Architecture of an ML

Lexicographic Database Processor

Gérard Huet

INRIA

Chalmers, May 2000
What is ML?

- **ML is not Scheme**
- **ML is not Haskell**
- **ML is not SML**
- **ML is not Ocaml**
The ML tester

type intlist = list(int);

exception Left;
exception Right;

value null _ = ();

value ml_tester () =
    try (null(raise Left, raise Right);
         print_string "Nonsense!")
    with [ Left -> print_string "You win"
           | Right -> print_string "You lose"];

- 3 -
I shall deny having shown you this

```caml
value my_favorite_hello_world_in_Caml =
let the_inverting_demon _ = () in
    the_inverting_demon(print_string "World!",
                        print_string "Hello ");
```

- 4 -
What ML could be

1 Ocaml V3.0
2 Without obtrusive labels
3 With left-to-right strict evaluation
4 With a reasonable syntax
5 With a fast stream processor
6 With a well-designed parsing package
7 With a well-supported literate programming tool

Note: 4-5-6 *free* with Camlp4
The Grind interface


The Proc interface

module type Process_signature = sig

value process_header : (Sanskrit.skt * Sanskrit.skt) -> unit;
value process_entry : Dictionary.entry -> unit;
value prelude : unit -> unit;
value postlude : unit -> unit;

end;
The Grind parser

value dictionary = Grammar.create (Lexer_dict.lexer ());

EXTEND
GLOBAL: database;
database:
[ [ EOI -> () | item_rec -> () ] ] ;
item_rec:
[ [ -> () | item_rec; item -> () ] ] ;
item:
[ [ e = entry -> Process.process_entry e |
    h = header -> Process.process_header h ] ] ;
The Grind parser (more)

sanskrit:

\[ [ [ t = \text{TEXT} \rightarrow \text{trad_skt}(t) ] ] ; \]

....

syntax:

\[ [ [ e = \text{entete}; v = \text{var}; oet = \text{OPT etym} \rightarrow (e,v,oet) ] ] ; \]

usage:

\[ [ [ "vb"; t = \text{TEXT}; v = \text{vbmeanings} \rightarrow \text{Verb}(t,v) \\
  | m = \text{meaning}; lm = \text{meanings} \rightarrow \text{Subst}([m::lm]) \\
  | s = \text{semantics} \rightarrow \text{Idiomatic}(s) \\
  ] ] ; \]

entry:

\[ [ [ s = \text{syntax}; u = \text{usage}; ore = \text{OPT reletyms} \rightarrow \text{Entry}(s,u,ore) \\
  | s = \text{syntax}; c = \text{crossref} \rightarrow \text{Crossref}(s,c) \\
  ] ] ; \]
The Grind parser (end)

There are several hundred such grammar rules, which process entries represented in abstract syntax. The main procedure is:

```
value main () =
   (Process.prelude ();
   let strm = Stream.of_channel stdin in
       try Grammar.Entry.parse database strm with
         [ Exc_located loc Exit -> ()
         | Exc_located loc (Token.Error msg) -> ... 
         | ... other possible exceptions ...
         ];
   try Process.postlude () with
         [ Warning s -> output_string stderr s ];
   flush stdout; report_statistics());
```
The Dictionary interface (abstract syntax)

module Dictionary : sig

type gender = [ Mas | Neu | Fem ];
type number = [ Singular | Dual | Plural ];
type case = [ Nom | Acc | Ins | Dat | Abl | Gen | Loc | Voc ];

type voice = [ Active | Reflexive ]
and mode = [ Indicative | Imperative | Causative | Intensive | Desiderative ]
and tense = [ Present of mode | Perfect | Imperfect | Aorist | Future ]
and nominal = [ Pp | Ppr of voice | Ppft | Ger | Infi | Peri ]
and verbal = [ Conjug of (tense * voice)
            | Passive | Optative of voice | Nominal of nominal
            | Absolutive | Conditional | Precative
            | Derived of (verbal * verbal) ];

- 10 -
The TeX Printing Process

module Dictex = Grind(Process_tex);


module Process_tex = Print_dict(Print_tex);

module Print_tex : Print.Printer_signature;

module Print_dict : functor(Printer:Print.Printer_signature)
  -> Proc.Process_signature;
The Printer Signature

module type Printer_signature = sig

value ps : string -> unit;
value pc : char -> unit;
value pi : int -> unit;
value line : unit -> unit;
value space : unit -> unit;
value skip : unit -> unit;
...

- 12 -
The Generic Dictionary Printer

module Print_dict (Printer:Print.Printer_signature) = struct

value print_gender = fun
  [ Mas -> ps "m."
  | Neu -> ps "n."
  | Fem -> ps "f."
  ];

value print_number = fun
  [ Singular -> ps "sg."
  | Dual   -> ps "du."
  | Plural -> ps "pl."
  ];

... a lot of generic prettyprinting procedures
The HTML Printing Process

module Dichtml = Grind(Process_html);


module Process_html = Print_dict(Print_html);

module Print_html : Print.Printer_signature;
Binding occurrences, no dangling pointers

module Print_html = struct

value print_skt s =
    (ps "<A CLASS="defb"; HREF=""
         print_anchor_in_file s; ps ""><I>
        ps (skt_to_html s); ps "</I></A>
        if mode.val=Checking then check s else ());

value print_skt_def s =
    (ps "<A CLASS="defg"; NAME=""
        print_anchor s; ps ""><I>
        ps (skt_to_html s); ps "</I></A>
        if mode.val=Recording then record s else ());
Another Process, for declension computations


module Process_decl = struct

value genders = ref (Empty:(trie (skt * gender)));

value record_stems s l =
    genders.val:=enter (code_skts s) l genders.val;

...

A second pass constructs the declined forms from grammar tables, in a big trie.
Tries for dictionary index structure

Tries store sparse sets of codes in lexical order with maximal prefix sharing.

module Trie = struct

type code = list int;

type trie =
  [ Empty
    | Spread of list node
  ]
and info = (int * bool)
and node = (info * trie);

Visit polytrie for more complex indexing maps.
Tries as Zippers

*Warning* Fragile audience discouraged.


type zipper =
    [ Top
        | Zip of ((list node) * zipper * info * (list node))
    ]

and state = (zipper * trie);

A state records the editing context as a zipper and the current subtrie.
value enter code t = enter_edit code t Top
  where rec enter_edit c t z = match c with
  [ [] -> close_zip t z
  | [n::rest] -> match t with
    [ Empty -> close_zip (trie_of c) z
    | Spread(nodes) -> let (left,right) = split n nodes
      in match right with
        [ [] -> close_zip Empty (Zip(left,z,(n,rest=[])),[]))
        | [((m,b),u)::upper] ->
          if m=n then
            if rest=[] then close_zip u (Zip(left,z,(n,True),upper))
            else enter_edit rest u (Zip(left,z,(m,b),upper))
          else close_zip (trie_of rest) (Zip(left,z,(n,rest=[])),right))
  ]
  ]
];
Summarizing

• Lexicographic data base = ASCII ISO-LATIN text file
• Syntax = Pseudo-TeX-macros \foo{x,y,z} YAML
• Could be converted to XML trivially <foo> x y z </foo>
•Parsed by the grinder into ML abstract syntax Foo(x,y,z)
• ML is used as lexer/parser tool (no need of LeX/Yacc/…)
• ML is also used as scripting language (no need of Sed/Awk/Perl)
• ML is an efficient truly general-purpose programming language
• Modular programming is fun!