Programming Language Engineering Master of Computer Science

Faculty of Science and Bio-Engineering Sciences
Vrije Universiteit Brussel

Section 5: Primitive Implementations

Theo D'Hondt
Software Languages Lab

"… taking into account the limitations of physical computational entities…"

A Quick Reminder

```scheme
(begin
  (define meta-level-eval eval)

  (define (loop output environment)
    (define rollback environment)

    (define (error message qualifier)
      (display message)
      (loop qualifier rollback))

    (define (wrap-native-procedure native-procedure)
      (lambda (arguments continue environment)
        (define native-value (apply native-procedure arguments))
        (continue native-value environment)))

    (define (bind-variable variable value environment)
      (define binding (cons variable value))
      (cons binding environment))

    (define (bind-parameters parameters arguments environment)
      (if (symbol? parameters)
        (bind-variable parameters arguments environment)
        (if (pair? parameters)
          (let*
            ((variable (car parameters))
             (value (car arguments ))
             (environment (bind-variable variable value environment)))
            (bind-parameters (cdr parameters) (cdr arguments) environment))
          environment)))

    (define (evaluate-sequence expressions continue environment)
      (define head (car expressions))
      (define tail (cdr expressions))
      (define (continue-with-sequence value environment)
        (evaluate-sequence tail continue environment))
      (if (null? tail)
        (evaluate head continue environment)
        (evaluate head continue-with-sequence environment)))
```

A Quick Reminder

```
(begin
 (define meta-level-eval eval)

 (define (loop output environment)
  (define rollback environment)

  (define (error message qualifier)
   (display message)
   (loop qualifier rollback))

  (define (wrap-native-procedure native-procedure)
   (lambda (arguments continue environment)
    (define native-value (apply native-procedure arguments))
    (continue native-value environment)))

  (define (bind-variable variable value environment)
   (define binding (cons variable value))
   (cons binding environment))

  (define (bind-parameters parameters arguments environment)
   (if (symbol? parameters)
    (bind-variable parameters arguments environment)
    (if (pair? parameters)
     (let*
      ((variable (car parameters))
       (value (car arguments ))
       (environment (bind-variable variable value environment)))
      (bind-parameters (cdr parameters) (cdr arguments) environment))
     environment)))

  (define (evaluate-sequence expressions continue environment)
   (define head (car expressions))
   (define tail (cdr expressions))
   (define (continue-with-sequence value environment)
    (evaluate-sequence tail continue environment))
   (if (null? tail)
    (evaluate head continue environment)
    (evaluate head continue-with-sequence environment)))
```

REP continuation

A Quick Reminder

```
(begin
 (define meta-level-eval eval)

 (define (loop output environment)
  (define rollback environment)

  (define (error message qualifier)
   (display message)
   (loop qualifier rollback))

  (define (wrap-native-procedure native-procedure)
   (lambda (arguments continue environment)
    (define native-value (apply native-procedure arguments))
    (continue native-value environment)))

  (define (bind-variable variable value environment)
   (define binding (cons variable value))
   (cons binding environment))

  (define (bind-parameters parameters arguments environment)
   (if (symbol? parameters)
    (bind-variable parameters arguments environment)
    (if (pair? parameters)
     (let*
      ((variable (car parameters))
       (value (car arguments ))
       (environment (bind-variable variable value environment)))
      (bind-parameters (cdr parameters) (cdr arguments) environment))
     environment)))

  (define (evaluate-sequence expressions continue environment)
   (define head (car expressions))
   (define tail (cdr expressions))
   (define (continue-with-sequence value environment)
    (evaluate-sequence tail continue environment))
   (if (null? tail)
    (evaluate head continue environment)
    (evaluate head continue-with-sequence environment)))
```

REP continuation

error continuation

A Quick Reminder

```
(begin
  (define meta-level-eval eval)

  (define (loop output environment)
    (define rollback environment)
```

REP continuation

```
    (define (error message qualifier)
      (display message)
      (loop qualifier rollback))
```

error continuation

```
    (define (wrap-native-procedure native-procedure)
      (lambda (arguments continue environment)
        (define native-value (apply native-procedure arguments))
        (continue native-value environment)))

    (define (bind-variable variable value environment)
      (define binding (cons variable value))
      (cons binding environment))

    (define (bind-parameters parameters arguments environment)
      (if (symbol? parameters)
        (bind-variable parameters arguments environment)
        (if (pair? parameters)
          (let*
            ((variable (car parameters))
             (value (car arguments ))
             (environment (bind-variable variable value environment)))
            (bind-parameters (cdr parameters) (cdr arguments) environment))
          environment)))
```

parameter binding

```
    (define (evaluate-sequence expressions continue environment)
      (define head (car expressions))
      (define tail (cdr expressions))
      (define (continue-with-sequence value environment)
        (evaluate-sequence tail continue environment))
      (if (null? tail)
        (evaluate head continue environment)
        (evaluate head continue-with-sequence environment)))
```

A Quick Reminder

```
(begin
 (define meta-level-eval eval)

 (define (loop output environment)
  (define rollback environment)
```

REP continuation

```
  (define (error message qualifier)
   (display message)
   (loop qualifier rollback))
```

error continuation

```
  (define (wrap-native-procedure native-procedure)
   (lambda (arguments continue environment)
    (define native-value (apply native-procedure arguments))
    (continue native-value environment)))

  (define (bind-variable variable value environment)
   (define binding (cons variable value))
   (cons binding environment))

  (define (bind-parameters parameters arguments environment)
   (if (symbol? parameters)
    (bind-variable parameters arguments environment)
    (if (pair? parameters)
     (let*
      ((variable (car parameters))
       (value (car arguments ))
       (environment (bind-variable variable value environment)))
      (bind-parameters (cdr parameters) (cdr arguments) environment))
     environment)))
```

parameter binding

```
  (define (evaluate-sequence expressions continue environment)
   (define head (car expressions))
   (define tail (cdr expressions))
   (define (continue-with-sequence value environment)
    (evaluate-sequence tail continue environment))
   (if (null? tail)
    (evaluate head continue environment)
    (evaluate head continue-with-sequence environment)))
```

expresion sequence

A Quick Reminder (cont’d)

```
(define (make-procedure parameters expressions environment)
  (lambda (arguments continue dynamic-environment)
    (define (continue-after-sequence value environment-after-sequence)
      (continue value dynamic-environment))
    (define lexical-environment (bind-parameters parameters arguments environment))
    (evaluate-sequence expressions continue-after-sequence lexical-environment)))

(define (evaluate-application operator)
  (lambda operands
    (lambda (continue environment)
      (define (continue-after-operator procedure environment-after-operator)
        (define (evaluate-operands operands arguments environment)
          (define (continue-with-operands value environment-with-operands)
            (evaluate-operands (cdr operands) (cons value arguments) environment-with-operands))
          (if (null? operands)
            (procedure (reverse arguments) continue environment)
            (evaluate (car operands) continue-with-operands environment)))
        (evaluate-operands operands '() environment-after-operator))
      (evaluate operator continue-after-operator environment))))

(define (evaluate-begin . expressions)
  (lambda (continue environment)
    (evaluate-sequence expressions continue environment)))

(define (evaluate-define pattern . expressions)
  (lambda (continue environment)
    (define (continue-after-expression value environment-after-expression)
      (define binding (cons pattern value))
      (continue value (cons binding environment-after-expression)))
    (if (symbol? pattern)
      (evaluate (car expressions) continue-after-expression environment))
      (let* ((binding (cons (car pattern) '()))
             (environment (cons binding environment))
             (procedure (make-procedure (cdr pattern) expressions environment)))
        (set-cdr! binding procedure)
        (continue procedure environment)))))
```

A Quick Reminder (cont’d)

```
(define (make-procedure parameters expressions environment)
  (lambda (arguments continue dynamic-environment)
    (define (continue-after-sequence value environment-after-sequence)
      (continue value dynamic-environment))
    (define lexical-environment (bind-parameters parameters arguments environment))
    (evaluate-sequence expressions continue-after-sequence lexical-environment)))
```

function closure

```
(define (evaluate-application operator)
  (lambda operands
    (lambda (continue environment)
      (define (continue-after-operator procedure environment-after-operator)
        (define (evaluate-operands operands arguments environment)
          (define (continue-with-operands value environment-with-operands)
            (evaluate-operands (cdr operands) (cons value arguments) environment-with-operands))
          (if (null? operands)
            (procedure (reverse arguments) continue environment)
            (evaluate (car operands) continue-with-operands environment)))
        (evaluate-operands operands '() environment-after-operator))
      (evaluate operator continue-after-operator environment))))

(define (evaluate-begin . expressions)
  (lambda (continue environment)
    (evaluate-sequence expressions continue environment)))

(define (evaluate-define pattern . expressions)
  (lambda (continue environment)
    (define (continue-after-expression value environment-after-expression)
      (define binding (cons pattern value))
      (continue value (cons binding environment-after-expression)))
    (if (symbol? pattern)
      (evaluate (car expressions) continue-after-expression environment))
      (let* ((binding (cons (car pattern) '()))
             (environment (cons binding environment))
             (procedure (make-procedure (cdr pattern) expressions environment)))
        (set-cdr! binding procedure)
        (continue procedure environment)))))
```

A Quick Reminder (cont’d)

```
(define (make-procedure parameters expressions environment)
  (lambda (arguments continue dynamic-environment)
    (define (continue-after-sequence value environment-after-sequence)
      (continue value dynamic-environment))
    (define lexical-environment (bind-parameters parameters arguments environment))
    (evaluate-sequence expressions continue-after-sequence lexical-environment)))
```

function closure

```
(define (evaluate-application operator)
  (lambda operands
    (lambda (continue environment)
      (define (continue-after-operator procedure environment-after-operator)
        (define (evaluate-operands operands arguments environment)
          (define (continue-with-operands value environment-with-operands)
            (evaluate-operands (cdr operands) (cons value arguments) environment-with-operands))
          (if (null? operands)
            (procedure (reverse arguments) continue environment)
            (evaluate (car operands) continue-with-operands environment)))
        (evaluate-operands operands '() environment-after-operator))
      (evaluate operator continue-after-operator environment))))
```

function application

```
(define (evaluate-begin . expressions)
  (lambda (continue environment)
    (evaluate-sequence expressions continue environment)))

(define (evaluate-define pattern . expressions)
  (lambda (continue environment)
    (define (continue-after-expression value environment-after-expression)
      (define binding (cons pattern value))
      (continue value (cons binding environment-after-expression)))
    (if (symbol? pattern)
      (evaluate (car expressions) continue-after-expression environment))
      (let* ((binding (cons (car pattern) '()))
             (environment (cons binding environment))
             (procedure (make-procedure (cdr pattern) expressions environment)))
        (set-cdr! binding procedure)
        (continue procedure environment))))
```

A Quick Reminder (cont’d)

```
(define (make-procedure parameters expressions environment)
  (lambda (arguments continue dynamic-environment)
    (define (continue-after-sequence value environment-after-sequence)
      (continue value dynamic-environment))
    (define lexical-environment (bind-parameters parameters arguments environment))
    (evaluate-sequence expressions continue-after-sequence lexical-environment)))
```

function closure

```
(define (evaluate-application operator)
  (lambda operands
    (lambda (continue environment)
      (define (continue-after-operator procedure environment-after-operator)
        (define (evaluate-operands operands arguments environment)
          (define (continue-with-operands value environment-with-operands)
            (evaluate-operands (cdr operands) (cons value arguments) environment-with-operands))
          (if (null? operands)
            (procedure (reverse arguments) continue environment)
            (evaluate (car operands) continue-with-operands environment)))
        (evaluate-operands operands '() environment-after-operator))
      (evaluate operator continue-after-operator environment))))
```

function application

```
(define (evaluate-begin . expressions)
  (lambda (continue environment)
    (evaluate-sequence expressions continue environment)))

(define (evaluate-define pattern . expressions)
  (lambda (continue environment)
    (define (continue-after-expression value environment-after-expression)
      (define binding (cons pattern value))
      (continue value (cons binding environment-after-expression)))
    (if (symbol? pattern)
      (evaluate (car expressions) continue-after-expression environment))
      (let* ((binding (cons (car pattern) '()))
             (environment (cons binding environment))
             (procedure (make-procedure (cdr pattern) expressions environment)))
        (set-cdr! binding procedure)
        (continue procedure environment))))
```

procedure

A Quick Reminder (cont’d)

```scheme
(define (evaluate-if predicate consequent . alternative)
  (lambda (continue environment)
    (define (continue-after-predicate boolean environment-after-predicate)
      (if (eq? boolean #f)
        (if (null? alternative)
          (continue '() environment-after-predicate)
          (evaluate (car alternative) continue environment-after-predicate))
        (evaluate consequent continue environment-after-predicate)))
    (evaluate predicate continue-after-predicate environment)))

(define (evaluate-lambda parameters . expressions)
  (lambda (continue environment)
    (continue (make-procedure parameters expressions environment) environment)))

(define (evaluate-quote expression)
  (lambda (continue environment)
    (continue expression environment)))

(define (evaluate-set! variable expression)
  (lambda (continue environment)
    (define (continue-after-expression value environment-after-expression)
      (define binding (assoc variable environment-after-expression))
      (if binding
        (set-cdr! binding value)
        (error "inaccessible variable: " variable))
      (continue value environment-after-expression))
    (evaluate expression continue-after-expression environment)))

(define (evaluate-variable variable continue environment)
  (define binding (assoc variable environment))
  (if binding
    (continue (cdr binding) environment)
    (let ((native-value (meta-level-eval variable (interaction-environment))))
      (if (procedure? native-value)
        (continue (wrap-native-procedure native-value) environment)
        (continue native-value environment)))))
```

A Quick Reminder (cont’d)

```scheme
(define (evaluate-if predicate consequent . alternative)
  (lambda (continue environment)
    (define (continue-after-predicate boolean environment-after-predicate)
      (if (eq? boolean #f)
        (if (null? alternative)
          (continue '() environment-after-predicate)
          (evaluate (car alternative) continue environment-after-predicate))
        (evaluate consequent continue environment-after-predicate)))
    (evaluate predicate continue-after-predicate environment)))

(define (evaluate-lambda parameters . expressions)
  (lambda (continue environment)
    (continue (make-procedure parameters expressions environment) environment)))
```

procedure

```scheme
(define (evaluate-quote expression)
  (lambda (continue environment)
    (continue expression environment)))

(define (evaluate-set! variable expression)
  (lambda (continue environment)
    (define (continue-after-expression value environment-after-expression)
      (define binding (assoc variable environment-after-expression))
      (if binding
        (set-cdr! binding value)
        (error "inaccessible variable: " variable))
      (continue value environment-after-expression))
    (evaluate expression continue-after-expression environment)))

(define (evaluate-variable variable continue environment)
  (define binding (assoc variable environment))
  (if binding
    (continue (cdr binding) environment)
    (let ((native-value (meta-level-eval variable (interaction-environment))))
      (if (procedure? native-value)
        (continue (wrap-native-procedure native-value) environment)
        (continue native-value environment)))))
```

A Quick Reminder (cont’d)

```
  (define (evaluate-while predicate . expressions)
    (lambda (continue environment)
      (define (iterate value environment)
        (define (continue-after-predicate boolean environment-after-predicate)
          (define (continue-after-sequence value environment-after-sequence)
            (iterate value environment))
          (if (eq? boolean #f)
            (continue value environment)
            (evaluate-sequence expressions continue-after-sequence environment)))
        (evaluate predicate continue-after-predicate environment))
      (iterate '() environment)))

  (define (evaluate expression continue environment)
    (cond
      ((symbol? expression)
       (evaluate-variable expression continue environment))
      ((pair? expression)
       (let ((operator (car expression))
             (operands (cdr expression)))
         ((apply
            (case operator
              ((begin)  evaluate-begin )
              ((define) evaluate-define)
              ((if)     evaluate-if    )
              ((lambda) evaluate-lambda)
              ((quote)  evaluate-quote )
              ((set!)   evaluate-set!  )
              ((while)  evaluate-while )
              (else     (evaluate-application operator))) operands) continue environment)))
      (else
        (continue expression environment))))

  (display output)
  (newline)
  (display ">>>")
  (evaluate (read) loop environment))

(loop "cpSlip version 0" '()))
```

A Quick Reminder (cont’d)

```
  (define (evaluate-while predicate . expressions)
    (lambda (continue environment)
      (define (iterate value environment)
        (define (continue-after-predicate boolean environment-after-predicate)
          (define (continue-after-sequence value environment-after-sequence)
            (iterate value environment))
          (if (eq? boolean #f)
            (continue value environment)
            (evaluate-sequence expressions continue-after-sequence environment)))
        (evaluate predicate continue-after-predicate environment))
      (iterate '() environment)))

  (define (evaluate expression continue environment)
    (cond
      ((symbol? expression)
       (evaluate-variable expression continue environment))
      ((pair? expression)
       (let ((operator (car expression))
             (operands (cdr expression)))
         ((apply
            (case operator
              ((begin)  evaluate-begin )
              ((define) evaluate-define)
              ((if)     evaluate-if    )
              ((lambda) evaluate-lambda)
              ((quote)  evaluate-quote )
              ((set!)   evaluate-set!  )
              ((while)  evaluate-while )
              (else     (evaluate-application operator))) operands) continue environment)))
      (else
        (continue expression environment))))
```

dispatcher

```
  (display output)
  (newline)
  (display ">>>")
  (evaluate (read) loop environment))

(loop "cpSlip version 0" '()))
```

A Quick Reminder (cont'd)

```
(define (evaluate-while predicate . expressions)
  (lambda (continue environment)
    (define (iterate value environment)
      (define (continue-after-predicate boolean environment-after-predicate)
        (define (continue-after-sequence value environment-after-sequence)
          (iterate value environment))
        (if (eq? boolean #f)
          (continue value environment)
          (evaluate-sequence expressions continue-after-sequence environment)))
      (evaluate predicate continue-after-predicate environment))
    (iterate '() environment)))
```

```
(define (evaluate expression continue environment)
  (cond
    ((symbol? expression)
     (evaluate-variable expression continue environment))
    ((pair? expression)
     (let ((operator (car expression))
           (operands (cdr expression)))
       ((apply
          (case operator
            ((begin)  evaluate-begin )
            ((define) evaluate-define)
            ((if)     evaluate-if    )
            ((lambda) evaluate-lambda)
            ((quote)  evaluate-quote )
            ((set!)   evaluate-set!  )
            ((while)  evaluate-while )
            (else     (evaluate-application operator))) operands) continue environment)))
    (else
      (continue expression environment))))
```

```
  (display output)
  (newline)
  (display ">>>")
  (evaluate (read) loop environment))

(loop "cpSlip version 0" '()))
```

top level

dispatcher

A Quick Reminder (cont'd)

```scheme
(define (evaluate-while predicate . expressions)
  (lambda (continue environment)
    (define (iterate value environment)
      (define (continue-after-predicate boolean environment-after-predicate)
        (define (continue-after-sequence value environment-after-sequence)
          (iterate value environment))
        (if (eq? boolean #f)
          (continue value environment)
          (evaluate-sequence expressions continue-after-sequence environment)))
      (evaluate predicate continue-after-predicate environment))
    (iterate '() environment)))

(define (evaluate expression continue environment)
  (cond
    ((symbol? expression)
     (evaluate-variable expression continue environment))
    ((pair? expression)
     (let ((operator (car expression))
           (operands (cdr expression)))
       ((apply
          (case operator
            ((begin)  evaluate-begin )
            ((define) evaluate-define)
            ((if)     evaluate-if    )
            ((lambda) evaluate-lambda)
            ((quote)  evaluate-quote )
            ((set!)   evaluate-set!  )
            ((while)  evaluate-while )
            (else     (evaluate-application operator))) operands) continue environment)))
    (else
      (continue expression environment))))

(display output)
(newline)
(display ">>>")
(evaluate (read) loop environment))

(loop "cpSlip version 0" '()))
```

dispatcher

top level

startup

Ad hoc Continuations

```
(define (fibonacci n continue)
  (define (continuation-1 p)
    (define (continuation-2 q)
      (continue (+ p q)))
    (fibonacci (- n 2) continuation-2))
  (if (> n 1)
    (fibonacci (- n 1) continuation-1)
    (continue 1)))
```

easy because of first-class closures

```
(fibonacci 15 display)
```

Ad hoc Continuations

```
(define (fibonacci n continue)
  (define (continuation-1 p)
    (define (continuation-2 q)
      (continue (+ p q)))
    (fibonacci (- n 2) continuation-2))
  (if (> n 1)
    (fibonacci (- n 1) continuation-1)
    (continue 1)))
```

nested continuation

easy because of first-class closures

```
(fibonacci 15 display)
```

Ad hoc Continuations

```
(define (fibonacci n continue)
 (define (continuation-1 p)
  (define (continuation-2 q)
   (continue (+ p q)))
  (fibonacci (- n 2) continuation-2))
 (if (> n 1)
  (fibonacci (- n 1) continuation-1)
  (continue 1)))
```

nested continuation

twice nested continuation

easy because of first-class closures

```
(fibonacci 15 display)
```

Ad hoc Continuations

```
(define (fibonacci n continue)
  (define (continuation-1 p)
    (define (continuation-2 q)
      (continue (+ p q)))
    (fibonacci (- n 2) continuation-2))
  (if (> n 1)
    (fibonacci (- n 1) continuation-1)
    (continue 1)))
```

nested continuation

twice nested continuation

easy because of first-class closures

```
(fibonacci 15 display)
```

top-level continuation

Ad-hoc Continuations (cont’d)

```
(define (function <parameters> top-continuation)
  ...
  (define (local-continuation <local-parameters>)
    ...
    <use parameters>
    <use local-parameters>
    ...
    (top-continuation local-arguments))
  ...
  <use parameters>
  <build arguments>
  ...
  (local-continuation <arguments>))
```

```
  (function <top-level-arguments> top-level-continuation)
```

Ad-hoc Continuations (cont’d)

```
(define (function <parameters> top-continuation)
  ...
  (define (local-continuation <local-parameters>)
    ...
    <use parameters>
    <use local-parameters>
    ...
    (top-continuation local-arguments))
  ...
  <use parameters>
  <build arguments>
  ...
  (local-continuation <arguments>))
```

tail calls

```
  (function <top-level-arguments> top-level-continuation)
```

Primitive Language Constraints

- No nested functions
- No garbage collection
- Static & weak typing
- No proper tail calls

Non-nested Continuations

```
(begin
  (define (factorial n)
    (if (> n 1)
      (* n (factorial (- n 1)))
      1))
  (factorial 10))
```

```
(begin
  (define (factorial n continue)
    (define (continuation p)
      (continue (* n p)))
    (if (> n 1)
      (factorial (- n 1) continuation)
      (continue 1)) )
  (factorial 10 display))
```

```
(begin
  (define (continuation p closure)
    (let* ((n (car closure))
           (continuation (cadr closure))
           (nested-closure (caddr closure)))
      (continuation (* n p) nested-closure)))

  (define (factorial . closure)
    (define n (car closure))
    (define nested-continuation (cadr closure))
    (define nested-closure (caddr closure))
    (if (> n 1)
      (factorial (- n 1) continuation closure)
      (nested-continuation 1 nested-closure)))

  (define (top-continuation p closure)
    (display p))

  (factorial 10 top-continuation '()))
```

Non-nested Continuations

```
(begin
  (define (factorial n)
    (if (> n 1)
      (* n (factorial (- n 1)))
      1))
  (factorial 10))
```

```
(begin
  (define (factorial n continue)
    (define (continuation p)
      (continue (* n p)))
    (if (> n 1)
      (factorial (- n 1) continuation)
      (continue 1)) )
  (factorial 10 display))
```

```
(begin
  (define (continuation p closure)
    (let* ((n (car closure))
           (continuation (cadr closure))
           (nested-closure (caddr closure)))
      (continuation (* n p) nested-closure)))

  (define (factorial . closure)
    (define n (car closure))
    (define nested-continuation (cadr closure))
    (define nested-closure (caddr closure))
    (if (> n 1)
      (factorial (- n 1) continuation closure)
      (nested-continuation 1 nested-closure)))

  (define (top-continuation p closure)
    (display p))

  (factorial 10 top-continuation '()))
```

Non-nested Continuations

```
(begin
  (define (factorial n)
    (if (> n 1)
      (* n (factorial (- n 1)))
      1))
  (factorial 10))
```

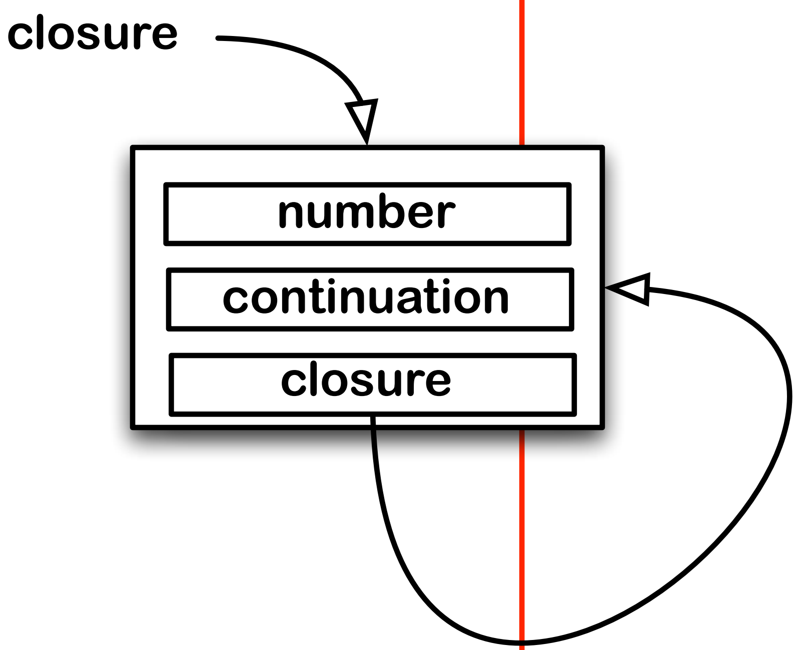
```
(begin
  (define (factorial n continue)
    (define (continuation p)
      (continue (* n p)))
    (if (> n 1)
      (factorial (- n 1) continuation)
      (continue 1)) )
  (factorial 10 display))
```

```
(begin
  (define (continuation p closure)
    (let* ((n (car closure))
           (continuation (cadr closure))
           (nested-closure (caddr closure)))
      (continuation (* n p) nested-closure)))

  (define (factorial . closure)
    (define n (car closure))
    (define nested-continuation (cadr closure))
    (define nested-closure (caddr closure))
    (if (> n 1)
      (factorial (- n 1) continuation closure)
      (nested-continuation 1 nested-closure)))

  (define (top-continuation p closure)
    (display p))

  (factorial 10 top-continuation '()))
```

require ad-hoc closures

Ad-hoc Continuations in C

```
#include <stdio.h>
#include <stdlib.h>

static int factorial(int n)
  { if (n > 1)
      return n * factorial(n - 1);
    else
      return 1; }

typedef
  struct cl * clos;

typedef
   void (* cont)(int, clos);

typedef
  struct cl { int n;
              cont continuation;
              clos closure; } cl;

static clos make_closure(int n, cont continuation, clos closure)
  { clos new_closure = malloc(sizeof(cl));
    new_closure->n = n;
    new_closure->continuation = continuation;
    new_closure->closure = closure;
    return new_closure; }
```



Ad-hoc Continuations in C

```c
static void continuation(int p, clos closure)
  { int n = closure->n;
    cont continuation = closure->continuation;
    clos nested_closure = closure->closure;
    free(closure);
    continuation(n * p, nested_closure); }

static void c_factorial(clos closure)
  { int n = closure->n;
    cont nested_continuation = closure->continuation;
    clos nested_closure = closure->closure;
    if (n > 1)
      c_factorial(make_closure(n - 1, continuation, closure));
    else
      nested_continuation(1, nested_closure); }

static void top_continuation(int p, clos closure)
  { printf("c_factorial(10) = %d\n", p); }

int main (int argc, const char * argv[])
  { printf("factorial(10)   = %d\n", factorial(10));
    c_factorial(make_closure(10, top_continuation, (clos)0));
    return 0; }
```

Ad-hoc Continuations in C

```
static void continuation(int p, clos closure)
 { int n = closure->n;
  cont continuation = closure->continuation;
  clos nested_closure = closure->closure;
  free(closure);
  continuation(n * p, nested_closure); }

static void c_factorial(clos closure)
 { int n = closure->n;
  cont nested_continuation = closure->continuation;
  clos nested_closure = closure->closure;
  if (n > 1)
   c_factorial(make_closure(n - 1, continuation, closure));
  else
   nested_continuation(1, nested_closure); }

static void top_continuation(int p, clos closure)
 { printf("c_factorial(10) = %d\n", p); }

int main (int argc, const char * argv[])
 { printf("factorial(10)   = %d\n", factorial(10));
  c_factorial(make_closure(10, top_continuation, (clos)0));
  return 0; }
```

Ad-hoc Continuations in C

```
static void continuation(int p, clos closure)
  { int n = closure->n;
    cont continuation = closure->continuation;
    clos nested_closure = closure->closure;
    free(closure);
    continuation(n * p, nested_closure); }

static void c_factorial(clos closure)
  { int n = closure->n;
    cont nested_continuation = closure->continuation;
    clos nested_closure = closure->closure;
    if (n > 1)
      c_factorial(make_closure(n - 1, continuation, closure));
    else
      nested_continuation(1, nested_closure); }

static void top_continuation(int p, clos closure)
  { printf("c_factorial(10) = %d\n", p); }

int main (int argc, const char * argv[])
  { printf("factorial(10)   = %d\n", factorial(10));
    c_factorial(make_closure(10, top_continuation, (clos)0));
    return 0; }
```

Ad-hoc Continuations in C

```
static void continuation(int p, clos closure)
  { int n = closure->n;
    cont continuation = closure->continuation;
    clos nested_closure = closure->closure;
    free(closure);
    continuation(n * p, nested_closure); }

static void c_factorial(clos closure)
  { int n = closure->n;
    cont nested_continuation = closure->continuation;
    clos nested_closure = closure->closure;
    if (n > 1)
      c_factorial(make_closure(n - 1, continuation, closure));
    else
      nested_continuation(1, nested_closure); }

static void top_continuation(int p, clos closure)
  { printf("c_factorial(10) = %d\n", p); }

int main (int argc, const char * argv[])
  { printf("factorial(10)   = %d\n", factorial(10));
    c_factorial(make_closure(10, top_continuation, (clos)0));
    return 0; }
```

Strategy for building a VM in C

version 1: straightforward code

version 2: using a trampoline

version 3: factored out environment

version 4: threaded continuations

version 5: functional continuations

version 6: partial evaluation

version 7: iterative constructs

version 8: lexical addressing

version 9: garbage collection

version 10: proper tail recursion

version 11: internal caches

version 12: optimizing C

Strategy for building a VM in C

- version 1: straightforward code
- version 2: using a trampoline
- version 3: factored out environment
- version 4: threaded continuations
- version 5: functional continuations
- version 6: partial evaluation
- version 7: iterative constructs
- version 8: lexical addressing
- version 9: garbage collection
- version 10: proper tail recursion
- version 11: internal caches
- version 12: optimizing C

translation
from
Scheme

Strategy for building a VM in C

version 1: straightforward code

version 2: using a trampoline

version 3: factored out environment

version 4: threaded continuations

version 5: functional continuations

version 6: partial evaluation

version 7: iterative constructs

version 8: lexical addressing

version 9: garbage collection

version 10: proper tail recursion

version 11: internal caches

version 12: optimizing C

countering
uncontrolled
C stack
growth

Strategy for building a VM in C

version 1: straightforward code

version 2: using a trampoline

version 3: factored out environment

version 4: threaded continuations

version 5: functional continuations

version 6: partial evaluation

version 7: iterative constructs

version 8: lexical addressing

version 9: garbage collection

version 10: proper tail recursion

version 11: internal caches

version 12: optimizing C

managing the environment as a global variable

Strategy for building a VM in C

version 1: straightforward code

version 2: using a trampoline

version 3: factored out environment

version 4: threaded continuations

version 5: functional continuations

version 6: partial evaluation

version 7: iterative constructs

version 8: lexical addressing

version 9: garbage collection

version 10: proper tail recursion

version 11: internal caches

version 12: optimizing C

managing continuations in an explicit stack

Strategy for building a VM in C

```
version 1: straightforward code

version 2: using a trampoline

version 3: factored out environment

version 4: threaded continuations

version 5: functional continuations

version 6: partial evaluation

version 7: iterative constructs

version 8: lexical addressing

version 9: garbage collection

version 10: proper tail recursion

version 11: internal caches

version 12: optimizing C
```

factoring out continue operations

Strategy for building a VM in C

version 1: straightforward code

version 2: using a trampoline

version 3: factored out environment

version 4: threaded continuations

version 5: functional continuations

version 6: partial evaluation

version 7: iterative constructs

version 8: lexical addressing

version 9: garbage collection

version 10: proper tail recursion

version 11: internal caches

version 12: optimizing C

introducing a rich abstract grammar

Strategy for building a VM in C

```
version 1: straightforward code

version 2: using a trampoline

version 3: factored out environment

version 4: threaded continuations

version 5: functional continuations

version 6: partial evaluation

version 7: iterative constructs

version 8: lexical addressing

version 9: garbage collection

version 10: proper tail recursion

version 11: internal caches

version 12: optimizing C
```

optimizing
iterative
evaluation
steps

Strategy for building a VM in C

version 1: straightforward code

version 2: using a trampoline

version 3: factored out environment

version 4: threaded continuations

version 5: functional continuations

version 6: partial evaluation

version 7: iterative constructs

version 8: lexical addressing

version 9: garbage collection

version 10: proper tail recursion

version 11: internal caches

version 12: optimizing C

introducing lexical addressing

Strategy for building a VM in C

version 1: straightforward code

version 2: using a trampoline

version 3: factored out environment

version 4: threaded continuations

version 5: functional continuations

version 6: partial evaluation

version 7: iterative constructs

version 8: lexical addressing

version 9: garbage collection

version 10: proper tail recursion

version 11: internal caches

version 12: optimizing C

weaving in the garbage collector

Strategy for building a VM in C

```
version 1: straightforward code
version 2: using a trampoline
version 3: factored out environment
version 4: threaded continuations
version 5: functional continuations
version 6: partial evaluation
version 7: iterative constructs
version 8: lexical addressing
version 9: garbage collection
version 10: proper tail recursion
version 11: internal caches
version 12: optimizing C
```

optimizing tail calls

Strategy for building a VM in C

version 1: straightforward code

version 2: using a trampoline

version 3: factored out environment

version 4: threaded continuations

version 5: functional continuations

version 6: partial evaluation

version 7: iterative constructs

version 8: lexical addressing

version 9: garbage collection

version 10: proper tail recursion

version 11: internal caches

version 12: optimizing C

caching continuation and environment frames

Strategy for building a VM in C

version 1: straightforward code

version 2: using a trampoline

version 3: factored out environment

version 4: threaded continuations

version 5: functional continuations

version 6: partial evaluation

version 7: iterative constructs

version 8: lexical addressing

version 9: garbage collection

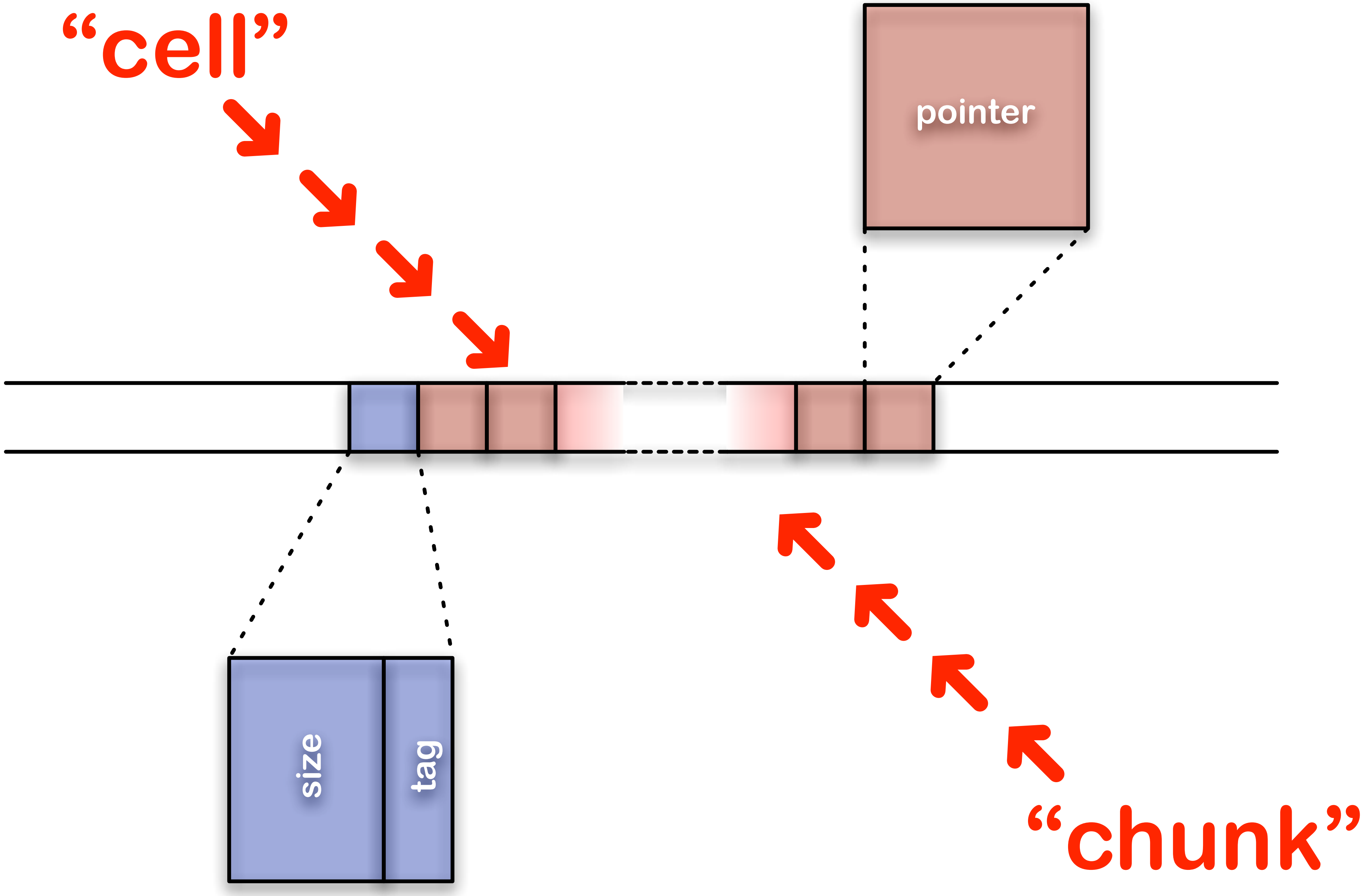
version 10: proper tail recursion

version 11: internal caches

version 12: optimizing C

code duplication, inlining, &c.

A Universal Memory Model



A Universal Memory Model (cont’d)

```
typedef           void * ADR_type;
typedef unsigned short   BYT_type;
typedef  unsigned long   CEL_type;
typedef           void   NIL_type;
typedef           char * TXT_type;
typedef  unsigned long   UNS_type;

typedef union PTR { CEL_type    cel;
                    union PTR * ptr; } * PTR_type;

static const UNS_type Memory_Cell_Size = sizeof(CEL_type);

UNS_type   Memory_Get_Size(PTR_type);
BYT_type    Memory_Get_Tag(PTR_type);
NIL_type Memory_Initialize(ADR_type,
                           UNS_type);
PTR_type Memory_Make_Chunk(BYT_type,
                           UNS_type);
```

A Universal Memory Model (cont'd)

```
typedef           void * ADR_type;
typedef unsigned short   BYT_type;
typedef  unsigned long   CEL_type;
typedef           void   NIL_type;
typedef           char * TXT_type;
typedef  unsigned long   UNS_type;

typedef union PTR { CEL_type    cel;
                    union PTR * ptr; } * PTR_type;

static const UNS_type Memory_Cell_Size = sizeof(CEL_type);

UNS_type   Memory_Get_Size(PTR_type);
BYT_type    Memory_Get_Tag(PTR_type);
NIL_type Memory_Initialize(ADR_type,
                           UNS_type);
PTR_type Memory_Make_Chunk(BYT_type,
                           UNS_type);
```

pointers and cells

A Universal Memory Model (cont’d)

```
/*------------------------------ private constants ---------------------------*/

static const TXT_type IMM_error_string = "insufficient memory";

static const UNS_type Size_max = 0x00FFFFFF;
static const CEL_type Tag_mask = 0x000000FF;
static const BYT_type PTR_size = sizeof(PTR_type);

enum { Size_Shift = 8 };

/*------------------------------ private variables ---------------------------*/

static PTR_type Free_pointer;
static PTR_type Head_pointer;
static PTR_type Tail_pointer;

/*------------------------------ private functions ---------------------------*/

static CEL_type make_header(BYT_type Tag,
                            UNS_type Siz)
  { return (Siz << Size_Shift) | Tag; }
```

A Universal Memory Model (cont’d)

```
/*------------------------------ private constants ---------------------------*/

static const TXT_type IMM_error_string = "insufficient memory";

static const UNS_type Size_max = 0x00FFFFFF;
static const CEL_type Tag_mask = 0x000000FF;
static const BYT_type PTR_size = sizeof(PTR_type);

enum { Size_Shift = 8 };

/*------------------------------ private variables ---------------------------*/

static PTR_type Free_pointer;
static PTR_type Head_pointer;
static PTR_type Tail_pointer;
```

memory management

```
/*------------------------------ private functions ---------------------------*/

static CEL_type make_header(BYT_type Tag,
                            UNS_type Siz)
  { return (Siz << Size_Shift) | Tag; }
```

A Universal Memory Model (cont’d)

```
/*---------------------------- public functions ----------------------------*/

UNS_type Memory_Get_Size(PTR_type Pointer)
  { return Pointer->cel >> Size_Shift; }

BYT_type Memory_Get_Tag(PTR_type Pointer)
  { return (Pointer->cel & Tag_mask); }

NIL_type Memory_Initialize(ADR_type Address,
                           UNS_type Size   )
  { Head_pointer = (PTR_type)Address;
    Free_pointer = Head_pointer + 1;
    Tail_pointer = Head_pointer + Size; }

PTR_type Memory_Make_Chunk(BYT_type Tag ,
                           UNS_type Size)
  { PTR_type pointer;
    pointer = Free_pointer;
    Free_pointer->cel = make_header(Tag,
                                    Size);
    Free_pointer += Size + 1;
    if (Free_pointer >= Tail_pointer)
      Main_Fatal_Error(IMM_error_string);
    return pointer; }
```

A Universal Memory Model (cont’d)

```
/*------------------------------ public functions ----------------------------*/

UNS_type Memory_Get_Size(PTR_type Pointer)
  { return Pointer->cel >> Size_Shift; }

BYT_type Memory_Get_Tag(PTR_type Pointer)
  { return (Pointer->cel & Tag_mask); }

NIL_type Memory_Initialize(ADR_type Address,
                           UNS_type Size    )
  { Head_pointer = (PTR_type)Address;
    Free_pointer = Head_pointer + 1;
    Tail_pointer = Head_pointer + Size; }

PTR_type Memory_Make_Chunk(BYT_type Tag ,
                           UNS_type Size)
  { PTR_type pointer;
    pointer = Free_pointer;
    Free_pointer->cel = make_header(Tag,
                                    Size);
    Free_pointer += Size + 1;
    if (Free_pointer >= Tail_pointer)
      Main_Fatal_Error(IMM_error_string);
    return pointer; }
```

initialization

A Universal Memory Model (cont'd)

```
/*---------------------------- public functions ---------------------------*/

UNS_type Memory_Get_Size(PTR_type Pointer)
  { return Pointer->cel >> Size_Shift; }

BYT_type Memory_Get_Tag(PTR_type Pointer)
  { return (Pointer->cel & Tag_mask); }

NIL_type Memory_Initialize(ADR_type Address,
                           UNS_type Size    )
  { Head_pointer = (PTR_type)Address;
    Free_pointer = Head_pointer + 1;
    Tail_pointer = Head_pointer + Size; }

PTR_type Memory_Make_Chunk(BYT_type Tag ,
                           UNS_type Size)
  { PTR_type pointer;
    pointer = Free_pointer;
    Free_pointer->cel = make_header(Tag,
                                    Size);
    Free_pointer += Size + 1;
    if (Free_pointer >= Tail_pointer)
      Main_Fatal_Error(IMM_error_string);
    return pointer; }
```

allocation

A Universal Memory Model (cont’d)

```
/*--------------------------- public functions ---------------------------*/

UNS_type Memory_Get_Size(PTR_type Pointer)
  { return Pointer->cel >> Size_Shift; }

BYT_type Memory_Get_Tag(PTR_type Pointer)
  { return (Pointer->cel & Tag_mask); }

NIL_type Memory_Initialize(ADR_type Address,
                           UNS_type Size    )
  { Head_pointer = (PTR_type)Address;
    Free_pointer = Head_pointer + 1;
    Tail_pointer = Head_pointer + Size; }

PTR_type Memory_Make_Chunk(BYT_type Tag ,
                           UNS_type Size)
  { PTR_type pointer;
    pointer = Free_pointer;
    Free_pointer->cel = make_header(Tag,
                                    Size);
    Free_pointer += Size + 1;
    if (Free_pointer >= Tail_pointer)
      Main_Fatal_Error(IMM_error_string);
    return pointer; }
```

getters

Abstract Grammar

```
typedef enum { CFN_tag ,
               CHA_tag ,
               CNT_tag ,
               ENV_tag ,
               FLS_tag ,
               NAT_tag ,
               NBR_tag ,
               NUL_tag ,
               PAI_tag ,
               PRC_tag ,
               REA_tag ,
               STR_tag ,
               SYM_tag ,
               TRU_tag ,
               USP_tag ,
               VEC_tag } TAG_type;
```

```
typedef   NIL_type * EXP_type;

typedef   EXP_type * VEC_type;

typedef NIL_type (* CCC_type) (EXP_type,
                               CNT_type,
                               ENV_type);
typedef NIL_type (* FUN_type) (VEC_type,
                               CNT_type,
                               ENV_type);
```

```
typedef struct CFN * CFN_type;
typedef struct CHA * CHA_type;
typedef struct CNT * CNT_type;
typedef struct ENV * ENV_type;
typedef struct FLS * FLS_type;
typedef struct NAT * NAT_type;
typedef struct NBR * NBR_type;
typedef struct NUL * NUL_type;
typedef struct PAI * PAI_type;
typedef struct PRC * PRC_type;
typedef struct REA * REA_type;
typedef struct STR * STR_type;
typedef struct SYM * SYM_type;
typedef struct TRU * TRU_type;
typedef struct USP * USP_type;
```

Abstract Grammar

```
typedef enum { CFN_tag ,
               CHA_tag ,
               CNT_tag ,
               ENV_tag ,
               FLS_tag ,
               NAT_tag ,
               NBR_tag ,
               NUL_tag ,
               PAI_tag ,
               PRC_tag ,
               REA_tag ,
               STR_tag ,
               SYM_tag ,
               TRU_tag ,
               USP_tag ,
               VEC_tag } TAG_typ
```

CFN	→	C-function
CHA	→	character
CNT	→	continuation
ENV	→	environment
FLS	→	false
NAT	→	native function
NBR	→	number
NUL	→	empty list
PAI	→	pair
PRC	→	procedure
REA	→	real
STR	→	string
SYM	→	symbol
TRU	→	true
USP	→	unspecified
VEC	→	vector

```
f struct CFN * CFN_type;
f struct CHA * CHA_type;
f struct CNT * CNT_type;
f struct ENV * ENV_type;
f struct FLS * FLS_type;
f struct NAT * NAT_type;
f struct NBR * NBR_type;
f struct NUL * NUL_type;
f struct PAI * PAI_type;
f struct PRC * PRC_type;
f struct REA * REA_type;
f struct STR * STR_type;
f struct SYM * SYM_type;
f struct TRU * TRU_type;
f struct USP * USP_type;
```

```
typedef   NIL_type * EXP_type;

typedef   EXP_type * VEC_type;

typedef NIL_type (* CCC_type)


typedef NIL_type (* FUN_type)
                                CNT_type,
                                ENV_type);
```

Abstract Grammar (cont'd)

```
typedef
  struct CFN { CEL_type hdr;
               CCC_type ccc; } CFN;
BYT_type   is_CFN(EXP_type);
CFN_type make_CFN(CCC_type);

typedef
  struct CHA { CEL_type hdr;
               CHR_type chr; } CHA;
BYT_type   is_CHA(EXP_type);
CHA_type make_CHA(CHR_type);

typedef
  struct CNT { CEL_type hdr;
               CFN_type cfn;
               CNT_type cnt;
               EXP_type exp[]; } CNT;
BYT_type   is_CNT(EXP_type);
CNT_type make_CNT(CFN_type,
                  CNT_type,
                  UNS_type);

typedef
  struct ENV { CEL_type hdr;
               SYM_type var;
               EXP_type val;
               ENV_type env; } ENV;
BYT_type   is_ENV(EXP_type);
ENV_type make_ENV(SYM_type,
                  EXP_type,
                  ENV_type);

typedef
  struct FLS { CEL_type hdr; } FLS;
BYT_type   is_FLS(EXP_type);
FLS_type make_FLS(NIL_type);
```

```
typedef
  struct NAT { CEL_type hdr;
               FUN_type fun;
               CHR_type nam[]; } NAT;
BYT_type   is_NAT(EXP_type);
NAT_type make_NAT(FUN_type,
                  TXT_type);

typedef
  struct NBR { CEL_type hdr;
               LNG_type lng; } NBR;
BYT_type   is_NBR(EXP_type);
NBR_type make_NBR(LNG_type);

typedef
  struct NUL { CEL_type hdr; } NUL;
BYT_type   is_NUL(EXP_type);
NUL_type make_NUL(NIL_type);

typedef
  struct PAI { CEL_type hdr;
               EXP_type car;
               EXP_type cdr; } PAI;
BYT_type   is_PAI(EXP_type);
PAI_type make_PAI(EXP_type,
                  EXP_type);

typedef
  struct PRC { CEL_type hdr;
               SYM_type nam;
               EXP_type par;
               EXP_type bod;
               ENV_type clo; } PRC;
BYT_type   is_PRC(EXP_type);
PRC_type make_PRC(SYM_type,
                  EXP_type,
                  EXP_type,
                  ENV_type);
```

```
typedef
  struct REA { CEL_type hdr;
               FLO_type flo; } REA;
BYT_type   is_REA(EXP_type);
REA_type make_REA(FLO_type);

typedef
  struct STR { CEL_type hdr;
               CHR_type txt[]; } STR;
BYT_type   is_STR(EXP_type);
STR_type make_STR(TXT_type);

typedef
  struct SYM { CEL_type hdr;
               CHR_type txt[]; } SYM;
BYT_type   is_SYM(EXP_type);
SYM_type make_SYM(TXT_type);

typedef
  struct TRU { CEL_type hdr; } TRU;
BYT_type   is_TRU(EXP_type);
TRU_type make_TRU(NIL_type);

typedef
  struct USP { CEL_type hdr; } USP;
BYT_type   is_USP(EXP_type);
USP_type make_USP(NIL_type);

BYT_type   is_VEC(EXP_type);
VEC_type make_VEC(UNS_type);
UNS_type size_VEC(VEC_type);
```

Abstract Grammar (cont’d)

```
typedef
  struct CFN { CEL_type hdr;
               CCC_type ccc; } CFN;
BYT_type   is_CFN(EXP_type);
CFN_type make_CFN(CCC_type);

typedef
  struct CHA { CEL_type hdr;
               CHR_type chr; } CHA;
BYT_type   is_CHA(EXP_type);
CHA_type make_CHA(CHR_type);

typedef
  struct CNT { CEL_type hdr;
               CFN_type cfn;
               CNT_type cnt;
               EXP_type exp[]; }
BYT_type   is_CNT(EXP_type);
CNT_type make_CNT(CFN_type,
                  CNT_type,
                  UNS_type);

typedef
  struct ENV { CEL_type hdr;
               SYM_type var;
               EXP_type val;
               ENV_type env; } E
BYT_type   is_ENV(EXP_type);
ENV_type make_ENV(SYM_type,
                  EXP_type,
                  ENV_type);

typedef
  struct FLS { CEL_type hdr; } FLS;
BYT_type   is_FLS(EXP_type);
FLS_type make_FLS(NIL_type);
```

```
typedef
  struct NAT { CEL_type hdr;
               FUN_type fun;
               CHR_type nam[]; } NAT;
BYT_type   is_NAT(EXP_type);
NAT_type make_NAT(FUN_type,
                  TXT_type);

typedef

typedef
  struct PRC { CEL_type hdr;
               SYM_type nam;
               EXP_type par;
               EXP_type bod;
               ENV_type clo; } PRC;
BYT_type   is_PRC(EXP_type);
PRC_type make_PRC(SYM_type,
                  EXP_type,
                  EXP_type,
                  ENV_type);
```

```
typedef
  struct REA { CEL_type hdr;
               FLO_type flo; } REA;
BYT_type   is_REA(EXP_type);
REA_type make_REA(FLO_type);

typedef
  struct STR { CEL_type hdr;
               CHR_type txt[]; } STR;
_type   is_STR(EXP_type);
_type make_STR(TXT_type);

edef
truct SYM { CEL_type hdr;
            CHR_type txt[]; } SYM;
_type   is_SYM(EXP_type);
_type make_SYM(TXT_type);

edef
truct TRU { CEL_type hdr; } TRU;
_type   is_TRU(EXP_type);
_type make_TRU(NIL_type);

edef
truct USP { CEL_type hdr; } USP;
_type   is_USP(EXP_type);
_type make_USP(NIL_type);

BYT_type   is_VEC(EXP_type);
VEC_type make_VEC(UNS_type);
UNS_type size_VEC(VEC_type);
```

```
typedef
  struct PRC { CEL_type hdr;
               SYM_type nam;
               EXP_type par;
               EXP_type bod;
               ENV_type clo; } PRC;
BYT_type   is_PRC(EXP_type);
PRC_type make_PRC(SYM_type,
                  EXP_type,
                  EXP_type,
                  ENV_type);
```

Abstract Grammar (cont’d)

```
typedef
  struct CFN { CEL_type hdr;
               CCC_type ccc; } CFN;
BYT_type   is_CFN(EXP_type);
CFN_type make_CFN(CCC_type);

typedef
  struct CHA { CEL_type hdr;
               CHR_type chr; } CHA;
BYT_type   is_CHA(EXP_type);
CHA_type make_CHA(CHR_type);

typedef
  struct CNT { CEL_type hdr;
               CFN_type cfn;
               CNT_type cnt;
               EXP_type exp[];
BYT_type   is_CNT(EXP_type);
CNT_type make_CNT(CFN_type,
                  CNT_type,
                  UNS_type);

typedef
  struct ENV { CEL_type hdr;
               SYM_type var;
               EXP_type val;
               ENV_type env; } E
BYT_type   is_ENV(EXP_type);
ENV_type make_ENV(SYM_type,
                  EXP_type,
                  ENV_type);

typedef
  struct FLS { CEL_type hdr; } FLS;
BYT_type   is_FLS(EXP_type);
FLS_type make_FLS(NIL_type);
```

```
typedef
  struct NAT { CEL_type hdr;
               FUN_type fun;
               CHR_type nam[]; } NAT;
BYT_type   is_NAT(EXP_type);
NAT_type make_NAT(FUN_type,
                  TXT_type);

typedef
```

```
typedef
  struct PRC { CEL_type hdr;
```

```
typedef
  struct STR { CEL_type hdr;
               CHR_type txt[]; } STR;
BYT_type   is_STR(EXP_type);
STR_type make_STR(TXT_type);
```

```
                  EXP_type,
                  EXP_type,
                  ENV_type);
```

```
typedef
  struct PRC { CEL_type hdr;
               SYM_type nam;
               EXP_type par;
               EXP_type bod;
               ENV_type clo; } PRC;
BYT_type   is_PRC(EXP_type);
PRC_type make_PRC(SYM_type,
                  EXP_type,
                  EXP_type,
                  ENV_type);
```

```
typedef
  struct REA { CEL_type hdr;
               FLO_type flo; } REA;
BYT_type   is_REA(EXP_type);
REA_type make_REA(FLO_type);

typedef
  struct STR { CEL_type hdr;
               CHR_type txt[]; } STR;
_type   is_STR(EXP_type);
_type make_STR(TXT_type);

edef
uct SYM { CEL_type hdr;
          CHR_type txt[]; } SYM;
type   is_SYM(EXP_type);
type make_SYM(TXT_type);

def
uct TRU { CEL_type hdr; } TRU;
type   is_TRU(EXP_type);
type make_TRU(NIL_type);

edef
truct USP { CEL_type hdr; } USP;
_type   is_USP(EXP_type);
_type make_USP(NIL_type);

BYT_type   is_VEC(EXP_type);
VEC_type make_VEC(UNS_type);
UNS_type size_VEC(VEC_type);
```

Abstract Grammar (cont’d)

```
typedef
  struct CFN { CEL_type hdr;
               CCC_type ccc; } CFN;
BYT_type   is_CFN(EXP_type);
CFN_type make_CFN(CCC_type);

typedef
  struct CHA { CEL_type hdr;
               CHR_type chr; } CHA;
BYT_type   is_CHA(EXP_type);
CHA_type make_CHA(CHR_type);

typedef
  struct CNT { CEL_type hdr;
               CFN_type cfn;
               CNT_type cnt;
               EXP_type exp[];
BYT_type   is_CNT(EXP_type);
CNT_type make_CNT(CFN_type,
                  CNT_type,
                  UNS_type);

typedef
  struct ENV { CEL_type hdr;
               SYM_type var;
               EXP_type val;
               ENV_type env; } E
BYT_type   is_ENV(EXP_type);
ENV_type make_ENV(SYM_type,
                  EXP_type,
                  ENV_type);

typedef
  struct FLS { CEL_type hdr; } FLS;
BYT_type   is_FLS(EXP_type);
FLS_type make_FLS(NIL_type);
```

```
typedef
  struct NAT { CEL_type hdr;
               FUN_type fun;
               CHR_type nam[]; } NAT;
BYT_type   is_NAT(EXP_type);
NAT_type make_NAT(FUN_type,
                  TXT_type);

typedef

typedef
  struct PRC { CEL_type hdr;
               SYM_type nam;
               EXP_type par;
               EXP_type bod;
               ENV_type clo; } PRC;
BYT_type   is_PRC(EXP_type);
PRC_type make_PRC(SYM_type,
                  EXP_type,
                  EXP_type,
                  ENV_type);
```

```
typedef
  struct REA { CEL_type hdr;
               FLO_type flo; } REA;
BYT_type   is_REA(EXP_type);
REA_type make_REA(FLO_type);

typedef
  struct STR { CEL_type hdr;
               CHR_type txt[]; } STR;
 _type   is_STR(EXP_type);
 _type make_STR(TXT_type);

edef
 uct SYM { CEL_type hdr;
               CHR_type txt[]; } SYM;
 type   is_SYM(EXP_type);
 type make_SYM(TXT_type);

 def
 uct TRU { CEL_type hdr; } TRU;
 type   is_TRU(EXP_type);
 type make_TRU(NIL_type);

edef
truct USP { CEL_type hdr; } USP;
_type   is_USP(EXP_type);
_type make_USP(NIL_type);

BYT_type   is_VEC(EXP_type);
VEC_type make_VEC(UNS_type);
UNS_type size_VEC(VEC_type);
```

```
typedef
  struct PRC { CEL_type hdr;

                  EXP_type,
                  EXP_type,
                  ENV_type);
```

```
typedef
  st
                                  STR;
BYT_
STR_type make_STR(TXT_type);
```

```
BYT_type   is_VEC(EXP_type);
VEC_type make_VEC(UNS_type);
UNS_type size_VEC(VEC_type);
```

Abstract Grammar (cont’d)

```
/*------------------------------ private macros ------------------------------*/

#define chunk_size(TYP)                                                        \
  (sizeof(TYP) / sizeof(CEL_type) - 1)

#define make_chunk_with_offset(TYP, SIZ)                                       \
  (TYP##_type)Memory_Make_Chunk(TYP##_tag, TYP##_size + SIZ)

#define make_chunk(TYP)                                                        \
  make_chunk_with_offset(TYP, 0)

/*------------------------------ private functions ---------------------------*/

static BLN_type member_of(EXP_type Expression,
                          TAG_type Tag)
  { BYT_type tag;
    tag = Tag_of(Expression);
    return (tag == Tag); }

static UNS_type string_to_expression_size(TXT_type Raw_string)
  { return strlen(Raw_string) / sizeof(CEL_type) + 1; }

/*-------------------------------------Tag -----------------------------------*/

TAG_type Tag_of(EXP_type Expression)
  { return (TAG_type)Memory_Get_Tag((PTR_type)Expression); }
```

Abstract Grammar (cont’d)

```
static const UNS_type PRC_size = chunk_size(PRC);

BYT_type is_PRC(EXP_type Expression)
  { return member_of(Expression,
                     PRC_tag); }

PRC_type make_PRC(SYM_type Name,
                  EXP_type Paramet
                  EXP_type Body,
                  ENV_type Closure
  { PRC_type procedure;
    procedure = make_chunk(PRC);
    procedure->nam = Name;
    procedure->par = Parameters;
```

```
static const UNS_type STR_size = chunk_size(STR);

BYT_type is_STR(EXP_type Expression)
  { return member_of(Expression,
                     STR_tag); }

STR_type make_STR(TXT_type Raw_string)
  { STR_type string;
    UNS_type size;
                                  on_size(Raw_string);
                                  offset(STR,
                                         size);
```

```
static const UNS_type VEC_size = 0;

BYT_type is_VEC(EXP_type Expression)
  { return member_of(Expression,
                     VEC_tag); }

VEC_type make_VEC(UNS_type Size)
  { VEC_type vector;
    vector = (VEC_type)make_chunk_with_offset(VEC,
                                              Size);
    return vector; }

UNS_type size_VEC(VEC_type Vector)
  { return (UNS_type) Memory_Get_Size((PTR_type)Vector); }
```

Abstract Grammar (cont’d)

```
static const UNS_type PRC_size = chunk_size(PRC);

BYT_type is_PRC(EXP_type Expression)
  { return member_of(Expression,
                     PRC_tag); }

PRC_type make_PRC(SYM_type Name,
                  EXP_type Parameters,
                  EXP_type Body,
                  ENV_type Closure)
  { PRC_type procedure;
    procedure = make_chunk(PRC);
    procedure->nam = Name;
    procedure->par = Parameters;
    procedure->bod = Body;
    procedure->clo = Closure;
    return procedure; }
```

```
_type STR_size = chunk_size(STR);

(EXP_type Expression)
r_of(Expression,
     STR_tag); }

R(TXT_type Raw_string)
ing;

on_size(Raw_string);
offset(STR,
       size);
```

```
  { return member_of(Expression,
                     VEC_tag); }

VEC_type make_VEC(UNS_type Size)
  { VEC_type vector;
    vector = (VEC_type)make_chunk_with_offset(VEC,
                                              Size);
    return vector; }

UNS_type size_VEC(VEC_type Vector)
  { return (UNS_type) Memory_Get_Size((PTR_type)Vector); }
```

Abstract Grammar (cont'd)

```
static const UNS_type PRC_size = chunk_size(PRC);

BYT_type is_PRC(EXP_type Expression)
  { return member_of(Expression,
                     PRC_tag); }

PRC_type make_PRC(SYM_type Name,
                  EXP_type Paramet
                  EXP_type Body,
                  ENV_type Closur
  { PRC_type procedure;
    procedure = make_chunk(PRC);
    procedure->nam = Name;
    procedure->par = Parameters;
```

```
static const UNS_type VEC_siz

BYT_type is_VEC(EXP_type Expr
  { return member_of(Expressi
                     VEC_tag)

VEC_type make_VEC(UNS_type Si
  { VEC_type vector;
    vector = (VEC_type)make_chunk_with_offset(VEC,
                                              Size);
    return vector; }

UNS_type size_VEC(VEC_type Vector)
  { return (UNS_type) Memory_Get_Size((PTR_type)Vector); }
```

```
static const UNS_type STR_size = chunk_size(STR);

BYT_type is_STR(EXP_type Expression)
  { return member_of(Expression,
                     STR_tag); }

STR_type make_STR(TXT_type Raw_string)
  { STR_type string;
    UNS_type size;
    size = string_to_expression_size(Raw_string);
    string = make_chunk_with_offset(STR,
                                    size);
    strcpy(string->txt,
           Raw_string);
    return string; }
```

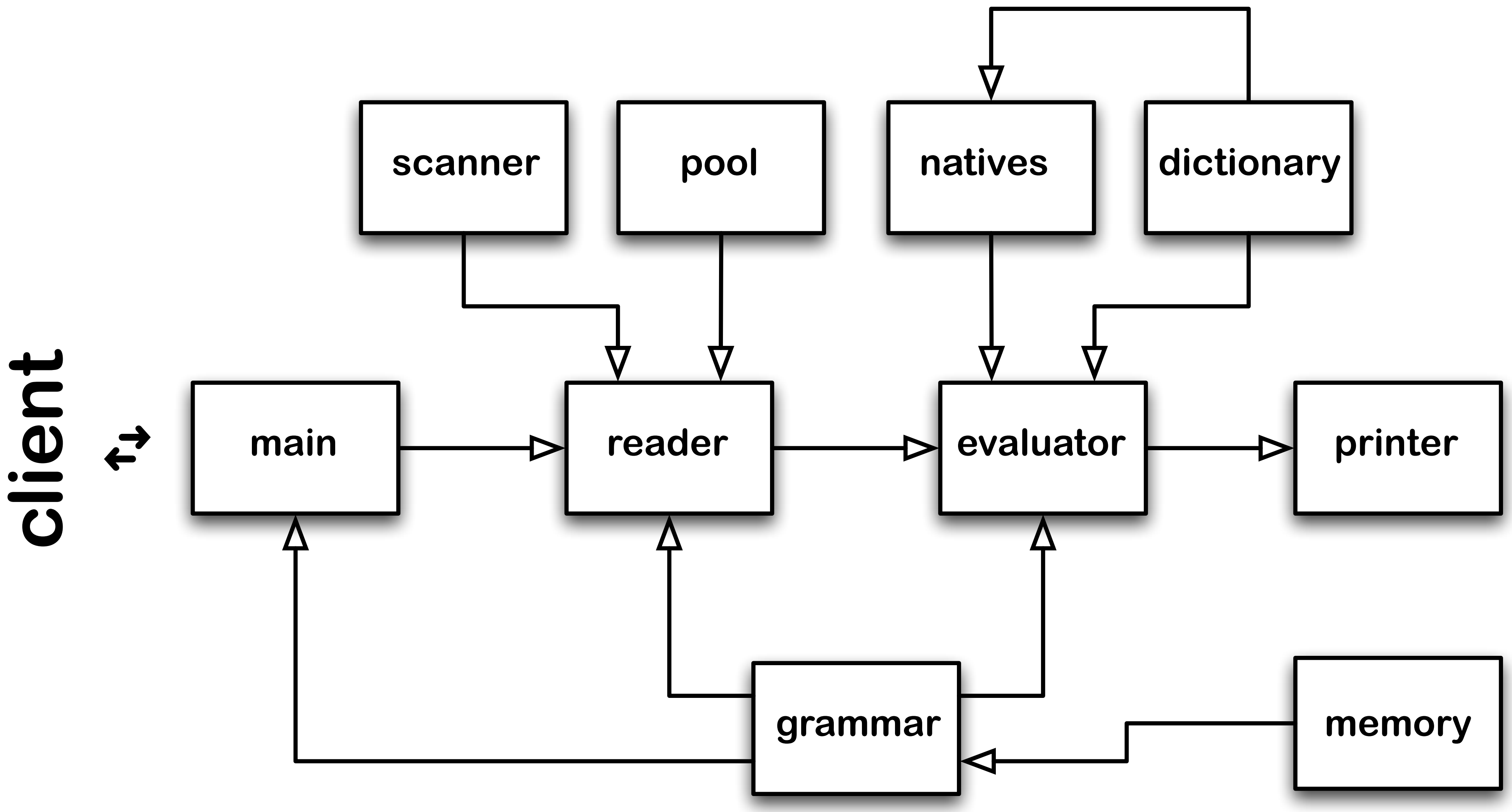
Abstract Grammar (cont'd)

```
static const UNS_type PRC_size = chunk_size(PRC);

BYT_type is_PRC(EXP_type Expression)
  { return member_of(Expression,
                     PRC_tag); }

PRC_type make_PRC(SYM_type Name,
                  EXP_type Paramet
                  EXP_type Body,
                  ENV_type Closure
  { PRC_type procedure;
    procedure = make_chunk(PRC);
    procedure->nam = Name;
    procedure->par = Parameters;
```

```
static const UNS_type STR_size = chunk_size(STR);

BYT_type is_STR(EXP_type Expression)
  { return member_of(Expression,
                     STR_tag); }

STR_type make_STR(TXT_type Raw_string)
  { STR_type string;
    UNS_type size;

                             on_size(Raw_string);
                             offset(STR,
                                    size);
```

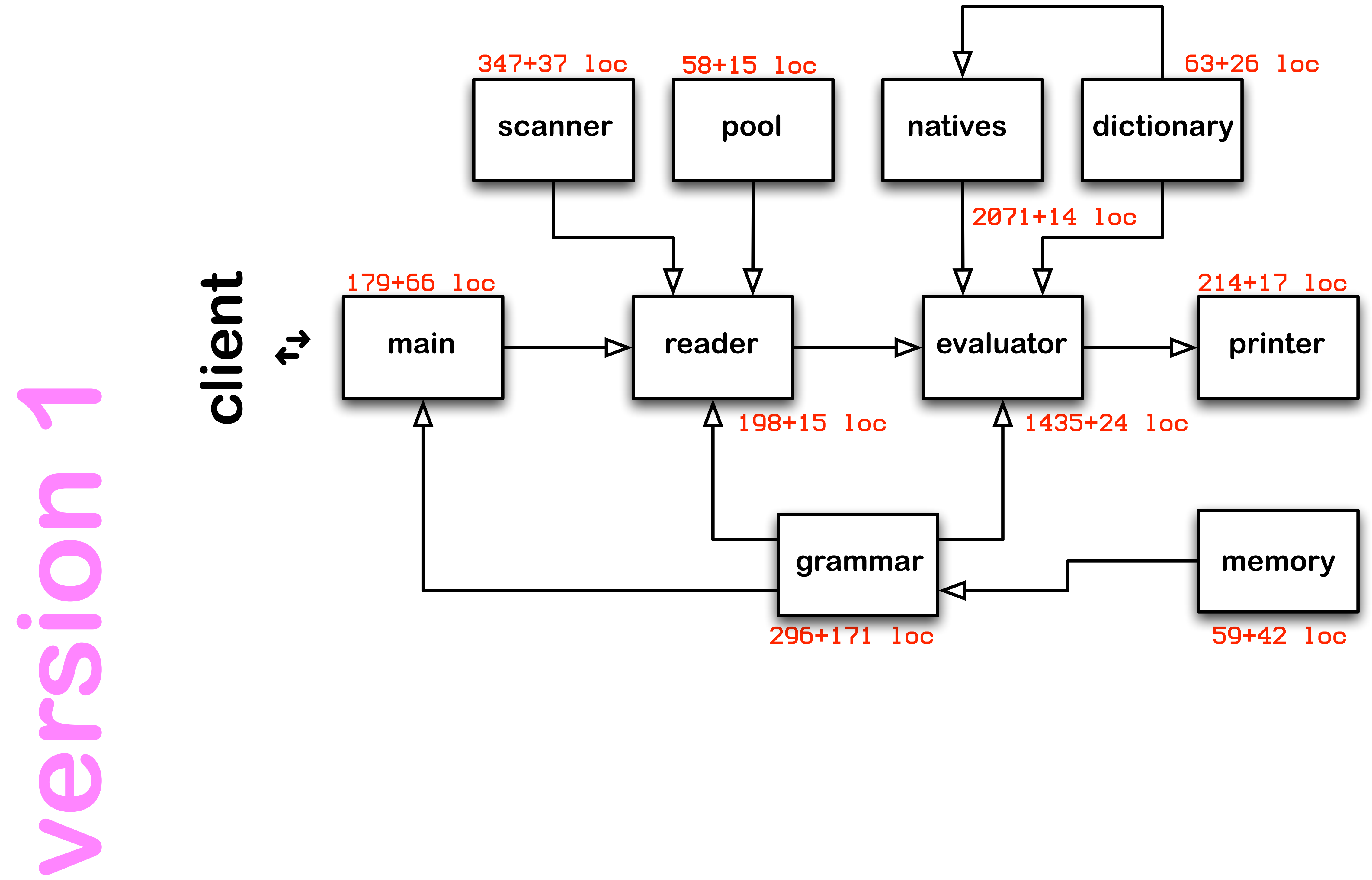
```
static const UNS_type VEC_size = 0;

BYT_type is_VEC(EXP_type Expression)
  { return member_of(Expression,
                     VEC_tag); }

VEC_type make_VEC(UNS_type Size)
  { VEC_type vector;
    vector = (VEC_type)make_chunk_with_offset(VEC,
                                              Size);
    return vector; }

UNS_type size_VEC(VEC_type Vector)
  { return (UNS_type) Memory_Get_Size((PTR_type)Vector); }
```

Virtual Machine Architecture



version 1

Virtual Machine Architecture



5347 loc

Client Interface

```
/*-----------------------------------*/
/*             >>>Slip<<<            */
/*            Theo D'Hondt           */
/*     VUB Software Languages Lab    */
/*             (c) 2010              */
/*-----------------------------------*/
/*  version 1: straightforward code  */
/*-----------------------------------*/
/*               Slip                */
/*-----------------------------------*/
```

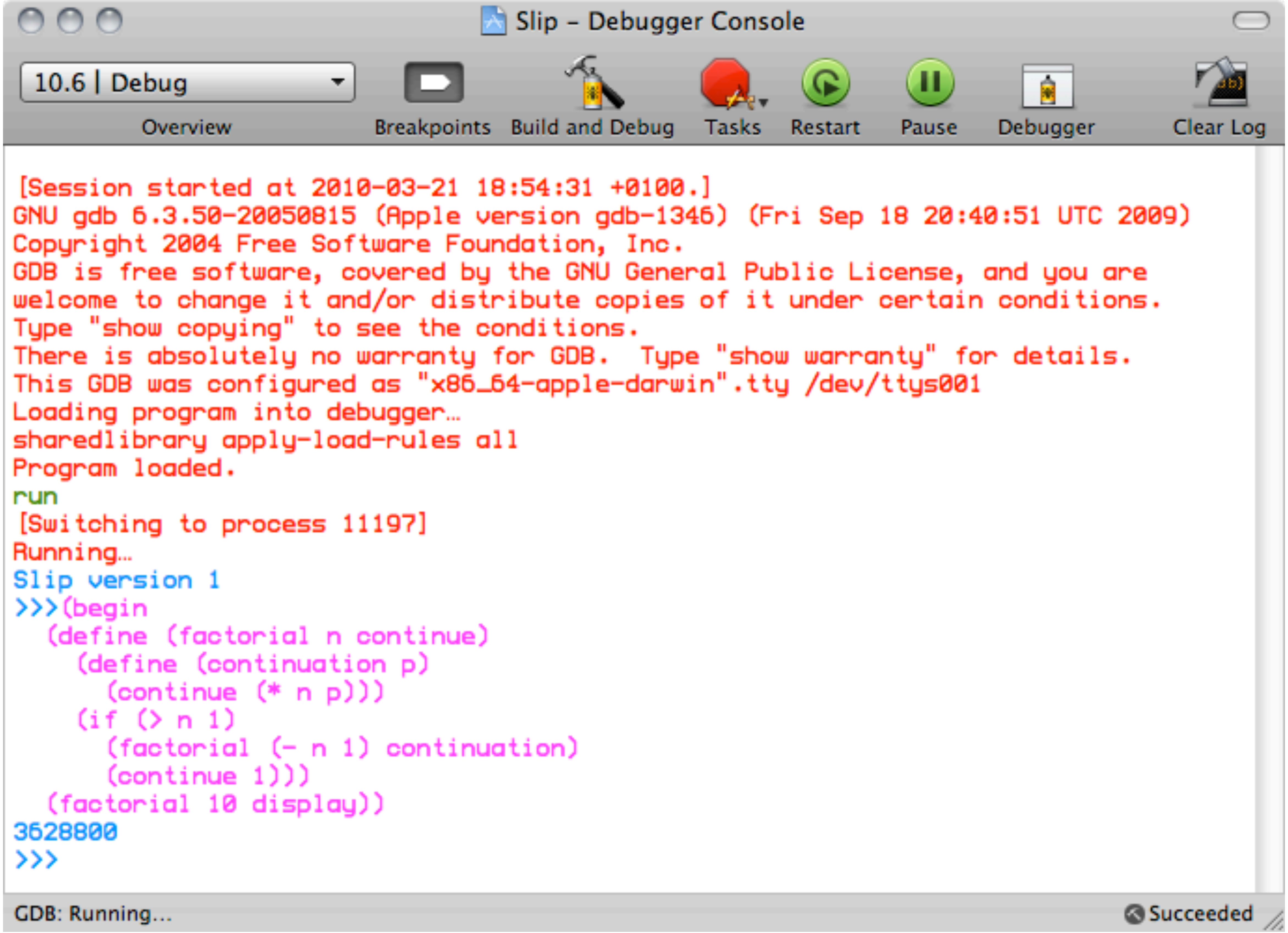
```
/*------------------------------ imported functions --------------------------
*/

void Slip_Load(char  *, char **);
void Slip_Print(char  *);
void Slip_Read(char **);

/*------------------------------ exported functions --------------------------
*/

void Slip_REP(char  *, int    );
```

Client Example

```
#include <stdio.h>

#include "SlipSlip.h"

enum { Buffer_size = 10000,
       Memory_size = 100000000 };

static char Buffer[Buffer_size];
static char Memory[Memory_size];

void Slip_Load(char  * text,
               char ** input)
  { FILE * stream = fopen(text, "r");
    *input = Buffer;
    Buffer[0] = 0;
    if (stream)
      { unsigned index;
        char character;
        for (index = 0;;)
          { int result = fscanf(stream, "%c", &character);
            if (result == EOF)
              { Buffer[index] = 0;
                fclose(stream);
                return; }
            else
              Buffer[index++] = character; }}
    else
      printf("%s not found\n", text); }
```

```
void Slip_Print(char * string)
  { printf("%s", string); }

void Slip_Read(char ** input)
  { unsigned index;
    char character;
    for (index = 0;;)
      { scanf("%c", &character);
        if (character == '\n')
          { Buffer[index] = 0;
            *input = Buffer;
            return; }
        Buffer[index++] = character; }}

int main (int argc, const char * argv[])
  { Slip_REP(Memory, Memory_size);
    return 0; }
```

Client Example (cont'd)



```
[Session started at 2010-03-21 18:54:31 +0100.]
GNU gdb 6.3.50-20050815 (Apple version gdb-1346) (Fri Sep 18 20:40:51 UTC 2009)
Copyright 2004 Free Software Foundation, Inc.
GDB is free software, covered by the GNU General Public License, and you are
welcome to change it and/or distribute copies of it under certain conditions.
Type "show copying" to see the conditions.
There is absolutely no warranty for GDB.  Type "show warranty" for details.
This GDB was configured as "x86_64-apple-darwin".tty /dev/ttys001
Loading program into debugger…
sharedlibrary apply-load-rules all
Program loaded.
run
[Switching to process 11197]
Running…
Slip version 1
>>>(begin
   (define (factorial n continue)
     (define (continuation p)
       (continue (* n p)))
     (if (> n 1)
       (factorial (- n 1) continuation)
       (continue 1)))
   (factorial 10 display))
3628800
>>>
```

Parsing Slip

```
‹expression›     ::=  ‹computation›|‹lambda›|‹quote›|‹vector›|
                      ‹variable›|‹literal›|‹null›
‹computation›    ::=  ‹definition›|‹assignment›|‹sequence›|
                      ‹conditional›|‹iteration›|‹application›
‹definition›     ::=  (define ‹variable› ‹expression›)
‹definition›     ::=  (define ‹pattern› ‹expression›+)
‹assignment›     ::=  (set! ‹variable› ‹expression›)
‹sequence›       ::=  (begin ‹expression›+)
‹conditional›    ::=  (if ‹expression› ‹expression› ‹expression›)
‹conditional›    ::=  (if ‹expression› ‹expression›)
‹iteration›      ::=  (while ‹expression› ‹expression›+)
‹application›    ::=  (‹expression›+)
‹lambda›         ::=  (lambda () ‹expression›+)
‹lambda›         ::=  (lambda ‹variable› ‹expression›+)
‹lambda›         ::=  (lambda (‹pattern›) ‹expression›+)
‹quote›          ::=  ' ‹expression› | ‹pattern›
‹vector›         ::=  [ (‹expression› | ‹pattern›)+ ]
‹variable›       ::=  «symbol»
‹pattern›        ::=  (‹variable›+)
‹pattern›        ::=  (‹variable›+ . ‹variable›)
‹literal›        ::=  «number» | «character» | «string» | #t | #f
‹null›           ::=  ()
```

Parsing Slip (cont’d)

```
/*------------------------------ public types --------------------------------*/

typedef enum { CHA_token =  0,                           /* character         */
               END_token =  1,                           /* eof               */
               FLS_token =  2,                           /* false             */
               IDT_token =  3,                           /* identifier        */
               LBR_token =  4,                           /* left bracket      */
               LPR_token =  5,                           /* left parenthesis  */
               NBR_token =  6,                           /* number            */
               PER_token =  7,                           /* period            */
               QUO_token =  8,                           /* quote             */
               RBR_token =  9,                           /* right bracket     */
               RPR_token = 10,                           /* right parenthesis */
               REA_token = 11,                           /* real              */
               STR_token = 12,                           /* string            */
               TRU_token = 13 } SCA_type;                /* true              */

/*------------------------------ public prototypes ---------------------------*/

NIL_type Scan_Initialize(NIL_type);
SCA_type       Scan_Next(NIL_type);
NIL_type     Scan_Preset(TXT_type);

/*------------------------------ public variables ----------------------------*/

extern CHR_type Scan_String[];
```

Parsing Slip (cont’d)

```
typedef enum { Apo =  0,       /* ' */
               Bks =  1,       /* \ */
               Dgt =  2,       /* 0 1 2 3 4 5 6 7 8 9 */
               Eol =  3,       /*   */
               Exp =  4,       /* e E */
               Fls =  5,       /* f F */
               Hsh =  6,       /* # */
               Ill =  7,       /*   */
               Opr =  8,       /* ! $ % & * / : < = > ? ^ _ ~ */
               Lbr =  9,       /* [ */
               Lpr = 10,       /* ( */
               Ltr = 11,       /* a A b B ... z Z */
               Mns = 12,       /* - */
               Nul = 13,       /*   */
               Per = 14,       /* . */
               Pls = 15,       /* + */
               Quo = 16,       /* " */
               Rbr = 17,       /* ] */
               Rpr = 18,       /* ) */
               Smc = 19,       /* ; */
               Tru = 20,       /* t T */
               Wsp = 21 } CAT_type;
```

Parsing Slip (cont’d)

```
typedef enum { Apo =  0,        /* ' */
               Bks =  1,        /* \ */
               Dgt =  2,        /* 0 1 2 3 4 5 6 7 8 9 */
               Eol =  3,        /*   */
               Exp =  4,        /* e E */
               Fls =  5,        /*
               Hsh =  6,        /*
               Ill =  7,        /*
               Opr =  8,        /*
               Lbr =  9,        /*
               Lpr = 10,        /*
               Ltr = 11,        /*
               Mns = 12,        /*
               Nul = 13,        /*
               Per = 14,        /*
               Pls = 15,        /*
               Quo = 16,        /*
               Rbr = 17,        /*
               Rpr = 18,        /*
               Smc = 19,        /*
               Tru = 20,        /*
               Wsp = 21 } CAT_type
```

```
static const CAT_type ASCII_Table[] =
      /*NUL SOH STX ETX EOT ENQ ACK BEL BS  HT  LF  VT  FF  CR  SO  SI */
      { Nul,Ill,Ill,Ill,Ill,Ill,Ill,Ill,Ill,Wsp,Eol,Ill,Wsp,Eol,Ill,Ill,
      /*DLE DC1 DC2 DC3 DC4 NAK SYN ETB CAN EM  SUB ESC FS  GS  RS  US */
        Ill,Ill,Ill,Ill,Ill,Ill,Ill,Ill,Ill,Ill,Ill,Ill,Ill,Ill,Ill,Ill,
      /*    !   "   #   $   %   &   '   (   )   *   +   ,   -   .   / */
        Wsp,Opr,Quo,Hsh,Opr,Opr,Opr,Apo,Lpr,Rpr,Opr,Pls,Ill,Mns,Per,Opr,
      /* 0   1   2   3   4   5   6   7   8   9   :   ;   <   =   >   ? */
        Dgt,Dgt,Dgt,Dgt,Dgt,Dgt,Dgt,Dgt,Dgt,Dgt,Opr,Smc,Opr,Opr,Opr,Opr,
      /* @   A   B   C   D   E   F   G   H   I   J   K   L   M   N   O */
        Ill,Ltr,Ltr,Ltr,Ltr,Exp,Fls,Ltr,Ltr,Ltr,Ltr,Ltr,Ltr,Ltr,Ltr,Ltr,
      /* P   Q   R   S   T   U   V   W   X   Y   Z   [   \   ]   ^   _ */
        Ltr,Ltr,Ltr,Ltr,Tru,Ltr,Ltr,Ltr,Ltr,Ltr,Ltr,Lbr,Bks,Rbr,Opr,Opr,
      /* `   a   b   c   d   e   f   g   h   i   j   k   l   m   n   o */
        Ill,Ltr,Ltr,Ltr,Ltr,Exp,Fls,Ltr,Ltr,Ltr,Ltr,Ltr,Ltr,Ltr,Ltr,Ltr,
      /* p   q   r   s   t   u   v   w   x   y   z   {   |   }   ~  DEL*/
        Ltr,Ltr,Ltr,Ltr,Tru,Ltr,Ltr,Ltr,Ltr,Ltr,Ltr,Ill,Ill,Ill,Opr,Ill,
      /*                                                               */
        Ill,Ill,Ill,Ill,Ill,Ill,Ill,Ill,Ill,Ill,Ill,Ill,Ill,Ill,Ill,Ill,
      /*                                                               */
        Ill,Ill,Ill,Ill,Ill,Ill,Ill,Ill,Ill,Ill,Ill,Ill,Ill,Ill,Ill,Ill,
      /*                                                               */
        Ill,Ill,Ill,Ill,Ill,Ill,Ill,Ill,Ill,Ill,Ill,Ill,Ill,Ill,Ill,Ill,
      /*                                                               */
        Ill,Ill,Ill,Ill,Ill,Ill,Ill,Ill,Ill,Ill,Ill,Ill,Ill,Ill,Ill,Ill,
      /*                                                               */
        Ill,Ill,Ill,Ill,Ill,Ill,Ill,Ill,Ill,Ill,Ill,Ill,Ill,Ill,Ill,Ill,
      /*                                                               */
        Ill,Ill,Ill,Ill,Ill,Ill,Ill,Ill,Ill,Ill,Ill,Ill,Ill,Ill,Ill,Ill,
      /*                                                               */
        Ill,Ill,Ill,Ill,Ill,Ill,Ill,Ill,Ill,Ill,Ill,Ill,Ill,Ill,Ill,Ill,
      /*                                                               */
        Ill,Ill,Ill,Ill,Ill,Ill,Ill,Ill,Ill,Ill,Ill,Ill,Ill,Ill,Ill,Ill };
```

Parsing Slip (cont’d)

```
category = ASCII_Table[character];
    switch (category)
      { case Apo: return Apo_fun();
        case Bks: return Ill_fun();
        case Dgt: return Nbr_fun();
        case Eol: return Wsp_fun();
        case Exp: return Idt_fun();
        case Fls: return Idt_fun();
        case Hsh: return Hsh_fun();
        case Ill: return Ill_fun();
        case Opr: return Idt_fun();
        case Lpr: return Lpr_fun();
        case Lbr: return Lbr_fun();
        case Ltr: return Idt_fun();
        case Mns: return Sgn_fun();
        case Nul: return Nul_fun();
        case Per: return Per_fun();
        case Pls: return Sgn_fun();
        case Quo: return Quo_fun();
        case Rbr: return Rbr_fun();
        case Rpr: return Rpr_fun();
        case Smc: return Smc_fun();
        case Tru: return Idt_fun();
        case Wsp: return Wsp_fun(); }
```

Parsing Slip (cont’d)

```
category = ASCII_Table[character];
    switch (category)
      { case Apo: return Apo_fun();
        case Bks: return Ill_fun();
        case Dgt: return Nbr_fun();
        case Eol: return Wsp_fun();
        case Exp: return Idt_fun();
        case Fls: return Idt_fun();
        case Hsh: return Hsh_fun();
        case Ill: return Ill_fun();
        case Opr: return Idt_fun();
        case Lpr: return Lpr_fun();
        case Lbr: return Lbr_fun();
        case Ltr: return Idt_fun();
        case Mns: return Sgn_fun();
        case Nul: return Nul_fun();
        case Per: return Per_fun();
        case Pls: return Sgn_fun();
        case Quo: return Quo_fun();
        case Rbr: return Rbr_fun();
        case Rpr: return Rpr_fun();
        case Smc: return Smc_fun();
        case Tru: return Idt_fun();
        case Wsp: return Wsp_fun(); }
```

```
static SCA_type number(NIL_type)
  { SCA_type token;
    token = NBR_token;
    copy_and_get_while(dig_allowed);
    if (check(per_allowed))
      { token = REA_token;
        copy_and_get_char();
        integer(); }
    if (check(exp_allowed))
      { token = REA_token;
        copy_and_get_char();
        if (check(sgn_allowed))
          copy_and_get_char();
        integer(); }
    return stop_copy_text_return(token); }
```

Parsing Slip (cont’d)

```
category = ASCII_Table[character];
    switch (category)
      { case Apo: return Apo_fun();
        case Bks: return Ill_fun();
        case Dgt: return Nbr_fun();
        case Eol: return Wsp_fun();
        case Exp: return Idt_fun();
        case Fls: return Idt_fun();
        case Hsh: return Hsh_fun();
        case Ill: return Ill_fun();
        case Opr: return Idt_fun();
        case Lpr: return Lpr_fun();
        case Lbr: return Lbr_fun();
        case Ltr: return Idt_fun();
        case Mns: return Sgn_fun(
        case Nul: return Nul_fun(
        case Per: return Per_fun(
        case Pls: return Sgn_fun(
        case Quo: return Quo_fun(
        case Rbr: return Rbr_fun(
        case Rpr: return Rpr_fun(
        case Smc: return Smc_fun(
        case Tru: return Idt_fun(
        case Wsp: return Wsp_fun(); }
```

```
static SCA_type number(NIL_type)
  { SCA_type token;
    token = NBR_token;
    copy_and_get_while(dig_allowed);
    if (check(per_allowed))
      { token = REA_token;
        copy_and_get_char();
        integer(); }
    if (check(exp_allowed))
      { token = REA_token;
        copy_and_get_char();
        if (check(sgn_allowed))
          copy_and_get_char();
        integer(); }
    return stop_copy_text_return(token); }
```

```
static SCA_type Hsh_fun(NIL_type)
  { next_character();
    if (check(tru_allowed))
      return next_character_return(TRU_token);
    if (check(fls_allowed))
      return next_character_return(FLS_token);
    if (check(bks_allowed))
      return character();
    return error_and_return(ILH_error_string); }
```

Parsing Slip (cont'd)

```
static EXP_type read_expression(NIL_type)
  { switch (Token)
      { case CHA_token:
          return read_character();
        case END_token:
          return read_end_program();
        case FLS_token:
          return read_false();
        case IDT_token:
          return read_symbol();
        case LBR_token:
          return read_vector();
        case LPR_token:
          return read_list();
        case NBR_token:
          return read_number();
        case PER_token:
          return read_period();
        case QUO_token:
          return read_quote();
        case RPR_token:
          return read_parenthesis();
        case REA_token:
          return read_real();
        case STR_token:
          return read_string();
        case TRU_token:
          return read_true(); }
    return Main_Null; }
```

198 loc

Parsing Slip (cont’d)

```
static EXP_type read_expression(NIL_type)
  { switch (Token)
      { case CHA_token:
          return read_character();
        case END_token:
          return read_end_program();
        case FLS_token:
          return read_false();
        case IDT_token:
          return read_symbol();
        case LBR_token:
          return read_vector();
        case LPR_token:
          return read_list();
        case NBR_token:
          return read_number();
        case PER_token:
          return read_period();
        case QUO_token:
          return read_quote();
        case RPR_token:
          return read_parenthesis();
        case REA_token:
          return read_real();
        case STR_token:
          return read_string();
        case TRU_token:
          return read_true(); }
    return Main_Null; }
```

```
EXP_type Read_Parse(TXT_type Input)
  { EXP_type expression;
    Scan_Preset(Input);
    next_token();
    expression = read_expression();
    if (Token != END_token)
      Main_Error(XCT_error_string);
    return expression; }
```

198 loc

Parsing Slip (cont’d)

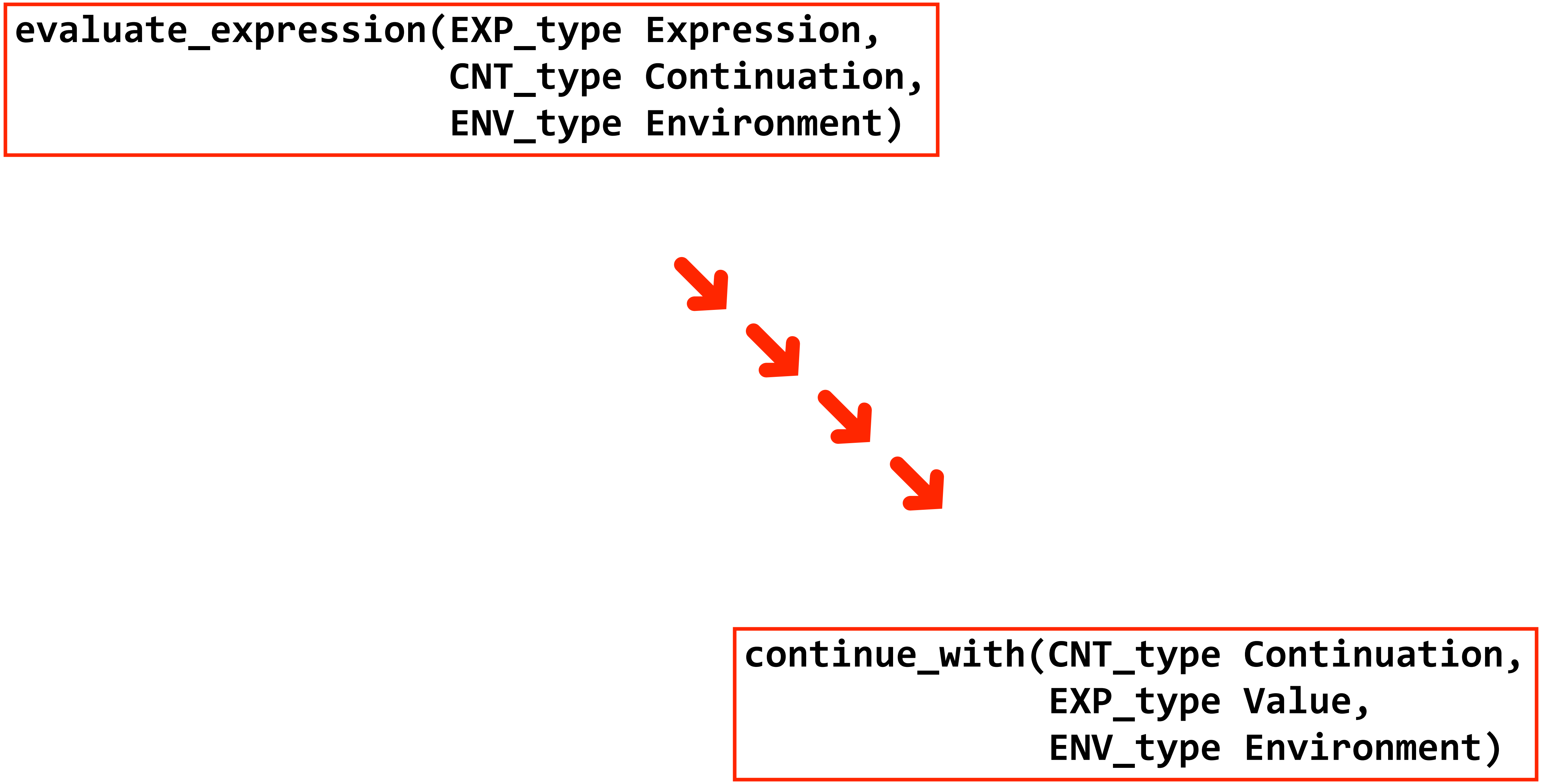
```
static EXP_type read_expression(NIL_type)
 { switch (Token)
   { case CHA_token:
     return read_character();
    case END_token:
     return read_end_program();
    case FLS_token:
     return read_false();
    case IDT_token:
     return read_symbol();
    case LBR_token:
     return read_vector();
    case LPR_token:
     return read_list();
    case NBR_token:
     return read_number();
    case PER_token:
     return read_period();
    case QUO_token:
     return read_quote();
    case RPR_token:
     return read_parenthesis();
    case REA_token:
     return read_real();
    case STR_token:
     return read_string();
    case TRU_token:
     return read_true(); }
  return Main_Null; }
```

```
EXP_type Read_Parse(TXT_type Input)
 { EXP_type expression;
  Scan_Preset(Input);
  next_token();
  expression = read_expression();
  if (Token != END_token)
   Main_Error(XCT_error_string);
  return expression; }
```

```
static PAI_type read_list(NIL_type)
 { EXP_type expression;
  PAI_type list;
  next_token();
  if (Token == RPR_token)
   list = Main_Empty_Pair;
  else
   { expression = read_expression();
    list = make_PAI(expression,
            Main_Null);
    build_list(list); }
  return next_token_and_return(list); }
```

198 loc

Evaluating Slip



```
evaluate_expression(EXP_type Expression,
                    CNT_type Continuation,
                    ENV_type Environment)
```

```
continue_with(CNT_type Continuation,
              EXP_type Value,
              ENV_type Environment)
```

Evaluating Slip

```
static NIL_type continue_with(CNT_type Continuation,
                              EXP_type Value,
                              ENV_type Environment)
  { CCC_type function;
    CFN_type c_function;
    c_function = Continuation->cfn;
    function = c_function->ccc;
    function(Value,
             Continuation,
             Environment); }
```

Evaluating Slip

```
typedef NIL_type (* CCC_type) (EXP_type,
                               CNT_type,
                               ENV_type);
```

```
static NIL_type continue_with(CNT_type Continuation,
                              EXP_type Value,
                              ENV_type Environment)
  { CCC_type function;
    CFN_type c_function;
    c_function = Continuation->cfn;
    function = c_function->ccc;
    function(Value,
             Continuation,
             Environment); }
```

Evaluating Slip

```
typedef NIL_type (* CCC_type) (EXP_type,
                               CNT_type,
                               ENV_type);
```

```
static NIL_type continue_with(CNT_type Continuation,
                              EXP_type Value,
                              ENV_type Environment)
  { CCC_type function;
    CFN_type c_function;
    c_function = Continuation->cfn;
    function = c_function->ccc;
    function(Value,
             Continuation,
             Environment); }
```

```
struct CFN { CEL_type hdr;
             CCC_type ccc; } CFN;
```

Evaluating Slip

```
typedef NIL_type (* CCC_type) (EXP_type,
                               CNT_type,
                               ENV_type);
```

```
static NIL_type continue_with(CNT_type Continuation,
                              EXP_type Value,
                              ENV_type Environment)
  { CCC_type function;
    CFN_type c_function;
    c_function = Continuation->cfn;
    function = c_function->ccc;
    function(Value,
             Continuation,
             Environment); }
```

```
struct CFN { CEL_type hdr;
             CCC_type ccc; } CFN;
```

```
struct CNT { CEL_type hdr;
             CFN_type cfn;
             CNT_type cnt;
             EXP_type exp[]; } CNT;
```

Evaluating Slip (cont’d)

```
static NIL_type evaluate_expression(EXP_type Expression,
                                    CNT_type Continuation,
                                    ENV_type Environment)
  { TAG_type tag;
    tag = Tag_of(Expression);
    switch (tag)
      { case PAI_tag:
            evaluate_form(Expression,
                          Continuation,
                          Environment);
        case SYM_tag:
          evaluate_symbol(Expression,
                          Continuation,
                          Environment);
        case CHA_tag:
        case FLS_tag:
        case NUL_tag:
        case NBR_tag:
        case REA_tag:
        case STR_tag:
        case TRU_tag:
        case VEC_tag:
           evaluate_value(Expression,
                          Continuation,
                          Environment); }
    Main_Error_Tag(IXT_error_string,
                   tag); }
```

Evaluating Slip (cont'd)

```
static NIL_type evaluate_expression(EXP_type Expression,
                                    CNT_type Continuation,
                                    ENV_type Environment)
  { TAG_type tag;
    tag = Tag_of(Expression);
    switch (tag)
      { case PAI_tag:
            evaluate_form(Expression,
                          Continuation,
                          Environment);
        case SYM_tag:
          evaluate_symbol(Expression,
                          Continuation,
                          Environment);
        case CHA_tag:
        case FLS_tag:
        case NUL_tag:
        case NBR_tag:
        case REA_tag:
        case STR_tag:
        case TRU_tag:
        case VEC_tag:
           evaluate_value(Expression,
                          Continuation,
                          Environment); }
    Main_Error_Tag(IXT_error_string,
                   tag); }
```

forms

Evaluating Slip (cont’d)

```
static NIL_type evaluate_form(PAI_type Form,
                              CNT_type Continuation,
                              ENV_type Environment)
  { EXP_type operands,
             operator;
    operator = Form->car;
    operands = Form->cdr;
    if (operator == Main_Begin)
          evaluate_begin(operands,
                         Continuation,
                         Environment);
    if (operator == Main_Define)
         evaluate_define(operands,
                         Continuation,
                         Environment);
    if (operator == Main_If)
             evaluate_if(operands,
                         Continuation,
                         Environment);
    if (operator == Main_Lambda)
         evaluate_lambda(operands,
                         Continuation,
                         Environment);
                             Environment);
```

```
    if (operator == Main_Quote)
          evaluate_quote(operands,
                         Continuation,
                         Environment);
    if (operator == Main_Set)
            evaluate_set(operands,
                         Continuation,
                         Environment);
    if (operator == Main_While)
          evaluate_while(operands,
                         Continuation,
    evaluate_application(operator,
                         operands,
                         Continuation,
                         Environment); }
```

Evaluating Slip (cont’d)

```
static NIL_type evaluate_expression(EXP_type Expression,
                                    CNT_type Continuation,
                                    ENV_type Environment)
  { TAG_type tag;
    tag = Tag_of(Expression);
    switch (tag)
      { case PAI_tag:
            evaluate_form(Expression,
                          Continuation,
                          Environment);
        case SYM_tag:
          evaluate_symbol(Expression,
                          Continuation,
                          Environment);
        case CHA_tag:
        case FLS_tag:
        case NUL_tag:
        case NBR_tag:
        case REA_tag:
        case STR_tag:
        case TRU_tag:
        case VEC_tag:
           evaluate_value(Expression,
                          Continuation,
                          Environment); }
    Main_Error_Tag(IXT_error_string,
                   tag); }
```

Evaluating Slip (cont'd)

```
static NIL_type evaluate_expression(EXP_type Expression,
                                    CNT_type Continuation,
                                    ENV_type Environment)
  { TAG_type tag;
    tag = Tag_of(Expression);
    switch (tag)
      { case PAI_tag:
            evaluate_form(Expression,
                          Continuation,
                          Environment);
        case SYM_tag:
          evaluate_symbol(Expression,
                          Continuation,
                          Environment);
        case CHA_tag:
        case FLS_tag:
        case NUL_tag:
        case NBR_tag:
        case REA_tag:
        case STR_tag:
        case TRU_tag:
        case VEC_tag:
           evaluate_value(Expression,
                          Continuation,
                          Environment); }
    Main_Error_Tag(IXT_error_string,
                   tag); }
```

symbols

Evaluating Slip (cont'd)

```
static NIL_type evaluate_expression(EXP_type Expression,
                                    CNT_type Continuation,
                                    ENV_type Environment)
  { TAG_type tag;
    tag = Tag_of(Expression);
    switch (tag)
      { case PAI_tag:
            evaluate_form(Expression,
                          Continuation,
                          Environment);
        case SYM_tag:
          evaluate_symbol(Expression,
                          Continuation,
                          Environment);
        case CHA_tag:
        case FLS_tag:
        case NUL_tag:
        case NBR_tag:
        case REA_tag:
        case STR_tag:
        case TRU_tag:
        case VEC_tag:
           evaluate_value(Expression,
                          Continuation,
                          Environment); }
    Main_Error_Tag(IXT_error_string,
                   tag); }
```

symbols

values

Evaluating Slip (cont’d)

```
static NIL_type evaluate_symbol(SYM_type Variable,
                                CNT_type Continuation,
                                ENV_type Environment)
  { EXP_type value;
    value = Dictionary_Lookup(Variable,
                              Environment);
    continue_with(Continuation,
                  value,
                  Environment); }
```

```
static NIL_type evaluate_value(EXP_type Value,
                               CNT_type Continuation,
                               ENV_type Environment)
  { continue_with(Continuation,
                  Value,
                  Environment); }
```

Evaluating Slip (cont’d)

```
static NIL_type evaluate_symbol(SYM_type Variable,
                                CNT_type Continuation,
                                ENV_type Environment)
  { EXP_type value;
    value = Dictionary_Lookup(Variable,
                              Environment);
    continue_with(Continuation,
                  value,
                  Environment); }
```

```
static NIL_type evaluate_value(EXP_type Value,
                               CNT_type Continuation,
                               ENV_type Environment)
  { continue_with(Continuation,
                  Value,
                  Environment); }
```

Simple Identity

Evaluating Slip (cont’d)

```
static NIL_type evaluate_symbol(SYM_type Variable,
                                CNT_type Continuation,
                                ENV_type Environment)
  { EXP_type value;
    value = Dictionary_Lookup(Variable,
                              Environment);
    continue_with(Continuation,
                  value,
                  Environment); }
```

Generating an immediate value

```
static NIL_type evaluate_value(EXP_type Value,
                               CNT_type Continuation,
                               ENV_type Environment)
  { continue_with(Continuation,
                  Value,
                  Environment); }
```

Evaluating a Slip set!

```
static NIL_type evaluate_set(PAI_type Operands,
                             CNT_type Continuation,
                             ENV_type Environment)
  { sET_type set_thread;
    CNT_type continuation;
    EXP_type expression;
    PAI_type expressions,
             residue;
    SYM_type variable;
    TAG_type tag;
    tag = Tag_of(Operands);
    switch (tag)
      { case NUL_tag:
          Main_Error_Text(MSV_error_string,
                          Main_Set_String);
        case PAI_tag:
          variable = Operands->car;
          if (is_SYM(variable))
            { expressions = Operands->cdr;
              tag = Tag_of(expressions);
              switch (tag)
                { case NUL_tag:
                    Main_Error_Text(E1X_error_string,
                                    Main_Set_String);
                  case PAI_tag:
                    expression = expressions->car;
                    residue    = expressions->cdr;
                    if (is_NUL(residue))
```

```
                                                    Main_Error_Text(TMX_error_string,
                                                                    Main_Set_String); }
                                              Main_Error_Text(ITF_error_string,
                                                              Main_Set_String); }
                                          Main_Error_Text(IVV_error_string,
                                                          Main_Set_String); }
                                    Main_Error_Text(ITF_error_string,
                                                    Main_Set_String); }
```

Evaluating a Slip set! (cont’d)

```
if (is_NUL(residue))
  { continuation = make_CNT(Continue_set,
                            Continuation,
                            sET_size);
    set_thread = (sET_type)continuation;
    set_thread->var = variable;
    evaluate_expression(expression,
                        continuation,
                        Environment); }
```

Evaluating a Slip set! (cont'd)

```
if (is_NUL(residue))
  { continuation = make_CNT(Continue_set,
                            Continuation,
                            sET_size);
    set_thread = (sET_type)continuation;
    set_thread->var = variable;
    evaluate_expression(expression,
                        continuation,
                        Environment); }
```

```
static CFN_type Continue_set;

typedef struct sET * sET_type;
typedef struct sET { CEL_type hdr;
                     CFN_type cfn;
                     CNT_type cnt;
                     SYM_type var; } sET;

static const UNS_type sET_size = chunk_size(sET);
```

Evaluating a Slip set! (cont’d)

```
if (is_NUL(residue))
  { continuation = make_CNT(Continue_set,
                            Continuation,
                            sET_size);
    set_thread = (sET_type)continuation;
    set_thread->var = variable;
    evaluate_expression(expression,
                        continuation,
                        Environment); }
```

```
static CFN_type Continue_set;

typedef struct sET * sET_type;
typedef struct sET { CEL_type hdr;
                     CFN_type cfn;
                     CNT_type cnt;
                     SYM_type var; } sET;

static const UNS_type sET_size = chunk_size(sET);
```

```
static NIL_type initialize_set(NIL_type)
  { Continue_set = make_CFN(continue_set); }
```

Evaluating a Slip set! (cont’d)

```
static NIL_type continue_set(EXP_type Value,
                             CNT_type Continuation,
                             ENV_type Environment)
  { sET_type set_thread;
    CNT_type continuation;
    SYM_type variable;
    set_thread = (sET_type)Continuation;
    continuation = set_thread->cnt;
    variable     = set_thread->var;
    Dictionary_Replace(variable,
                       Value,
                       Environment);
    continue_with(continuation,
                  Value,
                  Environment); }
```

Evaluating a Slip application

```
static NIL_type evaluate_application(EXP_type Operator,
                                     EXP_type Operands,
                                     CNT_type Continuation,
                                     ENV_type Environment)
  { aPL_type application_thread;
    CNT_type continuation;
    continuation = make_CNT(Continue_application,
                            Continuation,
                            aPL_size);
    application_thread = (aPL_type)continuation;
    application_thread->opd = Operands;
    evaluate_expression(Operator,
                        continuation,
                        Environment); }

static NIL_type initialize_application(NIL_type)
  { Continue_application = make_CFN(continue_application); }
```

Evaluating a Slip application

```
static NIL_type evaluate_application(EXP_type Operator,
                                     EXP_type Operands,
                                     CNT_type Continuation,
                                     ENV_type Environment)
  { aPL_type application_thread;
    CNT_type continuation;
    continuation = make_CNT(Continue_application,
                            Continuation,
                            aPL_size);
    application_thread = (aPL_type)continuation;
    application_thread->opd = Operands;
    evaluate_expression(Operator,
                        continuation,
                        Environment); }

static NIL_type initialize_applic
  { Continue_application = make_C
```

```
static CFN_type Continue_application;

typedef struct aPL * aPL_type;
typedef struct aPL { CEL_type hdr;
                     CFN_type cfn;
                     CNT_type cnt;
                     PAI_type opd; } aPL;

static const UNS_type aPL_size = chunk_size(aPL);
```

Evaluating a Slip application (cont'd)

```
static NIL_type continue_application(EXP_type Procedure,
                                     CNT_type Continuation,
                                     ENV_type Environment)
  { aPL_type application_thread;
    CNT_type continuation;
    PAI_type operands;
    TAG_type tag;
    application_thread = (aPL_type)Continuation;
    continuation = application_thread->cnt;
    operands     = application_thread->opd;
    tag = Tag_of(Procedure);
    switch (tag)
      { case NAT_tag:
          evaluate_native_call(Procedure,
                               operands,
                               continuation,
                               Environment);
        case PRC_tag:
             evaluate_bindings(Procedure,
                               operands,
                               continuation,
                               Environment); }
    Main_Error_Text(PNR_error_string,
                    Application_String); }
```

Evaluating a Slip application (cont'd)

```
static NIL_type evaluate_bindings(PRC_type Procedure,
                                  PAI_type Operands,
                                  CNT_type Continuation,
                                  ENV_type Environment)
  { ENV_type closure;
    EXP_type parameters;
    parameters = Procedure->par;
    closure    = Procedure->clo;
    binding(Procedure,
            parameters,
            Operands,
            closure,
            Continuation,
            Environment); }

static NIL_type initialize_bindings(NIL_type)
  { Continue_bindings = make_CFN(continue_bindings); }
```

Evaluating a Slip application (cont’d)

```
static NIL_type evaluate_bindings(PRC_type Procedure,
                                  PAI_type Operands,
                                  CNT_type Continuation,
                                  ENV_type Environment)
  { ENV_type closure;
    EXP_type parameters;
    parameters = Procedure->par;
    closure    = Procedure->clo;
    binding(Procedure,
            parameters,
            Operands,
            closure,
            Continuation,
            Environment); }

static NIL_type initialize_bindir
  { Continue_bindings = make_CFN(
```

```
static CFN_type Continue_bindings;

typedef struct bND * bND_type;
typedef struct bND { CEL_type hdr;
                     CFN_type cfn;
                     CNT_type cnt;
                     PRC_type prc;
                     PAI_type par;
                     PAI_type opd;
                     ENV_type clo; } bND;

static const UNS_type bND_size = chunk_size(bND);
```

Evaluating a Slip application (cont’d)

```
static NIL_type continue_bindings(EXP_type Value,
                                  CNT_type Continuation,
                                  ENV_type Environment)
  { bND_type binding_thread;
    CNT_type continuation;
    ENV_type closure;
    EXP_type parameter;
    PAI_type operands,
             parameters;
    PRC_type procedure;
    binding_thread = (bND_type)Continuation;
    continuation = binding_thread->cnt;
    procedure    = binding_thread->prc;
    parameters   = binding_thread->par;
    operands     = binding_thread->opd;
    closure      = binding_thread->clo;
    parameter = parameters->car;
    if (is_SYM(parameter))
      { parameters = parameters->cdr;
        closure = Dictionary_Define(parameter,
                                    Value,
                                    closure);
        binding(procedure,
                parameters,
                operands,
                closure,
                continuation,
                Environment); }
    Main_Error_Procedure(IPA_error_string,
                         procedure); }
```

Evaluating a Slip application (cont'd)

```
static NIL_type continue_bindings(EXP_type Value,
                                  CNT_type Continuation,
                                  ENV_type Environment)
  { bND_type binding_thread;
    CNT_type continuation;
    ENV_type closure;
    EXP_type parameter;
    PAI_type operands,
             parameters;
    PRC_type procedure;
    binding_thread = (bND_type)Continuation;
    continuation = binding_thread->cnt;
    procedure    = binding_thread->prc;
    parameters   = binding_thread->par;
    operands     = binding_thread->opd;
    closure      = binding_thread->clo;
    parameter = parameters->car;
    if (is_SYM(parameter))
      { parameters = parameters->cdr;
        closure = Dictionary_Define(parameter,
                                    Value,
                                    closure);
        binding(procedure,
                parameters,
                operands,
                closure,
                continuation,
                Environment); }
    Main_Error_Procedure(IPA_error_string,
                         procedure); }
```

auxiliary

Evaluating a Slip application (cont’d)

```
static NIL_type evaluate_body(PRC_type Procedure,
                              CNT_type Continuation,
                              ENV_type Environment,
                              ENV_type Closure)
  { bOD_type body_thread;
    CNT_type continuation;
    PAI_type body;
    continuation = make_CNT(Continue_body,
                            Continuation,
                            bOD_size);
    body_thread = (bOD_type)continuation;
    body_thread->env = Environment;
    body = Procedure->bod;
    evaluate_sequence(body,
                      continuation,
                      Closure); }

static NIL_type initialize_body(NIL_type)
  { Continue_body = make_CFN(continue_body); }
```

Evaluating a Slip application (cont’d)

```
static NIL_type evaluate_body(PRC_type Procedure,
                              CNT_type Continuation,
                              ENV_type Environment,
                              ENV_type Closure)
  { bOD_type body_thread;
    CNT_type continuation;
    PAI_type body;
    continuation = make_CNT(Continue_body,
                            Continuation,
                            bOD_size);
    body_thread = (bOD_type)continuation;
    body_thread->env = Environment;
    body = Procedure->bod;
    evaluate_sequence(body,
                      continuatio
                      Closure); }

static NIL_type initialize_body(N
  { Continue_body = make_CFN(cont
```

```
static CFN_type Continue_body;

typedef struct bOD * bOD_type;
typedef struct bOD { CEL_type hdr;
                     CFN_type cfn;
                     CNT_type cnt;
                     ENV_type env; } bOD;

static const UNS_type bOD_size = chunk_size(bOD);
```

Evaluating a Slip application (cont’d)

```
static NIL_type continue_body(EXP_type Value,
                              CNT_type Continuation,
                              ENV_type Ignore)
  { bOD_type body_thread;
    CNT_type continuation;
    ENV_type environment;
    body_thread = (bOD_type)Continuation;
    continuation = body_thread->cnt;
    environment  = body_thread->env;
    continue_with(continuation,
                  Value,
                  environment); }
```

Evaluating a Slip application (cont’d)

```
static NIL_type continue_body(EXP_type Value,
                              CNT_type Continuation,
                              ENV_type Ignore)
  { bOD_type body_thread;
    CNT_type continuation;
    ENV_type environment;
    body_thread = (bOD_type)Continuation;
    continuation = body_thread->cnt;
    environment  = body_thread->env;
    continue_with(continuation,
                  Value,
                  environment); }
```

local environment

Evaluating a Slip application (cont’d)

```
static NIL_type continue_body(EXP_type Value,
                              CNT_type Continuation,
                              ENV_type Ignore)
  { bOD_type body_thread;
    CNT_type continuation;
    ENV_type environment;
    body_thread = (bOD_type)Continuation;
    continuation = body_thread->cnt;
    environment  = body_thread->env;
    continue_with(continuation,
                  Value,
                  environment); }
```

local environment

restore environment

A Slip REP-loop

```
static NIL_type read_eval_print(


  { EXP_type expression;
    TXT_type input;
    Print_Print(Value);
    Slip_Print("\n>>>");
    Slip_Read(&input);
    Rollback = Environment;
    expression = Read_Parse(input);
    Evaluate_Evaluate(expression,
                      Continuation,
                      Environment); }
```

```
EXP_type Main_Error(TXT_type Error)
  { STR_type string;
    string = make_STR(Error);
    read_eval_print(string,
                    Read_eval_print_continuation,
                    Rollback);
    return Main_Unspecified; }
```

```
Read_eval_print_function = make_CFN(read_eval_print);
Read_eval_print_continuation = make_CNT(Read_eval_print_function,
                                        Main_Empty_Continuation,
                                        0);
Read_eval_print_continuation->cnt = Read_eval_print_continuation;

Slip_string = make_STR("Slip version 1");
read_eval_print(Slip_string,
                Read_eval_print_continuation,
                Dictionary_Environment); }
```

A Slip REP-loop

```
EXP_type Main_Error(TXT_type Error)
  { STR_type string;
```

```
t_continuation,
```

```
static NIL_type read_eval_print(EXP_type Value,
                                CNT_type Continuation,
                                ENV_type Environment)

  { EXP_type expression;
    TXT_type input;
    Print_Print(Value);
    Slip_Print("\n>>>");
    Slip_Read(&input);
    Rollback = Environment;
    expression = Read_Parse(input);
    Evaluate_Evaluate(expression,
                      Continuation,
                      Environment); }
```

```
    Read_eval_print_function = make_CFN(read_eval_print);
    Read_eval_print_continuation = make_CNT(Read_eval_print_function,
                                            Main_Empty_Continuation,
                                            0);
    Read_eval_print_continuation->cnt = Read_eval_print_continuation;

    Slip_string = make_STR("Slip version 1");
    read_eval_print(Slip_string,
                    Read_eval_print_continuation,
                    Dictionary_Environment); }
```

A Slip REP-loop

```
static NIL_type read_eval_print(


  { EXP_type expression;
    TXT_type input;
    Print_Print(Value);
    Slip_Print("\n>>>");
    Slip_Read(&input);
    Rollback = Environment;
    expression = Read_Parse(input);
    Evaluate_Evaluate(expression,
                      Continuation,
                      Environment); }
```

```
EXP_type Main_Error(TXT_type Error)
  { STR_type string;
    string = make_STR(Error);
    read_eval_print(string,
                    Read_eval_print_continuation,
                    Rollback);
    return Main_Unspecified; }
```

```
    Read_eval_print_function = make_CFN(read_eval_print);
    Read_eval_print_continuation = make_CNT(Read_eval_print_function,
                                            Main_Empty_Continuation,
                                            0);
    Read_eval_print_continuation->cnt = Read_eval_print_continuation;

    Slip_string = make_STR("Slip version 1");
    read_eval_print(Slip_string,
                    Read_eval_print_continuation,
                    Dictionary_Environment); }
```

A Slip REP-loop

```
static NIL_type read_eval_print(


  { EXP_type expression;
    TXT_type input;
    Print_Print(Value);
    Slip_Print("\n>>>");
    Slip_Read(&input);
    Rollback = Environment;
    expression = Read_Parse(input);
    Evaluate_Evaluate(expression,
                      Continuation,
                      Environment); }
```

```
EXP_type Main_Error(TXT_type Error)
  { STR_type string;
    string = make_STR(Error);
    read_eval_print(string,
                    Read_eval_print_continuation,
                    Rollback);
    return Main_Unspecified; }
```

```
Read_eval_print_function = make_CFN(read_eval_print);
Read_eval_print_continuation = make_CNT(Read_eval_print_function,
                                        Main_Empty_Continuation,
                                        0);
Read_eval_print_continuation->cnt = Read_eval_print_continuation;

Slip_string = make_STR("Slip version 1");
read_eval_print(Slip_string,
                Read_eval_print_continuation,
                Dictionary_Environment); }
```