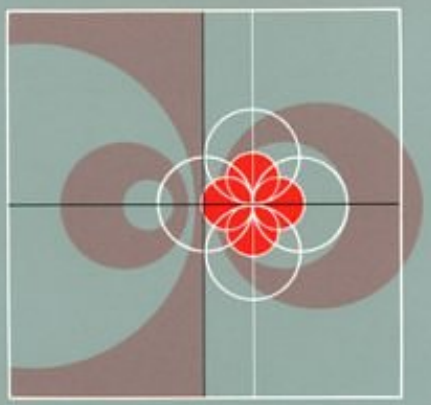
I. Prelude

Raphael

Disclaimer

GUY L. STEELE JR.

COMMON LISP



THE LANGUAGE

SECOND EDITION

```
; SLIME 2.19
CL-USER>
```

```
 2: 9 U×-*slime-repl sbcl*                          All REPL
Connected. Raphael, this could be the start of a beautiful program.
```

Thanks, Dmitry (:

```
[2, 4, 3]
```

```
define x := [2, 4, 3]
```

```
define x2 :=
  augment-rhythm [2, 4, 3] 2
```

```
define x2 := [4, 8, 6]
```

```
print augment-rhythm
```

```
lam seq n.
  map (lam x. figure-* x n) seq
```

```
define augment-rhythm := lam seq n.
  map (lam x. figure-* x n) seq
```

```
define diminish-rhythm := lam seq n.
  fold-from-end (lam x acc.
    case figure-/ x n
    | inl y => cons y acc
    | inr _ => acc)
    [] seq
```

```
define x-dur :=
  rhythm-duration [2, 4, 3]
```

```
define x-dur := 9
```

```
define rhythm-duration := lam seq.
  if empty-seq? seq
  then 0
  else
    let initial := (pos->int (at' seq 0)) in
    from 1 to length seq with initial
    accum lam i acc f.
      f (+ (pos->int (figure-duration (at' seq i)))
           acc)
```

Commands

Expressions

true false

()

- * + /

inl inr

(a, b)

II. Anticlimax

Yes

No input

No output

No unbounded loops

SECOND EDITION

THE C PROGRAMMING LANGUAGE

BRIAN W. KERNIGHAN
DENNIS M. RITCHIE

PRENTICE HALL SOFTWARE SERIES

Constant folding

Dead code elimination

```
int main(void) {
    return 0;
}
```

?

READ

EVAL

PRINT

LOOP

READ

EVAL

PRINT

LOOP

READ

EVAL

THINK

LOOP

III. Interlude

Divisive rhythm

Beat

x

$$\frac{x}{2}$$

$$\frac{x}{2} \quad \frac{x}{4} \quad \frac{x}{4}$$

$$\frac{x}{2}\ \frac{x}{2}\ \frac{x}{3}\ \frac{x}{3}\ \frac{x}{3}\ 2x$$

Additive rhythm

Shortest duration

k

2k

3k

3k 2k k 4k 3k

```
extract [3, 2, 1, 4, 3] with additive-rhythm
  minimum = 16
  meter = 13/16
  hide-meter = 1
end
```

Rhythm in the Music of Messiaen: an Algebraic Study and an Application in the Turangalîla Symphony

Julian L. Hook

IV. Liszt processing

Thanks, Mikey (:

```
(multi defclass+
  (s-term () position free-variables)
  (s-atomic-term (s-term))
  (s-flike (s-term))
  (s-int-op (s-flike))

  (s-binder () var-name expr))
```

```
(defmacro multi (macro &body args)
  (let ((head (if (listp macro)
                  macro
                  (list macro))))
    `(progn
       ,@(mapcar (lambda (x) `(,@head ,@x)) args))))
```

```
(macroexpand-1 '(multi defclass+
  (s-term () position free-variables)
  (s-atomic-term (s-term))
  (s-flike (s-term))
  (s-int-op (s-flike))

  (s-binder () var-name expr)))
```

```
(macroexpand-1 '(multi defclass+
  (s-term () position free-variables)
  (s-atomic-term (s-term))
  (s-flike (s-term))
  (s-int-op (s-flike))

  (binder () var-name expr)))
```

```
(progn
 (defclass+ s-term nil position free-variables)
 (defclass+ s-atomic-term (s-term))
 (defclass+ s-flike (s-term))
 (defclass+ s-int-op (s-flike))
 (defclass+ binder nil var-name expr))
```

```
(macrolet ((s-term (name &rest slots)
             `(defclass+ ,name (s-term)
                ,@slots)))

  (multi s-term
    (s-app func arg)
    (s-lam expr)
    (s-let expr body)
    (s-if test then else)
    (s-case expr inl inr)
    (s-tuple elems)))
```

```
(macrolet ((s-flike (name &rest slots)
             `(defclass+ ,name (s-flike)
                ,@slots)))

  (multi s-flike
    (s-rec-up start end initial f)
    (s-rec-down start end initial f)
    (s-compare x y)
    (s-= x y)
    (s-div floor x y)
    (s-inl value)
    (s-inr value)))
```

```
(macrolet ((s-int-op (name op &rest slots)
             `(defclass+ ,name (s-int-op)
                ((op ',op)) ,@slots)))

  (multi s-int-op
    (s-neg - x)
    (s-abs abs x)
    (s-add + x y)
    (s-sub - x y)
    (s-mul * x y)))
```

```
(macrolet ((s-atomic-term (name &rest slots)
             `(defclass+ ,name (s-atomic-term)
                ,@slots)))
  (multi s-atomic-term
    (s-bvar id)
    (s-fvar name)
    (s-true)
    (s-false)
    (s-unit)
    (s-int value)))
```

```
(s-bvar 1)
```

```
(s-fvar "x")
```

```
(s-int 5)
```

```
(s-true)
```

```
if true then x else y
```

```
(s-if (s-true) (s-fvar "x") (s-fvar "y"))
```

```
lam x. x
```

```
(s-lam (binder "x" (s-bvar 0)))
```

```
lam x y. x
```

```
(s-lam (binder "x"
  (s-lam (binder "y"
    (s-bvar 1)))))
```

Locally nameless representation

binder

```
(defclass+ s-lam (s-term) expr)
```

```
(defclass+ s-let (s-term) expr body)
```

```
(defclass+ s-case (s-term) expr inl inr)
```

```
(defclass+ binder () var-name expr))
```

```
let x := inl 2 in
case x
| inl y => y
| inr _ => -1
```

```
(s-let (s-inl (s-int 2))
       (s-binder "x"
                 (s-case (s-bvar 0)
                         (binder "y" (s-bvar 0))
                         (binder "_" (s-int -1)))))
```

Locally nameless representation

Bound variables x Free variables

Bound variables: de Bruijn indices

```
lam a. lam b. lam c. c
```

```
(s-lam
  (binder "a"
          (s-lam
            (binder "b"
                    (s-lam (binder "c"
                                   (s-bvar 0)))))))
```

```
lam a. lam b. lam c. b
```

```
(s-lam
  (binder "a"
          (s-lam
            (binder "b"
                    (s-lam (binder "c"
                                   (s-bvar 1)))))))
```

```
lam a. lam b. lam c. a
```

```
(s-lam
  (binder "a"
          (s-lam
            (binder "b"
                    (s-lam (binder "c"
                                   (s-bvar 2)))))))
```

```
lam a b c. a

(s-lam
  (binder "a"
          (s-lam
            (binder "b"
                    (s-lam (binder "c"
                                   (s-bvar 2)))))))
```

```
lam a. + a ((lam b. * a b) 3)

(s-lam
  (binder "a"
          (s-add
            (s-bvar 0)
            (s-app (s-lam
                     (binder "b"
                             (s-mul (s-bvar 1)
                                    (s-bvar 0))))
                   (s-int 3)))))
```

```
lam a b. > (rhythm-duration a) (rhythm-duration b)

(s-lam
  (binder
    "a"
    (s-lam
      (binder
        "b"
        (s-app
          (s-app (s-fvar ">")
                 (s-app (s-fvar "rhythm-duration")
                        (s-bvar 1)))
          (s-app (s-fvar "rhythm-duration")
                 (s-bvar 0)))))))
```

```
(defun free-vars-s (term)
  (labels ((impl (term acc)
             (etypecase term
               (s-fvar (pset-add (slot-value term 'name) acc))
               (s-atomic-term acc)
               (binder (impl (slot-value term 'expr) acc))
               (s-term (reduce (lambda (acc x)
                                 (impl x acc))
                               (slot-values term)
                               :initial-value acc)))))
    (impl term empty-pset)))
```

```
(defun fresh-name (name taken)
  (if (pset-member name taken)
      (fresh-name (concatenate 'string name "'")
                  taken)
      name))
```

```
(defun open-binder (repl binder)
  (labels ((impl (repl term k)
             (etypecase term
               (s-bvar (if (= k (slot-value term 'id))
                           repl
                           term))
               (s-atomic-term term)
               (binder (binder (slot-value term 'var-name)
                               (impl repl
                                     (slot-value term 'expr)
                                     (1+ k))))
               (s-term (map-obj (lambda (x)
                                  (impl repl x k))
                                term)))))
    (impl repl (slot-value binder 'expr) 0)))
```

```
(defun fvar-open-binder (binder)
  (with-slots (var-name expr) binder
    (open-binder (s-fvar (fresh-name var-name (free-vars-s expr)))
                 binder)))
```

```
(defun fvar-close-binder (str term &optional var-name)
  (labels ((impl (str term k)
             (etypecase term
               (s-fvar (if (equal str (slot-value term 'name))
                           (s-bvar k)
                           term))
               (s-atomic-term term)
               (binder (binder (slot-value term 'var-name)
                               (impl str
                                     (slot-value term 'expr)
                                     (1+ k))))
               (s-term (map-obj (lambda (x)
                                  (impl str x k)) term)))))
    (binder (or var-name str)
            (impl str term 0))))
```

```
(defun normalize (term env)
  (labels ((iter (t1 k)
             (let ((t2 (reduction-step t1 env)))
               (if t2
                   (iter t2 (1+ k))
                   (values t1 k)))))
    (iter term 0)))
```

```
(defgeneric reduction-step (term env))
```

```
(defmethod reduction-step ((term s-atomic-term) env)
  (declare (ignore term env))
  nil)
```

```
(defmethod reduction-step ((term s-fvar) env)
  (pmap-lookup (slot-value term 'name) env))
```

```
(defmethod reduction-step ((term s-fvar) env)
  (pmap-lookup (slot-value term 'name) env))
```

```
(multi (reduction-rule term env)

  ((s-app func arg)
   (single-step (func arg)
                (s-app func arg)
                (typecase func
                (s-lam (open-binder arg
                                    (slot-value func 'expr)))
                (s-flike (flike-bind-arg func arg)))))

  ((s-let expr body)
   (single-step (expr)
                (s-let expr body)
                (open-term expr body))))
```

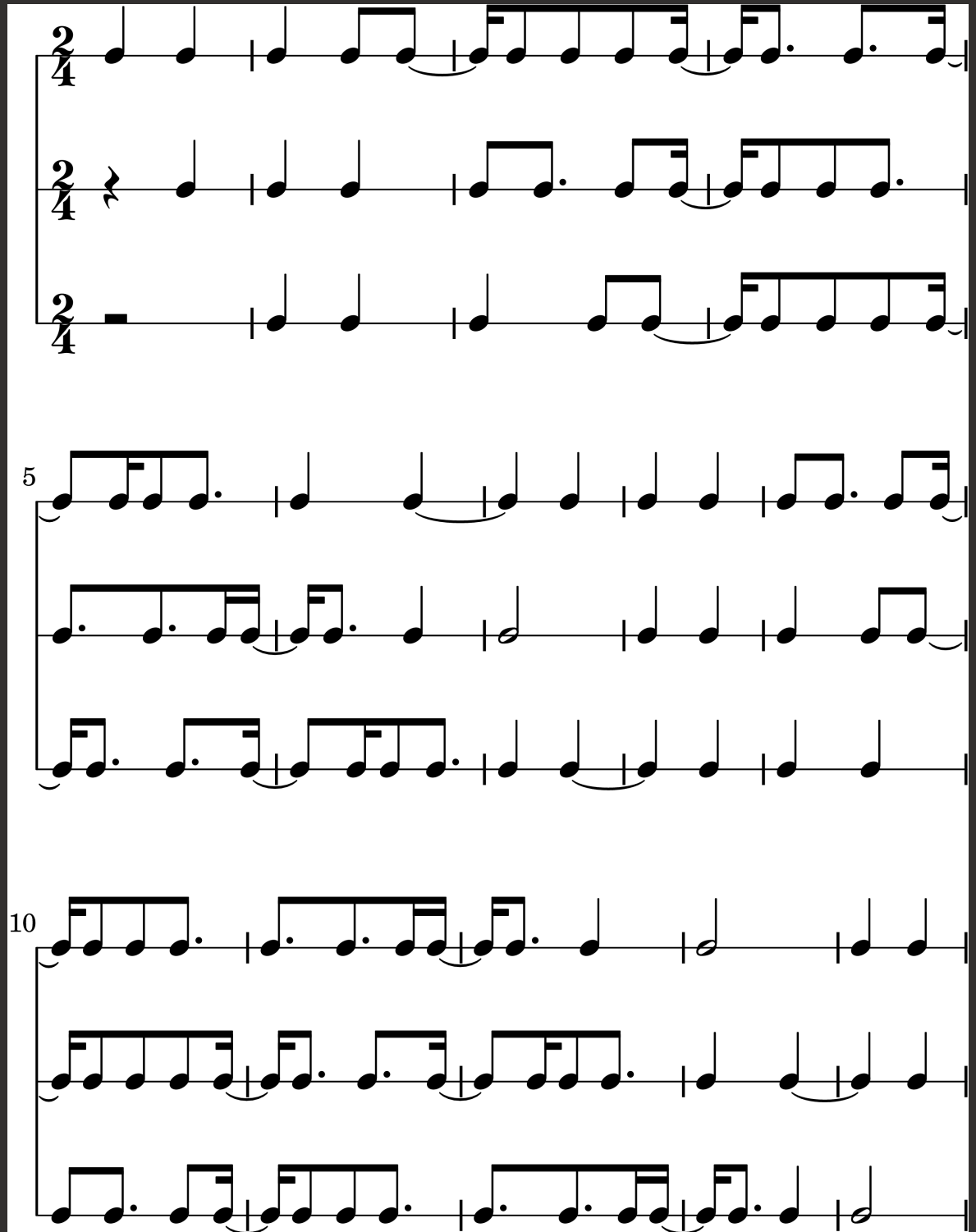
```
((s-if test then else)
 (single-step (test)
              (s-if test then else)
              (typecase test
                (s-true then)
                (s-false else))))
```

```
((s-case expr inl inr)
 (single-step (expr)
              (s-case expr inl inr)
              (typecase expr
                (s-inl (open-binder (slot-value expr 'value)
                                    inl))
                (s-inr (open-binder (slot-value expr 'value)
                                    inr))))))
```

```
((s-app func arg)
 (single-step (func arg)
              (s-app func arg)
              (typecase func
                (s-lam (open-binder arg
                                    (slot-value func 'expr)))
                (s-flike (flike-bind-arg func arg)))))
```

```
((s-let expr body)
 (single-step (expr)
              (s-let expr body)
              (open-binder expr body))))
```

```
define a := [4, 4, 4]

define c := diminish-rhythm a 2#pos

define b := incr-at c 1#int

define d := snoc (incr-all c) 1

define e := (rhythm-slice b 0 2)

define f := incr-at (augment-rhythm 2) 1#int

define x := append-seq [a, b, c, d, e, f]

define canon :=
  crop-rhythm (map (repeat-rhythm 2#nat)
    [x, delay x 4#pos, delay x 8#pos])

extract canon with additive-rhythm
  minimum = 16
  meter = 2/4
end
```


5
10

V. Interlude

Integers mod 12

Pitch classes

(pitch-class, duration)

(pitch-class, duration) — duration

(pitch-class, octave, duration) — duration

([(pitch-class, octave)], duration)

[pitch-class]

VI. LISt of $\mathbb{Z}$ (derived?) Types PROCESSING

Type checking as a form of interpretation

```
(multi defclass+
  (t-type ())
  (t-atomic-type (t-type))
  (t-env () var-count map)
  (t-scheme () vars type))
```

```
(macrolet ((t-type (name &rest slots)
             `(defclass+ ,name (t-type) ,@slots)))
(multi t-type
  (t-arrow domain codomain)
  (t-prod elems-t)
  (t-sum inl inr)
  (t-map key-t value-t)
  (t-abstract name params))
```

```
(macrolet ((t-atomic-type (name &rest slots)
             `(defclass+ ,name (t-atomic-type) ,@slots)))

(multi t-atomic-type
  (t-var id)
  (t-bool)
  (t-int)
  (t-unit))
```

```
(defun type-check (context env term)
  (mlet* (((type env2) (type-case context env term)))
    (generalize-type type env env2)))
```

```
(defgeneric type-case (term context env))
```

```
(defmacro typing-rule ((context env term) specializer &body body)
  (multiple-value-bind (type slots)
      (if (symbolp specializer)
          (values specializer nil)
          (values (car specializer) (cdr specializer)))
    `(defmethod type-case (,context ,env (,term ,type))
       (declare (ignorable ,context ,env ,term))
       (with-slots ,slots ,term ,@body))))
```

```
(multi (typing-rule context env term)
  (s-unit (values t-unit env))
  (s-int (values t-int env))
  (s-true (values t-bool env))
  (s-false (values t-bool env)))
```

```
(s-fvar (let ((scm (lookup-f-context term context)))
   (if scm
       (instantiate-t-scheme scm env)
       (liszp-type-error "Non existent variable: ~a~%" term))))

(s-bvar (let ((scm (lookup-b-context term context)))
   (if scm
       (instantiate-t-scheme scm env)
       (liszp-type-error "Non existent variable: ~a~%" term)))
```

```
((s-if test then else)
 (mlet* (((test-t env) (type-case context env test))
         ((then-t env) (type-case context env then))
         ((else-t env) (type-case context env else))
         ((env) (unify test-t t-bool env))
         ((env) (unify then-t else-t env)))
   (values then-t env)))
```

```
((s-lam expr)
 (mlet* (((domain env) (new-t-var env))
         ((codomain env) (type-case
                          (extend-b-context (t-scheme nil
                                                      domain)
                                            context)
                          env
                          expr)))
    (values (t-arrow domain codomain) env)))
```

```
((s-app func arg)
 (mlet* (((ft env) (type-case context env func))
         ((at env) (type-case context env arg))
         ((var env) (new-t-var env))
         ((env) (unify ft (t-arrow at var) env)))
    (values var env)))
```

```
(defun new-t-var (env)
  (with-slots (var-count map) env
    (values (t-var var-count)
            (t-env (1+ var-count) map))))
```

```
(defun new-t-var (env)
  (with-slots (var-count map) env
    (values (t-var var-count)
            (t-env (1+ var-count) map))))
```

```
(defun unify (t1 t2 env)
  (labels ((unify-var (id type env)
             (if (typep type 't-var)
                 (if (= id (slot-value type 'id))
                     env
                     (extend-t-env id type env))
                 (if (occurs-t-var id type env)
                     (liszp-type-error "Occurs check '~a' in '~a'" id type)
                     (extend-t-env id type env))))
           (unify-lists (l1 l2 env)
             (if l1
                 (mlet* (((env) (unify (first l1) (first l2) env)))
                   (unify-lists (cdr l1) (cdr l2) env))
                 env))
           (unify-abstracts (t1 t2 env)
             (unify-lists (slot-value t1 'params)
                          (slot-value t2 'params)
                          env))
           (impl (t1 t2 env)
                 (cond ((typep t1 't-var) (unify-var (slot-value t1 'id) t2 env))
                       ((typep t2 't-var) (unify-var (slot-value t2 'id) t1 env))
                       ((typep t1 (type-of t2))
                        (if (typep t1 t-abstract)
                            (unify-abstracts t1 t2 env)
                            (unify-lists (slot-values t1)
                                         (slot-values t2)
                                         env)))
                       (t (liszp-type-error "Mismatching types: '~a' '~a'" t1 t2)))))
    (impl (shallow-subst-t-var t1 env)
          (shallow-subst-t-var t2 env)
          env)))
```

```
check lam x. x

=> t0 -> t0
```

```
check lam x. 2

=> t0 -> int
```

```
check lam x y. x

=> t0 -> t1 -> t0
```

```
check (lam x y. x) true

=> t1 -> bool
```

```
check lam x. if x then 1 else 0

=> bool -> int
```

```
check lam f x. if f x then 1 else 0

=> (t0 -> bool) -> t0 -> int
```

```
abstract mod12

define int-to-mod12 : int -> mod12 := lam x.
  mod x 12

define mod12-to-int : mod12 -> int := lam x.
  x

define mod+ : mod12 -> mod12 -> mod12 := lam x y.
  int-to-mod12 (+ x y)

register int-reader int-to-mod12

end
```

Extraction methods register the types that they support

VII. Postlude

Indexing

refine

Extracting code