Parsers and Printers as Stream Destructors and Constructors Embedded in Functional Languages

Michel Mauny
INRIA
B.P. 105 F-78153 Le Chesnay CEDEX
mauny@inria.inria.fr

1 Introduction

Functional languages are nice supports to build prototypes. Strong polymorphic typing ([11]) provides genericity and programming security. User definable types permit to design complex and readable data structures and again, programming security is enhanced. Pattern-matching definitions of functions are a concise and readable way to express algorithms. It is thus possible, in the process of designing prototype languages, to describe their abstract syntax by means of structured types and to write semantic functions such as compilers and program transformers. However, defining a concrete syntax for a prototype language is far from being a trivial task (especially when that language is often evolving), and is rarely integrated to functional languages. Parser generators have a quite crude syntax and are usually completely independent from our favorite functional language. Particularly from the typing point of view, we cannot ask a parser generator to integrate our favorite language type system.

We propose in this paper an extension of an ML dialect to treat parser definitions. The key point of the extension is the adjunction of stream expressions and patterns with the possibility of matching actual streams against these patterns. This particular form of pattern-matching is implemented using syntax analysis techniques and the functions using this pattern-matching virtually build annotated parse trees and return the computed value of their attributes. We call them *parsers*.

Parametric parsers (we call them *higher-order* parsers) may be defined in this framework. They permit modular design of grammars.

Writing a (pretty) printer for a language may also be difficult. Formatting primitives may be gathered in a "language of boxes" as in PPML ([13]) which provides considerable help. However, it is useful to notice that when a parser for a language has been written in a high-level language, a pretty-printer for that language is very easy to write because the

structures of parsing and printing algorithms may be very similar in the cases where the output of pretty-printer must be a valid input to the corresponding parser.

A simplified version of these features has been implemented in CAML ([13]) and a system of macros for ML is described as an application.

In section 2 we give the syntax of the new features together with examples. Typing rules for stream expressions and parsers are given in section 3 and an informal semantics of parsers is given in section 4. Implementation of macros is described in section 5. The current implementation of parsers in CAML is briefly described in section 6 where the history of "object languages" manipulation in CAML is recalled. The printer definition feature, conceptually simpler, will not be fully developed here; it has also been integrated to the CAML system ([8]).

2 Syntax

We extend the kernel of an ML dialect by allowing parser and printer definitions. Both definition styles are statically typechecked and then parsers are analyzed by a parser generator and compiled as usual function definitions. Printers behave as usual function definitions.

2.1 The kernel of the language

The considered language possesses (basic or functional) constants as numbers, addition, etc, a conditional and pattern-matching abstractions. This last feature presupposes the existence of structured types. For simplicity, we will only assume the existence of a (sum) type for lists and a (product) type for pairs. We give in figure 1 the syntax of this part of the language in a BNF style.

The syntactic category <*Basic constants*> corresponds to numbers, boolean values (denoted by `true` and `false`) and strings (such as `"foo"`). Predefined functional constants are addition, multiplication and subtraction denoted respectively by the infix symbols "`+`", "`*`" and "`-`", and string concatenation "`^`".

Function bodies have one or more cases and local declarations may be multiple and recursive. Variable names are usual identifiers and are introduced by their binding occurrence in a local declaration or in a pattern of an abstraction. The syntax of patterns is given in figure 2. In both expressions and patterns, parentheses may be used freely for disambiguating purposes.

We often note `[`e_1`;`$\ldots$`;`e_n`]` the expression e_1`::`$\ldots$`::`e_n`::[]`. We also use `(*` and `*)` as comment delimitors.

We suppose that the language has a call-by-value semantics. We have done no special effort about evaluation strategies, but we believe that this work could fit well with lazy evaluation because we consider streams as values[1].

2.2 Extension to Parsers and Printers

The key point of the extension is the introduction of a new kind of structured values and patterns together with a special semantics for both matching against these patterns and value construction.

These expressions and patterns have a special syntax because they represent segments

[1] Actually, stream construction would be more reasonable when submitted to lazy evaluation.

```
<Expr>   ::=  <Basic constants>
          |  <Functional constants>
          |  <Expr> , <Expr>                          Pairs
          |  []                                      Empty list
          |  <Expr> :: <Expr>                        Non empty lists
          |  <Identifier>
          |  <Expr> <Expr>                           Applications
          |  <Expr> <Infix operator> <Expr>          Application of an infix function
          |  function  <Pattern> -> <Expr>           Pattern-matching abstractions
                       ...
                   |   <Pattern> -> <Expr>
          |  if <Expr> then <Expr> else <Expr>       Conditional expressions
          |  let [rec] <Identifier> = <Expr>         Local [recursive] declarations
             ...
             and <Identifier> = <Expr>
             in <Expr>
```

Figure 1: The syntax of expressions in the kernel language

```
<Pattern>   ::=  <Basic constants>
             |   _                                   Wild card
             |   <Pattern> , <Pattern>               Pairs
             |   []                                  Empty list
             |   <Pattern> :: <Pattern>              Non empty list
             |   <Identifier> <Pattern>              Other structured patterns
             |   <Pattern> as <Identifier>
```

Figure 2: The syntax of patterns in the kernel language

of streams. We first extend the syntax of expressions as shown in figure 3. Patterns appearing in the left hand side of an arrow "`->`" in a parser body are restricted to be stream patterns (either empty or not) defined in figure 4, and these are the only possible occurrences of stream patterns (in particular, no usual function may attempt to match its arguments against a stream pattern).

We do not extend the syntax of patterns, but we define a new syntactic category *<Stream pattern>* given in figure 4.

In the following, we will only consider parsers working on streams whose elements belong to a particular type `lexeme` defined below. These elements correspond to terminals and the associated input (obtained from a character stream) is given as a comment in the definition. Typing is described in section 3.

It is theoretically possible to define lexical analysers using the notation of parser definitions[2] ([3]). However, for efficiency pur-

[2] We give in appendix an example of parser defini-

```
<Expr>   ::=   ...
            |  [< >]                                                  Empty stream
            |  [< <Stream component> ... ; <Stream component> >]      Non empty stream
            |  parse  <Stream pattern> -> <Expr>                      Parser body
                      ...
                  |   <Stream pattern> -> <Expr>


<Stream component>   ::=   <Expr>     Substream
                         |   '<Expr>   Stream element
```

Figure 3: Syntax of expression extended to streams and parsers

```
<Stream pattern>   ::=   [< >]
                       |  [< <Stream pattern component> ... ; <Stream pattern component> >]


<Stream pattern component>   ::=   <Expr> <Identifier>
                                 |   '<Pattern>
```

Figure 4: Syntax of stream patterns

poses, it is preferable to have a special notation for lexical analysers definitions. Because of lack of space, we do not introduce here such a notation.

```
type lexeme =
  Num of num       (* Numbers *)
| Bool of bool     (* "true" or "false" *)
| Ident of string  (* Identifiers *)
| Star             (* "*" *)
| Plus             (* "+" *)
| Sharp            (* "#" *)
| Comma            (* "," *)
| Semic            (* ";" *)
| Lsqb             (* "[" *)
| Rsqb             (* "]" *)
| Lpar             (* "(" *)
| Rpar             (* ")" *)
| Lfquote          (* "<<" *)
| Rfquote          (* ">>" *)
```

tion where the lexical part is handled by a parser.

2.3 Examples

In the examples, we write the type of each function as a comment. We first define a printer for lists of numbers by using a predefined function `pnum` which builds a stream of character strings from its numeric argument.

```
(* print_num_list : num list → string stream *)
let print_num_list =
 let rec pns = function
         [] -> [< >]
       | [n] -> [< pnum n >]
       | n::L -> [< pnum n; '";"; pns L >]
 in function L -> [< '"["; pns L; '"]" >]
in print_num_list [1;2;3]
```

Intuitively, `print_num_list` builds a

stream whose elements are character strings. Stream elements are expressions preceded by the "'" character, other stream components are substreams which are expanded inside the stream being built.

We may also define a parser for a list of numbers (as `"[1;2;3]"`):

```
(* parse_num_list : lexeme stream → num list *)
let parse_num_list =
 let rec pns = parse
     [< 'Num n >] -> [n]
   | [< 'Num n; 'Semic; pns l >] -> n::l
 in parse [< 'Lsqb; 'Rsqb> >] -> []
        | [< 'Lsqb; pns L; 'Rsqb >] -> L
in ...
```

The `parse_num_list` function must be seen as matching a stream segment composed of elements of type `lexeme`. Such elements stand for terminals and represent respectively numbers, semicolons, left and right square brackets.

When applied to a stream, `parse_num_list` will match its first element against `Lsqb`. In case of failure, the whole function call fails. Otherwise, if the second element is matched by `Rsqb`, then the first rule is chosen and the empty list `[]` is returned as result. If the second element is not `Rsqb`, then the `pns` function matches it against the pattern `Num n`, binding the identifier `n` to the numeric value held by the constructor `Num`. Then the next element of the stream is matched against `Semic`, and so on. When the recursive call of `pns` returns a list of numbers, the identifier `l` gets bound to the list and `n::l` is returned. The variable `L` then gets bound to `n::l` which becomes the result of the whole parser call. If one of the matches above fails, the parser call fails.

Expressions appearing in stream patterns denote thus calls to matching functions on the current stream argument, and the variable following them gets bound to their result; patterns preceded by the "'" character are used to match stream elements (terminals).

The next example generalizes the above parser for lists of numbers to arbitrary lists by having a non terminal as formal parameter.

```
(* star : (lexeme stream → α)
          → lexeme stream → α list *)
let star = function p ->
 let rec pns =
  parse [< p x >] -> [x]  (* a singleton *)
      | [< p x; 'Semic; pns L >] -> x::L
 in parse [< 'Lsqb; 'Rsqb> >] -> []
        | [< 'Lsqb; pns L; 'Rsqb >] -> L
in ...
```

It takes a parser as argument and iterates that parser on its stream argument. For example, the partial application:

```
star (parse [< 'Num n >] -> n)
    : lexeme stream → num list
```

recognizes sequences of numbers such as "`[1;2;3]`" (and returns lists of numbers).

Notice that it is also possible to define parsers whose semantic actions contain formal parameters.

3 Typing

In this section, we will assign types to parsers and printers. We assume given the typing rules for expressions and patterns of the kernel language. Typing rules are classically written as inference rules. Rules for expressions allow the deduction of a type for a given expression under some set of hypotheses. Rules for patterns build a set of hypotheses (i.e. a typing environment) from a pattern under some set of hypotheses. We will only extend these sets of rules in order to assign types to parser and printer definitions.

$$(1)\quad \frac{}{\Gamma \vdash_{\mathrm{Exp}} \texttt{[< >]} : \alpha\ \texttt{stream}}$$

$$(2)\quad \frac{\Gamma \vdash_{\mathrm{SEC}} sc_i : \tau\ \texttt{stream} \quad i = 1 \ldots, n}{\Gamma \vdash_{\mathrm{Exp}} \texttt{[<}sc_1 \ldots; sc_n\texttt{>]} : \tau\ \texttt{stream}}$$

$$(3)\quad \frac{\Gamma \vdash_{\mathrm{Exp}} e : \tau}{\Gamma \vdash_{\mathrm{SEC}} \texttt{'}e : \tau\ \texttt{stream}}$$

$$(4)\quad \frac{\Gamma \vdash_{\mathrm{Exp}} e : \tau\ \texttt{stream}}{\Gamma \vdash_{\mathrm{SEC}} e : \tau\ \texttt{stream}}$$

Figure 5: Typing stream expressions

$$(5)\quad \frac{}{\Gamma, \emptyset \vdash_{\mathrm{SPat}} \texttt{[< >]} : \alpha\ \texttt{stream}}$$

$$(6)\quad \frac{\Gamma, \Delta_i \vdash_{\mathrm{SPC}} sc_i : \tau\ \texttt{stream} \quad \cap_i \overline{\Delta}_i = \emptyset \quad i = 1, \ldots n}{\Gamma, (\cup_i \Delta_i) \vdash_{\mathrm{SPat}} \texttt{[<}sc_1 \ldots; sc_n\texttt{>]} : \tau\ \texttt{stream}}$$

$$(7)\quad \frac{\Gamma, \Delta \vdash_{\mathrm{Pat}} p : \tau}{\Gamma, \Delta \vdash_{\mathrm{SPC}} \texttt{'}p : \tau\ \texttt{stream}}$$

$$(8)\quad \frac{\Gamma \vdash_{\mathrm{Exp}} e : \tau\ \texttt{stream} \rightarrow \sigma \quad \Gamma, \Delta \vdash_{\mathrm{Pat}} p : \sigma}{\Gamma, \Delta \vdash_{\mathrm{SPC}} e\ p : \tau\ \texttt{stream}}$$

Figure 6: Typing stream patterns

We first assign types to stream expressions. Streams belong to the type **stream**. We build typing environments from stream patterns, and then we will be ready to assign types to parsers bodies.

3.1 Stream Expressions

Let us recall that stream expressions represent either the empty stream or a stream built up from some ML expressions. The typing rules that we need extend the relation $\vdash_{\mathrm{Exp}}$ for typing ML expressions; they are given at figure 5. The relation $\vdash_{\mathrm{Exp}}$ uses an auxiliary relation $\vdash_{\mathrm{SEC}}$ which assigns a type to components of stream expressions (elements or substreams).

Rules (1) and (2) assign types to stream expressions and auxiliary rules (3) and (4) give types to stream components (either substreams or stream elements). It is thus possible to build (homogeneous) streams whose elements belong to an arbitrary type. Polymorphic formatting tools could also be integrated to the concrete syntax of streams: they should be interpreted by output routines on streams. This has been implemented in the CAML system ([13], [8]) but will not be described here.

$$(5)\quad \frac{\Gamma, \Delta_i \vdash_{\mathrm{SPat}} sp_i : \sigma\ \texttt{stream} \quad \Gamma_{\overline{\Delta_i}} \cup \Delta_i \vdash_{\mathrm{Exp}} e_i : \tau \quad i = 1 \ldots, n}{\Gamma \vdash_{\mathrm{Exp}} (\texttt{parse}\ sp_1\ \texttt{->}\ e_1\ \ldots \mid\ sp_n\ \texttt{->}\ e_n) : \sigma\ \texttt{stream} \rightarrow \tau}$$

Figure 7: Typing parser bodies

3.2 Stream Patterns

We define in figure 6 a new relation $\vdash_{\mathrm{SPat}}$ analoguous to the relation $\vdash_{\mathrm{Pat}}$ building typing environments and checking the type of patterns. The meaning of "$\Gamma, \Delta \vdash_{\mathrm{SPat}} sp : \tau$" could be expressed as: "under the set of hypotheses Γ, the stream pattern sp has type τ and produces the typing environment Δ". The meaning of "$\Gamma, \Delta \vdash_{\mathrm{Pat}} p : \tau$" is "under the set of hypotheses Γ, the pattern p has type τ and produces the typing environment Δ".

The relation $\vdash_{\mathrm{SPat}}$ uses the auxiliary relation $\vdash_{\mathrm{SPC}}$ which assigns types to components of stream patterns.

Note that stream patterns must be *linear*: every variable bound by a (stream) pattern must occur only once in the whole (stream) pattern. The "$\cup$" operator denotes set union and "$\cap$" set intersection. For a given set of hypotheses Δ, $\overline{\Delta}$ denotes the set of identifiers to which Δ gives a type.

Notice also that the typing of a pattern component $e\ p$ (rule (8)) differs from a typing rule for application. The expression e will be applied to the actual stream argument, and the variable pattern p will be bound to the result of this application.

3.3 Parsers and Printers

It is now trivial to assign types to printer and parser bodies since they must be typed as function bodies. There is no special rule for printers since they are only introduced by stream expressions. We note Γ_V the set of hypotheses obtained from Γ by removing all type assumptions on identifiers belonging to the set V. The rule for parser bodies is given in figure 7. In order to type a parser body, each stream pattern sp_i produces a typing environment Δ_i which is used to assign a type τ_i to the corresponding expression e_i. As in function bodies, all the τ_i are unified producing a type τ and the type of the parser body is the functional type σ `stream` $\rightarrow \tau$, where σ `stream` is the common type of the sp_i's.

3.4 Example

We refer here to the example given in section 2.3. We may verify the correctness of the types of the `star` function and of its partial application to a parser of numbers.

Another application of `star` is such that:

```
parse_list_gen
    (parse_list_gen (parse [< 'Num n >] -> n))
  : lexeme stream → (num list) list
```

A slightly complex example where different non terminals possess different types could be:

```
(* num_or_bool : lexeme stream -> num *)
let num_or_bool =

    (* pnum : lexeme stream -> num *)
    let pnum = parse [< 'Num n >] -> n

    (* pbool : lexeme stream -> bool *)
```

```
  and pbool = parse [< 'Bool b >] -> b

  in
parse [< pnum n; pbool b >]
       -> if b then n else -n
```

4 Semantics

In this section, we outline the semantics of parser definitions. Actually, the precise semantics (eliminating classical non determinism of syntax analysis) can only be given when a particular parsing technique has been chosen.

Two semantic points should be precised: the first one concerns the matching of actual streams by stream patterns and the second concerns the values returned by a parser. In this section, we try to relate the pattern-matching of streams to context-free grammars. This does not constitute an operational semantics (which depends on parsing and parser generation techniques). These grammar specifications may then be seen as attribute grammars when semantic actions are added. The value returned by a parser call is thus the value of the synthetized attribute of the root node of (one of) the parse tree(s) produced.

4.1 From Parser Definitions to Grammar Specifications

We give only the intuition allowing to see a parser body as a context-free grammar specification. More information about classical parsing techniques and their implementation may be found in [3]. Unlike grammar specifications, we have a notion of scope and auxiliary parsers may have to be renamed in order to be put all in the same scope.

We may thus transform the definition or `parse_num_list` (which we recall here):

```
(* parse_num_list : lexeme stream → num list *)
let parse_num_list =
 let rec pns = parse
     [< 'Num n >] -> [n]
   | [< 'Num n; 'Semic; pns l >] -> n::l
 in parse [< 'Lsqb; 'Rsqb> >] -> []
        | [< 'Lsqb; pns L; 'Rsqb >] -> L
```

into the grammar specification:

```
parse_num_list → Lsqb Rsqb
parse_num_list → Lsqb pns Rsqb
pns → Num
pns → Num Semic pns
```

where **Num**, **Semic**, **Lsqb** and **Rsqb** are terminals while other identifiers are non-terminals. Renaming occurs when a local declaration is hidden by another one.

We need also to give names to anonymous parsers. So:

```
let p = parse
 [< 'Num n; (parse 'Comma -> ()
                | 'Semic -> ()) _ >] -> n
```

translates into:

```
p → Num aux
aux → Comma
aux → Semic
```

where **aux** is a new non-terminal name. In order to generate the context-free grammar specification from a parser definition, it is necessary to reduce expressions appearing in stream patterns until parser names or parser bodies are obtained. This reduction may be realized by the usual semantic rules of functional languages. Here is an example needing such a transformation:

```
(* p_bools_or_nums :
            lexeme stream → (num list) *)
let p_bools_or_nums =
 let rec star = function p ->
   (parse [< >] -> []
        | [< p x; (star p) L >] -> x::L)
in parse [< 'Lsqb; (star (parse [< 'Num n >]
                              -> n)) L;
            'Rsqb >] -> L
       | [< 'Lsqb;
            (star (parse [< 'Bool b >] ->
                     if b then 1 else 0)) L;
            'Rsqb >] -> L
```

This definition translates into the grammar specification:

```
p_bools_or_nums → Lsqb star_num Rsqb
p_bools_or_nums → Lsqb star_aux Rsqb
star_num →
star_num → Num star_num
aux → Bool
star_aux →
star_aux → aux star_aux
```

where `aux` and `star_aux` are new non-terminal names.

The semantics of a higher-order parser taking a parser as argument could be written as a parameterized context-free grammar specification.

4.2 Semantic Actions as Attributes

So far, we forgot the values returned by parsers. Once the context-free grammar specification is generated, we must add semantic actions to the grammar specification and classical methods can be used to synthetize ML expressions associated to parse trees from annotated parse trees ([3]).

In some usages of parsers (cf. section 5), there may be also an interaction between the context calling a parser in order to select a particular parse tree. For example, a type-checker could eliminate parse trees whose synthetized attribute value is not matched by the type expected at that occurrence ([1]).

4.3 Operational Semantics and Compilation issues

The choice between deterministic and non deterministic parsing techniques influence the evaluation of semantic actions. With deterministic techniques, a semantic action may be executed as soon as the rule reduction occurs, whereas execution of semantic actions has to be delayed until a particular parse tree has been selected in the non deterministic case.

Code generation for parsers (consisting for example in building a parsing table) should occur at compile time. Nevertheless, in presence of unknown non terminals, incomplete parsing tables should be generated, and missing information should be provided by application of higher-order parsers to a specific parser. When arbitrary expressions are used as non terminals, the situation is even more complicated: when the parser is called, then this kind of non terminals must be evaluated to parsers in order to provide information to the parser. At first sight, parsers must be represented in a special way in order to be distinguished of usual functions: type information is not sufficient. Non terminals which are not parsers (i.e. which do not attempt to match their argument) such as:

```
function (s : lexeme stream) -> 1
```

should be treated as semantic actions.

In case of first-order parser definition, classical parser generation techniques may be used,

producing parsing tables which may be interpreted by a parsing engine. In CAML, we have chosen to analyze parser definitions through the Yacc ([9]) parser generator, imposing some restrictions on the parsers definable in the language.

At our knowledge, none of the classical parser generation techniques considers the problem of unknown non terminals. The problem consists in generating an incomplete description of a parsing table for a higher-order parser which will be completed by the application to its arguments.

This should be related to incremental parser generation techniques which have been proposed in [7]. These techniques rely on Tomita's pseudo-parallel LR parsing algorithm ([12]). It seems also possible to relate the problem of unknown non terminals to the work on the parsing of incomplete sentences described in [10]. Both works use sophisticated techniques to optimize sharing of common substructures of parse trees built in parallel. A description of such techniques with complexity results may be found in [4].

5 Macro Expressions and Macro Evaluation

In this section, we consider extensions of the ML parser itself by user-defined parsers. This constitutes an application of the integration of parser definition in functional languages.

We will assume the existence of a type `dyn` of values of dynamic type (the disjoint union of all possible types). More information about dynamic types may be found in [2] and [13]).

5.1 Parsers producing ML programs

It is often desirable to give a concrete syntax to objects of a specific structured type. As an example, consider the CAML type definition for abstract syntax trees for arithmetic expressions:

```
type aexpr =
  Constant of num
| Addition of (aexpr * aexpr)
| Multiplication of (aexpr * aexpr)
```

Instead of typing in:

```
Addition (Constant 1, Constant 2)
```

we would like to write something like:

```
1+2
```

maybe after telling the toplevel parser that another parser has to be called. Assuming that we have defined a parser for arithmetic expressions, this is possible in CAML by writing:

```
<< 1+2 >>
```

where "<<" and ">>" have to be read as french quotation marks.

We define the parser `pexpr` for arithmetic expressions by:

```
(* pexpr : lexeme stream → aexpr *)
let rec pexpr =
 parse [< 'Num n >] -> n
     | [< 'Lpar; pexpr e; 'Rpar >] -> e
     | [< pexpr e1; 'Star; pexpr e2 >]
                        -> Mult(e1,e2)
     | [< pexpr e1; 'Plus; pexpr e2 >]
                        -> Add(e1,e2)
```

Note that the underlying grammar is ambiguous but this does not matter for our purpose.

We now define a function `pexpr_to_ML` calling that parser and transforming the result into a value of type `ML` in order to form a legal piece of abstract syntax. Before doing this, we must define a constructor called `MLquote` of type (`dyn → ML`). So the definition of the type `ML` looks like:

```
type ML = ...
        | MLquote of dyn
          ...
```

Typechecking and compiling `MLquote` expressions are trivial tasks since their type and value are hidden inside a dynamic value. The definition of `pexpr_to_ML` is thus:

```
(* pexpr_to_ML : iostream → ML *)
let pexpr_to_ML = function
  str -> MLquote (dynamic (pexpr str))
```

Its type is `lexeme stream → ML`. The user now may put it in a special system reference in order that the toplevel parser will be able to find it.

Let us now take a look at the ML toplevel parser. We only give here the rules concerning calls to user-defined parsers:

```
let rec parse_ML =
 let rec parse_MLdecl = ...
  ...
 and parse_MLexpr =
  parse ...
      | [< 'Lfquote; (default_parser()) e;
           'Rfquote >] -> e
      | ...
```

where the evaluation of `default_parser()` looks in the system reference containing the default user-defined parser and returns it.

We also would like to be able to write things like:

```
let e1 = <<1+2>> and e2 = <<3+4>>
in <<#e1 + #e2>>
```

or:

```
function <<#e1 + #e2>> -> "addition"
       | <<#e1 * #e2>> -> "multiplication"
```

where the expressions immediately following "`#`" are ML expressions and *not* expressions belonging to the arithmetic language. In other words, since the language of arithmetic expressions may be considered as an object language, we want "escapes" to its metalanguage. The "`#`" character designates such an escape. It is similar to commas used in Lisp in the scope of a "backquote" character. In this case, the phrase `<<1+2>>` does not represent the *constant value* `Add(Const 1, Const 2)` but the *ML program* building this value. More precisely, the result returned by `pexpr_to_ML` is the abstract syntax tree of "`Add (Const 1, Const 2)`".

We must define a parser returning programs for arithmetic expressions instead of values by:

```
(* pexpr : lexeme stream → ML *)
let rec pexpr =
 parse [< Num n >] -> MLnum_const n
     | [< 'Sharp; parse_MLexpr0 e >] -> e
     | [< 'Lpar; pexpr e; 'Rpar >] -> e
     | [< pexpr e1; 'Star; pexpr e2 >]
          -> MLapply (MLvar "Mult",
                      MLpair (e1,e2))
     | [< pexpr e1; 'Plus;
          pexpr e2 >]
          -> MLapply (MLvar "Add",
                      MLpair (e1,e2))
```

The parser `parse_MLexpr0` recognizes either atomic expressions (as numbers, identifiers, boolean values ...) or completely parenthesized ML expressions.

The `pexpr` parsing function may be called by the toplevel parser since it already produces abstract syntax trees for ML expressions.

5.2 Macros

Since we are able to use any parser and to make it available as a macro-syntax facility (i.e. between "`<<`" and "`>>`"), we may also do it for the ML toplevel parser itself.

Suppose that the default grammar is the one of ML itself. The following declaration binds the identifier `MAX` to the *syntax of* `256` of type `ML`:

```
let MAX = <<256>>
```

We may add a special rule to the ML parser of expressions:

```
  ...
and parse_MLexpr0 =
 parse ...
     | [< 'Sharp; parse_MLexpr0 e >]
       -> some code evaluating e
       ...
```

This rule says "after the keyword '`#`', recognize an atomic ML expression; then evaluate it, make sure it is of type `ML`, and return it as result". This rule is recursively available in other rules for ML expressions, declarations etc, so it specifies general macro calls. It is of course possible to add similar rules everywhere a macro (with maybe a type such as, for example, `MLdecl` for declarations) is desirable.

If the declaration of the macro `MAX` has been made at toplevel, we may parse:

```
... if x = #MAX then ... else ...
```

exactly as:

```
... if x = 256 then ... else ...
```

If we have a version of the ML parser which produces *programs* building abstract syntax trees instead of directly their values, we might also define *macro functions*.

6 Implementation

Parser and printer definitions have been partially implemented in CAML. Instead of having anonymous or local parsers, we have parser declarations only as a toplevel syntactic construct. It is also currently impossible to define higher-order parsers.

Parser definitions are statically typechecked, and compilation goes through the Yacc ([9]) parser generator which produces parsing tables sent as arguments to a unique parsing system function.

6.1 History

Object languages manipulation has quite a long history in ML from earlier versions of ML available at INRIA to recent versions of CAML. The first integration to an ML system has been realized by P. Lechenadec at INRIA ([6]). At this time, the implementation was already borrowed from D. MacQueen's interface between Lisp and Yacc adapted to Le_Lisp by G. Berry. The ML version has then been ported to CAML by F. Dupont and A. Suárez. The author's work started by giving a better syntax to grammar specifications and adding a static typechecker to parser definitions. Macros came for free as soon as evaluation functions were made available.

6.2 Current implementation

The current implementation of parser definitions possesses two distinct syntactic con-

structs for parsers producing ML programs and parsers producing values. Although it is not as powerful as the features described in this paper, it is still useful and produces efficient parsers. The CAML toplevel parser itself has been defined in this way.

The compilation of parsing functions relies on the Yacc parser generator ([9]). Yacc produces deterministic LALR(1) parsers which enable semantic actions to be executed as soon as a grammar rule is reduced.

There is no lexical analyser definition facility in the current implementation. There is only one lexical analyser in CAML; it takes as parameter a set of keywords which is specific to each grammar.

Using input/output features of CAML together with (system or user-defined) evaluation functions, it is very easy for the user to write toplevel loops reading from standard input or from files.

The CAML toplevel may know several user-defined parsers (producing values of type `ML`): the "`<< ...>>`" has been extended to "`<:` *parser name* `< ...>>`" in order to call a specific parser.

Calls to other parsers from a specific parsers are possible through semantic actions. The advantage is that grammar specifications are analysed separately and parsing functions produced are efficient.

The possibility to define parsers in CAML is widely used. As non trivial examples (excepted CAML parsers themselves), the SAM compiler generator ([14]) and the Calculus of Constructions proof system ([5]) use intensively these features.

The style of recursive definitions working on patterns written in concrete syntax is particularly readable; for example, consider a compiling function written as:

```
let rec pascal_compiler = function
      <<if #test then #stat1 else #stat2>>
         -> (pascal_compiler test)
             @[Branch (compile_stat stat1,
                       compile_stat stat2)]
       | <<while #test do #stat>> -> ...
         ...
```

It is more readable than the version manipulating explicit abstract syntax trees, and thus easier to maintain.

We did not emphasize on printer specifications. In fact, they are defined, typed and compiled as usual functions. In CAML, a syntax similar to the one of stream expressions is implemented. This syntax is enriched by boxes, breaks and indentation annotations making pretty-printers fairly easy to write ([13], [8]).

6.3 Future Work

Much work remains to be done in order to give an efficient compilation to higher-order parsers. An operational semantics of stream expressions and parser definitions would represent an important step in this direction.

A possible extension of this work is to provide a Typol-like ([14]) facility in order to program the semantic treatments of object formalisms. It would then be possible for the user to relate the semantic and syntactical aspects of any object language and to program, for example, the disambiguating role of typing over parsing, obtaining results similar to those of [1].

7 Related Works

Recent related works have been done at Chalmers University of Göteborg ([1]). Although powerful parsing techniques are used,

parsers produced are not made available to the user. Furthermore, the user cannot choose its semantic actions.

We still believe that it is interesting to have parsers as ML values together with the possibility of parameterizing parsers. The macro-syntax technique given at section 5 provides the same possibilities as syntax extensions described in [1] and interaction between the type-checker and parsers could be added as done in [1] in case of usage of a non-deterministic parsing technique.

8 Conclusion

We have proposed an integration of parser definitions into functional languages. We have given a syntactic extension of usual ML syntax. It provides the ability to define statically typechecked parsing functions of arbitrary order of functionality. The type-checking of parser definitions has been described and an informal non deterministic semantics was given. Macro-syntax facilities consistent with ML type discipline were obtained as applications.

9 References

[1] A. Aasa, K. Petersson, D. Synek, Concrete Syntax for Data Objects in Functional Languages. In *Proceedings of the 1988 ACM Conference on Lisp and Functional Programming*, Snowbird, 1988, pp. 96-105.

[2] M. Abadi, L. Cardelli, B. Pierce, G. Plotkin, Dynamic Typing in a Statically-Typed Language. In *Proceedings of ACM Symposium on Principles of Programming Languages*, Austin 1989, pp. 213-227.

[3] A. Aho, R. Sethi, J. Ullman, *Compilers: Principles, Techniques and Tools*. Addison Wesley 1986.

[4] S. Billot, B. Lang, The Structure of Shared Forests in Ambiguous Parsing. To appear in *Proceedings of the 27th Annual Meeting of the Association for Computational Linguistics, Vancouver (Canada)*, June 1989.

[5] T. Coquand, G. Huet, The Calculus of Constructions, Information and Computation 76,2/3, 1988.

[6] G. Cousineau, G. Huet, L. Paulson, *The ML handbook*, INRIA 1985.

[7] J. Heering, P. Klint, J.G. Rekers, Incremental generation of parsers, CWWI Report CS-R8822, May 1988.

[8] O. Jehl, Rapport de stage de troisième année, ENST, 1989.

[9] S.C. Johnson, Yacc: Yet Another Compiler-compiler. In *Unix Programmer's Manual* Vol. 2B.

[10] B. Lang, Parsing Incomplete Sentences. In *Proceedings of the 12th Int. Conf. on Computational Linguistics*, COLING'88, Budapest (Hungary), August 1988.

[11] R. Milner, A theory of type polymorphism in programming. In *J. Comput. Syst. Sci. 17, pp. 348-375, 1978.*

[12] M. Tomita, An Efficient Augmented-Context-Free Parsing Algorithm, Computational Linguistic 13 (1-2), 1987.

[13] P. Weis, M.V. Aponte, A. Laville, M. Mauny, A. Suárez, *The CAML V2-6 Reference Manual*. INRIA 1989.

[14] P. Weis, *Métacompilation très efficace à l'aide d'Opérateurs Sémantiques*, Thèse de Doctorat, Université de Paris VII, 1987.

[14] Clément D., Despeyroux J., Despeyroux T., Hascouët L., Kahn G., *Natural Semantics on the Computer*, INRIA Research Report RR 416, INRIA-Sophia-Antipolis, 1985.

Appendix

We give as appendix the complete specification of a parser for pure lambda expressions.

For simplicity, we allow only mono-character identifiers. We define two functions: a simple lexical analyser (defined as a parser) and a parser for streams of lexemes. We note constant characters by `'a'`, `'b'` etc.

```
(* Lexical part *)
type lexeme =
  Ident of char | Lambda | Dot | Lpar | Rpar

(* spaces : char stream → unit *)
let rec spaces =
 parse [< ' '; spaces _ >] -> ()
     | [< '           t'; spaces _ >] -> ()
     | [< '          n'; spaces _  >] -> ()
     | [< >] -> ()

(* lexm : char stream → lexeme *)
let lexm =
 parse [< 'a' >]  -> Ident 'a'
     | ...
     | [< 'Z' >]  -> Ident 'Z'
     | [< '.' >]  -> Dot
     | [< '(' >]  -> Lpar
     | [< ')' >]  -> Rpar
     | [< '                   ' >] -> Lambda

(* lex : char stream → lexeme stream *)
let rec lex =
 parse [< spaces _; lexm l; lex s >] -> [< 'l; s >]

(* Syntactic part *)
type lambda_term = Var of char
                 | Abs of (char * lambda_term)
                 | App of (lambda_term * lambda_term)

(* plambda : lexeme stream → lambda_term *)
let rec plambda =
 parse [< 'Lambda; 'Ident v; 'Dot; plambda M >]
                    -> Abs(v,M)
     | [< plapp M >] -> M

and plapp =
 parse [< plapp M; plambda0 N >] -> App(M,N)
     | [< plambda0 M >] -> M

and plambda0 = parse [< 'Ident v >] -> Var v
                   | [< 'Lpar; plambda M; 'Rpar >] -> M
```