```lisp
(in-package "MESYSTEM")

; defuna


(defun translate (arg)
  (cond ((cl:listp arg) (mapcar #'(lambda (z) (translate z)) arg))
        ((cl:numberp arg) (cl-fixnum arg))
        (t arg)))

(defmacro defuna (&rest body)
  `(defun ,@(translate body)))

(defmacro defparametera (&rest body)
  `(defparameter ,@(translate body)))


;

(defmacro watch (var)
  (list 'format 't (concatenate 'string (cl:symbol-name var) " = ~S~% ") var))



; die translatoren

(setf *max-cont* (- (expt 2 32) 1))

(defun two-comp30 (x)
  (if (>= x (expt 2 29))
      (- x (expt 2 30))
    x))

(defun fixnum-cl (i)
  (two-comp30 (ash i -2)))

(defun cl-fixnum (i)
  (logand (ash i 2) *max-cont*))


(defun cl-to (x)
  (cond ((cl:null x) nil)
        ((cl:characterp x) (cl-char x))
        ((cl:numberp x) (cl-fixnum x))
        ((cl:stringp x) (cl-string x))
        ((cl-user::is-pretemplate x) (cl-user::asm-templ x))
        ((cl:consp x) (cl-cons x))
        ((cl:arrayp x) (cl-array x))
        ((cl:symbolp x) (cl-symbol x))))

(defun cl-cons (p)
  (cons (cl-to (cl:car p)) (cl-to (cl:cdr p))))

(defun cl-char (char)
  (code-char (cl-fixnum (cl:char-code char))))


(defun cl-symbol (sym)
  (intern (cl-to (cl:symbol-name sym))))
```



```lisp
(defun cl-string (str)
  (let ((len (cl:length str)))
    (let ((new-str (make-string (cl-fixnum len))))
      (do ((i 0 (+ i 1)))
          ((>= i len) new-str)
        (set-char new-str (cl-fixnum i) (cl-char (cl:char str i)))))))

(defun cl-array (arr)
  (let ((len (cl:length arr)))
    (let ((new-arr (make-vector (cl-to len))))
      (do ((i 0 (+ i 1)))
          ((>= i len) new-arr)
        (set-svref new-arr (cl-to i) (cl-to (cl:aref arr i)))))))

(defun cl-closure (clos)
  )

(defun cl-template (templ)
  )



; me nach cl

(defun to-cl (x)
  (cond ((null x) cl:nil)
        ((characterp x) (char-cl x))
        ((numberp x) (fixnum-cl x))
        ((stringp x) (string-cl x))
        ((consp x) (cons-cl x))
        ((arrayp x) (array-cl x))
        ((symbolp x) (symbol-cl x))
        ((closurep x) (closure-cl x))
        ((templatep x) (cl-user::disasm-templ x))))



(defun char-cl (c)
  (cl:code-char (fixnum-cl (char-code c))))

(defun string-cl (s)
  (let ((len (fixnum-cl (length s))))
    (let ((str (cl:make-string len)))
      (do ((i 0 (+ i 1)))
          ((>= i len) str)
        (setf (cl:char str i) (char-cl (char s (cl-fixnum i))))))))

(defun cons-cl (p)
  (cl:cons (to-cl (car p)) (to-cl (cdr p))))

(defun array-cl (v)
  (let ((len (fixnum-cl (length v))))
    (let ((arr (cl:make-array len)))
      (do ((i 0 (+ i 1)))
          ((>= i len) arr)
        (setf (cl:aref arr i) (to-cl (svref v (cl-fixnum i))))))))

(defun symbol-cl (p)
  (let ((np (symbol-name p)))
    (cl:make-symbol (string-cl np))))

(defun closure-cl (p)
  (list '&closure (to-cl (closure-ref p 0)) (to-cl (closure-ref p 4))))

(defun template-cl (p)
```



```lisp
  (let ((d-len (fixnum-cl (length-d p))))
    (let ((arr (cl:make-array d-len)))
      (do ((i 0 (+ i 1)))
          ((>= i d-len))
        (setf (cl:aref arr i) (to-cl (templ-ref-d p (cl-fixnum i)))))
      (list '&template arr))))













(defun log-to (x)
  (cond (x (cl-to 1))
        (t nil)))


(defparametera *tst* 7)
(defuna tst (n) (* n 9))


(setf nil 0)
(setf *alloc-pointer* 65536)


; memory-funktionen

(defparameter *max-cont* (- (expt 2 32) 1))

(defparameter *addr-mask* (- (expt 2 32) 1 3))
(defparameter *tag-mask* 3)

(defparameter *base* 256)



(defun shiftl (n i)
  (* n (expt 2 i)))

(defun shiftr (n i)
  (let ((x (floor (/ n (expt 2 i)))))
    x))

(defun align (x i)
  (let ((mask (- (expt 2 i) 1)))
    (let ((low (logand x mask)))
      (if (zerop low)
          x
        (+ (logand x (- *max-cont* mask)) mask 1)))))

(defun get-byte (paddr offset)
  (let ((offseti (fixnum-cl offset)))
```



```lisp
    (let ((peff (+ (logand paddr *addr-mask*) offseti)))
      (shiftl (aref *memory* peff) 2))))

(defun get-hword (paddr offset)
  (let ((offseti (fixnum-cl offset)))
    (let ((peff (+ (logand paddr *addr-mask*) offseti)))
      (cl-fixnum (+ (* (aref *memory* peff) *base*) (aref *memory* (+ peff 1))))
)))

(defun get-word (paddr offset)
  (let ((offseti (fixnum-cl offset)))
    (let ((peff (+ (logand paddr *addr-mask*) offseti)))
      (+ (* (fixnum-cl (get-hword peff 0)) *base* *base*) (fixnum-cl (get-hword
peff (cl-fixnum 2)))))))


(defun set-byte (paddr offset i)
  (let ((offseti (fixnum-cl offset)))
    (let ((peff (+ (logand paddr *addr-mask*) offseti)))
      (setf (aref *memory* peff) (shiftr i 2)))))

(defun set-hword (paddr offset i)
  (let ((offseti (fixnum-cl offset)))
    (let ((peff (+ (logand paddr *addr-mask*) offseti)))
      (setf (aref *memory* peff) (logand (shiftr i 10) 255))
      (setf (aref *memory* (+ peff 1)) (logand (shiftr i 2) 255))
      i)))

(defun set-word (paddr offset pval)
  (let ((offseti (fixnum-cl offset)))
    (let ((peff (+ (logand paddr *addr-mask*) offseti)))
      (set-hword peff 0 (cl-fixnum (shiftr pval 16)))
      (set-hword peff (cl-fixnum 2) (cl-fixnum (logand pval 65535)))
      pval)))

; unsichere Funktionen

(defun p-constr (v-high v-low)
  (+ (shiftl v-high 14) (shiftr v-low 2)))

(defun p-add (p i)
  (+ p (fixnum-cl i)))

(defun p-shiftl (p i)
  (shiftl p (fixnum-cl i)))

(defun p-shiftr (n i)
  (shiftr n (fixnum-cl i)))

(defun p-align (x i)
  (align x (fixnum-cl i)))


; tag-opx bezieht sich auf die Bits 0 und 1

(defun tag-and (p tag-code)
  (let ((tag-codei (fixnum-cl tag-code)))
    (logior (logand p *addr-mask*) (logand (logand p *tag-mask*) (logand tag-cod
ei *tag-mask*)))))

(defun tag-andc1 (p tag-code)
  (let ((tag-codei (fixnum-cl tag-code)))
    (logior (logand p *addr-mask*) (logandc1 (logand p *tag-mask*) (logand tag-c
odei *tag-mask*)))))
```

```
(defun tag-ior (p tag-code)
  (let ((tag-codei (fixnum-cl tag-code)))
    (logior (logand p *addr-mask*) (logior (logand p *tag-mask*) (logand tag-cod
ei *tag-mask*)))))


(defun tag-xor (p tag-code)
  (let ((tag-codei (fixnum-cl tag-code)))
    (logior (logand p *addr-mask*) (logxor (logand p *tag-mask*) (logand tag-cod
ei *tag-mask*)))))


(defun tag-not (p)
  (logior (logand p *addr-mask*) (lognot (logand p *tag-mask*))))


(defun tag-eq (p tag-code)
  (let ((tag-codei (fixnum-cl tag-code)))
    (= (logand p *tag-mask*) (logand tag-codei *tag-mask*))))

(defun get-tag (p)
  (shiftl (logand p *tag-mask*) 2))

(defun tag-set (p tag-code)
  (let ((tag-codei (fixnum-cl tag-code)))
    (logior (logand p *addr-mask*) (logand tag-codei *tag-mask*))))


; addr-opx bezieht sich auf die Bits 2 bis 31
; (diese bits machen eine fixnum aus)

(defun addr-plus (p i)
  )


(defun addr-minus (p i)
  )

(defun addr-eq (p1 p2)
  (= (logand p1 *addr-mask*) (logand p2 *addr-mask*)))

(defun addr-eqi (p1 i)
  (= (logand p1 *addr-mask*) i))

(defun addr-set (p1 p2)
  )


(defun addr-and (p i)
  (logior (logand p i) (logand p *tag-mask*)))

(defun addr-andc1 (p i)
  (logior (logand p *tag-mask*) (logandc1 (logior (logand p *addr-mask*) *tag-ma
sk*) i)))

(defun addr-ior (p i)
  (logior (logior p i) (logand p *tag-mask*)))


(defun addr-xor (p i)
  )

(defun addr-not (p)
  )
```



```
(defun addr-shiftl (p i)
  )

(defun addr-shiftr (p i)
  )

(defun get-addr (p)
  (logand p *addr-mask*))


(defparameter *low-byte-mask* 63) ; 11.11.11|00

; neu zu definierende Kontrollstrukturen im Interpreter



; vergleichsfunktionen allgemein


(defun eq (x y)
  (= x y))

(defuna null (x)
  (eq x nil))

(defuna atom (x)
  (not (consp x)))

; logische funktionen


(defuna not (x)
  (cond ((null x) t)
        (t nil)))



; die conses, lese und schreibfunktionen


(defuna cons (x y)
  (let ((p (%allocate-space 8)))
    (set-word p 0 x)
    (set-word p 4 y)
    (setf p (tag-set p 1))
    p))


(defuna car (x)
  (if (null x)
        nil
      (get-word x 0)))


(defuna cdr (x)
  (if (null x)
      nil
    (get-word x 4)))


(defuna set-car (x v)
  (set-word x 0 v))
```



```
(defuna set-cdr (x v)
  (set-word x 4 v))

; die conses, testfunktionen

(defparametera *cons-tag* 1)

(defuna consp (x)
  (tag-eq x *cons-tag*))

(defuna listp (x)
  (or (null x) (consp x)))



; die strings, lesen und schreiben von zeichenpositionen


(defparametera *string-head-tag* 2)
(defparametera *string-tag* 2)

(defuna make-string (n)
  (let ((len (%div (%align n 2) 4)))
    (let ((n-align (%align (%+ (%* len 4) 8) 3)))
      (let ((p (%allocate-space n-align)))
        (set-word p 0 (tag-set (addr-ior (p-shiftl n-align 2) 1) *string-head-ta
g*))
        (set-word p 4 n)
        (do ((i 0 (%+ i 1)))
            ((%>= i n))
          (set-char p i (code-char 0)))
        (setq p (tag-set p *string-tag*))
        p))))


(defuna char (str i)
  (let ((byte-val (get-byte str (%+ i 8))))
    (code-char byte-val)))

(defuna set-char (str i c)
  (let ((byte-val (char-code c)))
    (set-byte str (%+ i 8) byte-val)))


; die strings, testfunktionen


(defuna stringp (x)
  (and (tag-eq x *string-tag*)
       (let ((h (get-word x 0)))
         (and
          (addr-eqi (addr-and h 2) 0)
          (tag-eq h *string-head-tag*)))))



; die strings, vergleichsfunktionen

(defuna length (s)
  (if (or (stringp s) (arrayp s))
      (get-word s 4)
    -1))
```



```
(defuna string< (x y)
  )

(defuna string> (x y)
  )

(defuna string= (x y)
  )

(defuna string-cmp-code (x y)
  (let ((len-x (length x))
        (len-y (length y))
        (res-code 0))
    (do ((i 0 (%+ i 1)))
        ((or (%>= i len-x) (%>= i len-y) (%/= res-code 0)) res-code)
      (let ((cc-1 (char-code (char x i)))
            (cc-2 (char-code (char y i))))
        (if (%< cc-1 cc-2)
            (setq res-code -1)
          (if (%> cc-1 cc-2)
              (setq res-code +1)))))
    (if (%= res-code 0)
        (cond ((%< len-x len-y) -1)
              ((%= len-x len-y) 0)
              ((%> len-x len-y) +1))
      res-code)))

(defuna get-hash (str n)
  (let ((len (length str))
        (res 2))
    (do ((i 0 (%+ i 2)))
        ((%>= i (%- len 1)) (%rem res n))
      (setq res (%rem (%+ res (%+ (char-code (char str i)) (%* (char-code (char
str (%+ i 1))) 256))) n)))))


; die vectoren, lesen und schreiben von elementen

(defparametera *array-head-tag* 1)
(defparametera *array-tag* 2)
(defparametera *array-lead* 12)


(defuna make-vector (n)
  (let ((n-bytes (%align (%+ (%* 4 n) *array-lead*) 3)))
    (let ((p (%allocate-space n-bytes)))
      (set-word p 0 (tag-set (addr-ior (p-shiftl n-bytes 2) 1) *array-head-tag*)
)
      (set-word p 4 n)
      (set-word p 8 0)
      (do ((i 0 (%+ i 1)))
          ((%>= i n))
        (set-svref p i nil))
      (setq p (tag-set p *array-tag*)))))

(defuna svref (vect i)
  (get-word vect (%+ (%* i 4) *array-lead*)))

(defuna set-svref (vect i val)
  (set-word vect (%+ (%* i 4) *array-lead*) val))

; die vectoren, testfunktionen
```

```lisp
(defuna arrayp (x)
  (and (tag-eq x *array-tag*)
       (let ((h (get-word x 0)))
         (and
          (addr-eqi (addr-and h 2) 0)
          (tag-eq h *array-head-tag*)))))



; die chars

(defparametera *char-mark* 0)
(defparametera *char-tag* 3)


(defuna char-code (c)
  (p-shiftr c 14))

(defuna code-char (x)
  (tag-set (p-shiftl x 14) *char-tag*))

(defuna characterp (x)
  (tag-eq x *char-tag*))



; die atome


(defparametera *symbol-head-addr* 3)
(defparametera *symbol-head-tag* 2)
(defparametera *symbol-tag* 2)


(defuna make-symbol (str)
  (let ((p (%allocate-space 24)))
    (set-word p 0 (tag-set *symbol-head-addr* *symbol-head-tag*))
    (set-word p 4 str)
    (set-word p 8 nil)
    (set-word p 12 nil)
    (set-word p 16 nil)
    (set-word p 20 nil)
    (tag-set p *symbol-tag*)))

(defuna intern (str)
  (let ((hash-val (get-hash str *base-pkg-len*)))
    (let ((slot (svref *base-pkg* hash-val))
          (p-old nil)
          (p-akt nil)
          (str-cmp-val -1)
          (test-str str))
      (do ((akt-atom slot (cdr akt-atom)))
          ((or (null akt-atom) (%>= str-cmp-val 0)))
        (format t "sym-name car akt-atom = ~S~%" (to-cl (symbol-name (car akt-atom)))
)
        (setq str-cmp-val (string-cmp-code str (symbol-name (car akt-atom))))
        (watch str-cmp-val)
        (setq p-old p-akt)
        (setq p-akt akt-atom))
      (format t "p-old = ~S~%" (to-cl p-old))
      (format t "p-akt = ~S~%" (to-cl p-akt))
      (format t "str-cmp-val = ~S ~%" (to-cl str-cmp-val))
      (if (%= 0 str-cmp-val)
          (return-from intern (car p-akt)))
```



```lisp
      (if (%>= str-cmp-val 0)
          (let* ((new-sym (make-symbol str))
                 (new-holder (cons new-sym p-akt)))
            (set-symbol-package new-sym *base-pkg*)
            (if (null p-old)
                (let ()
                  (set-svref *base-pkg* hash-val new-holder))
              (set-cdr p-old new-holder))
            (return-from intern new-sym)))
      (if (%< str-cmp-val 0)
          (let* ((new-sym (make-symbol str))
                 (new-holder (cons new-sym nil)))
            (set-symbol-package new-sym *base-pkg*)
            (format t "new-holder = ~S~%" (to-cl new-holder))
            (if (null p-akt)
                (let ()
                  (set-svref *base-pkg* hash-val new-holder))
              (set-cdr p-akt new-holder))
            (return-from intern new-sym))))))


; die atom attribute, lesen

(defuna symbol-name (sym)
  (get-word sym 4))

(defuna symbol-value (sym)
  (get-word sym 8))

(defuna symbol-function (sym)
  (get-word sym 12))

(defuna symbol-plist (sym)
  (get-word sym 16))

(defuna symbol-package (sym)
  (get-word sym 20))

; die atom attribute, schreiben

(defuna set-symbol-value (sym val)
  (set-word sym 8 val))

(defuna set-symbol-function (sym val)
  (set-word sym 12 val))

(defuna set-symbol-plist (sym val)
  (set-word sym 16 val))

(defuna set-symbol-package (sym val)
  (set-word sym 20 val))


; die atome, testfunktion

; symbol hat |10 als tag im zeiger und 1|00 im header


(defuna symbolp (x)
  (and (tag-eq x *symbol-tag*)
       (let ((h (get-word x 0)))
         (and
          (addr-eqi (addr-and h 2) 2)
          (tag-eq h *symbol-head-tag*)))))
```



```lisp

; die (fix)nums

(defparametera *fixnum-tag* 0)


(defun %+ (x y)
  (cl-fixnum (+ (fixnum-cl x) (fixnum-cl y))))

(defun %- (x y)
  (cl-fixnum (- (fixnum-cl x) (fixnum-cl y))))


(defun %* (x y)
  (cl-fixnum (* (fixnum-cl x) (fixnum-cl y))))

(defun %div (x y)
  (let ((z (floor (fixnum-cl x) (fixnum-cl y ))))
    (cl-fixnum z)))

(defuna %rem (x y)
  (%- x (%* (%div x y) y)))

(defuna numberp (x)
  (tag-eq x *fixnum-tag*))

(defun %/= (x y)
  (cl:not (%= x y)))

(defun %= (x y)
  (= (fixnum-cl x) (fixnum-cl y)))


(defun %> (x y)
  (> (fixnum-cl x) (fixnum-cl y)))


(defun %< (x y)
  (< (fixnum-cl x) (fixnum-cl y)))


(defun %<= (x y)
  (<= (fixnum-cl x) (fixnum-cl y)))


(defun %>= (x y)
  (>= (fixnum-cl x) (fixnum-cl y)))

(defuna %align (x i)
  (p-shiftl (p-align (p-shiftr x 2) i) 2))


; die closures

(defparametera *closure-head-addr* 3)
(defparametera *closure-head-tag* 1)
(defparametera *closure-tag* 2)
(defparametera *closure-slots* 2)

(defuna make-closure (templ vect)
  (let ((p (%allocate-space (%+ (%* *closure-slots* 4) 8))))
    (set-word p 0 (tag-set *closure-head-addr* *closure-head-tag*))
    (set-closure-ref p 0 templ)
    (set-closure-ref p 1 vect)
```



```lisp
    (setq p (tag-set p *closure-tag*))
    p))


(defuna closure-ref (cl i)
  (get-word cl (%+ 8 (%* i 4))))

(defuna set-closure-ref (cl i x)
  (set-word cl (%+ 8 (%* i 4)) x))

(defuna closurep (x)
  (and (tag-eq x *closure-tag*)
       (let ((h (get-word x 0)))
         (and
          (addr-eqi (addr-and h 2) 2)
          (tag-eq h *closure-head-tag*)))))



; die templates

(defparametera *template-head-tag* 3)
(defparametera *template-tag* 2)

(defparametera *template-leader* 4)

; d und c sind anzahl von 32 bit worten

(defuna make-template (d c)
  (let ((n-bytes (%align (%+ (%* (%+ d c) 4) (%* 4 *template-leader*)) 3)))
    (let ((p (%allocate-space n-bytes)))
      (set-word p 0 (tag-set (addr-ior (p-shiftl n-bytes 2) 1) *template-head-ta
g*))
      (set-word p 4 (p-add p (%+ (%* *template-leader* 4) (%* d 4))))
      (set-word p 8 d)
      (set-word p 12 c)
      (do ((i 0 (+ i 1)))
          ((%>= i d))
        (set-templ-ref-d p i nil))
      (do ((i 0 (+ i 1)))
          ((%>= i c))
        (set-templ-ref-c p i 0 0))
      (setq p (tag-set p *template-tag*))
      p)))

(defuna length-d (templ)
  (get-word templ 8))

(defuna templ-ref-d (templ i)
  (get-word templ (%+ (%* *template-leader* 4) (%* i 4))))


(defuna templ-ref-c (templ i)
  (let ((pc (get-word templ 4)))
    (get-word pc (%* i 4))))


(defuna set-templ-ref-d (templ i val)
  (set-word templ (%+ (%* *template-leader* 4) (%* i 4)) val))

(defuna set-templ-ref-c (templ i v-high v-low)
  (let ((pc (get-word templ 4)))
    (set-word pc (%* i 4) (p-constr v-high v-low))))
```

```
(defuna templatep (x)
  (and (tag-eq x *template-tag*)
       (let ((h (get-word x 0)))
         (and
          (addr-eqi (addr-and h 2) 0)
          (tag-eq h *template-head-tag*)))))



; allgemeiner heap allokator. ruft (später: inkrementellen) garbage collector au
f

(setf *memory* (make-array (* 512 1024)))


(defuna %allocate-space (n)
  (let ((ret *alloc-pointer*))
    (setf *alloc-pointer* (+ *alloc-pointer* (p-shiftr n 2)))
    ret))




; system-erzeugung


(defun create-nil (str)
  (let ((nil-add (make-symbol str)))
    (set-symbol-value nil-add nil-add)
    (set-symbol-function nil-add nil-add)
    (set-symbol-plist nil-add nil-add)
    (set-symbol-package nil-add nil-add)
    nil-add))

(setf nil (create-nil (cl-string "nil")))

(defparametera *base-pkg-len* 53)
(setf *base-pkg* (make-vector *base-pkg-len*))

;(setf str (cl-string "juergen"))



; tests

(setf *type-fun-l* '(consp listp symbolp numberp characterp stringp arrayp closu
rep templatep))

(defun show-types (x)
  (loop for f in *type-fun-l* do
        (format t "~S(~S)=~S~%" f x (funcall (cl:symbol-function f) x))))


(setf *tst-l* (cl:list 1 #\a (cl:cons 2 3) 'utz (cl:make-array 5) "juergen" ))



(intern (cl-string "read-error"))
(intern (cl-string "next"))
(intern (cl-string "same"))
(intern (cl-string "peek"))
```



```
(mapcar #'(lambda (z) (intern (cl-string z)))
        '("cond" "labels" "let" "setq" "progn" "do" "do-until" "funcall" "funarg" "car" "cdr" "
cons" "eq" "defun" "labels" "let"
                  "+" "-" "*" "divrem" ">" "<" "=" ">=" "<=" "null" "atom"))
```