MBase tutorial

Meta Alternative Ltd.

Jul. 2008

Contents

1 How to design low level mini–languages 2
1.1 00calc.al: immediate interpretation 3
1.2 01calc.al: two passes 4
1.3 02calc.al: extending the language 6
1.4 03calc.al: compiler 8
1.5 04calc.al: IL compiler 9
1.6 05calc.al: register machine target 10

1 How to design low level mini–languages

MBase is originally designed as a framework for developing various domain specific languages. It provides many high level features and many precooked language components, but often one needs a simple, low level interpreter or translator. Here we will show the variety of approaches to such a low level design in MBase, from a simple interpreter to a compiler targeting complicated register machine.

This quick tour will cover some of the most useful MBase programming techniques, patterns and idiomatic designs. Each section below is a literate program. See the attached files `00calc.al` to `05calc.al` for details.

1.1 00calc.al: immediate interpretation

This version of a calculator mini–language consists of a parser only. This is the simplest, widely used design, which is suitable for many types of interpreted DSLs.

MBase provides a number of parsing tools. The most basic one is `<r>` macro, which is mostly used in the form of short regular expressions that recognises lexemes. Our lexer will need a regular expression to recognise the floating point numbers syntax, so here is a definition:

```
(define p.double
  (<r> ((p.digit +*) (?? ("." (p.digit +*))))
       -> list->string))
```

`<r>` is a special mini–language macro. It unrolls into a code, which uses recursive descent parsing combinators. “ `+*`” postfix operator generates “one–or–many” combinator, “ `??`” is for “maybe” combinator, “ `->`” infix defines a transformation combinator for a parsing result, in our case – collecting recognised characters into a string.

A higher level macro `make-simple-lexer` is provided for defining lexers. It is suitable in case your language has whitespaces to ignore (including comments), identifiers, keywords which forms a subset of identifiers, simple string lexemes and regular expression recogniseable lexemes. Most languages fits into this scheme, including even 2–dimensional syntaxes like Python.

```
(make-simple-lexer calclexer
```

Simple lexemes are just strings. Definitions goes as pair, at first a string and then a token. I.e., here “ `-`*” lexeme produces* `MINUS` *token.*

```
  (simple-tokens
   "-" MINUS "(" LB ")" RB
   )
```

These are more complex cases, token recognisers are defined using the language of `<r>` *macro. Definitions are also paired in the same way as for* `simple-tokens`*.*

```
  (regexp-tokens
   (("+") -> list->symbol) OP1
   (("*" | "/") -> list->symbol) OP2
   p.double number)
```

And here are regular expressions to ignore totally, i.e. whitespaces:

```
  (ignore p.whitespace)
  )
```

Tokens from a lexer are represented as symbols (N.B. `list->symbol` used above), so we can dispatch them into a floating point arightmetic functions easily:

```
(function getop (x)
  (case x
    ((+) f+) ((-) f-) ((*) f*) ((/) f/)))
```

And now we will implement a parser and an interpreter in single pass. MBase provides a special mini–language for defining simple parsers in BNF–like format. First argument of `bnf-parser` macro is a list of entry points to be exported as functions. It means that `calcparser` will be defined for a parser that accepts `expr` syntax. All the rest inside `bnf-parser` is a list of entries, each entry contains patterns and expressions to be evaluated if a pattern is matched.

```
(bnf-parser ((expr calcparser))
```

Top level expression entry, recognising low priority operations ("+", "-") first. "`MINUS`" is a separate entity here because it can be seen as unary operation as well.

```
  (expr
   ((term:l MINUS expr:r)   (f- l r) )
   ((term:l OP1:o expr:r)   ((getop o) l r) )
   ((term)                   $0))
```

Intermediate expressions entry, dealing with highest priority operations ("", "/").*

```
  (term
   ((fact:l OP2:o term:r)   ((getop o) l r))
   ((fact)                   $0))
```

Atomic expressions, including unary negation.

```
  (fact
   ((LB expr:x RB)           x)
   ((number)                (flt:parse $0))
   ((MINUS fact:e)          (f- (f# "0.0") e)))
  )
```

And now we can use this interpreter by calling a special function `lex-and-parse`, which applies a lexer to a string or a stream and applies a given parser to a resulting tokens stream.

```
(writeline
 (lex-and-parse
  calclexer
  calcparser
  "(2+2*2)/1.1"))
```

1.2 01calc.al: two passes

Now, to illustrate a more complex approach, we will separate a parsing pass and an evaluation pass. To do this, we need to define an abstract syntax tree structure. MBase provides a special mini–language for defining algebraic data types and verifiable transforms over them: `def:ast` macro.

```
(def:ast calc01 ( )
  (*TOP* <expr>) ; An entry node
  (expr ; Single variant node
   (|
    (plus <expr:a> <expr:b>)
    (minus <expr:a> <expr:b>)
    (mult <expr:a> <expr:b>)
    (div <expr:a> <expr:b>)
    (const <number:v>))))
```

If you have programmed in ML or Haskell, you will find the definition above somewhat familiar.

`expr` is a variant data type, and `plus`, `minus`, etc. are constructor tags. `expr` is a recursive type, referencing to itself.

We need such a definition not only for a documenting purposes, but also for defining visitors and iterators over these AST structures. The following function, `eval`, takes `calc01` AST as a source and returns a floating point number, an interpreted value of this tree. In languages like ML you would have to write a recursive function and use pattern matching, but MBase way is different — a visitor recursive function will be generated automatically, and you only have to declare certain nodes and variant entries transforms explicitly.

```
(function eval (e)
 ; Visits e assuming it contains expr node:
  (calc01:visit expr e
   (expr DEEP ; Transforms expr nodes using depth–first strategy
    ((const v) ; all expr variants are listed here
     (plus (f+ a b))
     (minus (f- a b))
     (mult (f* a b))
     (div (f/ a b))
     ))))
```

Here and further we will omit definitions that are identical to previous versions, giving only modified entries. Lexer is the same as in `00calc.al`.

Our dispatch function is different now, it translates tokens into tags, not functions.

```
(function getop (x)
  (case x
    ((+) 'plus) ((-) 'minus) ((*) 'mult) ((/) 'div)))
```

And the parser is different, it produces `calc01` AST instead of evaluating values immediately.

```
(bnf-parser ((expr calcparser))
  (expr
   ((term:l MINUS expr:r)   `(minus ,l ,r) )
   ((term:l OP1:o expr:r)   `(,(getop o) ,l ,r) )
   ((term)                   $0))
  (term
   ((fact:l OP2:o term:r)   `(,(getop o) ,l ,r))
   ((fact)                   $0))
  (fact
   ((LB expr:x RB)           x)
   ((number)                `(const ,(flt:parse $0)))
   ((MINUS fact:e)          `(minus (const ,(f# "0")) ,e)))
  )
```

The usage is pretty much the same, we only add `eval` function call here:

```
(writeline (eval (lex-and-parse
                  calclexer
                  calcparser
                  "(2+2*2)/1.1")))
```

1.3 02calc.al: extending the language

Now we will introduce a somewhat more complex language with variables. The design is similar to the previous version, i.e., a two pass interpreter, with a parser generating an AST and an interpreter over the AST.

The AST is just slightly different — `let` and `var` variants are added to `expr` node:

```
(def:ast calc02 ( )
  (*TOP* <expr>)
  (expr
   (|
    (plus <expr:a> <expr:b>)
    (minus <expr:a> <expr:b>)
    (mult <expr:a> <expr:b>)
    (div <expr:a> <expr:b>)
    (let <ident:nm> <expr:val> <expr:body>)
    (var <ident:nm>)
    (const <number:v>))))
```

The interpreter is slightly more complicated, with an explicit recursion for `let` variant.

```
(function eval (ex)
  ; loop is a recursive function, starting with an empty environment:
  (let loop ((env nil) (e ex))
    (calc02:visit expr e
```

We are using a different visiting strategy here: with `DEEP` *it will process inner* *nodes first, and with " _" it will not go into recursion for listed variants. If* `else-deep` *is specified,* `DEEP` *behaviour will be turned on for the rest of patterns.*

```
      (expr _
        ((const v)
         (var  (lookup-env-car env nm))
         (let
            (loop `((,nm ,(loop env val)) ,@env)
                  body))
```

So here `const`*,* `var` *and* `let` *were processed with " _" strategy, and for* `let`*, internal subnodes* `val` *and* `body` *were processed by explicit calls to a* `loop` *function, taking care of a correct environment contents. All the rest are processed recursively with* `DEEP`*–strategy.*

```
         (else-deep
          (
           (plus (f+ a b))
           (minus (f- a b))
           (mult (f* a b))
           (div (f/ a b))
           (else nil) ; To suppress a coverage warning
           )))))))
```

A lexer is different now, it defines a syntax for identifiers and lists special identifier values which form keywords (`let` and `in`). Also a new simple token `EQ` is introduced.

```
(make-simple-lexer calclexer
  ; Defines a single pair: a regular expression to match identifiers ([[p.ident0]])
and a token name for them ([[var]])
  (ident-or-keyword p.ident0 var)
  ; Gives a list of identifiers that should be keywords with identical token
names
  (keywords let in)
  (simple-tokens
   "-" MINUS "(" LB ")" RB "=" EQ
   )
  (regexp-tokens
   (("+") -> list->symbol) OP1
   (("*" | "/") -> list->symbol) OP2
   p.double number)
  (ignore p.whitespace)
  )
```

The parser is just a little bit different from the previous one, providing two more patterns for an atomic expression entry `fact`.

```
(bnf-parser ((expr calcparser))
  (expr
   ((term:l MINUS expr:r)    `(minus ,l ,r) )
   ((term:l OP1:o expr:r)    `(,(getop o) ,l ,r) )
   ((term)                    $0))
  (term
   ((fact:l OP2:o term:r)    `(,(getop o) ,l ,r))
   ((fact)                    $0))
  (fact
   ((let var:v EQ expr:e in expr:b)
    `(let ,v ,e ,b))
   ((var)                    `(var ,$0))
   ((LB expr:x RB)            x)
   ((number)                 `(const ,(flt:parse $0)))
   ((MINUS fact:e)           `(minus (const ,(f# "0")) ,e)))
  )
```

And the usage is still the same:

```
(writeline (eval (lex-and-parse
                  calclexer
                  calcparser
                  "let x = 2*2 in let y = 1.1 in (2+x)/y")))
```

1.4 03calc.al: compiler

In previous examples the AST was interpreted. Now we will translate it into MBase language so it can be compiled and executed as MBase code.

This transform is simple again, simpler than the previous interpreter, since it generates MBase code directly and MBase itself will deal with variable bindings.

```
(function compile (ex)
    (calc03:visit expr ex
      (expr DEEP
        ((const `(f# ,v))
         (var   nm)
         (let   `(alet ,nm ,val ,body))
         (plus  `(f+ ,a ,b))
         (minus `(f- ,a ,b))
         (mult  `(f* ,a ,b))
         (div   `(f/ ,a ,b))
         ))))
```

The parser is just slightly different, numbers in it are not parsed but represented as strings instead — MBase backend will parse them later.

```
(bnf-parser ((expr calcparser))
  (expr
   ((term:l MINUS expr:r)   `(minus ,l ,r) )
   ((term:l OP1:o expr:r)   `(,(getop o) ,l ,r) )
   ((term)                   $0))
  (term
   ((fact:l OP2:o term:r)   `(,(getop o) ,l ,r))
   ((fact)                   $0))
  (fact
   ((let var:v EQ expr:e in expr:b)
    `(let ,v ,e ,b))
   ((var)                   `(var ,$0))
   ((LB expr:x RB)           x)
   ((number)                `(const ,$0))
   ((MINUS fact:e)          `(minus (const "0.0") ,e)))
  )
```

And in order to add this language to MBase and to be able to use MBase as a backend, we will define a macro. This example illustrates how blurred the border between metaprogramming and compilation is.

```
(macro calc03# (str)
  (compile (lex-and-parse calclexer calcparser str)))
```

The usage is quite trivial:

```
(writeline (calc03# "let x = 2*2 in let y = 1.1 in (2+x)/y"))
```

As a side effect of this way of compilation, all the MBase variables are visible:

```
(writeline (let ((x (f# "2.0")))
             (calc03# "(2+x*x)/1.1")))
```

1.5 04calc.al: IL compiler

And finally, here is a “real” compiler which generates .NET IL instructions directly. Surprisingly, it is almost as easy as the previous compiler that generated a high level language code, thanks to the fact that .NET is a stack machine.

`compile` function transforms an AST into a flat list of IL instructions.

```
(function compile (ex)
    (calc04:visit expr ex
      (expr DEEP
        ((const `((Ldc_R8 ,(flt:parse v))))
         (var   `((Ldloc (var ,nm))))
         (let   `((local ,nm ,t_Double)
                  ,@val
                  (Stloc (var ,nm))
                  ,@body))
         (plus  `(,@a ,@b (Add)))
         (minus `(,@a ,@b (Sub)))
         (mult  `(,@a ,@b (Mul)))
         (div   `(,@a ,@b (Div)))
         ))))
```

MBase has a possibility to inline IL code (`n.asm`), so we will wrap our compiler in a macro.

```
(macro calc04# (str)
  `(n.asm ()
      ,@(compile (lex-and-parse calclexer calcparser str))
      ; Return a boxed object:
      (Box ,t_Double)))
```

And usage is the same:

```
(writeline (calc04# "let x = 2*2 in let y = 1.1 in (2+x)/y"))
```

A curious reader can print out the compiled code:

```
(iter writeline
      (compile
       (lex-and-parse calclexer calcparser
                      "let x = 2*2 in let y = 1.1 in (2+x)/y"
                      )))
```

1.6 05calc.al: register machine target

The most complicated form of compilation in this tutorial will be targeting a register machine.

We are using an imaginable architecture, which resembles the design of some of the popular register architectures. It has only three floating point registers called R1, R2 and R3. We will also refer to a frame pointer FP and a stack pointer SP. Mov* instructions may have memory or register as destination. If mov* instruction destination is a register, a source could be a constant. For arithmetic operations, only registers could be destination and registers and memory could be used as a source.

Compilation will need a number of intermediate representations. First intermediate AST will be defined for "imperativised" version of a code, distinguishing operators from expressions. The same representation will be used for a flattened 3–address code as well.

```
(def:ast calc05x ( )
  (*TOP* <topexprs>)
  (topexprs <*topexpr:es>)
  (topexpr
   (|
    (def <ident:nm> <expr:val>)
    (return <expr:val>)))
  (expr
   (|
    (plus <expr:a> <expr:b>)
    (minus <expr:a> <expr:b>)
    (mult <expr:a> <expr:b>)
    (div <expr:a> <expr:b>)
    (var <ident:nm>)
    (const <number:v>))))
```

Nested `let`–s should be transformed into a flat sequence of definitions. A "splitter" pattern is used here. As in `02calc.al`, the visitor stops on `let` nodes, and for each sub–expression `loop` function produces a pair: a transformed sub–expression and a list of raised definitions. Since only `let` can raise, it efficiently moves all the variable declarations to the flat top level. This pattern is very common in code transforms.

```
(function stage0 (ex)
  (alet res
   (let loop ((e ex))
    (collector (raiseadd raiseget)
     (cons ; return a pair: expression itself and raised definitions list
      (calc05:visit expr e
        (expr _
          ((let
               (let ((nval (loop val))
                     (nbody (loop body)))
                 (iter raiseadd (cdr nval))
                 (raiseadd `(def ,nm ,(car nval)))
                 (iter raiseadd (cdr nbody))
                 (car nbody)))
           (else-deep ((else node))))))
      (raiseget))))
   `(,@(cdr res)
     (return ,(car res)))))
```

`stage1lift` is a helper function which lifts all the complex sub–expressions of a given expression, using a provided `add` collector. Another common trick is used here: temporary node renaming. When a visitor starts on `ex`, it assumes that it is a node of a virtual type `d_expr`, inheriting all the properties of `expr`.

But all the `expr` sub–nodes referenced from this node are still of `expr` type, and processed with `expr` pattern. This means that an outer expression is untouched, and all the sub–expressions are raised unless they are variables. For example, `(plus (const ...) (const ...))` will be translated into `(plus (var ...) (var ...)` with two raised variables bindings for constants.

```
(recfunction stage1lift (add ex)
  (calc05x:visit d_expr ex
        ((d_expr expr) DEREF (forall node))
        (expr DEEP
           ((var node)
            (else (alet nm (gensym)
                        (add `(def ,nm ,(stage1lift add node)))
                        `(var ,nm)))))))
```

The next function will lift inner complex expressions in sequence: innermost lifts will be processed first.

```
(function stage1 (tex)
  (collector (add get)
    (foreach (ex tex)
      (add (calc05x:visit topexpr ex
              (expr _
                    (forall (stage1lift add node))))))
    (get)))
```

`stage2schedule` function compiles a flat code into an MBase register scheduling mini–language in order to get an optimal variables distribution. It is easy since we only have one datatype: a double precision floating point number.

```
(function stage2schedule (tex)
  (calc05x:visit topexprs tex
    (topexpr DEEP
      ((def `(genkill ((V ,nm double)) ,val))
       (return `(gen ,@val))))
    (expr DEEP
      ((const nil)
       (var `((V ,nm double)))
       (plus `(,@a ,@b))
       (minus `(,@a ,@b))
       (mult `(,@a ,@b))
       (div `(,@a ,@b))))))
```

Now we will rename variables using the provided register scheduling plan. A context sensitive `ast:mknode` macro is used here. It is useful only withing transforms that translate an AST into another AST of the same structure. `ast:mknode` generates another instance of a currently processed node or variant, changing only named sub–node values.

```
(function stage2 (tex)
  (let* ((plan (stage2schedule tex))
         (all (r3:allocateregisters nil (r3:lgraphs plan)))
         (newnm (fun (n) (hashget all n))))
    (calc05x:visit topexprs tex
       (expr DEEP
         ((var (ast:mknode (nm (newnm nm))))
          (else node)))
       (topexpr DEEP
         ((def (ast:mknode (nm (newnm nm))))
          (else node))))))
```

Perform all stages above in the right order:

```
(function compile (e)
  (stage2 (stage1 (stage0 e))))
```

Our imaginable register target machine has only 3 floating point registers, and all other variables should be spilled into a stack frame.

Here we will just count the register use frequency in order to decide what to spill.

```
(function stage3count (es)
  (with-hash (ht)
    (calc05x:iter topexprs es
      (expr DEEP
         ((var (alet h0 (alet h00 (ht> nm) (if h00 h00 0))
                       (ht! nm (+ h0 1))))
          (else nil))))
```

At this point `ht` *contains variable usage frequencies, so we can now sort the list in the descending order.*

```
    (qsort (fun (a b) (> (car a) (car b)))
           (hashmap (fun (nm cnt) `(,cnt ,(Sm<< nm))) ht))))
```

If we have just 3 variables, all are mapped to registers. Otherwise one register is reserved for spills, and two others are bound. This is not the most efficient way, but all the complex optimisations are beyond the scope of this tutorial.

In a simple case variables are just renamed to registers.

```
(function stage3rename (es mp)
  (with-hash (ht)
    (iter-over mp (fmt (nm vl) (ht! nm vl)))
    (calc05x:visit topexprs es
        (topexpr DEEP
          ((def (ast:mknode (nm (ht> nm))))
           (else node)))
        (expr DEEP
          ((var (ast:mknode (nm (ht> nm))))
           (else node))))))
```

Otherwise we have to perform a spilling. R3 is a temporary substitution for variables that are spilled out.

```
(function stage3spills (es r1 r2 sps)
  (collector (add get)
  (with-hash (ht)
    (iter-over sps (fmt (nm num) (ht! nm `(SP ,num))))
    (ht! r1 'R1)
    (ht! r2 'R2)
    (calc05x:visit topexprs es
        (topexpr DEEP
           ((def (if (list? (ht> nm))
                     (begin
                       (add `(def R3 ,val))
                       (add `(mov R3 ,(ht> nm))))
                     (add (ast:mknode (nm (ht> nm))))))
            (else (add node))))
        (expr DEEP
           ((var (ast:mknode (nm (ht> nm))))
            (else node)))))
  (get)))
```

`stage3` *function decides wheter it is renaming or spilling. It returns a pair: null or a number of spilled variables and the generated intermediate code.*

```
(function stage3 (es)
  (alet count (stage3count es)
    (cond
     ((< (length count) 4)
      (cons nil
       (stage3rename es (zip (map cadr count)
                             '(R1 R2 R3))))
      )
     (else
      (format count ((_ r1) (_ r2) . spilled)
        (cons (length spilled)
         (stage3spills es r1 r2
                       (zip (map cadr spilled)
                            (fromto 0 (length spilled)))))
        )))))
```

The code above has generated a slightly modified version of our AST, introducing a new toplevel instruction, so here we are patching it to reflect this changes.

```
(def:ast calc05t ( (calc05x) )
  (topexpr
   (|
    (mov <ident:n1> <ident:n2>)
    (def <ident:nm> <expr:val>)
    (return <expr:val>))))
```

stage4var emits a register or frame pointer reference argument.

```
(function stage4var (nm)
  (p:match nm
    ((SP $cnt) ; spilled into stack
     (if (> cnt 0)
         (S<< "[FP:" (* 8 (cadr nm)) "]")
         "[FP]"))
    (else (S<< nm))))
```

Next function emits simple instructions.

```
(function stage4emit (instr tgt)
  (calc05x:visit d_expr instr
    ((d_expr expr) DEREF
      ((var
        (if (eqv? nm tgt) "NOP"
            (S<< "MOVR8 " (stage4var nm) ", " tgt)))
       (const (S<< "MOVR8 #F" v ", " tgt))
       (plus (S<< "ADDR8 " a ", " b ", " tgt))
       (minus (S<< "SUBR8 " a ", " b ", " tgt))
       (mult (S<< "MULR8 " a ", " b ", " tgt))
       (div (S<< "DIVR8 " a ", " b ", " tgt))))
    (expr DEEP
       ((var (stage4var nm))
        (else nil)))
    ))
```

And to bind everything together, to emit correct stack frame initialisation and return statement:

```
(function stage4 (cnt es)
  (collector (add get)
    (if cnt (begin
              (add "MOVL SP, FP")
              (add (S<< "ADDL SP, " (* 8 cnt) ", SP"))))
    (foreach (e es)
      (calc05t:iter topexpr e
        (topexpr _
         ((mov (add (S<< "MOVR8 " (stage4var n1)
                         ", "  (stage4var n2))))
          (def (add (stage4emit val nm)))
          (return (begin
                    (add (stage4emit val 'R1))
                    (if cnt (add "MOVL FP, SP"))
                    (add "PUSHR8 R1")
                    (add "RET")))))))
    (get)))
```

emit is the interface function which takes a parsed AST and returns a list of assembler language strings.

```
(function emit (src)
  (alet c (stage3 (compile src))
     (stage4 (car c) (cdr c))))
```

And we can now print out the compiled code. Unlike other examples, there is no actual evaluation here, our target machine does not exist in reality.

```
(define code
  (S<< "let x = let t = 1+1+1 in t*t in "
         "let y = 1.1+(-x*t) in "
         "(1-y)+x/2+x*x-(2+x)/y"))

(define src
  (lex-and-parse calclexer calcparser code))

(iter println (emit src))
```