

Minim Compiler Example

Meta Alternative Ltd.

Oct. 2007

Minim compiler is an implementation of Minim language, which was suggested in `comp.lang.lisp` newsgroup as a shootout for compiler and interpreter implementation techniques. The language itself is quite useless and very simple, but it is suitable for demonstrating some basic programming approaches, typical for Domain Specific Languages implementations within MBase framework.

This must be a first expression in the code: instruction for compiler to build an exe assembly.

```
(n.module mcomp exe)
```

We can use an algebraic data type to define an abstract syntax tree for Minim language. It reflects a source in a very straightforward way.

```
(def:ast minim ()
; entry point, used internally:
  (*TOP* <program>)
; Program is a list of statements
  (program <*statement:sts>)
; Statement is a variant
  (statement
    (| (Ass <ident:var> <val:val>)
      (++) <ident:var>
      (--) <ident:var>
      (Cnd <test:tst> <statement:tr> <statement:fl>)
      (Gto <ident:tag>)
      (Tag <ident:tag>)
      (PrntStr <string:val>)
      (PrntVal <val:val>)
      (PrntNL)
      (Input <ident:v>)
    ))
; Test is a conditional expression inside if.
  (test
    (| (Comp cmp <val:left> <val:right>)
      (And <test:left> <test:right>)
      (Or <test:left> <test:right>)
      (Not <test:t>)
    ))
; Value is either a variable or a number.
  (val (| (V <ident:v>) (C num)))
)
```

Since we are going to generate .NET IL code we will need some method stubs to print strings and numbers and read numbers from standard input, as defined in Minim language specification.

```
(define _print_mtd (r_mtd "System.Console" "Write" object))
(define _readline_mtd (r_mtd "System.Console" "ReadLine"))
(define _parse_mtd (r_mtd "System.Int32" "Parse" string))
```

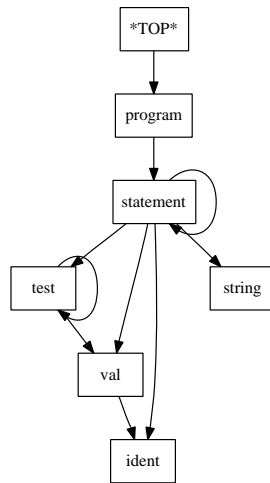


Figure 1: Minim AST

Now the interesting bit: compiling the AST defined above into a flat list of .NET IL instructions. It is easy to compile into a stack based VM, since order does not matter, and we can use a “visitor” to rebuild the source AST into an instructions list. Each statement node is substituted with a list of relevant instructions, and all the nested statements are already compiled (DEEP visitor processes the deepest nodes first) and could be just appended into relevant places of the compiled code.

```

(function minim->cli ( expr )
  (<> expr ; Apply something to [[expr]].
  (ast:visit minim program
    (program DEEP ; flatten the compiled toplevel statements list
      (foldl append '() sts))

```

statement is a variant, so we must mention all the possible node types here.

```

(statement DEEP (
; Minim variables are represented as .NET local variables.
  (Ass '((local ,var ,t_Int32)
        ,@val
        (Stloc (var ,var))))
; Increment is straightforward: load a value, add 1, store it back.
  (++) '((Ldloc (var ,var))
        ,(_ldc_i4 1)
        (Add)
        (Stloc (var ,var))))
; Same for decrement
  (--) '((Ldloc (var ,var))
        ,(_ldc_i4 1)
        (Sub)
        (Stloc (var ,var))))

```

Condition statement is compiled with two generated unique labels and an appropriate conditional jump. Test is a special expression, not a statement.

```

(Cnd (with-syms (lend lf1)
      '(@tst
        (Brfalse (label ,lf1))
        ,@tr
        (Br (label ,lend))
        (label ,lf1)
        ,@fl
        (label ,lend)
        )))
; Another one straightforward transform
  (Gto '((Br (label ,tag))))
; And goto tags are naturally labels.
  (Tag '((label ,tag)))
; This is where we need that method stubs.
  (PrntStr
    '((Ldstr ,val) (Call ,_print_mtd)))
  (PrntVal
    '(@val (Box ,t_Int32)
      (Call ,_print_mtd)))
  (PrntNl
    '((Ldstr "\n") (Call ,_print_mtd)))

```

Minim input declares a new local variable. We do not bother handling any input and parsing errors here, user will be happy enough to see a generic .NET unhandled exception message.

```

      (Input
        '((local ,v ,t_Int32)
          (Call ,_readline_mtd)
          (Call ,_parse_mtd)
          (Stloc (var ,v))))))
; Handling local variables and numeric constants.
      (val DEEP (
        (V '((Ldloc (var ,v))))
        (C '(',_ldc_i4 num))))))

```

Test is another variant node. It is always compiled into .NET IL code producing an unboxed boolean on stack.

```

      (test DEEP (
        (Comp
          '(@left ,@right
            ,(case cmp ((>) '(Cgt)) ((<) '(Clt)) ((=) '(Ceq))))))
        (And '(@left ,@right (And)))
        (Or '(@left ,@right (Or)))
        (Not '(@left ,@right (Not))))))
      )))

```

MBase is a Lisp-like language and it obviously have a pre-cooked S-expressions parser which we can reuse. But here we will take another route and implement a parser from scratch: normally any DSL will have its own syntax, and it is important to have an easy way of implementing simple parsers. As a free advantage we will have some more syntax validity checks than with a generic S-expressions input.

```

; Simple lexer: splits a stream into a list of tokens.
(make-simple-lexer minim-lexer
; Most languages have the same syntax for keywords and identifiers.
  (ident-or-keyword
    (p.alpha ((p.alpha | p.digit) *))
    ident)
; Now we can choose the subset of identifiers to be recognised as keywords.
  (keywords input print nl goto if then else and or not is)
; Some characters or substrings forms simple tokens.
  (simple-tokens
    "[" LB "]" RB
    "(" LB ")" RB
    ">" > "<" < "=" = "++" ++ "--" --)
; Some tokens are defined by more complicated regexps:
  (regexp-tokens
    ("\" \" ((#\ \\ #\ ") | (! #\ ") *) "\" ->
      (M@ list->string cuttail cdr)) string
    ("\' \" ((#\ \\ #\' ) | (! #\' ) *) \"\' ->
      (M@ list->string cuttail cdr)) string
    p.integer.p          number)
; And all the whitespace characters are ignored.
  (ignore p.whitespace))

```

Lexer produces a flat stream of tokens, which should be converted now into an AST of a structure defined above. LL(1) pushdown automaton will do this job.

```

(bnf-parser ((prograg parse-minim))
; In Minim examples all programs was enclosed into parenthesis, so let's follow
this convention:
  (prograg
    ((LB program RB) $1))

```

Program is a list of statements or just one statement.

```

(program
  ((statement program) (cons $0 $1))
  ((statement) (list $0)))

```

All the different statement flavours can be defined in a single BNF pattern:

```

(statement
  ((LB ident:va is val:v1 RB) '(Ass ,va ,v1))
  ((LB ++ ident:va RB) '(++ ,va))
  ((LB -- ident:va RB) '(-- ,va))
  ((LB goto ident:tag RB) '(Gto ,tag))
  ((LB if test:tst then statement:s1 else statement:s2 RB)
   '(Cnd ,tst ,s1 ,s2))
  ((LB print string:str RB)
   '(PrntStr ,str))
  ((LB print val:v RB)
   '(PrntVal ,v))
  (nl)
  '(PrntNl))
  ((LB input ident:va RB)
   '(Input ,va))
  ((ident)
   '(Tag , $0)))

```

Any value is either an identifier (variable name) or a numerical literal (a constant)

```

(val
  ((ident) '(V , $0))
  ((number) '(C , $0)))

```

Tests are different from other statements and have a special infix syntax.

```

(test
  ((LB val:v1 comp:c val:v2 RB) '(Comp ,c ,v1 ,v2))
  ((LB test:l and test:r RB) '(And ,l ,r))
  ((LB test:l or test:r RB) '(Or ,l ,r))
  ((LB not test:t RB) '(Not ,t)))

```

; Minim allows only the following comparison operations:

```

(comp
  (<) '<
  (>) '>
  (=) '=)
)

```

That is all with the declarative part of Minim implementation, now it is a time for some low level programming. We will use the possibility of including a .NET IL code into MBase expressions to sneak our new compiled Minim code into the code generated by MBase compiler backend. The following macro creates a function of a given name, stuffed with .NET IL produced from a Minim source read from a given file. Any exception — syntax or parsing error, file input error, whatever else will be reported gracefully, and a doing-nothing code will be produced instead.

```

(macro include-minim-f (nm fname)
  (try
    '(function ,nm ()
      (n.asm ()
        ,@(minim->cli
          (lex-and-parse minim-lexer parse-minim
            (read-file-list fname)))
        (Ldnull)))
      t_MBaseException
      (fun (e)
        (writeln '(Exception in minim loader: ,(mbaseerror e)))
        'nil)))

```

And in order to be able to produce standalone executables out of Minim programs we can reuse the MBase compiler: instruct it to build an exe module, generate a `main` function with a compiled Minim code and force a dumping of all the generated code into an exe file.

```

(function main ( )
  (let ((fnm ([ 0 ] *CMDLINE*)))
    (read-int-eval '(n.module minim exe))
    (read-compile-eval
      '(include-minim-f main ,fnm))
    (read-int-eval '(save-module))
  ))

```

That's all. Now we have a compiler, capable of producing standalone .NET executables for a toy language Minim. Most real world DSLs are in fact not much more complicated than a toy Minim, sometimes even simpler, and all the techniques seen above can be efficiently applied to any real world DSL implementation task.