```lisp
(in-package "CL-USER")

;; COMP-GEN *********** code erzeugung ****************************************
************************************************

(defun c-error (n proc &rest lis)
  (format t "error: ~A in ~A " n proc)
  (mapcar #'(lambda (z) (format t " ~A" z)) lis)
  (format t "~%~%"))


(defvar *stat* (make-array 4))


(defparameter *stk-idx* 3)

(defun get-stack (stat)
  (aref stat *stk-idx*))

(defun set-stack (stat new-stack)
  (format t "stack = ~A ~%" new-stack)
  (setf (aref stat *stk-idx*) new-stack))

(defparameter *ilist-idx* 2)

(defun get-ilist (stat)
  (aref stat *ilist-idx*))

(defun set-ilist (stat new-ilist)
  (setf (aref stat *ilist-idx*) new-ilist))


(defparameter *lits-idx* 1)

(defun get-lits (stat)
  (aref stat *lits-idx*))

(defun set-lits (stat new-lits)
  (setf (aref stat *lits-idx*) new-lits))


(defparameter *parms-idx* 0)

(defun get-parms (stat)
  (aref stat *parms-idx*))

(defun set-parms (stat new-parms)
  (setf (aref stat *parms-idx*) new-parms))


(defun stat-init (stat)
  (let ()
    (set-stack stat nil)
    (set-ilist stat nil)
    (set-lits stat nil)
    (set-parms stat nil)
    stat))

(stat-init *stat*)


(defun comp-gen (e stat)
```



```lisp
  (let ()
;    (format t "e= ~A~%stat= ~A~%+++++++++++++++++++++++++++++~%" e stat)
  (cond ((symbolp e) (comp-gen-symbol e stat))
        ((characterp e) (comp-gen-character e stat))
        ((numberp e) (comp-gen-number e stat))
        ((stringp e) (comp-gen-string e stat))
        ((listp e)
         (cond
          ((is-quote e)
           (comp-gen-quote e stat))
          ((is-function e)
           (comp-gen-function e stat))
          ((is-setq e)
           (comp-gen-setq e stat))
          ((is-if e)
           (comp-gen-if e stat))
          ((is-progn e)
           (comp-gen-progn e stat))
          ((eq (car e) 'do-until)
           (comp-gen-do-until e stat))
          ((eq (car e) 'vector)
           (comp-gen-vector e stat))
          ((eq (car e) 'apply)
           (comp-gen-apply e stat))
          ((eq (car e) 'funcall)
           (comp-gen-funcall e stat))
          ((is-primitive e)
           (comp-gen-primitive e stat))
          ((is-lambda (car e))
           (comp-gen-let e stat))
          ((is-lambda e)
           (comp-gen-lambda e stat))
          ((is-form e)
           (comp-gen-form e stat))
          (t (comp-gen-generic e stat)))))))


(defparameter *dummy* 0)

(defun increment-stack (n stat)
  (let ((stack (get-stack stat)))
    (setf stack (append (make-list n :initial-element *dummy*) stack))
    (set-stack stat stack)))

(defun decrement-stack (n stat)
  (let ((stack (get-stack stat)))
    (setf stack (do ((i 0 (+ i 1)) (stk stack (cdr stk))) ((>= i n) stk)))
    (set-stack stat stack)))

(defun add-vars-stack (vars stat)
  (let ((stack (get-stack stat)))
    (setf stack (append vars stack))
    (set-stack stat stack)))

(defun enter-opcode (opc ext stat)
  (let ((ilist (get-ilist stat)))
    (setf ilist (append ilist (list (list opc ext))))
    (set-ilist stat ilist)))

(defun enter-label (label stat)
  (let ((ilist (get-ilist stat)))
    (setf ilist (append ilist (list (list 'label label))))
    (set-ilist stat ilist)))
```



```lisp
(defun get-lit-index (e stat)
  (let ((lits (get-lits stat)))
    (let ((idx (position e lits)))
      idx)))


(defun enter-literal (e stat)
  (let ((lits (get-lits stat)))
    (let ((idx (get-lit-index e stat)))
    (if (null idx)
        (let ()
          (setf lits (append lits (list e)))
          (set-lits stat lits))))))



(defun comp-gen-number (e stat)
  (enter-literal e stat)
  (enter-opcode 'LITIDX (get-lit-index e stat) stat)
  (increment-stack 1 stat))

(defun comp-gen-string (e stat)
  (enter-literal e stat)
  (enter-opcode 'LITIDX (get-lit-index e stat) stat)
  (increment-stack 1 stat))

(defun comp-gen-character (e stat)
  (enter-literal e stat)
  (enter-opcode 'LITIDX (get-lit-index e stat) stat)
  (increment-stack 1 stat))



(defun comp-gen-quote (e stat)
  (let ((val (cadr e)))
    (enter-literal val stat)
    (enter-opcode 'LITIDX (get-lit-index val stat) stat)
    (increment-stack 1 stat)))

(defun comp-gen-function (e stat)
  (let ((val (cadr e)))
;    (format t "val = ~A~%" val)
    (if (listp val)
        (let ()
          (if (eq (car val) '%make-closure)
              (let ()
                (comp-gen val stat))
                ; (pprint stat))
            (if (eq (car val) '%symbol-value)
                (let ((new-expr (cons '%symbol-function (cdr val))))
                  (comp-gen new-expr stat))
              (c-error 2 'comp-gen-function "function of non closure"))))
      (if (atom val)
          (comp-gen val stat)
        (c-error 1 'comp-gen-function "function of non closure")))))

(defun get-pos (var lis)
  (position var lis))


(defun get-stack-index (var stat)
  (let ((stk (get-stack stat)))
    (let ((res (get-pos var stk)))
      res)))
```



```lisp
(defun comp-gen-symbol (e stat)
  (if (eq e nil)
      (let ()
        (enter-opcode 'LODNIL 0 stat)
        (increment-stack 1 stat))
    (let ((idx (get-stack-index e stat)))
      (when (null idx)
        (c-error 1 'comp-gen-symbol "symbol" e "not found on stack"))
      (enter-opcode 'LOD idx stat)
      (increment-stack 1 stat))))

(defun comp-gen-setq (e stat)
  (let ((v (car (cdr e)))
        (val (cadr (cdr e))))
    (comp-gen val stat)
    (enter-opcode 'LOD 0 stat)
    (increment-stack 1 stat)
    (let ((idx (get-stack-index v stat)))
      (when (null idx)
          (c-error 1 'comp-gen-setq "symbol" v "not found on stack"))
      (enter-opcode 'STO idx stat)
      (decrement-stack 1 stat))))


(defun comp-gen-progn (e stat)
  (maplist #'(lambda (z)
               (let ()
                 (comp-gen (car z) stat)
                 (if (not (null (cdr z)))
                     (let ()
                       (enter-opcode 'INCSP 1 stat)
                       (decrement-stack 1 stat))))) (cdr e)))

(defun comp-gen-let (e stat)
  (let ((head (car e))
        (args (cdr e)))
    (let ((vars (lambda-vars head))
          (body (lambda-body head)))
      (mapcar #'(lambda (z) (comp-gen z stat)) args)
      (decrement-stack (length vars) stat)
      (when (not (null vars))
        (add-vars-stack (reverse vars) stat))
      (setf body (cons 'progn body))
      (comp-gen body stat)
      (when (not (null vars))
        (enter-opcode 'STO (length vars) stat)
        (decrement-stack 1 stat)
        (enter-opcode 'INCSP (- (length vars) 1) stat)
        (decrement-stack (length vars) stat)
        (increment-stack 1 stat)))))


(defun comp-gen-primitive (e stat)
  (let ((head (car e))
        (args (cdr e)))
    (mapcar #'(lambda (z) (comp-gen z stat)) args)
    (enter-opcode head 0 stat)
    (decrement-stack (- (fun-arity head) 1) stat)))


(defun fun-arity (fun)
  (let ((i (emu::get-opcode (find-symbol (symbol-name fun) "EMU"))))
    (cadr (aref emu::*opcode-vector* i))))
```

```
(defun comp-gen-do-until-old (e stat)
  (let ((scnd (cadr e))
        (body (cddr e)))
    (let ((cnd (car scnd))
          (res (cdr scnd))
          (label-test (gensym))
          (label-goon (gensym))
          (label-ex (gensym)))
      (enter-label label-test stat)
      (comp-gen cnd stat)
      (enter-opcode 'JNIL label-goon stat)
      (decrement-stack 1 stat)
      (if (not (null res))
          (comp-gen (cons 'progn res) stat)
        (comp-gen 'nil stat))
      (enter-opcode 'JMP label-ex stat)
      (decrement-stack 1 stat)
      (enter-label label-goon stat)
      (comp-gen (cons 'progn body) stat)
      (enter-opcode 'INCSP 1 stat)
      (decrement-stack 1 stat)
      (enter-opcode 'JMP label-test stat)
      (enter-label label-ex stat)
      (increment-stack 1 stat))))



(defun comp-gen-do-until-new (e stat)
  (let ((scnd (cadr e))
        (body (cddr e)))
    (let ((cnd (car scnd))
          (res (cdr scnd))
          (label-test (gensym))
          (label-goon (gensym))
          (label-ex (gensym)))
      (enter-label label-test stat)
      (comp-gen cnd stat)
      (enter-opcode 'JNIL label-goon stat)
      (decrement-stack 1 stat)
      (enter-opcode 'JMP label-ex stat)
      (enter-label label-goon stat)
      (comp-gen (cons 'progn body) stat)
      (enter-opcode 'INCSP 1 stat)
      (decrement-stack 1 stat)
      (enter-opcode 'JMP label-test stat)
      (enter-label label-ex stat)
      (if (not (null res))
          (comp-gen (cons 'progn res) stat)
        (comp-gen 'nil stat)))))


(defun comp-gen-do-until (e stat)
  (comp-gen-do-until-old e stat))


(defun comp-gen-vector (e stat)
  (let ((args (cdr e)))
    (let ((nargs (length (cdr e))))
      (enter-literal nargs stat)
      (enter-opcode 'LITIDX (get-lit-index nargs stat) stat)
      (increment-stack 1 stat)
      (enter-opcode '%make-vector 0 stat)
      (do ((i 0 (+ i 1))
```



```
           (p args (cdr p)))
          ((null p))
        (enter-opcode 'LOD 0 stat)
        (increment-stack 1 stat)
        (enter-literal i stat)
        (enter-opcode 'LITIDX (get-lit-index i stat) stat)
        (increment-stack 1 stat)
        (comp-gen (car p) stat)
        (enter-opcode '%set-svref 0 stat)
        (decrement-stack 2 stat)
        (enter-opcode 'INCSP 1 stat)
        (decrement-stack 1 stat)))))

(defun comp-gen-funcall (e stat)
  (comp-gen-generic (cdr e) stat))



(defun comp-gen-apply (e stat)
  (let ((e1 (cdr e)))
    (let ((fun (car e1))
          (args (cdr e1))
          (nargs nil)
          (label-loop (gensym))
          (label-end (gensym))
          (q nil))
      (labels ((translate-fun (fun) (if (and (length=2 fun) (eq (car fun) '%symb
ol-value))
                                        (cons '%symbol-function (cdr fun))
                                      fun)))
        (setf fun (translate-fun fun)))
      (comp-gen fun stat)
      (setf q args)
      (setf nargs (do ((p args (cdr p))
                       (i 1 (+ i 1)))
                      ((length=1 p) i)
                    (comp-gen (car p) stat)
                    (setf q (cdr q))))
      (enter-literal nargs stat)
      (enter-opcode 'LITIDX (get-lit-index nargs stat) stat)
      (increment-stack 1 stat)
      (enter-opcode 'PUSHCS 0 stat)
      (enter-opcode 'INCSP 1 stat)
      (decrement-stack 1 stat)
      (comp-gen (car q) stat)
      (enter-opcode 'PUSHCS 0 stat)
      (enter-label label-loop stat)
      (enter-opcode 'JNIL label-end stat)
      (enter-opcode 'LODCS 0 stat)
      (enter-opcode 'CAR 0 stat)
      (enter-opcode 'LODCS 1 stat)
      (enter-literal 1 stat)
      (enter-opcode 'LITIDX (get-lit-index 1 stat) stat)
      (enter-opcode '%+ 0 stat)
      (enter-opcode 'STOCS 1 stat)
      (enter-opcode 'LODCS 0 stat)
      (enter-opcode 'CDR 0 stat)
      (enter-opcode 'LOD 0 stat)
      (enter-opcode 'STOCS 0 stat)
      (enter-opcode 'JMP label-loop stat)
      (enter-label label-end stat)
      (enter-opcode 'LODCS 1 stat)
      (enter-opcode 'INCCSP 2 stat)
      (enter-opcode 'CALLCLOSV 0 stat)
```



```
      (decrement-stack (+ nargs 1) stat)
      (increment-stack 1 stat))))


(defun comp-gen-if (e stat)
  (let ((tst (if-expr-tst e))
        (e1 (if-expr-branch-t e))
        (e2 (if-expr-branch-f e))
        (label (gensym))
        (label-ex (gensym)))
    (comp-gen tst stat)
    (enter-opcode 'JNIL label stat)
    (decrement-stack 1 stat)
    (comp-gen e1 stat)
    (enter-opcode 'JMP label-ex stat)
    (decrement-stack 1 stat)
    (enter-label label stat)
    (comp-gen e2 stat)
    (enter-label label-ex stat)))


(defun generate-rest-prolog (nvars stat)
  (let ((label-loop (gensym))
        (label-start (gensym))
        (label-ex (gensym)))
  (enter-opcode 'PUSHCS 0 stat)
  (enter-literal nvars stat)
  (enter-opcode 'LITIDX (get-lit-index nvars stat) stat)
  (increment-stack 1 stat)
  (enter-opcode '%< 0 stat)
  (decrement-stack 1 stat)
  (enter-opcode 'JNIL label-start stat)
  (decrement-stack 1 stat)
  (enter-opcode 'EXCEPT 0 stat)
  (enter-label label-start stat)
  (enter-opcode 'LODNIL 0 stat)
  (increment-stack 1 stat)
  (enter-label label-loop stat)
  (enter-opcode 'CONS 0 stat)
  (decrement-stack 1 stat)
  (enter-opcode 'LODCS 0 stat)
  (increment-stack 1 stat)
  (enter-literal 1 stat)
  (enter-opcode 'LITIDX (get-lit-index 1 stat) stat)
  (increment-stack 1 stat)
  (enter-opcode '%- 0 stat)
  (decrement-stack 1 stat)
  (enter-opcode 'INCCSP 1 stat)
  (enter-opcode 'PUSHCS 0 stat)
  (enter-opcode 'LITIDX (get-lit-index nvars stat) stat)
  (increment-stack 1 stat)
  (enter-opcode '%<  0 stat)
  (decrement-stack 1 stat)
  (enter-opcode 'JNIL label-loop stat)
  (decrement-stack 1 stat)
  (enter-opcode 'INCCSP 1 stat)))

(defun generate-standard-prolog (nvars stat)
  (let ()
    (enter-opcode 'INCSP 1 stat)
    (decrement-stack 1 stat)))



(defun comp-gen-lambda (e stat)
```



```
  (let ((vars (lambda-vars e))
        (body (lambda-body e)))
    (setf body (cons 'progn body))
    (let ((ilist (get-ilist stat))
          (lits (get-lits stat))
          (stack (get-stack stat))
          (nvars (length vars))
          (rest nil))

      (set-stack stat nil)
      (set-ilist stat nil)
      (set-lits stat nil)

      (setf rest (member '&rest vars))
      (cond ((length=2 rest)
             (setf nvars (- nvars 1))
             (setf vars (remove '&rest vars)))
            ((null rest))
            (t (c-error 1 'comp-gen-lambda "illegal formed &rest")))

      (add-vars-stack (reverse vars) stat)
      (increment-stack 1 stat)

      (if (length=2 rest)
          (generate-rest-prolog nvars stat)
        (generate-standard-prolog nvars stat))

      (comp-gen body stat)
      (enter-opcode 'STO nvars stat)
      (decrement-stack 1 stat)
      (enter-opcode 'INCSP (- nvars 1) stat)
      (decrement-stack nvars stat)
      (increment-stack 1 stat)
      (enter-opcode 'RET 0 stat)
      (let ((templ (make-pretemplate stat)))
        (set-stack stat stack)
        (set-ilist stat ilist)
        (set-lits stat lits)
        (enter-literal templ stat)
        (enter-opcode 'LITIDX (get-lit-index templ stat) stat)
        (increment-stack 1 stat)
       templ))))


(defun comp-gen-generic (e stat)
  (let ((len (length e)))
    (mapcar #'(lambda (z) (comp-gen z stat)) e)
    (enter-opcode 'CALLCLOS len stat)
    (decrement-stack len stat)
    (increment-stack 1 stat)))



(defun comp-gen-embrace (stat)
  (enter-opcode 'HALT 0 stat))





(defun is-pretemplate (x)
```

```lisp
  (and (listp x) (eq (car x) 'pre-template)))



(defun make-pretemplate (stat)
  (let ((lits (get-lits stat))
        (ilist (get-ilist stat)))
    (list 'pre-template lits ilist)))


; asm assembliert eine pretemplate zu einer template, die er direkt im speicher
(des emu-systems) anlegt

(defun is-label-stmt (stmt)
  (eq (car stmt) 'label))

(defun get-opc (stmt labl-assoc idx)
  (if (null (find-symbol (symbol-name (car stmt)) "EMU"))
      (c-error 1 'get-opc "opcode" stmt " not resolvable"))
  (let ((opc (emu::get-opcode (find-symbol (symbol-name (car stmt)) "EMU"))))
    (let ((d (cadr stmt)))
      (if (symbolp d)
          (let ((subs (assoc d labl-assoc)))
            (if (null subs)
                (c-error 1 'get-opc " label " d " cannot be resolved"))
            (setf d (- (cdr subs) idx))))
      (cons (logand opc 65535) (logand 65535 d)))))

(defun asm-templ (ptl)
  (let ((data (cadr ptl))
        (code (caddr ptl)))
    (let ((d-len (length data))
          (c-len nil)
          (labl-assoc nil)
          (stmt nil)
          (templ nil)
          (opc nil))
      (setf labl-assoc (do ((idx 0 idx)
                            (p code (cdr p)))
                           ((null p) labl-assoc)
                         (setf stmt (car p))
                         (if (is-label-stmt stmt)
                             (setf labl-assoc (cons (cons (cadr stmt) idx) labl-
assoc))
                           (setf idx (+ idx 1)))))
      (setf code (remove-if #'(lambda (z) (is-label-stmt z)) code))
      (setf c-len (length code))
      (setf templ (msys::make-template (msys::cl-to d-len) (msys::cl-to c-len)))
      (do ((i 0 (+ i 1))
           (p data (cdr p)))
          ((>= i d-len))
        (msys::set-templ-ref-d templ (msys::cl-to i) (msys::cl-to (car p))))
      (do ((i 0 (+ i 1))
           (p code (cdr p)))
          ((>= i c-len))
        (setf opc (get-opc (car p) labl-assoc i))
        (msys::set-templ-ref-c templ (msys::cl-to i) (msys::cl-to (car opc)) (ms
ys::cl-to (cdr opc))))
      templ)))

(defun disasm-opc (val)
  (let ((opc (logand (ash (logand val (ash 65535 16)) -16) 65535))
        (d (logand val 65535)))
    (let ((opc1 (car (aref emu::*opcode-vector* opc))))
```



```lisp
      (setf opc1 (make-symbol (symbol-name opc1)))
      (list opc1 d))))

(defun disasm-templ (tpl)
  (let ((d-len (msys::to-cl (msys::length-d tpl)))
        (c-len (msys::to-cl (msys::get-word tpl (msys::cl-to 12))))
        (data nil)
        (code nil)
        (res nil))
    (do ((i 0 (+ i 1)))
        ((>= i d-len))
      (setf data (cons (msys::to-cl (msys::templ-ref-d tpl (msys::cl-to i))) dat
a)))
    (setf data (reverse data))
    (do ((i 0 (+ i 1)))
        ((>= i c-len))
      (setf code (cons (disasm-opc (msys::templ-ref-c tpl (msys::cl-to i))) code
)))
    (setf code (reverse code))
    (list 'pre-template data code)))
```