(define (eval exp env)

(cond ((number? exp) (eval-number exp env))

((variable? exp) (eval-variable exp env))

((lambda? exp) (eval-lambda exp env))

((if? exp) (eval-if exp env))

((+? exp) (eval-+ exp env))

((\*? exp) (eval-\* exp env))

(else (eval-call exp env))))

(define (compute exp)

(eval exp (empty-env)))

(define (eval-number exp env)

exp)

(define (eval-variable exp env)

(env-lookup exp env))

(define (eval-lambda exp env)

(lambda (val)

(eval (lambda-body exp)

(env-extend env (lambda-variable exp) val))))

(define (eval-call exp env)

((eval (call-operator exp) env)

(eval (call-operand exp) env)))

(define (eval-if exp env)

(if (eval (if-condition exp) env)

(eval (if-consequent exp) env)

(eval (if-alternative exp) env)))

(define (eval-+ exp env)

(+ (eval (op-arg1 exp) env)

(eval (op-arg2 exp) env)))

(define (eval-\* exp env)

(\* (eval (op-arg1 exp) env)

(eval (op-arg2 exp) env)))

図1.1：インタプリタ

(define (empty-env) '())

(define (env-lookup var env)

(let ((entry (assq var env)))

(if entry

(error "Unbound variable: " var)

(right entry))))

(define (env-extend var val env)

(pair (pair var val) env))

図1.2：環境ADT

(define variable? symbol?)

(define (lambda? exp)

(eq? 'lambda (first exp)))

(define lambda-variable second)

(define lambda-body third)

(define call-operator first)

(define call-operand second)

(define (if? exp)

(eq? 'if (first exp)))

(define if-condition second)

(define if-consequent third)

(define if-alternative fourth)

(define (+? exp)

(eq? '+ (first exp)))

(define (\*? exp)

(eq? '\* (first exp)))

(define op-arg1 second)

(define op-arg2 third)

図1.3：表現の述語と選択子

(define (%num x) x)

(define (%var name) name)

(define (%lambda name exp) (list 'lambda var exp))

(define (%call e1 e2) (list e1 e2))

(define (%if e1 e2 e3) (list 'if e1 e2 e3))

(define (%+ e1 e2) (list '+ e1 e2))

(define (%\* e1 e2) (list '\* e1 e2))

図1.4：表現の構築子

;; Den = Env → Val

;; Proc = Val → Val

(define ((%num n) env)

n)

(define ((%var name) env)

(env-lookup name env))

(define ((%lambda name den) env)

(lambda (val)

(den (env-extend env var val))))

(define ((%call d1 d2) env)

((d1 env) (d2 env)))

(define ((%if d1 d2 d3) env)

(if (d1 env) (d2 env) (d3 env)))

(define ((%+ d1 d2) env)

(+ (d1 env) (d2 env)))

(define ((%\* d1 d2) env)

(\* (d1 env) (d2 env)))

図1.6：表示的な実装

(define (D exp)

(cond ((number? exp) (%num exp))

((variable? exp) (%var exp))

((lambda? exp)

(%lambda (lambda-variable exp)

(D (lambda-body exp))))

((if? exp)

(%if (D (if-condition exp))

(D (if-consequent exp))

(D (if-alternative exp))))

((+? exp)

(%+ (D (op-arg1 exp))

(D (op-arg2 exp))))

((\*? exp)

(%\* (D (op-arg1 exp))

(D (op-arg2 exp))))

(else

(%call (D (call-operator exp))

(D (call-operand exp))))))

図1.7：構文から意味論への写像

;; Den = Env → Sto → Val × Sto

;; Proc = Val → Sto → Val × Sto

(define (((%num n) env) sto)

(pair n sto))

(define (((%var name) env) sto)

(pair (env-lookup name env) sto))

(define (((%lambda name den) env) sto)

(pair (lambda (val) (den (env-extend env name val)))

sto))

(define (((%call d1 d2) env) sto)

(with-pair ((d1 env) sto)

(lambda (v1 s1)

(with-pair ((d2 env) s1)

(lambda (v2 s2)

((v1 v2) s2))))))

(define (((%if d1 d2 d3) env) sto)

(with-pair ((d1 env) sto)

(lambda (v1 s1)

(if v1

((d2 env) s1)

((d3 env) s1)))))

図1.8：モノリシックインタプリタ（その１）

(define ((((make-op op) d1 d2) env) sto)

(with-pair ((d1 env) sto)

(lambda (v1 s1)

(with-pair ((d2 env) s1)

(lambda (v2 s2)

(pair (op v1 v2) s2))))))

(define %+ (make-op +))

(define %\* (make-op \*))

(define (((%begin d1 d2) env) sto)

((d2 env) (right ((d1 env) sto))))

(define (((%fetch loc) env) sto)

(pair (store-fetch loc sto) sto))

(define (((%store loc den) env) sto)

(with-pair ((den env) sto)

(lambda (val sto)

(pair 'unit (store-store loc val sto)))))

(define (with-pair p k)

(k (left p) (right p)))

図1.9：モノリシックインタプリタ（その２）

(define (empty-store) '())

(define (store-fetch loc sto)

(let ((entry (assq loc sto)))

(if entry

(error "Empty location: " loc)

(right entry))))

(define (store-store loc val sto)

(pair (pair loc val) sto))

図1.10：ストア（Store）ADT

(define ((lift-p1-a0 unit bind op) p1)

(unit (op p1)))

(define ((lift-p1-a1 unit bind op) d1)

(bind d1

(lambda (v1)

(unit (op v1)))))

(define ((lift-p0-a2 unit bind op) d1 d2)

(bind d1

(lambda (v1)

(bind d2

(lambda (v2)

(unit (op v1 v2)))))))

(define ((lift-p1-a1 unit bind op) p1 d1)

(bind d1

(lambda (v1)

(unit (op p1 v1)))))

(define ((lift-if unit bind op) d1 d2 d3)

(bind d1

(lambda (v1)

(op v1 d2 d3))))

図1.11：持ち上げ（Lifting）演算子

;;; V = Val

(define computeV id)

(define %numV id)

(define %+V +)

(define %\*V \*)

(define (%ifV d1 d2 d3)

(if d1 d2 d3))

図1.12：値（value）レベル

;;; S = Sto → V x Sto

;; Store monad

(define (unitS v)

(lambda (sto)

(pair v sto)))

(define (bindS s f)

(lambda (sto)

(let ((v\*sto (s sto)))

(let ((v (left v\*sto))

(sto (right v\*sto)))

((f v) sto)))))

;; Lifted operators

(define (computeS den)

(computeV (left (den (empty-store)))))

(define %numS (lift-p1-a0 unitS bindS %numV))

(define %+S (lift-p0-a2 unitS bindS %+V))

(define %\*S (lift-p0-a2 unitS bindS %\*V))

(define %ifS (lift-if unitS bindS %ifV))

;; New operators

(define ((%fetchS loc) sto)

(pair (store-fetch loc sto) sto))

(define ((%storeS loc den) sto)

(let ((v\*s (den sto)))

(let ((v (left v\*s))

(s (right v\*s)))

(pair 'unit

(store-store loc v s)))))

(define ((%beginS d1 d2) sto)

(d2 (right (d1 sto))))

図1.13：ストア（Store）レベル

;;; E = Env → S

;;; Proc = V → S

;; Environment monad

(define (unitE s)

(lambda (env) s))

(define (bindE e f)

(lambda (env)

((f (e env)) env)))

;; Lifted operators

(define (compute den)

(comuteS (den (empty-env))))

(define %num (lift-p1-a0 unitE bindE %numS))

(define %+ (lift-p0-a2 unitE bindE %+S))

(define %\* (lift-p0-a2 unitE bindE %\*S))

(define %if (lift-if unitE bindE %ifS))

(define %fetch (lift-p1-a0 unitE bindE %fetchS))

(define %store (lift-p1-a1 unitE bindE %storeS))

(define %begin (lift-p0-a2 unitE bindE %beginS))

;; New operators

(define ((%var name) env)

(unitS (env-lookup name env)))

(define ((%lambda name den) env)

(unitS

(lambda (val)

(den (env-extend name val env)))))

(define ((%call d1 d2) env)

(bindS (d1 env)

(lambda (v1)

(bindS (d2 env)

(lambda (v2)

(v1 v2))))))

図1.14：環境レベル

;; E = Env → S

;; S = Sto → V x Sto

;; V = Val

(define ((unitSE s) env)

s)

(define ((unitVS v) sto)

(pair v sto))

(define (((unitVE v) env) sto)

(pair v sto))

(define ((bindSE t f) env)

((f (t env)) env))

(define (((bindVE t f) env) sto)

(let ((p ((t env) sto)))

(let ((v (left p))

(s (right p)))

(((f v ) env) s))))

図1.15：レベル間交渉演算子

;; E = Env → S

;; S = Sto → V x Sto

;; V = Val

;; Proc = V → S

(define (%num v)

(unitVE v))

(define ((%var name) env)

(unitVS (env-lookup env name)))

(define ((%lambda name den) env)

(unitVS

(lambda (val)

(den (env-extend env name val)))))

(define (%call d1 d2)

(bindVE d1

(lambda (v1)

(bindVE d2

(lambda (v2)

(unitSE (v1 v2)))))))

(define (%if d1 d2 d3)

(bindVE d1

(lambda (v1)

(if v1 d2 d3))))

図1.16：モジュラーインタプリタ（その１）

(define ((make-op op) d1 d2)

(bindVE d1

(lambda (v1)

(bindVE d2

(lambda (v2)

(unitVE (op v1 v2)))))))

(define %+ (make-op +))

(define %\* (make-op \*))

(define (%begin d1 d2)

(beindVE d1

(lambda (v1)

d2)))

(define (%fetch loc)

(unitSE

(lambda (sto)

(pair (store-fetch loc sto) sto))))

(define (%store loc den)

(bindVE den

(lambda (val)

(unitSE

(lambda (sto)

(pair 'unit (store-store loc val sto)))))))

図1.17：モジュラーインタプリタ（その２）

;; Computation ADT

(define computations

(make-computations

cbv-environments stores continuations nondeterminism errors))

;; Language ADT

(load "error-exceptions" "numbers" "booleans" "numeric-predicates"

"amb" "procedures" "environments" "stores" "while" "callcc")

;; Basic Semantics

(show-computations)

=> (-> Env

(-> Sto

(let A0 (\* Val Sto)

(let A1 (+ (list A0) Err)

(-> (-> A0 A1) A1)))))

;; Sample expressions

(compute

(%call (%lambda 'x (%+ (%var 'x) (%var 'x)))

(%amb (%num 1) (%num 2))))

=> (2 4) ; would be (2 3 3 4) in call-by-name

(compute

(%begin

(%store 'n (%amb (%num 4) (%num 5)))

(%store 'r (%num 1))

(%call/cc

(%lambda 'exit

(%while (%true)

(%begin

(%if (%zero? (%fetch 'n))

(%call (%var 'exit) (%fetch 'r))

(%unit))

(%store 'r (%\* (%fetch 'r) (%fetch 'n)))

(%store 'n (%- (%fetch 'n) (%num 1)))))))))

=> (24 120)

図1.18：仕様と表現の例

(define %let

(let ((unitE (get-unit 'envs 'top))

(bindE (get-bind 'envs 'top))

(bindV (get-bind 'env-values 'top)))

(lambda (name c1 c2)

(bindV c1

(lambda (v1)

(bindE c2

(lambda (e2)

(unitE

(lambda (env)

(e2 (env-extend env name v1)))))))))))

図1.19：%letのソースコードでの定義

(lambda (name c1 c2)

(lambda (env)

(lambda (sto)

(lambda (k)

(((c1 env) sto)

(lambda (a) ; Val x Sto

(((c2 (env-extend env name (left a))) (right a)) k)))))))

図1.20：単純化された%letの定義

(define %amb

(let ((unit (get-unit 'lists 'top))

(bind (get-bind 'lists 'top)))

(lambda (x y)

(bind x

(lambda (lx)

(bind y

(lambda (ly)

(unit (append lx ly)))))))))

図1.21：%ambのソースコードでの定義

;; Computation ADT

(define computations

(make-computations environments continuations nondeterminism))

;; Basic semantics

(-> Env (let AO (List Ans) (-> (-> Val AO) AO)))

;; Simplified %amb

(lambda (x y)

(lambda (env)

(lambda (k)

(reduce append ()

(map k (append ((x env) list) ((y env) list)))))))

;; Example

(compute

(%+ (%num 1)

(%call/cc

(%lambda 'k

(&\* (%num 10)

(%amb (%num 3) (%call (%var 'k) (%num 4))))))))

;; => (31 51)

図1.22：%amb version 1

;; Computation ADT

(define computations

(make-computations environments continuations2 nondeterminism))

;; Basic semantics

(-> Env (let AO (List Ans) (-> (-> Val AO) AO)))

;; Simplified %amb

(lambda (x y)

(lambda (env)

(lambda (k)

(append ((x env) k) ((y env) k)))))

;; Example

(compute

(%+ (%num 1)

(%call/cc

(%lambda 'k

(%\* (%num 10)

(%amb (%num 3) (%call (%var 'k) (%num 4))))))))

;; => (31 5)

図1.23：%amb version 2

;; Computation ADT

(define computations

(make-computations environments nondeterminism continuations))

;; Basic semantics

(-> Env (let AO (List Ans) (-> (-> (List Val) AO) AO)))

;; Simplified %amb

(lambda (x y)

(lambda (env)

(lambda (k)

((x env)

(lambda (a)

((y env)

(lambda (a0)

(k (append a a0)))))))))

;; Example

(compute

(%+ (%num 1)

(%call/cc

(%lambda 'k

(%\* (%num 10)

(%amb (%num 3) (%call (%var 'k) (%num 4))))))))

;; => (5)

図1.24：%amb version 3

;; Computation and language ADTs

(define computations

(make-computations cbn-environments exp-environments))

(load "error-values" "numbers" "booleans" "numeric-predicates"

"environmens" "exp-environments")

;; Simplified %evar and %elet

(lambda (name)

(lambda (env)

(lambda (eenv)

(if (env-lookup eenv name)

((right (env-lookup eenv name)) eenv) ; \*\*\*

(in 'errors (unbound-error name))))))

(lambda (name c1 c2)

(lambda (env)

(lambda (eenv)

((c2 env) (env-extend eenv name (c1 env))))))

図1.25：パラメータ化の単一化システム

(compute

(%let 'f (%\* (%evar 'x) (%evar 'x))

(%+ (%elet 'x (%num 3) (%var 'f))

(%elet 'x (%num 4) (%var 'f)))))

;; => 25

(compute

(%let 'g (%+ (%evar 'a) (%evar 'a))

(%let 'f (%elet 'a (%\* (%evar 'x) (%evar 'x))

(%var 'g))

(%elet 'x (%num 3) (%var 'f)))))

;; => 18

図1.26：単一化されたパラメータの例

;; Computation and language ADTs

(define computations

(make-computations resumptions stores lists))

(load "error-values" "numbers" "booleans" "begin" "while"

"products" "numeric-predicates" "amb" "stores" "resumptions")

;; Examples

(compute

(%par (%num 1) (%num 2) (%num 3)))

;; => (1 2 1 3 2 3)

(compute

(%seq

(%store 'x (%unit))

(%par

(%store 'x (%pair (%num 3) (%fetch 'x)))

(%store 'x (%pair (%num 2) (%fetch 'x)))

(%store 'x (%pair (%num 1) (%fetch 'x))))

(%fetch 'x)))

;; =>

;; ((pair 3 (pair 2 (pair 1 unit)))

;; (pair 2 (pair 3 (pair 1 unit)))

;; (pair 3 (pair 1 (pair 2 unit)))

;; (pair 1 (pair 3 (pair 2 unit)))

;; (pair 2 (pair 1 (pair 3 unit)))

;; (pair 1 (pair 2 (pair 3 unit))))

(compute

(%seq

(%store 'x (%num 1))

(%store 'go (%true))

(%par

(%store 'go (%false))

(%while (%and (%fetch 'go)

(%< (%fetch 'x) (%num 7)))

(%pause (%store 'x (%1+ (%fetch 'x))))))

(%fetch 'x)))

;; => (2 3 4 5 6 7 7 1)

図1.27：再開機能を用いた並列言語

;; Identity: T(A) = A

(define (unit a)

a)

(define (bind ta f)

(f ta))

;; Lists: T(A) = List(A)

(define (unit a)

(list a))

(define (bind ta f)

(reduce append '() (map f ta)))

;; Environments: T(A) = Env -> A

(define (unit a)

(lambda (env) a))

(define (bind ta f)

(lambda (env)

((f (ta env)) env)))

;; Stores: T(A) = Sto -> A x Sto

(define (unit a)

(lambda (sto) (pair a sto)))

(define (bind ta f)

(lambda (sto)

(let ((a\*s (ta sto)))

(let ((a (left a\*s))

(s (right a\*s)))

((f a) s)))))

図2.1：モナドの例（その１）

;; Exceptions: T(A) = A + X

(define (unit a)

(in-left a))

(define (bind ta f)

(sum-case ta

(lambda (a) (f a))

(lambda (x) (in-right x))))

;; Monoids: T(A) = A x M

(define (unit a)

(pair a monoid-unit))

(define (bind ta f)

(let ((a1 (left ta))

(m1 (right ta)))

(let ((a\*m (f a1)))

(let ((a2 (left a\*m))

(m2 (right a\*m)))

(pair a2 (monoid-product m1 m2))))))

;; Continuations: T(A) = (A -> Ans) -> Ans

(define (unit a)

(lambda (k) (k a)))

(define (bind ta f)

(lambda (k) (ta (lambda (a) ((f a) k)))))

;; Resumptions: T(A) = fix(X)(A + X)

(define (unit a)

(in-left a))

(define (bind ta f)

(sum-case ta

(lambda (a) (f a))

(lambda (ta) (bind ta f))))

図2.2：モナドの例（その２）

;; S(A) = EnvS -> A

;; T(A) = EnvT -> A

;; ST(A) = EnvS -> Env T -> A

(define ((joinS ssa) envS)

((ssa envS) envS))

(define ((joinT tta) envT)

((tta envT) envT))

(define (((joinST ststa) envS) envT)

((((ststa envS) envT) envS) envT))

図2.3：モナドは組み合わされたものではない

;; F(T)(A) = Env -> T(A)

(define (environment-transformer m)

(let ((unitT (monad-unit m))

(mapT (monad-map m))

(joinT (monad-join m)))

(define (unit a)

(lambda (env) (unitT a)))

(define ((map f) fta)

(lambda (env)

((mapT f) (fta env))))

(define (join ftfta)

(lambda (env)

(joinT

((mapT (lambda (fta) (fta env)))

(ftfta env)))))

(make-monad unit map join)))

図2.4：環境モナド変換子