatively small programs, often written by domain experts rather than professional programmers. For such uses, the high level of discourse and concise syntax provided by functional languages are particularly appreciated, while their inefficiencies are suffered more easily.

This paper describes several experiments in language implementation by embedding. The first is a language for describing simple pictures, inspired by a well-known domain-specific language, the `pic` preprocessor for `troff` [13]. The remainder are all examples of *program-generating languages*. This is a category of languages in which programs are actually specifications for programs in other languages. Perhaps the best-known examples are the parser generators, such as `yacc` [14]. From our point of view, these are just languages produced by embedding, that is, by adding program-generating functions to a functional language. Our examples include a simple parser generator for which we give all details, a more complicated one for which we give only examples, and a language for specifying certain types of tree-structured data types.

All of our examples use Standard ML as the host language. The program generators produce C++ code.

The alternative to language implementation by embedding is the traditional approach in which a grammar is designed and a parser written (or generated), and syntax-directed translation of the parse tree produces the desired effect. By comparison, the embedding approach has two advantages: it is easier to do, and it produces a powerful language as its result.

We view the second of these advantages — the quality of the resulting language — as by far the more important, for reasons we would like to explain. Domain-specific languages are most often implemented by the traditional method, with great pains taken to provide a syntax natural for domain experts. However, beyond this domain-specific syntax, they tend to be weak and poorly designed; such programming features as are provided are added haphazardly. The justification is that such features are supposedly not needed, since the domain-specific features cover everything needed by the domain expert, the intended user of the language.

Yet, time and time again — especially when the domain-specific language achieves widespread use — we see that programming features *are* needed, and then it is often too late. The beauty of language design by embedding is that the programming features come automatically and for free. This, in our view, is the real point of the method. Accordingly, our presentation emphasizes the power obtained "for free" from the host language. Even in the case of parser generators, we give examples showing the power of the programming features.

On the other hand, language embedding has its drawbacks, including syntax that is often far from optimal, poor error messages, and an inability to perform domain-specific optimizations and transformations. These issues, which are the topics of current research, are discussed in the conclusions.

The paper assumes knowledge of Standard ML.

2 FPIC

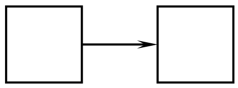
FPIC [12] is a language for drawing simple pictures. It is inspired by the Unix utility `pic` [13], a widely used preprocessor for `troff`. FPIC attempts to preserve the flavor of `pic`'s syntax, though it differs in detail.

FPIC is embedded in Standard ML and consists of approximately 1200 lines of ML. Our claim is that writing these lines represents a modest effort for the power of the resulting language.

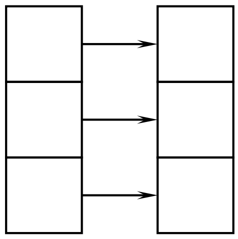
To illustrate, we give a collection of examples that use the following operations (a subset of the roughly 100 operations currently provided with the FPIC distribution). We name the type of each operation, in the hope that the intent of these types is self-evident, but we do not in this paper give the definitions either of the types or the operations; see references [12,11] for details. All the binary operations listed below are infix, except `harrow` and `line`.

Operation & type	**Description**
`square`: $\mathit{real} \to \mathit{Picture}$	Draw a square of a given size
`circle`: $\mathit{real} \to \mathit{Picture}$	Draw a circle of a given radius
`line`: $\mathit{Point} \to \mathit{Point} \to \mathit{Picture}$	Draw a line between two points
`lines`: $\mathit{Point\ list} \to \mathit{Picture}$	Draw a line between each pair of points in the list
`harrow`: $\mathit{real} \to \mathit{real} \to \mathit{Picture}$	Draw an arrow to the right, at a given height and of a given length
`seq`: $\mathit{Picture} \times \mathit{Picture} \to \mathit{Picture}$	Superimpose one picture on another
`hseq`: $\mathit{Picture} \times \mathit{Picture} \to \mathit{Picture}$	Draw one picture next to another
`vseq`: $\mathit{Picture} \times \mathit{Picture} \to \mathit{Picture}$	Draw one picture above another
`seqlist`: $\mathit{Picture\ list} \to \mathit{Picture}$	Superimpose all pictures in the list
`empty`: $\mathit{Picture}$	An empty picture, useful as a right identity for sequencing operations
`offsetBy`: $\mathit{Picture} \times (\mathit{real} \times \mathit{real}) \to \mathit{Picture}$	Move picture by a given amount
`scale`: $\mathit{Picture} \times \mathit{real} \to \mathit{Picture}$	Stretch a picture horizontally and vertically by certain amount
`scaleTo`: $\mathit{Picture} \times (\mathit{real} \times \mathit{real}) \to \mathit{Picture}$	Resize a picture to fit within a given sized box
`centeredAt`: $\mathit{Picture} \times \mathit{Point} \to \mathit{Picture}$	Move a picture so that its center is at a given point

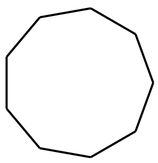
The following examples are intended merely to show how the programming capabilities of Standard ML, combined with the few primitives listed above, add up to a powerful programming language for pictures. For example, the function-plotting operations defined below — `plot` and `xyplot` — could easily be supplemented with operations to read function values from a file, to draw a grid, to include a legend, and so on, forming a plotting library comparable to, but far more powerful than, say, `gnuplot` [15].



```
(* Draw two squares connected by a horizontal arrow *)
val sq = square 1.0;
val boxes = sq hseq harrow 0.5 1.0 hseq sq;
boxes;
```

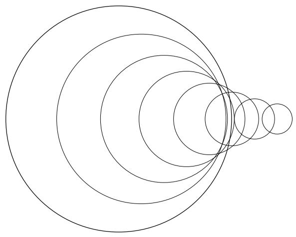


```
(* Use ML's foldr operation to draw several copies of boxes *)
foldr (op vseq) empty [boxes, boxes, boxes];
```



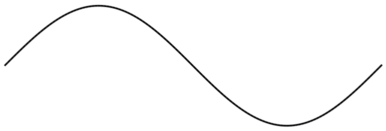
```
(* Draw a regular polygon with n sides *)
fun regular_poly n =
  let val rn = toReal n
      val realintvl = map toReal (intvl 0 n)
      val angles = map (fn k => k * (2.0*Math.pi/rn)) realintvl
      val points = map (fn theta => (Math.cos theta, Math.sin theta))
                       angles
  in lines points
  end;

regular_poly 9;
```



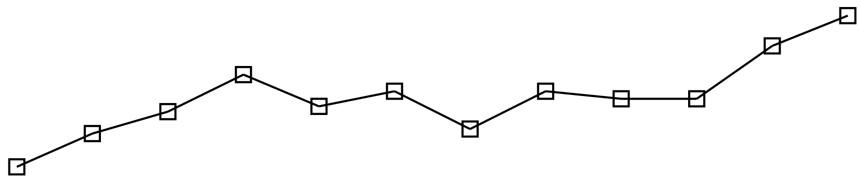
```
(* Draw n copies of P, moving each by (dx,dy) and scaling by s *)
fun copies P n (dx, dy) s =
  seqlist (map (fn k => P offsetBy (k*dx,k*dy) scale (Math.pow(s,k)))
             (map toReal (intvl 0 (n-1))));

copies (circle 5.0) 8 (1.0,0.0) 0.75;
```



```
(* Plot function f in the range x0..x1, sampling at each *)
(* interval of size dx; fit the plot into an area of size w by h *)
fun plot w h (x0:real) x1 dx f =
  let fun drawpoints x fx = if x>x1 then empty
                            else let val fdx = f (x+dx)
                                 in (line (x, fx) (x+dx, fdx))
                                    seq (drawpoints (x+dx) fdx)
                                 end
  in (drawpoints x0 (f x0)) scaleTo (w,h)
  end;

plot 5.0 5.0 0.0 (2.0*Math.pi) 0.05 Math.sin;
```

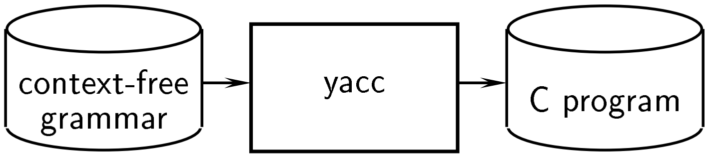


```
(* Connect a list of points by lines, with a picture drawn at each point *)
fun xyplot ptlis ptpic =
  (lines ptlis) seq (seqlist (map (fn p => ptpic centeredAt p) ptlis));

val djia = [(1.0,7.000), (2.0, 7.440), (3.0, 7.730), (4.0, 8.220),
            (5.0,7.800), (6.0, 8.000), (7.0, 7.500), (8.0, 8.000),
            (9.0,7.900), (10.0, 7.900), (11.0, 8.600), (12.0, 9.000)];
xyplot djia (square 0.2);
```

3 Program generation

A *program generator* is a language processor whose input is a program specification and whose output is a program. A well-known example is the parser generator `yacc` [14]:



In line with our philosophy of language implementation by embedding, we propose to create program generators by adding program-manipulating combinators to Standard ML. Thus, just as an FPIC picture specification is an ML expression of type `Picture`, so in these languages program specifications are ML expressions of type `Program`.

The `Program` type — actually, we use more descriptive type names, like `Parser` — can represent programs in whatever target language we choose. We have written generators that produce Java code, HTML, TeX, and even Standard ML. However, most of our examples generate C++ code, and in the following sections we confine ourselves to such examples.

Three examples will suffice to give the flavor of our approach. The first is a generator for simple recursive descent parsers, the second an LL(1) parser generator, and the third a generator of C++ class definitions for abstract syntax trees. In each case, we will provide examples in the language and some indication of the size of the language implementation (in lines of ML), but only for the first will we actually show the implementation.

4 Simple parser generator

Parsers are a classic example of language embedding in functional languages [8,9]. Traditionally, a parser is a function from an input stream to a syntax tree (we're simplifying somewhat for expository purposes). Combinators like `oo` and `||` can be defined and used to form parser-valued expressions like[2]

```
val A = term a oo B oo B || B
and B = term b || term c oo A;
```

representing the grammar with rules $A \to aBB|B$ and $B \to b|cA$. This parser can then be applied to an input stream to produce a parse tree. All of this is in ML; there is no program generation being done here.

Similar combinators can be defined to generate a parser in C++. Some care is required in the types of the combinators. A "parser" in this language is a C++ function (or, ambiguously, a sequence of C++ functions). Each non-terminal has an associated parsing function, and the various parts of the context-free rules for that non-terminal represent the body of this function. Thus, the value of the right-hand side of a rule is of a different type from the rule as a whole. Specifically, the types of the combinators are:

`||`: $\mathit{RHS} \times \mathit{RHS} \to \mathit{RHS}$
`oo`: $\mathit{RHS} \times \mathit{RHS} \to \mathit{RHS}$
`term`: $\mathit{Token} \to \mathit{RHS}$
`nonterm`: $\mathit{Name} \to \mathit{RHS}$
`::=`: $\mathit{Name} \times \mathit{RHS} \to \mathit{Parser}$

The `nonterm` combinator is needed to turn a name into a call to the appropriate parser function; it is not needed in the pure ML version because each non-terminal is the ML name of a parsing function. The `::=` combinator is also new. It is the combinator that produces the C++ function definition; it takes the place of recursion in the functional language itself (see above). When recursion in the host language is used in a language's embedding, it will necessarily need to be replaced by a combinator that emits the appropriate target language code.

The grammar give above is rewritten as

```
"A" ::= term a oo nonterm "B" oo nonterm "B"
     || nonterm "B" ;
"B" ::= term b
     || term c oo nonterm "A" ;
```

These are expressions of type `Parser`. Their values are these two C++ functions:

```
int parseA () {
```

[2] In ML, `A` and `B` need to be defined as functions, in order to avoid non-termination, though this would not be necessary in a lazy language like Haskell; we have elided this "eta-expansion" step to simplify the example.

```
   int pos = current;
   if (tokens[current] == a)
       current++;
   else
       goto L0;
   if (!parseB()) goto L0;
   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 give the definitions of the combinators in section 4.2.

4.1 Using the parser-generator language

In the introduction to this paper, we placed strong emphasis on the power of the language that one obtains from the embedding approach. In this case, one gets the ability to manipulate grammars, a feature totally absent from `yacc`. Yet it is often stated that domain-specific languages do not *need* a programming capability, and indeed `yacc` has survived quite nicely without one. Here we give two examples to demonstrate that programming features can be useful even in a parser generator.

Bear in mind that all the programming features we use in these examples — excepting only the parsing combinators listed above — are obtained *for free*.

(Before continuing, we warn the reader that these examples do not quite work, in the sense that the grammars resulting from the transformations we will make are not necessarily amenable to our simple parsing method — indeed, very few grammars are. The point is that we can use the programming facilities of the host language to manipulate grammars. With a stronger parsing method — such as the one implemented in the next section — the examples would be more likely to produce working parsers.)

For our first example, consider the following problem: Assume the language we wish to parse is an expression language which, like Standard ML itself, has token classes `op1`, `op2`, ..., `op9`, representing binary operators of increasing precedence. Grammars that incorporate precedence, are unambiguous, and can be parsed top-down are tricky to write. Here is the classic example of expressions over `+` and `*`:

$$
\begin{aligned}
&E \rightarrow T\ E' \\
&E' \rightarrow \texttt{+}\ E \ \mid\ \epsilon \\
&T \rightarrow P\ T' \\
&T' \rightarrow \texttt{*}\ T \ \mid\ \epsilon \\
&P \rightarrow \texttt{id}
\end{aligned}
$$

Thus, our grammar will have a sequence of rules of the form:

$$
\begin{aligned}
&\mathit{Expr1} \rightarrow \mathit{Expr2}\ \mathit{Expr1}' \\
&\mathit{Expr1}' \rightarrow \texttt{op1}\ \mathit{Expr1} \ \mid\ \epsilon \\
&\mathit{Expr2} \rightarrow \mathit{Expr3}\ \mathit{Expr2}' \\
&\mathit{Expr2}' \rightarrow \texttt{op2}\ \mathit{Expr2} \ \mid\ \epsilon \\
&\quad\vdots
\end{aligned}
$$

(*Expr10* is a separate case that must be written out by hand, like P above.)

We can avoid writing such a long list of rules by using the facilities of the host language to write a rule-generating function:

```
fun leveln (n:int) =
   let val ntn = "Expr"^(toString n)
       val ntn' = "Expr"^(toString n)^"prime"
       val ntn'' = "Expr"^(toString (n+1))
       val opn = "op"^(toString n)
   in [ntn ::= nonterm ntn'' oo nonterm ntn',
       ntn' ::= term opn oo nonterm ntn || empty
```

```
    ]
  end;
```

Now we can generate all 27 rules with the call

```
map leveln (intvl 1 9)
```

As another example, suppose we wanted to add the capability of using regular-right-part rules, that is, rules with regular expressions in their right-hand sides.

There is a well-known translation from a regular-right-part rule to a set of ordinary rules parsing the same sentences. We will formalize this translation as follows: Consider a production $A \to R$, where R is a regular expression over grammar symbols. Note that, since right-hand sides can use alternation, we can make the restriction, without loss of generality, that every non-terminal has a single production. We translate R into a pair $\widehat{R}$, containing an ordinary right-hand side for A and a set of new, ordinary productions (from new non-terminals). Thus, if A has the one production $A \to R$, and if $\widehat{R} = (\alpha, \Phi)$, then the set of productions $\{A \to \alpha\} \cup \Phi$ derive the same sentences (from A) as the one original production.

Here, then, is the definition of $\widehat{R}$, by induction on the structure of R:

$$
\begin{aligned}
\widehat{RS} &= (\widehat{R}_1\widehat{S}_1, \widehat{R}_2 \cup \widehat{S}_2) \\
\widehat{R|S} &= (B, \{B \to \widehat{R}_1|\widehat{S}_1\} \cup \widehat{R}_2 \cup \widehat{S}_2), \text{ where } B \text{ is new} \\
\widehat{R*} &= (B, \{B \to \widehat{R}_1 B|\epsilon\} \cup \widehat{R}_2, \text{ where } B \text{ is new} \\
\widehat{X} &= (X, \{\}), \text{ if } X \text{ is a token, non-terminal, or } \epsilon
\end{aligned}
$$

This translation generates only ordinary rules, because for every R, $\widehat{R}_1$ contains no Kleene stars or alternation, and the new productions in $\widehat{R}_2$ contain no Kleene stars and alternation only at the top level.

With the programming facilities of the host language, we can write these translations:

```
type RRP = RHS * Parser list;

val rpempty (*: RRP *) = (empty, []);
val rpterm (*: Token -> RRP *) = fn t => (term t, []);
val rpnonterm (*: Name -> RRP *) = fn n => (nonterm n, []);

infix 3 ooo; (* RRP * RRP -> RRP *)
fun (rhs1,rl1) ooo (rhs2,rl2) = (rhs1 oo rhs2, rl1 @ rl2);

infix 2 |||; (* RRP * RRP -> RRP *)
```

```
fun (rhs1,rl1) ||| (rhs2,rl2) =
    let val B = genName ()
    in (nonterm B,
        [B ::= rhs1 || rhs2] @ rl1 @ rl2)
    end;

val star (* RRP -> RRP *)
  = fn (rhs,rl) => let val B = genName ()
                   in (nonterm B,
                       (B ::= rhs oo nonterm B || empty) :: rl)
                   end;

infix 1 ::==; (* :name * RRP -> Parser list *)
fun A ::== (rhs,rl) = (A ::= rhs) :: rl;
```

4.2 Defining the combinators

The parsing method we implement is recursive-descent. Recursive descent parsing is not a powerful method. Or, rather, it is powerful only insofar as it is used *informally* and can be modified manually in specific cases. The LL(1) method presented in the next section is far more powerful. However, this example is much easier to explain, as it includes only about 50 lines of ML (of which about 15 is C++ code to be emitted). We will present all the code for this example, and not for the more elaborate examples to follow.

The basic idea of *top-down parsing* [1] is this: We are at all times attempting to find a substring of the input that can be derived from a particular non-terminal. By looking at the next token of the input, we decide which rule for that non-terminal is most appropriate, and then proceed to try to find strings derivable from each part of that right-hand side; this in turn leads to attempts to find strings for the non-terminals occurring in that right-hand side, and so on. *Recursive descent* parsing is a method of implementing top-down parsing in which each non-terminal is represented by a parsing function, which performs the actions just described. That is, it decides which right-hand side is appropriate and then attempts to match a string derivable from that right-hand side; if a non-terminal occurs in the right-hand side, the parsing function corresponding to that non-terminal is called. Thus, one obtains a set of mutually recursive parsing functions.

The key question in formalizing this method is, how does a parsing function determine which right-hand side is appropriate? For this example, we give a very simple answer: it checks the next input token against the first symbol of each rule that starts with a token; if none match the input token, then the rule that does not begin with a token is used. This version of the method requires that only one right-hand side for a given non-terminal can start with a non-terminal (or ϵ); we further require that this rule be presented as the last rule for that non-terminal.[3]

As we have said, a "parser" in this language is a C++ function:

```
type Parser = CFunction;
```

Each non-terminal in a grammar has an associated parsing function. Thus, the collection of all rules for a non-terminal will be an expression of type `Parser`. The individual rules, on the other hand, denote parts of that parsing function. More precisely, each right-hand side denotes an *attempt* to parse the input, which may fail and have to jump to the next right-hand side. Thus,

```
type RHS = Label -> CCommand;
```

The combinators have the following types, as given earlier:

`||`: $RHS \times RHS \to RHS$
`oo`: $RHS \times RHS \to RHS$
`term`: $Token \to RHS$
`nonterm`: $Name \to RHS$
`::=`: $Name \times RHS \to Parser$

The simplest case is the code corresponding to a token — `term` t — which just compares the current token to t and either succeeds or jumps to the failure label. Assuming the entire input is in an array `tokens` and the integer variable `current` is the index of the next token, the piece of code is

```
if (tokens[current]) == t)
  current++;
else
  goto failure-label;
```

Abstracting from both the token and the failure label, we get the definition of `term` (the carat (^) is ML's string concatenation operator):

```
fun term (t:Token) = fn (lab:Label) =>
  "if (tokens[current]) == " ^ t ^ ")\n"
  ^ "current++;\n"
  ^ "else\n"
  ^ "goto " ^ lab ^ ";\n" ;
```

We are treating the C++ program simply as a string, and indeed we will continue to do so in all of our program generators.

We can enhance the readability of this code by using ML's *anti-quotation* feature. With this feature, one can write[4]

[3] Of course, recursive descent could never be used in practice if these restrictions were enforced. Again, the method is normally used informally, with "lookahead" added as needed. Formalizing this lookahead leads eventually to LL(1) parsing, which we will implement separately. The idea here is to give a simple formalization of recursive descent so that we can illustrate the program generation method.

[4] The feature is more general than what we are presenting; see [17] for details.

```
    %`... ^x ...`         for   "..." ^ x ^ "..."

and %`... ^(---) ...`     for   "..." ^ (---) ^ "..."
```

That is, within anti-quotation brackets (`%‘` and `‘`), an anti-quoted expression (surrounded by `^(` and `)`) is evaluated (to a string) and spliced in. As an abbreviation, if the expression “`---`” consists of a single identifier, the parentheses can be elided. Furthermore, newlines can be embedded within anti-quotation brackets.

Using anti-quotation, the definition of `term` becomes:

```
(* term: Token -> RHS *)
fun term (t:Token) = fn (lab:Label) =>
   %`if (tokens[current] == ^t)
       current++;
     else
       goto ^lab;` ;
```

Here are the other combinators. Note how the alternation combinator creates a label for the second alternate, and the rule-forming combinator `::=` provide the final failure label.

```
(* empty: RHS *)
val empty = fn lab => %``;

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

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

(* ||: 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;

(* ::= : Name -> RHS -> CFunction *)
infix 1 ::= ;
fun (v:Name) ::= (rhs:RHS) =
```

```
%`int parse^v () {
  int pos = current;
  ^(rhs "L1000")
  return true;
L1000:
  current = pos;
  return false;
}` ;
```

5 LL(1) parser generator

An LL(1) parser [1] operates by keeping a (statically generated) table M which maps non-terminals and tokens to productions. When attempting to parse a string derived from non-terminal A, when the current token is t, $M[A,t]$ gives the unique production from A — it has to be unique, or the grammar is not LL(1) — that can derive a string starting with t, if any. The construction of M is somewhat complex and is the heart of the parser construction process.

We do not wish to explain the construction of LL(1) parsing tables, but we can say this much: The construction involves the calculation of two functions, $First : \textit{Non-terminals} \Rightarrow \wp(\textit{tokens} \cup \{\epsilon\})$ and $Follow : \textit{Non-terminals} \Rightarrow \wp(\textit{tokens} \cup \{\textit{eof}\})$. $First(A)$ contains every token that can possibly be the initial token in a string derived from A; it includes "ϵ" if A can derive the empty string. $Follow(A)$ contains all the tokens that can immediately follow A in any string derivable from the start symbol (that is, in any sentential form); if A can appear as the last symbol in a sentential form, then $Follow(A)$ contains *eof*. Note that these sets cannot be determined solely from the productions for A; $Follow(A)$, for example, can be determined only by looking at all the productions of the grammar in which A occurs.

These "global" calculations can be induced from the meanings of individual phrases in the grammar, as long as those meanings are rich enough. For this parser generator, we have had to change the combinator types a bit. Here, a `RHSPart` refers to a fragment of a right-hand side, a `RHS` is one or more complete right-hand sides for a single non-terminal, and a `Rule` is a non-terminal together with all its right-hand sides. Thus, the combinators are:

`||`: $\textit{RHS} \times \textit{RHS} \rightarrow \textit{RHS}$
`oo`: $\textit{RHSPart} \times \textit{RHSPart} \rightarrow \textit{RHSPart}$
`term`: $\textit{Token} \rightarrow \textit{RHSPart}$
`nonterm`: $\textit{Name} \rightarrow \textit{RHSPart}$
`prod`: $\textit{RHSPart} \rightarrow \textit{RHS}$
`::=`: $\textit{Name} \times \textit{RHS} \rightarrow \textit{Rule}$

Note the new combinator `prod`, which coerces a `RHSPart` to a `RHS`. A new function, called `grammar`, converts a list of *Rule*s to a list of parsing functions in C++. The grammar used as an example in the previous section becomes:

```
"A" ::= prod (term a oo nonterm "B" oo nonterm "B")
     || prod (nonterm "B") ;
"B" ::= prod (term b)
     || prod (term c oo nonterm "A") ;
```

The global nature of the table construction entails that the combinator definitions be quite a bit more complex than in the previous section. Here we present only the types:

```
type RHSPart =
           (unit -> token list)    (* calculate FIRST set *)
         * (token list -> unit)    (* add to FOLLOW set *)
         * CCommand;               (* code to parse rhs, or fail *)

type RHS = (unit -> (token list) list) (* FIRST sets of all RHS's *)
         * (token list -> unit)    (* add to FOLLOW set *)
         * CCommand list;          (* code for all RHS's *)

type Rule = (unit->unit)   (* add to First set for lhs *)
          * (unit->unit)   (* add to Follow set for lhs *)
          * (unit->(token list) list) (* get First sets for all rhs's *)
          * (unit -> int -> CArrayInit) (* calculate array M *)
          * CCommand;             (* calculate C++ parser fcn *)
```

The entire set of combinators (including auxiliary functions and type declarations) comes to about 350 lines of ML code. As a point of comparison, the Bison LR parser generator [3] is about 7000 lines of C. Granted, the LALR(1) construction that Bison uses is more complicated than the LL(1) construction used here, and Bison includes some additional facilities such as ambiguity resolution. However, it contains nothing analogous to the programming facilities whose use we illustrated in the previous section. (Those examples need some minor changes to work with the current set of combinators.)

6 Abstract syntax tree generation

Language processors usually begin their work by parsing their input and constructing an *abstract syntax tree* [1], basically a simplified version of the parse tree. Abstract syntax trees (AST's) are trees whose nodes are labelled with *abstract syntax operators*. Each operator σ has a signature $\tau_1 \times \tau_2 \times \cdots \tau_n \rightarrow \tau$ In an AST, the type of a node labelled with σ is τ, and it must have exactly n children with types $\tau_1, \ldots, \tau_n$. An *abstract syntax* is a finite set of abstract syntax operators. In short, then, an AST is a tree whose structure is constrained by the set of operators in its abstract syntax.

C++ has excellent facilities for defining tree-like data and hiding their representation. Given an abstract syntax, it is a simple matter to write a class whose objects are AST nodes. There are two basic styles, which we'll call the *single-class* style and the *subclass* style. In the single-class style, the class `ASTNode` contains a tag field (the name of the operator), a union type (the children corresponding to each operator), and a collection of constructors, accessors, and auxiliary functions, like `print`. The latter is written as a `switch` statement, dispatching on the tag (that is, the abstract syntax operator) of the node. In the subclass style, `ASTNode` is an abstract base class, and each operator is implemented as a derived class of `ASTNode`; each such class defines its own constructors and destructors, and its own part of functions like `print`, eliminating the `switch` statement in favor of dynamic method dispatch.

Either way is straightforward, but each has disadvantages. The subclass method makes it easy to add a new abstract syntax operator, since the required changes are completely localized: just add a new subclass. On the other hand, the single-class method facilitates the addition of new auxiliary functions, since these can be added as single function definitions in a single class, instead of as separate parts of a function definition spread across all the subclasses. Thus, either method can result in code that is difficult to maintain, depending upon the types of changes that need to be made.

Yet in either case the implementation of abstract syntax trees is highly stylized; an experienced programmer can write such classes almost without thinking. Thus, this would appear to be a natural domain for program generation.

We have written a program generator that generates (single-class) implementations of abstract syntax trees given a list of the abstract syntax operators. As an example, suppose we have an abstract syntax with one type (*Expr*) and three operators:

$const$: $int \to Expr$
$plus$: $Expr \times Expr \to Expr$
$negate$: $Expr \to Expr$

We would specify this abstract syntax as follows:

```
genAbsSyn "Expr"     (* name of the abstract syntax *)
          ["Expr"]   (* AST types that are being defined *)
          ["const" oftype "int" --> "Expr",
           "plus" oftype "Expr" ** "Expr" --> "Expr",
           "negate" oftype "Expr" --> "Expr"
          ] ;
```

This function call produces two C++ files, `Expr.h` and `Expr.C`. As usual, the "`.h`" file gives the representation and some small function definitions, while the "`.C`" file contains the remaining function definitions. Specifically, the class defines constructors, accessors, settors, and a print function. In total, the two files contain about one hundred lines of C++ code.

The program generator is written in about 180 lines of ML.

The use of a program generator has some decided advantages over either of the representations that a C++ programmer might use. The addition of new abstract syntax operators is as simple as it could possibly be, even simpler than in the subclass implementation. Adding new auxiliary functions — there are countless possibilities — is perhaps harder than it would be using the single-class implementation, because it requires modifying the program generator, but the advantage is that, once the change is made, it is made in *all* the AST classes generated by the generator. Similarly, a change in representation requires changes in the program generator, but once made, is made everywhere.

7 Conclusions

Though much has been learned about the structure of programming languages and their processors, the simple question "how should I design a language appropriate for my application?" has no easy answer. Embedding in a functional language is a method that is relatively easy and produces good results.

However, these results are far from perfect, and many issues remain before the method can be very widely used. A functional language designed for embedding would need to meet the needs of the domain-specific language user more fully. Some needed accommodations are:

Better syntax. The syntax of all of our embedded languages is more verbose than it would be if the language were defined from scratch. An example is the syntax of context-free grammars used in our parser generators; it is "isomorphic" to the `yacc` syntax, but still about twice as long simply due to syntactic issues such as having to place an operator between each symbol in the right-hand sides.

Better error messages. Users can receive error messages that are utterly incomprehensible, because they are designed for users of the host language rather than users of the embedded language. For example, if a user enters the following grammar in the recursive-descent parser generator:

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

(typing `:=` instead of `::=`), our embedded implementation produces this fearsome response:

```
Error: operator and operand don't agree [tycon mismatch]
  operator domain: 'Z ref * 'Z
  operand:         string * (Label -> string)
  in expression:
    "A" := term a
```

The error message can be understood only by a user who not only knows ML, but also knows how values in the embedded language are represented.

(An even worse result occurs if the user types:

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

mistyping the first `oo` as `o`. `o` is an infix operator in ML, representing function composition, and the above expression, as it happens, is both syntactically correct and type correct. Indeed, the value of this expression is a syntactically legal C++ function. Unfortunately, that function does not parse the grammar that the user intended to enter.)

Domain-specific analyses. The FPIC plotting function draws a curve by emitting a long list of line-drawing commands. Depending upon the output device, some other representation may be more efficient. Of course, representation transformations could be done as a separate pass, but ideally the plotting function would have such an optimization built in. In particular, this would allow the embedded language processor to implement representation optimizations in separately compiled program segments, just as the ML compiler optimizes individual functions.

The program-generating languages suggest an entirely new set of domain-specific analyses, namely analyses of the program that is to be generated. For instance, one would like to be able to ensure the syntactic and type correctness of generated programs *a priori*, before generating any actual programs. (The MetaML language [18] does just this for program generators from ML to ML, written in a certain style.) As things stand, these analyses are done *by the C++ compiler*, but building them into the generator would give earlier feedback; furthermore, the generator could perform some optimizations, based on its knowledge of the programs it is generating, that the C++ compiler could not be expected to perform.

We are currently exploring ways in functional language processors could be customized in these ways.

Acknowledgement

The author gratefully acknowledges the support provided by the Oregon Graduate Institute, where he was on sabbatical during the preparation of this paper.

References

[1] Aho, A. V., R. Sethi and J. D. Ullman, "Compilers: Principles, Techniques, and Tools," Addison-Wesley, 1986.

[2] Carlson, W. E., P. Hudak and 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.

[3] Donnelly, C. and R. Stallman, "The Bison Manual: Using the YACC-compatible Parser Generator," Free Software Foundation, 1995.

[4] Elliott, C., *Modeling interactive 3D and multimedia animation with an embedded language,* Proc. USENIX Conf. on Domain-Specific Languages, Santa

Barbara, Oct. 1997, pp. 285–296.

[5] Hudak, P., S. Peyton Jones and P. Wadler (eds.). *Report on the Programming Language Haskell (Version 1.2),* ACM SIGPLAN Notices **27**(5), May 1992.

[6] Hudak, P., *Building domain-specific embedded languages,* Computing Surveys, **28A**(4).

[7] Hudak, Paul, Tom Makucevich, Syam Gadde and Bo Whong, *Haskore music notation: An algebra of music,* J. Func. Prog. **6**(3) (1996), pp. 465-483.

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

[9] Hutton, G. and E. Meijer, *A Haskell library of monadic parser combinators,* Web page at `www.cs.nott.ac.uk/Department/Staff/gmh/pearl.hs`, April, 1997.

[10] Kamin, S., *The Challenge of Language Technology Transfer,* ACM Computing Surveys **28A**(4) (1996).

[11] Kamin, S., FPIC documentation, Web page at `www-sal.cs.uiuc.edu/~kamin/fpic/`.

[12] Kamin, S. and D. Hyatt, *A special-purpose language for picture-drawing,* Proc. USENIX Conf. on Domain-Specific Languages, Santa Barbara, Oct. 1997, pp. 297–310.

[13] Kernighan, B. W., *PIC: A crude graphics language for typesetting,* Bell Laboratory, 1981.

[14] Levine, John R., Tony Mason and Doug Brown, "Lex & Yacc," 2nd Ed. O'Reilly & Associates, 1992.

[15] Liaw, Andy and Dick Crawford, "`gnuplot` 3.5 User's Guide," Available by anonymous ftp at `picard.tamu.edu` in directory `pub/gnuplot`.

[16] Milner, Robin, Mads Tofte and Robert Harper, "The Definition of Standard ML," The MIT Press, Cambridge, MA, 1990.

[17] Standard ML of New Jersey User's Guide, Available at `cm.bell-labs.com/cm/cs/what/smlnj/index.html`, 1997.

[18] Taha, Walid and Tim Sheard, *Multi-stage programming with explicit annotations,* Proc. ACM SIGPLAN Symp. on Partial Evaluation and Semantics-Based Program Manipulation (PEPM '97), Amsterdam, June 12–13, 1997, SIGPLAN Notices **32**(12) (1997), pp. 203-217.