SLIP: a simple language implementation platform

Deep into Smalltalk
INRIA Lille Nord Europe
March 7th - 11th, 2011

Theo D’Hondt
Software Languages Lab
Faculty of Sciences - Vrije Universiteit Brussel

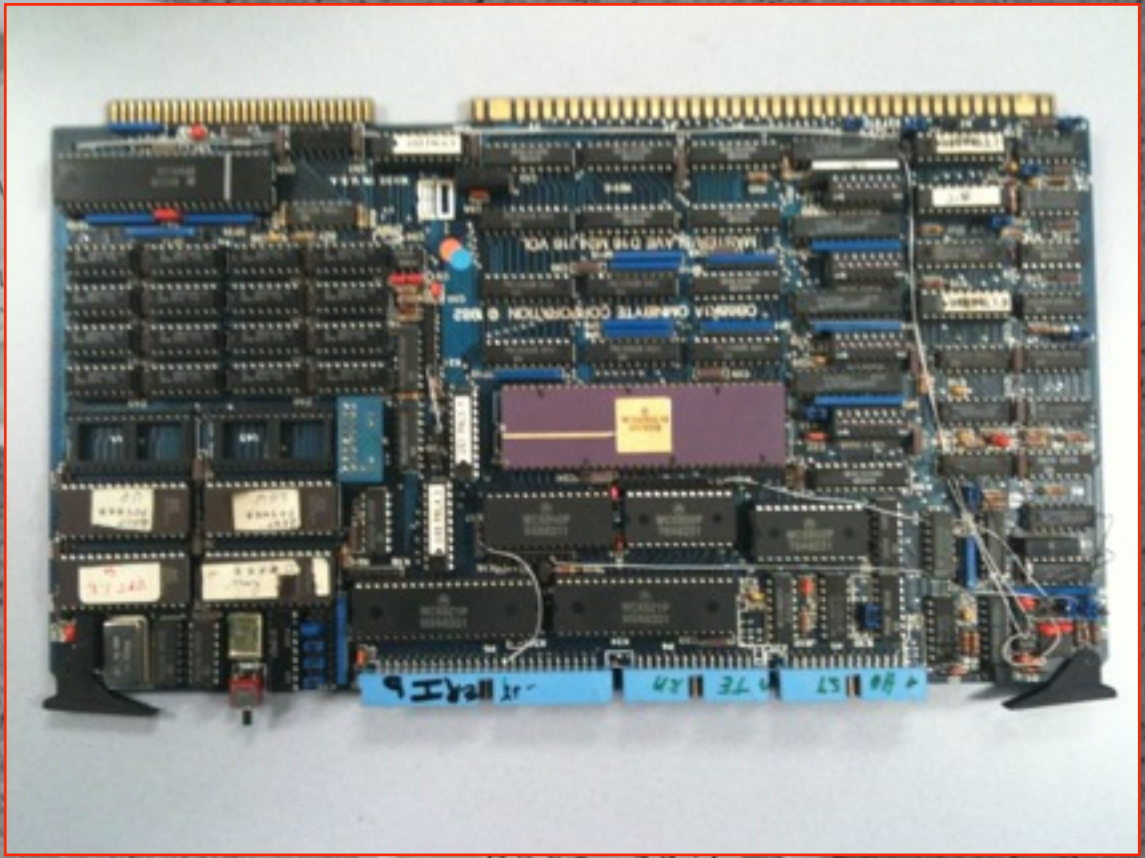
http://soft.vub.ac.be

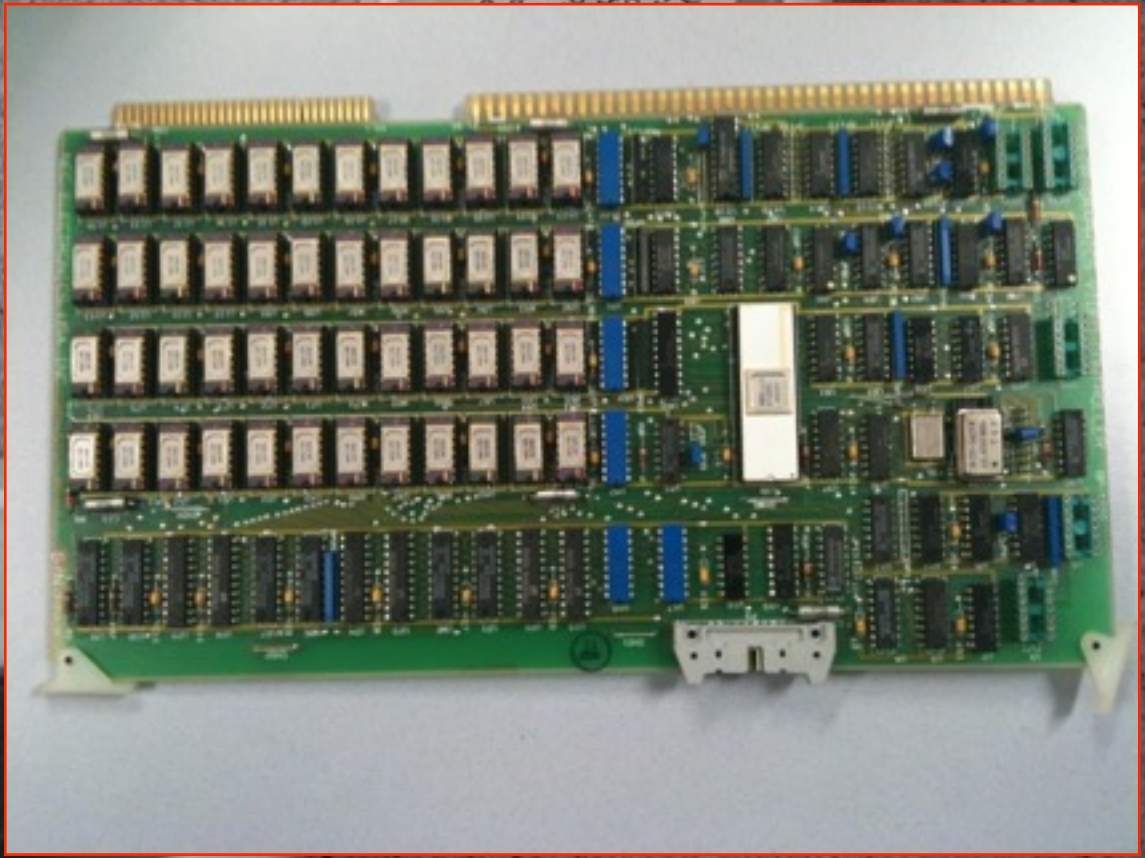
SMALLTALK-80™ Version 2
Copyright © Xerox Corp. 1983. All Rights Reserved.
Under License from Xerox Corporation
3333 Coyote Hill Road, Palo Alto, CA 94304

9-track,1600 bpi
ST80418

MEMOREX
MRX V
100% Tested
Compatible thru 6250 BPI

Copyright © 1983 by Xerox Corporation. All rights
reserved. The Virtual Image description.
reproduced The Virtual Image was described and/or
but has not in this document within the created in 1981
the copyright been published the meaning of
under a license law. The Virtual Image is furnished of
disclosed license and may not be used, copied,
and/or distributed except in accordance
with the terms of said license





Abstract

SLIP is a minimalist platform for the instruction and exploration of language implementations. It is a distillation of two previous platforms - Pico and 'skēm - but is not intended as a language in its own right. SLIP is of course a language: a very minimal but complete version of LISP - with a distinct Scheme flavour. But SLIP is also a sequence of interpreters, starting with a 100 line fully metacircular version. In the spirit of Friedman's EOPL, this version is rewritten in continuation passing style and translated into C in a straightforward way. Twelve successive versions introduce features such as trampolines, lexical addressing and garbage collection, to end with a fully optimized version that executes a simple benchmark on par with the PLT Scheme interpreter.

SLIP is a chain of implementations presented as an instruction tool. It is also an experimentation tool, and this presentation will present a 13th version that introduces simple futures into SLIP in view of experimenting with multi-core systems.

SLIP has been used in the Francqui chair (soft.vub.ac.be/francqui) in 2010, and already twice in the Programming Language Engineering (soft.vub.ac.be/PLE) course. It will again be used - including version 13 - in this year's issue of PLE.

Abstract

SLIP is a minimalist platform for the instruction and exploration of language implementations. It is a distillation of two previous platforms - Pico and 'skēm - but is not intended as a language in its own right. SLIP is of course a language: a very minimal but complete version of LISP - with a distinct Scheme flavour. But SLIP is also a sequence of interpreters, starting with a 100 line fully metacircular version. In the spirit of Friedman's EOPL, this version is rewritten in continuation passing style and translated into C in a straightforward way. Twelve successive versions introduce features such as trampolines, lexical addressing and garbage collection, to end with a fully optimized version that executes a simple benchmark on par with the PLT Scheme interpreter.

SLIP is a chain of implementations presented as an instruction tool. It is also an experimentation tool, and this presentation will present a 13th version that introduces simple futures into SLIP in view of experimenting with multi-core systems.

SLIP has been used in the Francqui chair (soft.vub.ac.be/francqui) earlier this year, and already twice in the Programming Language Engineering (PLE) course. It will again be used - including version 13 - in this year's issue of PLE.

Abstract

SLIP is a minimalist platform for the instruction and exploration of language implementations. It is a distillation of two previous platforms - Pico and 'skēm - but is not intended as a language in its own right. SLIP is of course a language: a very minimal but complete version of LISP - with a distinct Scheme flavour. But SLIP is also a sequence of interpreters, starting with a 100 line fully metacircular version. In the spirit of Friedman's EOPL, this version is rewritten in continuation passing style and translated into C in a straightforward way. Twelve successive versions introduce features such as trampolines, lexical addressing and garbage collection, to end with a fully optimized version that executes a simple benchmark on par with the PLT Scheme interpreter.

SLIP is a chain of implementations presented as an instruction tool. It is also an experimentation tool, and this presentation will present a 13th version that introduces simple futures into SLIP in view of experimenting with multi-core systems.

SLIP has been used in the Francqui chair (soft.vub.ac.be/francqui) earlier this year, and already twice in the Programming Language Engineering (PLE) course. It will again be used - including version 13 - in this year's issue of PLE.

Abstract

SLIP is a minimalist platform for the instruction and exploration of language implementations. It is a distillation of two previous platforms - Pico and 'skēm - but is not intended as a language in its own right. SLIP is of course a language: a very minimal but complete version of LISP - with a distinct Scheme flavour. But SLIP is also a sequence of interpreters, starting with a 100 line fully metacircular version. In the spirit of Friedman's EOPL, this version is rewritten in continuation passing style and translated into C in a straightforward way. Twelve successive versions introduce features such as trampolines, lexical addressing and garbage collection, to end with a fully optimized version that executes a simple benchmark on par with the PLT Scheme interpreter.

SLIP is a chain of implementations presented as an instruction tool. It is also an experimentation tool, and this presentation will present a 13th version that introduces simple futures into SLIP in view of experimenting with multi-core systems.

SLIP has been used in the Francqui chair (soft.vub.ac.be/francqui) earlier this year, and already twice in the Programming Language Engineering (PLE) course. It will again be used - including version 13 - in this year's issue of PLE.

Abstract

SLIP is a minimalist platform for the instruction and exploration of language implementations. It is a distillation of two previous platforms - Pico and 'skēm - but is not intended as a language in its own right. SLIP is of course a language: a very minimal but complete version of LISP - with a distinct Scheme flavour. But SLIP is also a sequence of interpreters, starting with a 100 line fully metacircular version. In the spirit of Friedman's EOPL, this version is rewritten in continuation passing style and translated into C in a straightforward way. Twelve successive versions introduce features such as trampolines, lexical addressing and garbage collection, to end with a fully optimized version that executes a simple benchmark on par with the PLT Scheme interpreter.

SLIP is a chain of implementations presented as an instruction tool. It is also an experimentation tool, and this presentation will present a 13th version that introduces simple futures into SLIP in view of experimenting with multi-core systems.

SLIP has been used in the Francqui chair (soft.vub.ac.be/francqui) earlier this year, and already twice in the Programming Language Engineering (PLE) course. It will again be used - including version 13 - in this year's issue of PLE.

Abstract

SLIP is a minimalist platform for the instruction and exploration of language implementations. It is a distillation of two previous platforms - Pico and 'skēm - but is not intended as a language in its own right. SLIP is of course a language: a very minimal but complete version of LISP - with a distinct Scheme flavour. But SLIP is also a sequence of interpreters, starting with a 100 line fully metacircular version. In the spirit of Friedman's EOPL, this version is rewritten in continuation passing style and translated into C in a straightforward way. Twelve successive versions introduce features such as trampolines, lexical addressing and garbage collection, to end with a fully optimized version that executes a simple benchmark on par with the PLT Scheme interpreter.

SLIP is a chain of implementations presented as an instruction tool. It is also an experimentation tool, and this presentation will present a 13th version that introduces simple futures into SLIP in view of experimenting with multi-core systems.

SLIP has been used in the Francqui chair (soft.vub.ac.be/francqui) earlier this year, and already twice in the Programming Language Engineering (PLE) course. It will again be used - including version 13 - in this year's issue of PLE.

Abstract

SLIP is a minimalist platform for the instruction and exploration of language implementations. It is a distillation of two previous platforms - Pico and 'skēm - but is not intended as a language in its own right. SLIP is of course a language: a very minimal but complete version of LISP - with a distinct Scheme flavour. But SLIP is also a sequence of interpreters, starting with a 100 line fully metacircular version. In the spirit of Friedman's EOPL, this version is rewritten in continuation passing style and translated into C in a straightforward way. Twelve successive versions introduce features such as trampolines, lexical addressing and garbage collection, to end with a fully optimized version that executes a simple benchmark on par with the PLT Scheme interpreter.

SLIP is a chain of implementations presented as an instruction tool. It is also an experimentation tool, and this presentation will present a 13th version that introduces simple futures into SLIP in view of experimenting with multi-core systems.

SLIP has been used in the Francqui chair (soft.vub.ac.be/francqui) earlier this year, and already twice in the Programming Language Engineering (PLE) course. It will again be used - including version 13 - in this year's issue of PLE.

Abstract

SLIP is a minimalist platform for the instruction and exploration of language implementations. It is a distillation of two previous platforms - Pico and 'skēm - but is not intended as a language in its own right. SLIP is of course a language: a very minimal but complete version of LISP - with a distinct Scheme flavour. But SLIP is also a sequence of interpreters, starting with a 100 line fully metacircular version. In the spirit of Friedman's EOPL, this version is rewritten in continuation passing style and translated into C in a straightforward way. Twelve successive versions introduce features such as trampolines, lexical addressing and garbage collection, to end with a fully optimized version that executes a simple benchmark on par with the PLT Scheme interpreter.

SLIP is a chain of implementations presented as an instruction tool. It is also an experimentation tool, and this presentation will present a 13th version that introduces simple futures into SLIP in view of experimenting with multi-core systems.

SLIP has been used in the Francqui chair (soft.vub.ac.be/francqui) earlier this year, and already twice in the Programming Language Engineering (PLE) course. It will again be used - including version 13 - in this year's issue of PLE.

Abstract

SLIP is a minimalist platform for the instruction and exploration of language implementations. It is a distillation of two previous platforms - Pico and 'skēm - but is not intended as a language in its own right. SLIP is of course a language: a very minimal but complete version of LISP - with a distinct Scheme flavour. But SLIP is also a sequence of interpreters, starting with a 100 line fully metacircular version. In the spirit of Friedman's EOPL, this version is rewritten in continuation passing style and translated into C in a straightforward way. Twelve successive versions introduce features such as trampolines, lexical addressing and garbage collection, to end with a fully optimized version that executes a simple benchmark on par with the PLT Scheme interpreter.

SLIP is a chain of implementations presented as an instruction tool. It is also an experimentation tool, and this presentation will present a 13th version that introduces simple futures into SLIP in view of experimenting with multi-core systems.

SLIP has been used in the Francqui chair (soft.vub.ac.be/francqui) earlier this year, and already twice in the Programming Language Engineering (PLE) course. It will again be used - including version 13 - in this year's issue of PLE.

•••update•••

```
(begin
  (define (Sort V Low High Recurse)

           ...

    (Recurse Left Right))

  (define (SingleCore-QuickSort V Low High)
    (define (SingleCore-Recurse Left Right)
      (if (< Low Right)
        (SingleCore-QuickSort V Low Right))
      (if (> High Left)
        (SingleCore-QuickSort V Left High)))
    (Sort V Low High SingleCore-Recurse))

  (define (MultiCore-QuickSort Depth V Low High)
    (define (MultiCore-Recurse Left Right)
      (if (> Depth 0)
        (begin
          (define promise
            (if (< Low Right)
              (spawn (MultiCore-QuickSort (- Depth 1) V Low Right))))
          (if (> High Left)
            (MultiCore-QuickSort (- Depth 1) V Left High))
          (sync promise))
        (begin
          (SingleCore-QuickSort V Low High)
          (collect))))
    (Sort V Low High MultiCore-Recurse))
```

•••update•••

```
(begin
  (define (Sort V Low High Recurse)

           ...

    (Recurse Left Right))

  (define (SingleCore-QuickSort V Low High)
    (define (SingleCore-Recurse Left Right)
      (if (< Low Right)
        (SingleCore-QuickSort V Low Right))
      (if (> High Left)
        (SingleCore-QuickSort V Left High)))
    (Sort V Low High SingleCore-Recurse))

  (define (MultiCore-QuickSort Depth V Low High)
    (define (MultiCore-Recurse Left Right)
      (if (> Depth 0)
        (begin
          (define promise
            (if (< Low Right)
              (spawn (MultiCore-QuickSort (- Depth 1) V Low Right))))
          (if (> High Left)
            (MultiCore-QuickSort (- Depth 1) V Left High))
          (sync promise))
        (begin
          (SingleCore-QuickSort V Low High)
          (collect))))
    (Sort V Low High MultiCore-Recurse))
```

•••update•••

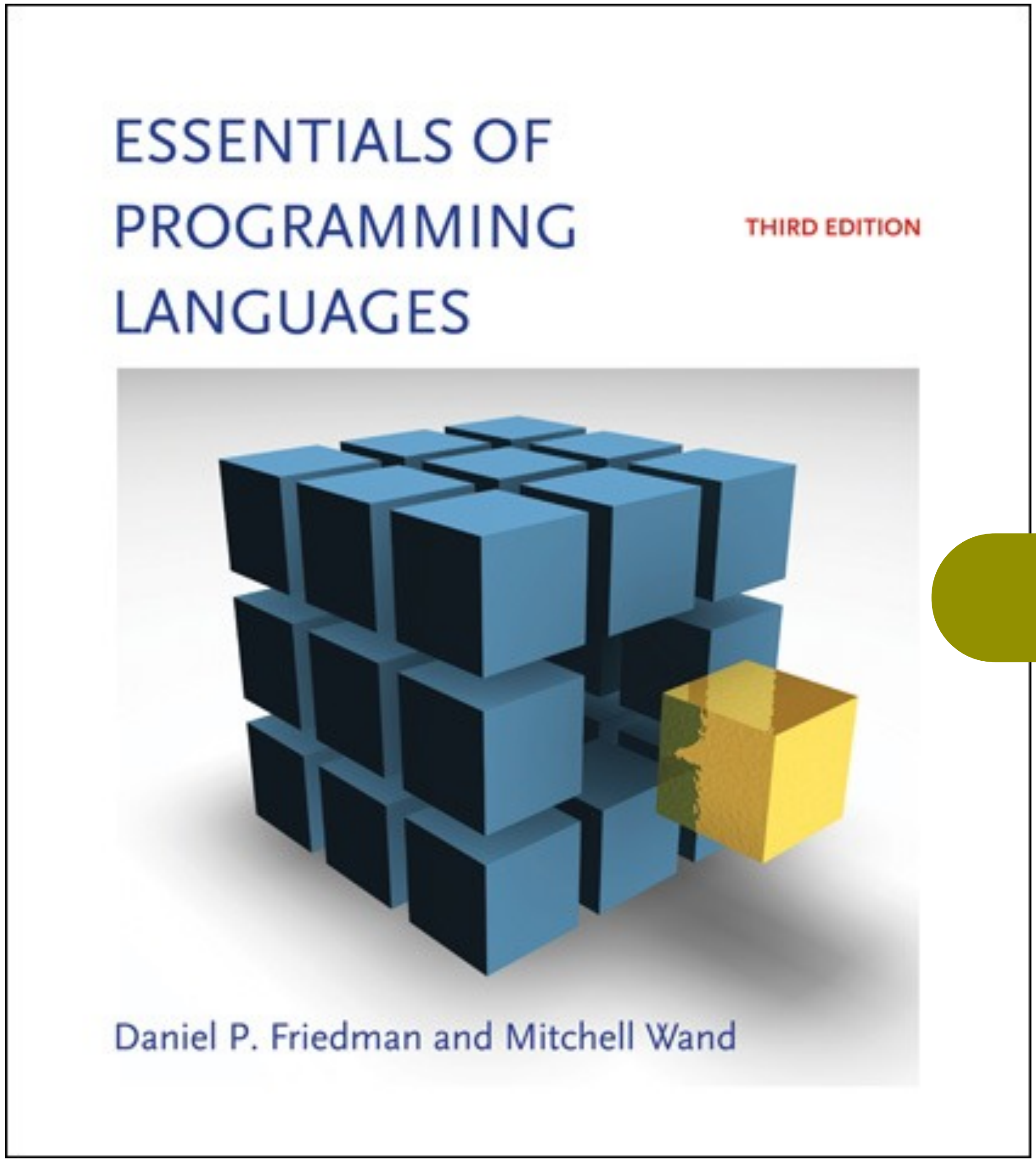
```
(begin
 (define (Sort V Low High Recurse)

      ...

  (Recurse Left Right))

 (define (SingleCore-QuickSort
  (define (SingleCore-Recurse
   (if (< Low Right)
    (SingleCore-QuickSort V
   (if (> High Left)
    (SingleCore-QuickSort V
  (Sort V Low High SingleCore

 (define (MultiCore-QuickSort
  (define (MultiCore-Recurse Left Right)
   (if (> Depth 0)
    (begin
     (define promise
      (if (< Low Right)
       (spawn (MultiCore-QuickSort (- Depth 1) V Low Right))))
     (if (> High Left)
      (MultiCore-QuickSort (- Depth 1) V Left High))
     (sync promise))
    (begin
     (SingleCore-QuickSort V Low High)
     (collect))))
  (Sort V Low High MultiCore-Recurse))
```

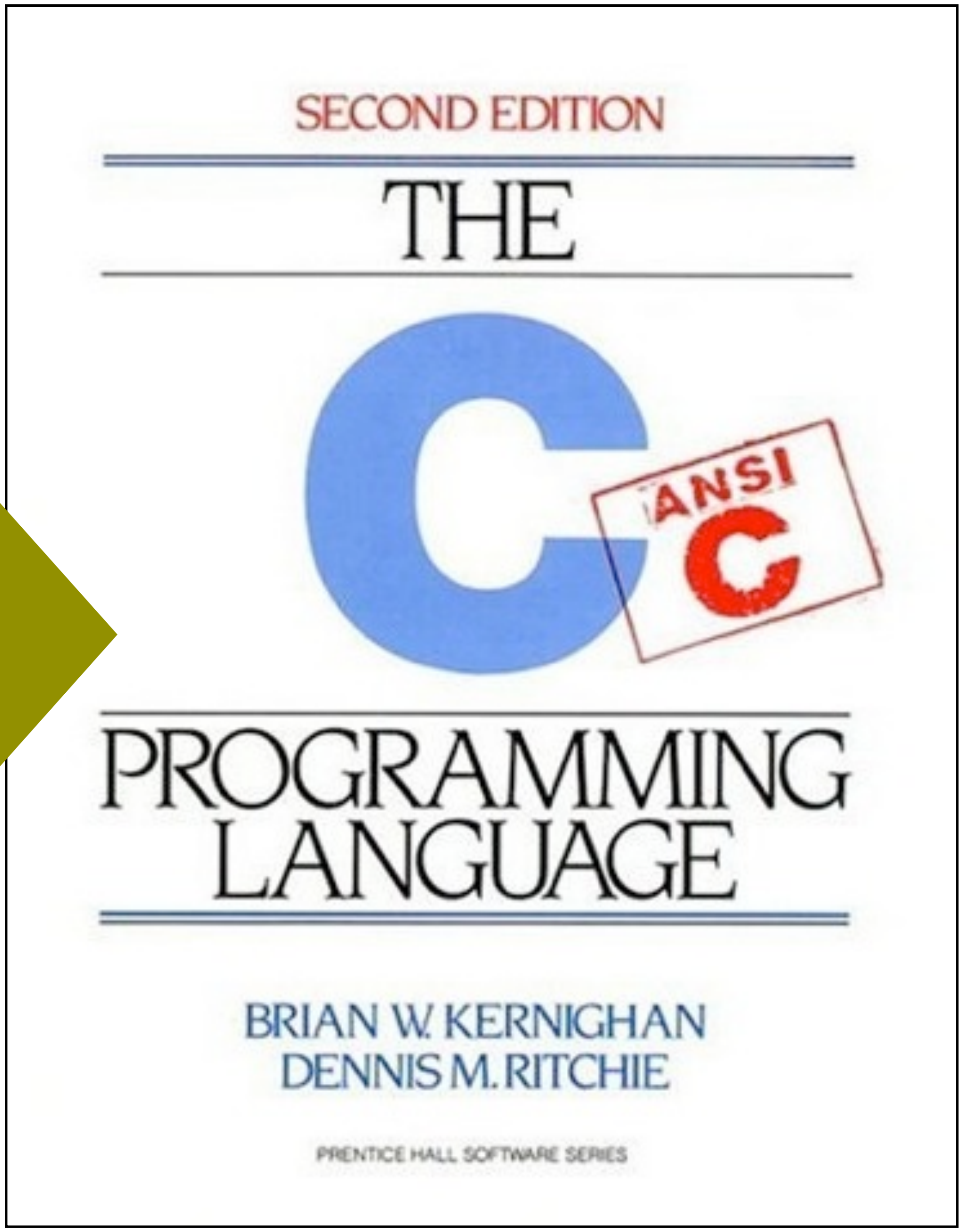
```
cpSlip/c version 13: multithreading
>>>(eval (read "quadcoreQuickSort.scm"))
quadcore quicksort of 100000 integers

... Collecting 109145 cells into 103092 cells in 0.003919 seconds

... Collecting 339952 cells into 106248 cells in 0.006443 seconds

... Collecting 940109 cells into 105837 cells in 0.01088 seconds

... Collecting 501743 cells into 105810 cells in 0.003370 seconds
  elapsed time = 12 secs
```

Agenda

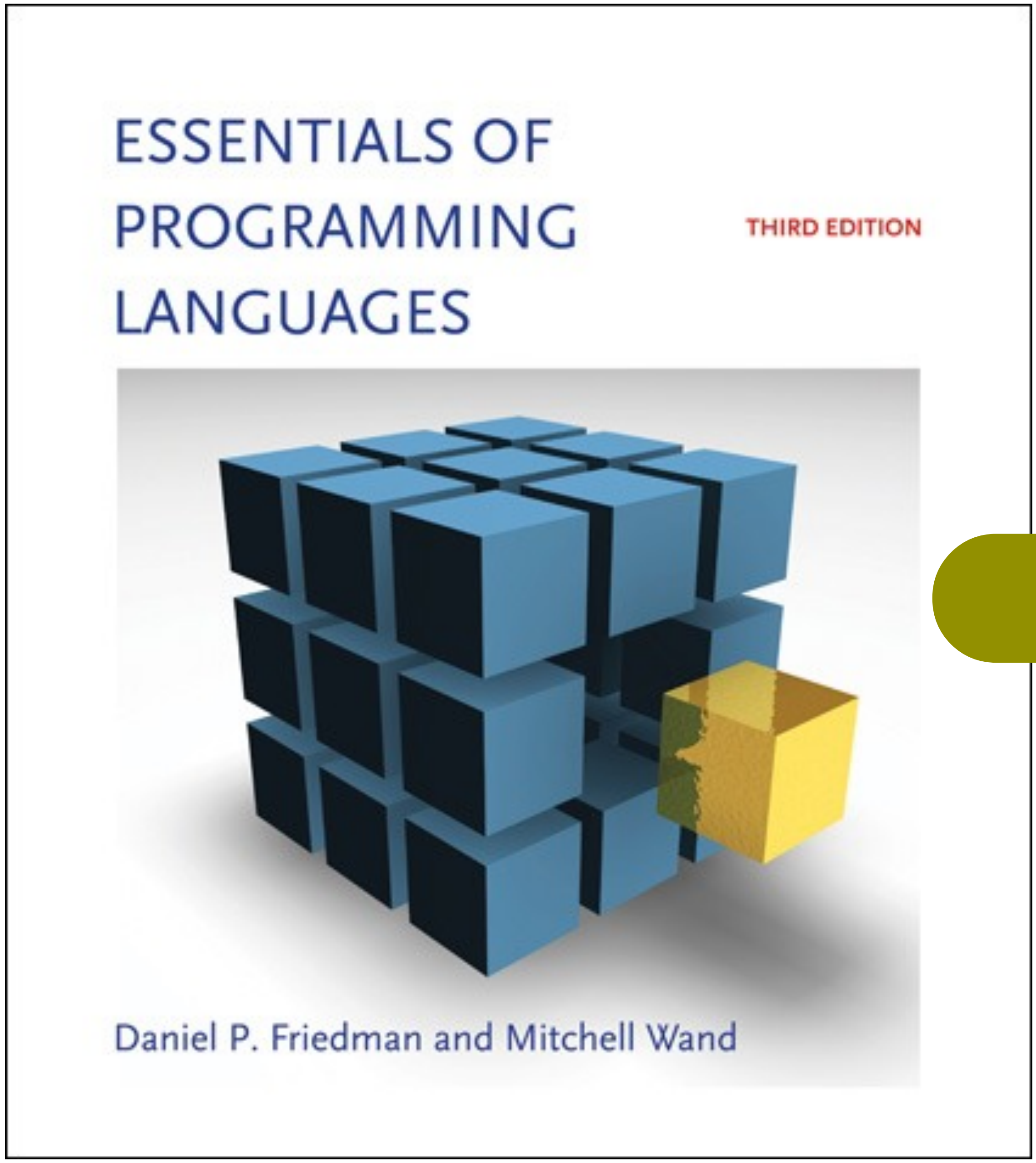
- ☑ motivation
- ☑ history: Pico (1&2), Pic%, 'skēm
- ☑ SLIP
- ☑ SLIP in cps
- ☑ SLIP in C
- ☑ multicore SLIP

Motivation


ESSENTIALS OF
PROGRAMMING
LANGUAGES
THIRD EDITION
Daniel P. Friedman and Mitchell Wand


SECOND EDITION
THE
C
ANSI
C
PROGRAMMING
LANGUAGE
BRIAN W. KERNIGHAN
DENNIS M. RITCHIE
PRENTICE HALL SOFTWARE SERIES

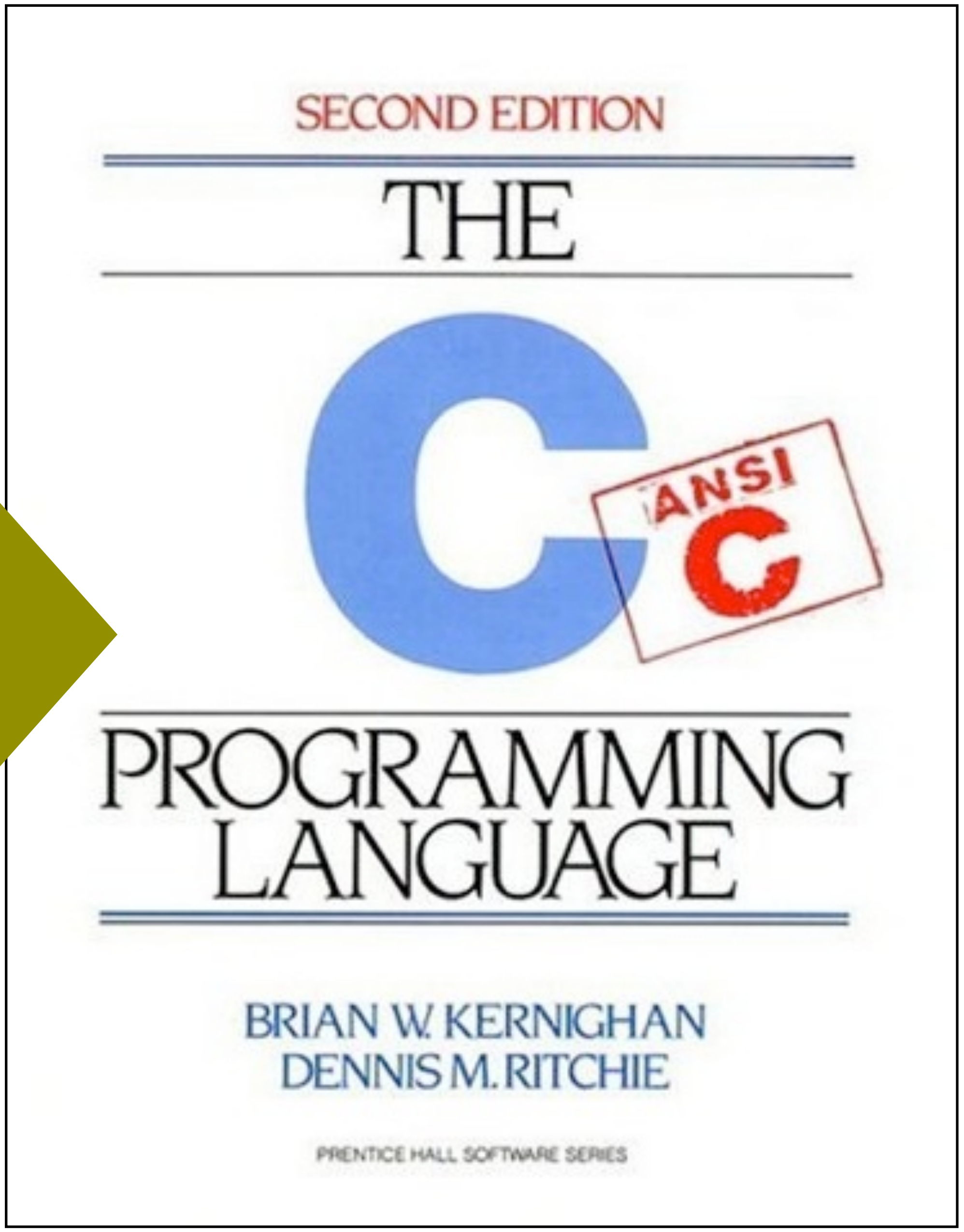
Motivation



ESSENTIALS OF PROGRAMMING LANGUAGES

THIRD EDITION

Daniel P. Friedman and Mitchell Wand



SECOND EDITION

THE C PROGRAMMING LANGUAGE

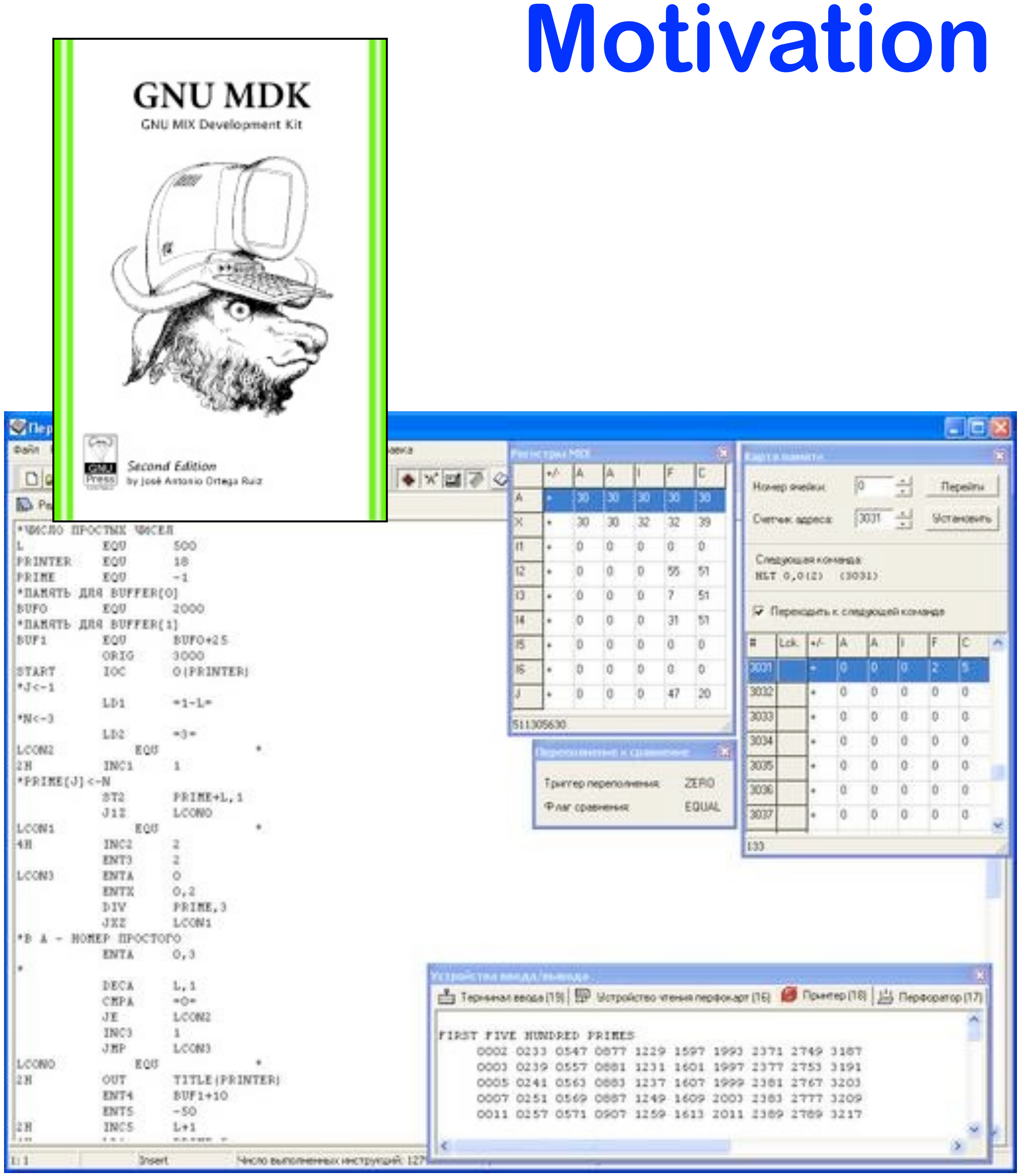
BRIAN W. KERNIGHAN
DENNIS M. RITCHIE

PRENTICE HALL SOFTWARE SERIES

First principles

Bare metal

Motivation (cont'd)



```
*ЧИСЛО ПРОСТЫХ ЧИСЕЛ
L            EQU       500
PRINTER      EQU       18
PRIME        EQU       -1
*ПАМЯТЬ ДЛЯ BUFFER[0]
BUF0         EQU       2000
*ПАМЯТЬ ДЛЯ BUFFER[1]
BUF1         EQU       BUF0+25
             ORIG      3000
START        IOC       0(PRINTER)
*J<-1
             LD1       =1-L=
*N<-3
             LD2       =3=
LCON2            EQU              *
2H           INC1      1
*PRIME[J]<-N
             ST2       PRIME+L,1
             J1Z       LCON0
LCON1            EQU              *
4H           INC2      2
             ENT3      2
LCON3        ENTA      0
             ENTX      0,2
             DIV       PRIME,3
             JXZ       LCON1
*B A - НОМЕР ПРОСТОГО
             ENTA      0,3
*
             DECA      L,1
             CMPA      =0=
             JE        LCON2
             INC3      1
             JMP       LCON3
LCON0            EQU              *
2H           OUT       TITLE(PRINTER)
             ENT4      BUF1+10
             ENT5      -50
2H           INC5      L+1
```

1:1 Insert Число выполненных инструкций: 127

	+/-	A	A	I	F	C
A	+	30	30	30	30	30
X	+	30	30	32	32	39
I1	+	0	0	0	0	0
I2	+	0	0	0	55	51
I3	+	0	0	0	7	51
I4	+	0	0	0	31	51
I5	+	0	0	0	0	0
I6	+	0	0	0	0	0
J	+	0	0	0	47	20

511305630

Триггер переполнения: ZERO
Флаг сравнения: EQUAL

Номер ячейки: 0
Счетчик адреса: 3031

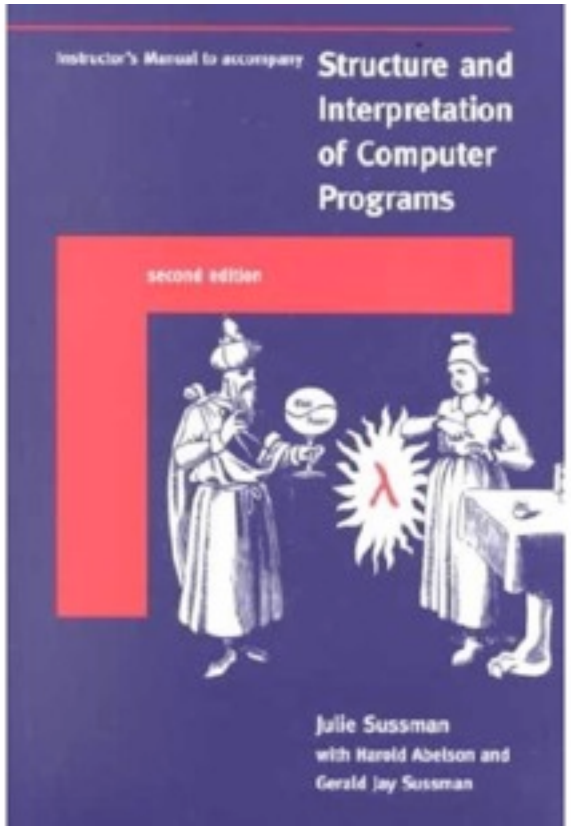
Следующая команда:
HLT 0,0(2) (3031)

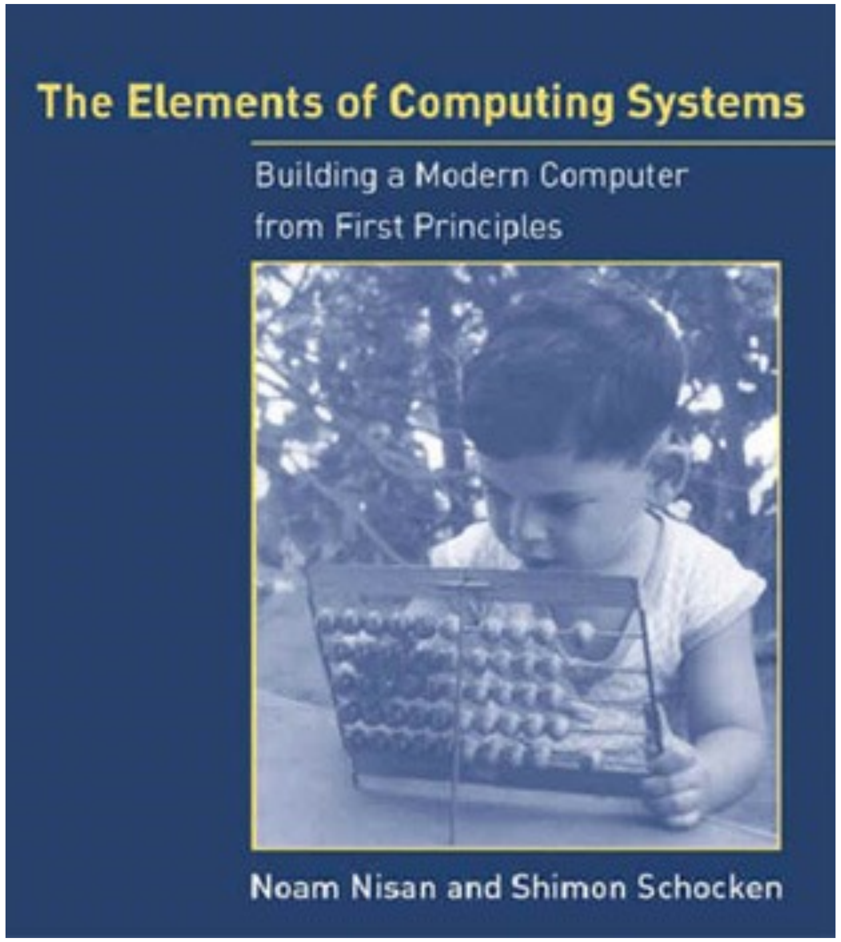
☑ Переходить к следующей команде

#	Lck.	+/-	A	A	I	F	C
3031		+	0	0	0	2	5
3032		+	0	0	0	0	0
3033		+	0	0	0	0	0
3034		+	0	0	0	0	0
3035		+	0	0	0	0	0
3036		+	0	0	0	0	0
3037		+	0	0	0	0	0

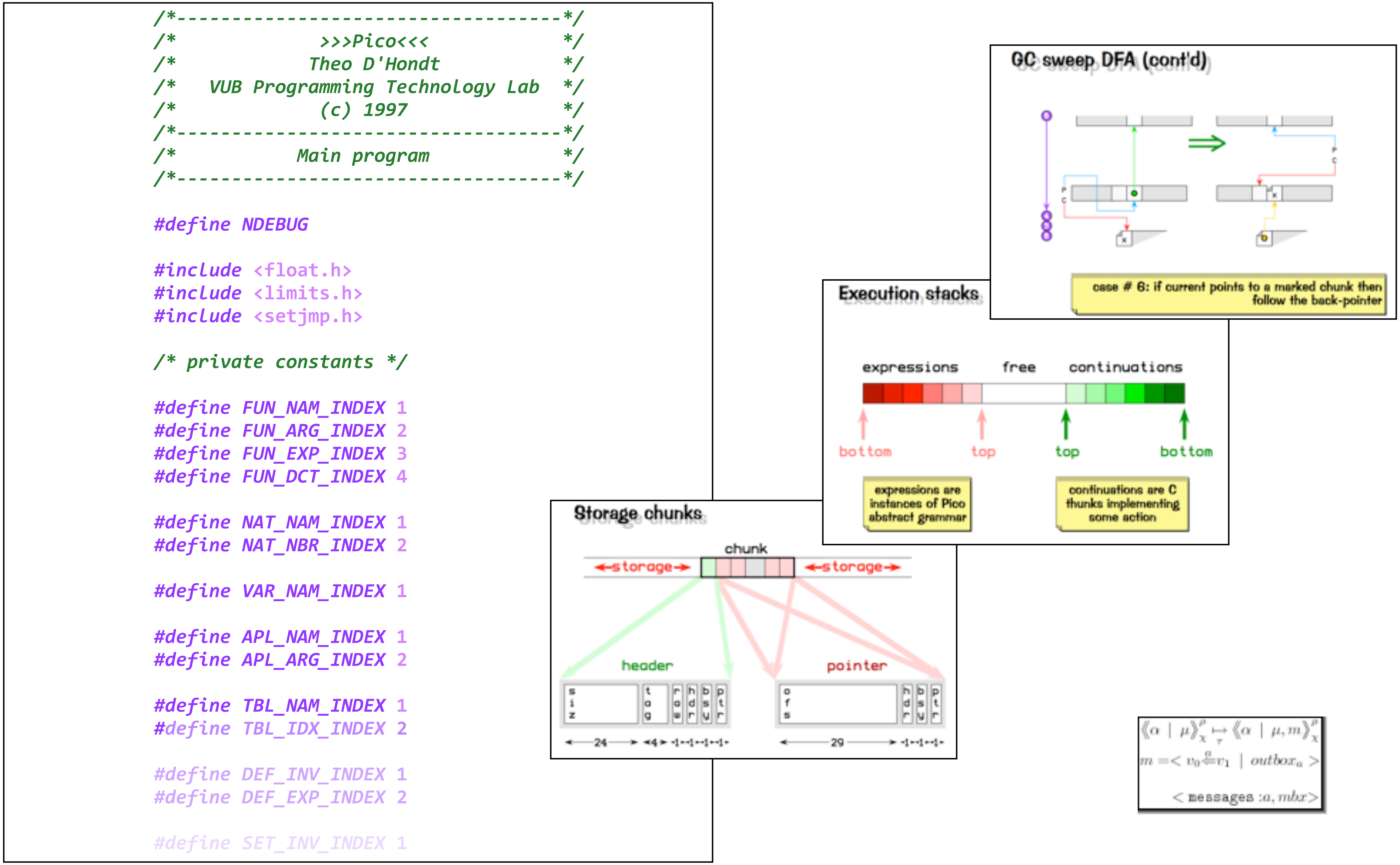
133

```
FIRST FIVE HUNDRED PRIMES
     0002 0233 0547 0877 1229 1597 1993 2371 2749 3187
     0003 0239 0557 0881 1231 1601 1997 2377 2753 3191
     0005 0241 0563 0883 1237 1607 1999 2381 2767 3203
     0007 0251 0569 0887 1249 1609 2003 2383 2777 3209
     0011 0257 0571 0907 1259 1613 2011 2389 2789 3217
```


Instructor's Manual to accompany
Structure and Interpretation of Computer Programs
second edition
λ
Julie Sussman
with Harold Abelson and
Gerald Jay Sussman


The Elements of Computing Systems
Building a Modern Computer from First Principles
Noam Nisan and Shimon Schocken

History: Pico 1



```
/*-----------------------------------*/
/*             >>>Pico<<<            */
/*            Theo D'Hondt           */
/*   VUB Programming Technology Lab  */
/*             (c) 1997              */
/*-----------------------------------*/
/*            Main program           */
/*-----------------------------------*/

#define NDEBUG

#include <float.h>
#include <limits.h>
#include <setjmp.h>

/* private constants */

#define FUN_NAM_INDEX 1
#define FUN_ARG_INDEX 2
#define FUN_EXP_INDEX 3
#define FUN_DCT_INDEX 4

#define NAT_NAM_INDEX 1
#define NAT_NBR_INDEX 2

#define VAR_NAM_INDEX 1

#define APL_NAM_INDEX 1
#define APL_ARG_INDEX 2

#define TBL_NAM_INDEX 1
#define TBL_IDX_INDEX 2

#define DEF_INV_INDEX 1
#define DEF_EXP_INDEX 2

#define SET_INV_INDEX 1
```

GC sweep DFA (cont'd)

case # 6: if current points to a marked chunk then follow the back-pointer

Execution stacks

expressions are instances of Pico abstract grammar

continuations are C thunks implementing some action

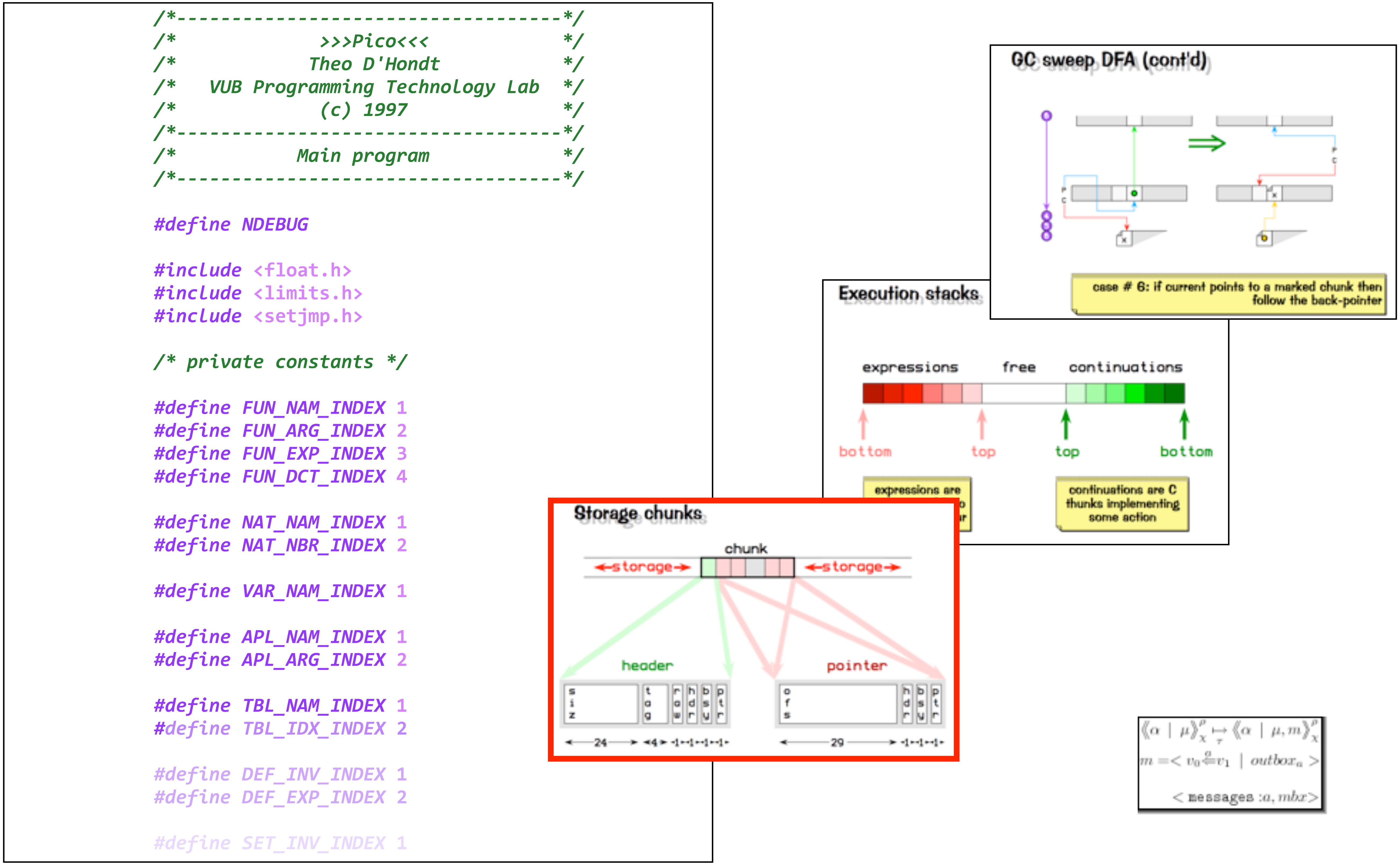
Storage chunks

$\langle\!\langle \alpha \mid \mu \rangle\!\rangle^{\rho}_{\chi} \underset{\tau}{\mapsto} \langle\!\langle \alpha \mid \mu, m \rangle\!\rangle^{\rho}_{\chi}$

$m = < v_0 \overset{a}{\Leftarrow} v_1 \mid outbox_a >$

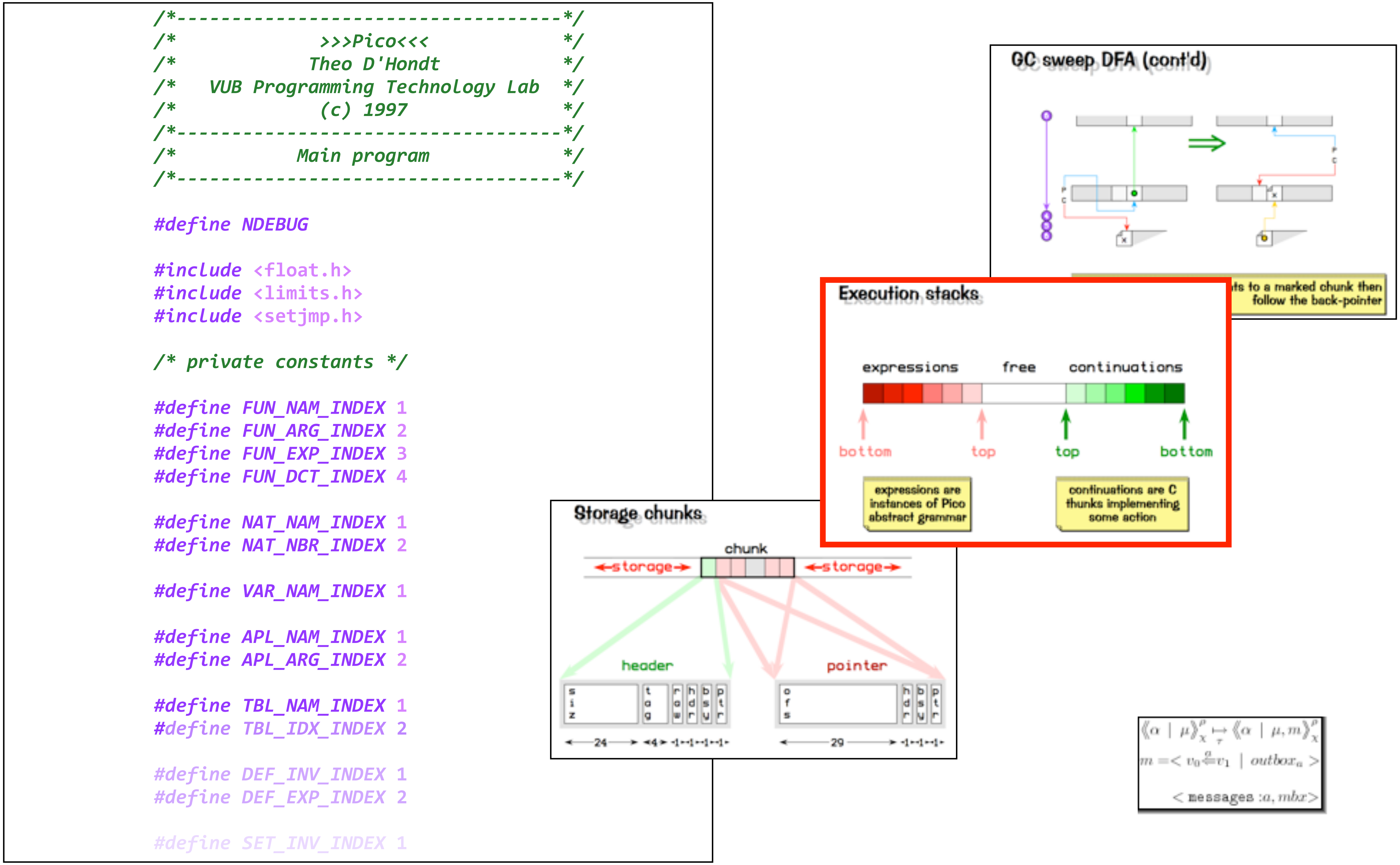
$< \texttt{messages} : a, mbx >$

History: Pico 1



```
/*-----------------------------------*/
/*              >>>Pico<<<           */
/*             Theo D'Hondt          */
/*   VUB Programming Technology Lab  */
/*              (c) 1997             */
/*-----------------------------------*/
/*             Main program          */
/*-----------------------------------*/

#define NDEBUG

#include <float.h>
#include <limits.h>
#include <setjmp.h>

/* private constants */

#define FUN_NAM_INDEX 1
#define FUN_ARG_INDEX 2
#define FUN_EXP_INDEX 3
#define FUN_DCT_INDEX 4

#define NAT_NAM_INDEX 1
#define NAT_NBR_INDEX 2

#define VAR_NAM_INDEX 1

#define APL_NAM_INDEX 1
#define APL_ARG_INDEX 2

#define TBL_NAM_INDEX 1
#define TBL_IDX_INDEX 2

#define DEF_INV_INDEX 1
#define DEF_EXP_INDEX 2

#define SET_INV_INDEX 1
```

History: Pico 1



```
/*-----------------------------------*/
/*             >>>Pico<<<            */
/*            Theo D'Hondt           */
/*   VUB Programming Technology Lab  */
/*             (c) 1997              */
/*-----------------------------------*/
/*            Main program           */
/*-----------------------------------*/

#define NDEBUG

#include <float.h>
#include <limits.h>
#include <setjmp.h>

/* private constants */

#define FUN_NAM_INDEX 1
#define FUN_ARG_INDEX 2
#define FUN_EXP_INDEX 3
#define FUN_DCT_INDEX 4

#define NAT_NAM_INDEX 1
#define NAT_NBR_INDEX 2

#define VAR_NAM_INDEX 1

#define APL_NAM_INDEX 1
#define APL_ARG_INDEX 2

#define TBL_NAM_INDEX 1
#define TBL_IDX_INDEX 2

#define DEF_INV_INDEX 1
#define DEF_EXP_INDEX 2

#define SET_INV_INDEX 1
```

GC sweep DFA (cont'd)

…ts to a marked chunk then follow the back-pointer

Execution stacks

expressions are instances of Pico abstract grammar

continuations are C thunks implementing some action

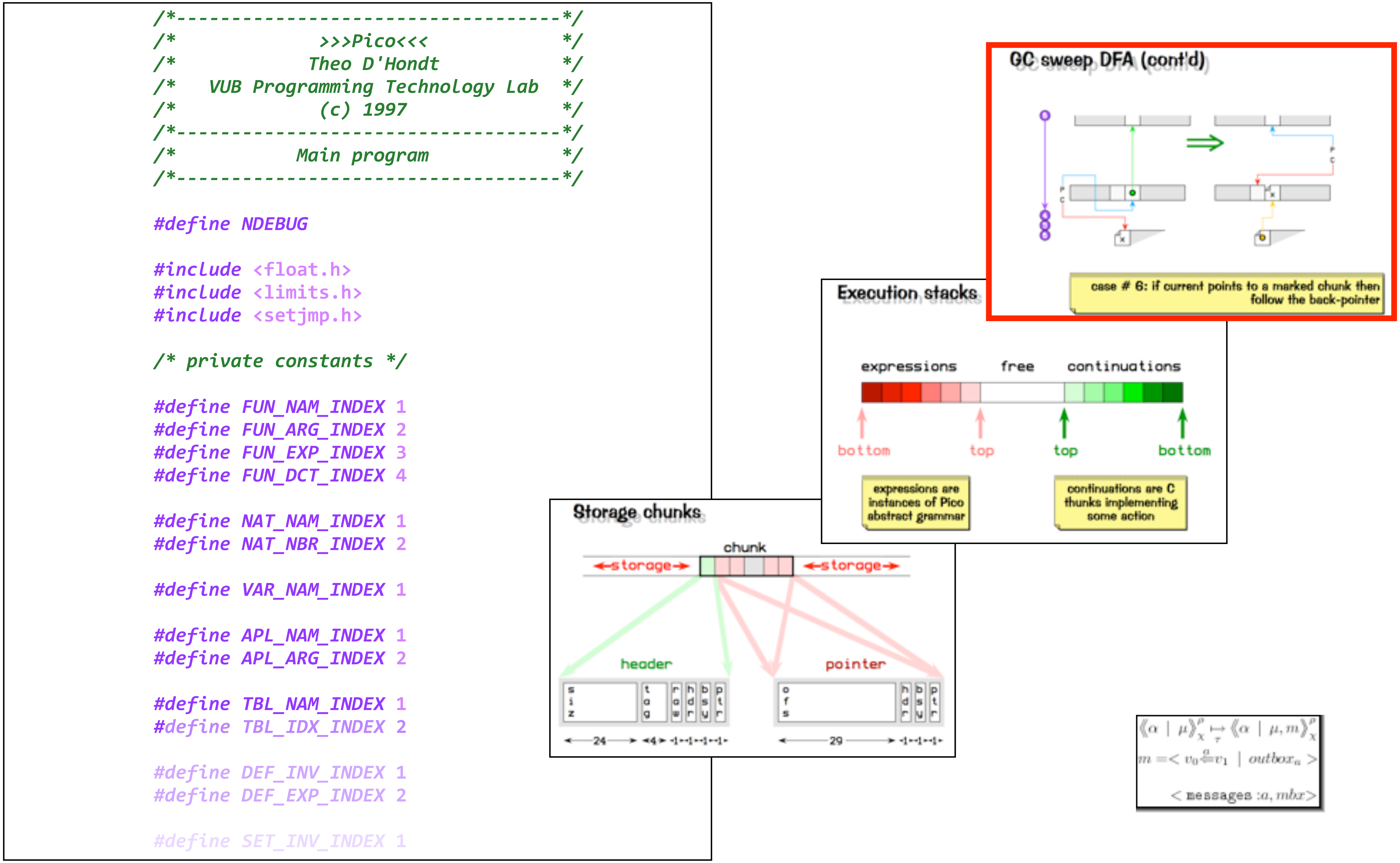
Storage chunks

$$\langle\!\langle \alpha \mid \mu \rangle\!\rangle^{\rho}_{\chi} \underset{\tau}{\mapsto} \langle\!\langle \alpha \mid \mu, m \rangle\!\rangle^{\rho}_{\chi}$$

$$m = < v_0 \overset{a}{\Leftarrow} v_1 \mid outbox_a >$$

$$< \texttt{messages} :a, mbx>$$

History: Pico 1



```
/*-----------------------------------*/
/*             >>>Pico<<<            */
/*            Theo D'Hondt           */
/*   VUB Programming Technology Lab  */
/*             (c) 1997              */
/*-----------------------------------*/
/*            Main program           */
/*-----------------------------------*/

#define NDEBUG

#include <float.h>
#include <limits.h>
#include <setjmp.h>

/* private constants */

#define FUN_NAM_INDEX 1
#define FUN_ARG_INDEX 2
#define FUN_EXP_INDEX 3
#define FUN_DCT_INDEX 4

#define NAT_NAM_INDEX 1
#define NAT_NBR_INDEX 2

#define VAR_NAM_INDEX 1

#define APL_NAM_INDEX 1
#define APL_ARG_INDEX 2

#define TBL_NAM_INDEX 1
#define TBL_IDX_INDEX 2

#define DEF_INV_INDEX 1
#define DEF_EXP_INDEX 2

#define SET_INV_INDEX 1
```

GC sweep DFA (cont'd)

case # 6: if current points to a marked chunk then follow the back-pointer

Execution stacks

expressions are instances of Pico abstract grammar

continuations are C thunks implementing some action

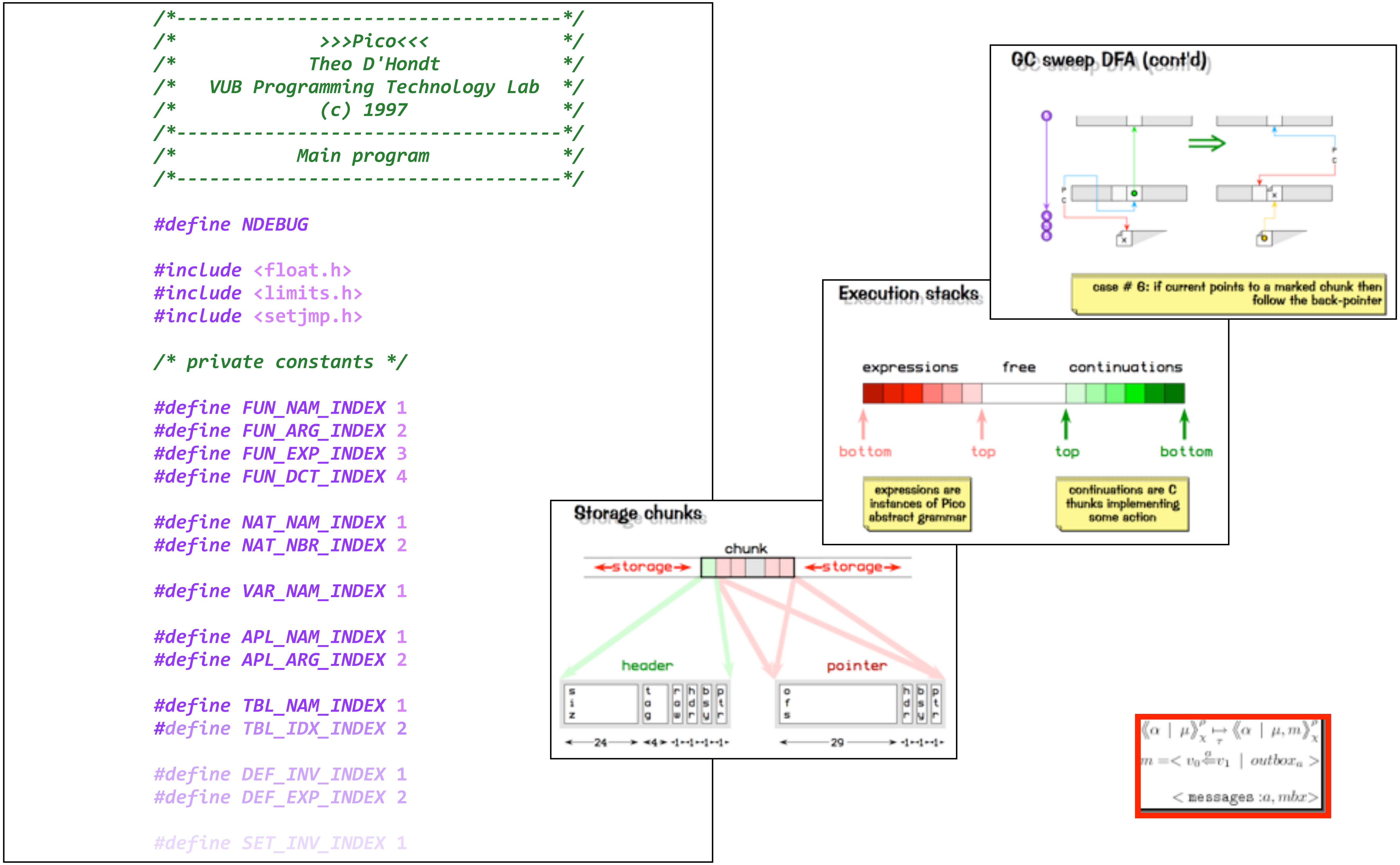
Storage chunks

$$\langle\!\langle \alpha \mid \mu \rangle\!\rangle^{\rho}_{\chi} \mapsto_{\tau} \langle\!\langle \alpha \mid \mu, m \rangle\!\rangle^{\rho}_{\chi}$$

$$m = < v_0 \overset{a}{\Leftarrow} v_1 \mid outbox_a >$$

$$< \mathtt{messages} : a, mbx >$$

History: Pico 1



```
/*-----------------------------------*/
/*             >>>Pico<<<            */
/*            Theo D'Hondt           */
/*    VUB Programming Technology Lab */
/*             (c) 1997              */
/*-----------------------------------*/
/*           Main program            */
/*-----------------------------------*/

#define NDEBUG

#include <float.h>
#include <limits.h>
#include <setjmp.h>

/* private constants */

#define FUN_NAM_INDEX 1
#define FUN_ARG_INDEX 2
#define FUN_EXP_INDEX 3
#define FUN_DCT_INDEX 4

#define NAT_NAM_INDEX 1
#define NAT_NBR_INDEX 2

#define VAR_NAM_INDEX 1

#define APL_NAM_INDEX 1
#define APL_ARG_INDEX 2

#define TBL_NAM_INDEX 1
#define TBL_IDX_INDEX 2

#define DEF_INV_INDEX 1
#define DEF_EXP_INDEX 2

#define SET_INV_INDEX 1
```

GC sweep DFA (cont'd)

case # 6: if current points to a marked chunk then follow the back-pointer

Execution stacks

expressions are instances of Pico abstract grammar

continuations are C thunks implementing some action

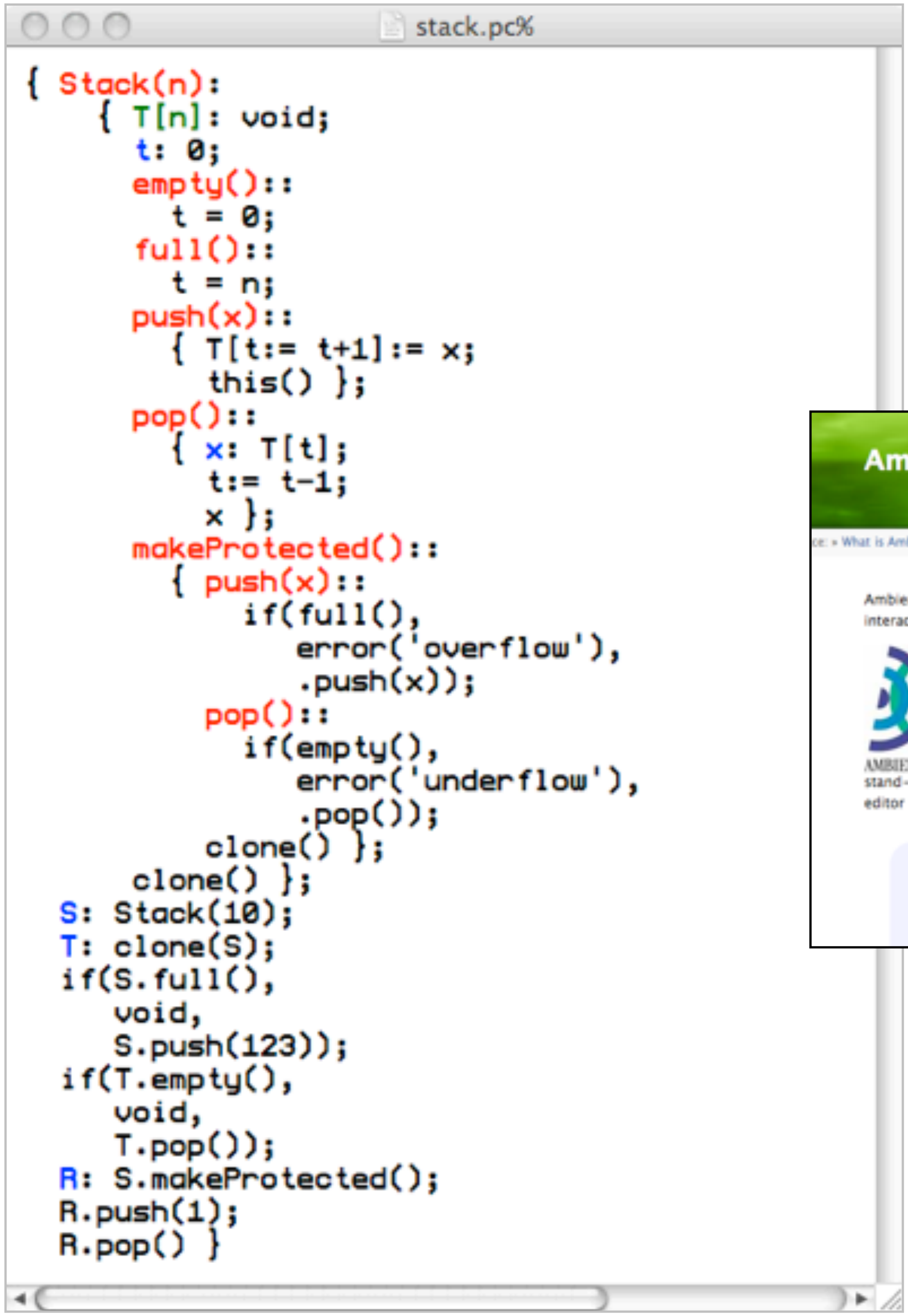
Storage chunks

$\langle\!\langle \alpha \mid \mu \rangle\!\rangle^{\mu'}_{\chi} \mapsto_{\tau} \langle\!\langle \alpha \mid \mu, m \rangle\!\rangle^{\mu'}_{\chi}$

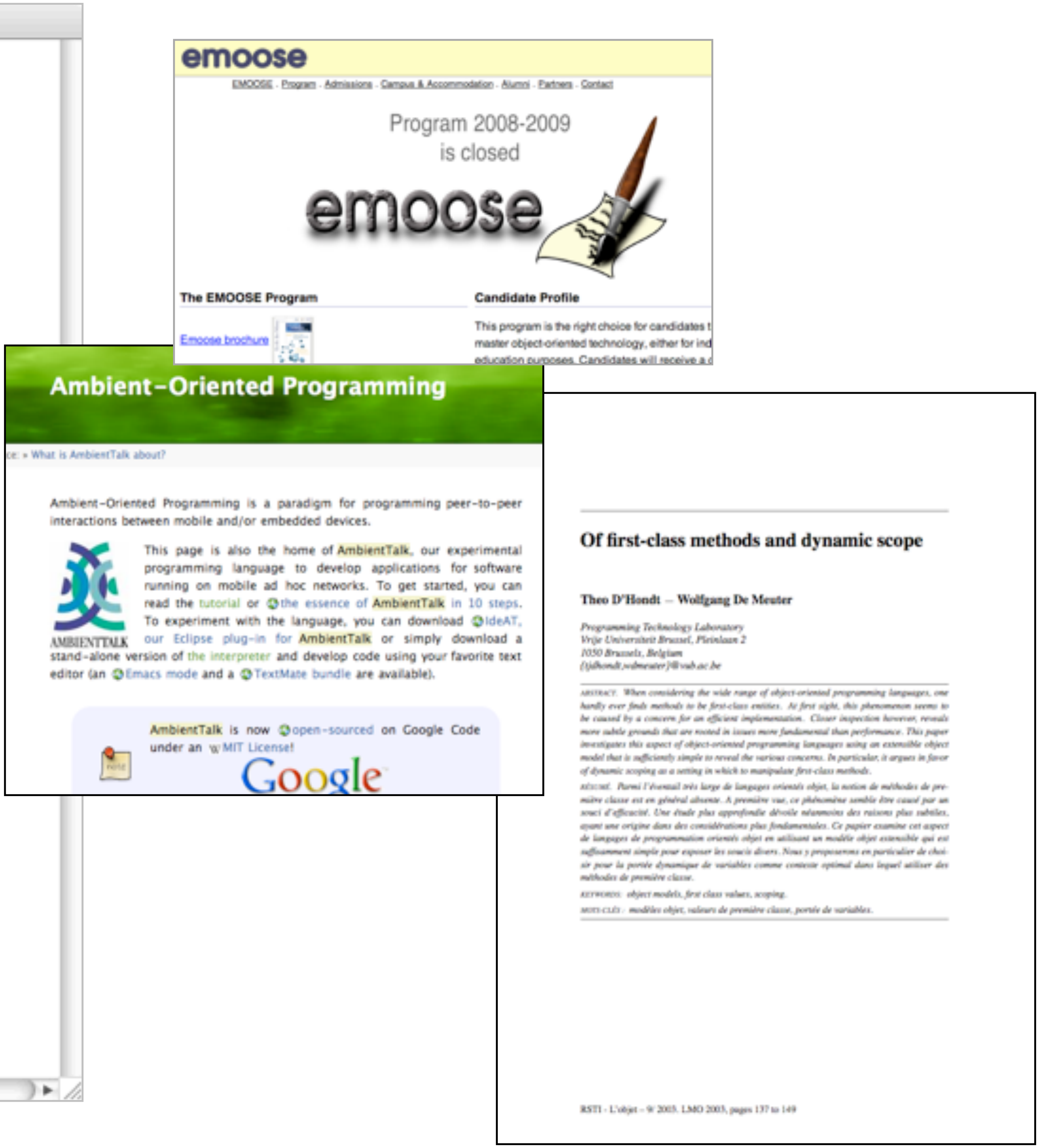
$m = < v_0 \overset{a}{\Leftarrow} v_1 \mid outbox_a >$

$< \texttt{messages} : a, mbx >$

History: Pic%



```
{ Stack(n):
  { T[n]: void;
   t: 0;
   empty()::
    t = 0;
   full()::
    t = n;
   push(x)::
    { T[t:= t+1]:= x;
     this() };
   pop()::
    { x: T[t];
     t:= t-1;
     x };
   makeProtected()::
    { push(x)::
       if(full(),
          error('overflow'),
          .push(x));
      pop()::
       if(empty(),
          error('underflow'),
          .pop());
      clone() };
   clone() };
 S: Stack(10);
 T: clone(S);
 if(S.full(),
   void,
   S.push(123));
 if(T.empty(),
   void,
   T.pop());
 R: S.makeProtected();
 R.push(1);
 R.pop() }
```


emoose
EMOOSE . Program . Admissions . Campus & Accommodation . Alumni . Partners . Contact
Program 2008-2009
is closed
emoose
Emoose brochure
Ambient-Oriented Programming
ce: » What is AmbientTalk about?
AMBIENTTALK
Google

The EMOOSE Program

Candidate Profile

This program is the right choice for candidates t
master object-oriented technology, either for ind
education purposes. Candidates will receive a c

Ambient-Oriented Programming is a paradigm for programming peer-to-peer interactions between mobile and/or embedded devices.

This page is also the home of AmbientTalk, our experimental programming language to develop applications for software running on mobile ad hoc networks. To get started, you can read the tutorial or the essence of AmbientTalk in 10 steps. To experiment with the language, you can download IdeAT, our Eclipse plug-in for AmbientTalk or simply download a stand-alone version of the interpreter and develop code using your favorite text editor (an Emacs mode and a TextMate bundle are available).

AmbientTalk is now open-sourced on Google Code under an MIT License!

Of first-class methods and dynamic scope

Theo D'Hondt — Wolfgang De Meuter

Programming Technology Laboratory
Vrije Universiteit Brussel, Pleinlaan 2
1050 Brussels, Belgium
{tjdhondt,wdmeuter}@vub.ac.be

ABSTRACT. *When considering the wide range of object-oriented programming languages, one hardly ever finds methods to be first-class entities. At first sight, this phenomenon seems to be caused by a concern for an efficient implementation. Closer inspection however, reveals more subtle grounds that are rooted in issues more fundamental than performance. This paper investigates this aspect of object-oriented programming languages using an extensible object model that is sufficiently simple to reveal the various concerns. In particular, it argues in favor of dynamic scoping as a setting in which to manipulate first-class methods.*

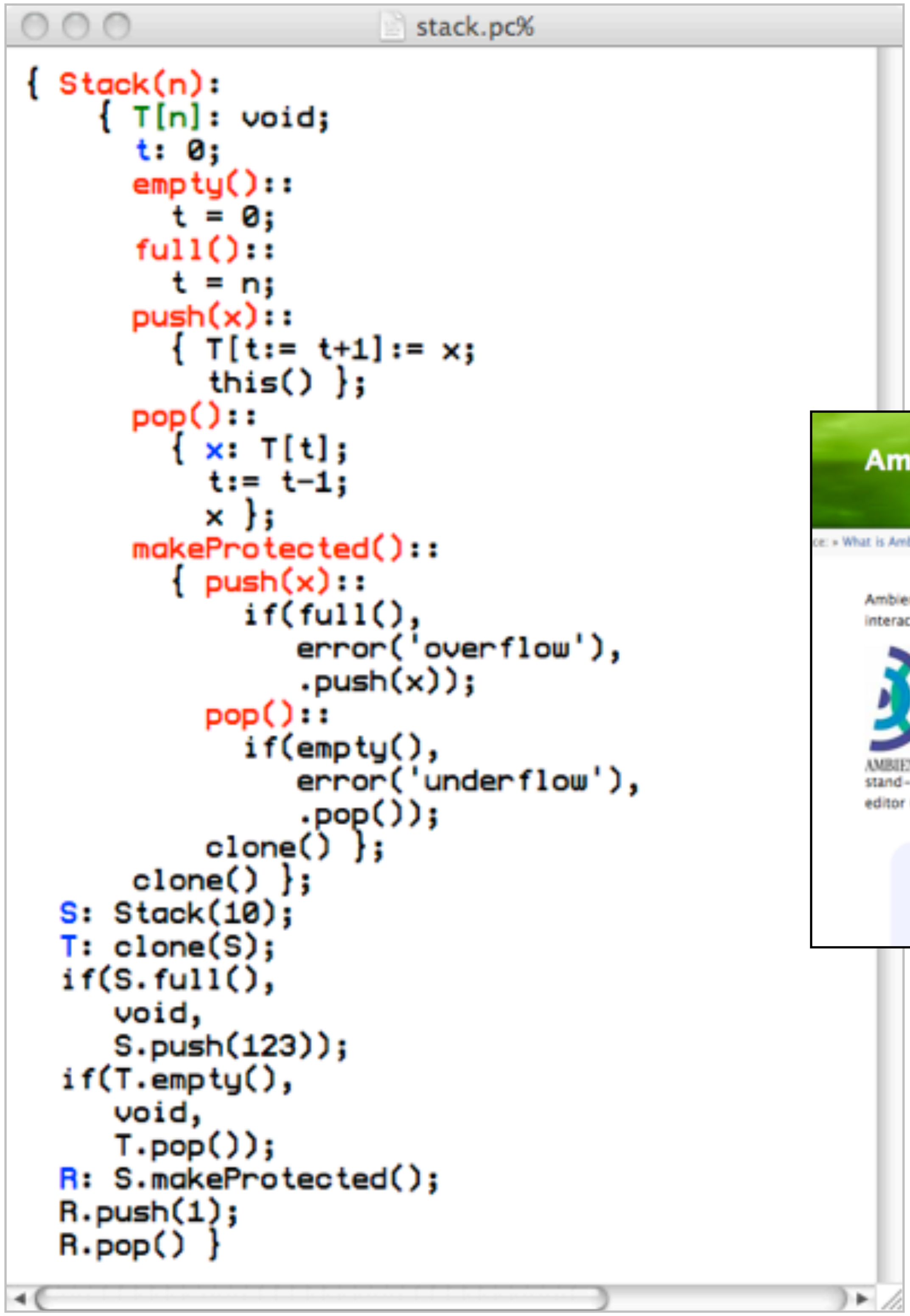
RÉSUMÉ. *Parmi l'éventail très large de langages orientés objet, la notion de méthodes de première classe est en général absente. A première vue, ce phénomène semble être causé par un souci d'efficacité. Une étude plus approfondie dévoile néanmoins des raisons plus subtiles, ayant une origine dans des considérations plus fondamentales. Ce papier examine cet aspect de langages de programmation orientés objet en utilisant un modèle objet extensible qui est suffisamment simple pour exposer les soucis divers. Nous y proposerons en particulier de choisir pour la portée dynamique de variables comme contexte optimal dans lequel utiliser des méthodes de première classe.*

KEYWORDS: *object models, first class values, scoping.*

MOTS-CLÉS : *modèles objet, valeurs de première classe, portée de variables.*

RSTI - L'objet – 9/2003. LMO 2003, pages 137 to 149

History: Pic%



```
{ Stack(n):
    { T[n]: void;
      t: 0;
      empty()::
        t = 0;
      full()::
        t = n;
      push(x)::
        { T[t:= t+1]:= x;
          this() };
      pop()::
        { x: T[t];
          t:= t-1;
          x };
      makeProtected()::
        { push(x)::
            if(full(),
               error('overflow'),
               .push(x));
          pop()::
            if(empty(),
               error('underflow'),
               .pop());
          clone() };
      clone() };
  S: Stack(10);
  T: clone(S);
  if(S.full(),
     void,
     S.push(123));
  if(T.empty(),
     void,
     T.pop());
  R: S.makeProtected();
  R.push(1);
  R.pop() }
```

emoose

EMOOSE . Program . Admissions . Campus & Accommodation . Alumni . Partners . Contact

Program 2008-2009
is closed

emoose

The EMOOSE Program

Emoose brochure

Candidate Profile

This program is the right choice for candidates
master object-oriented technology, either for in
education purposes. Candidates will receive a

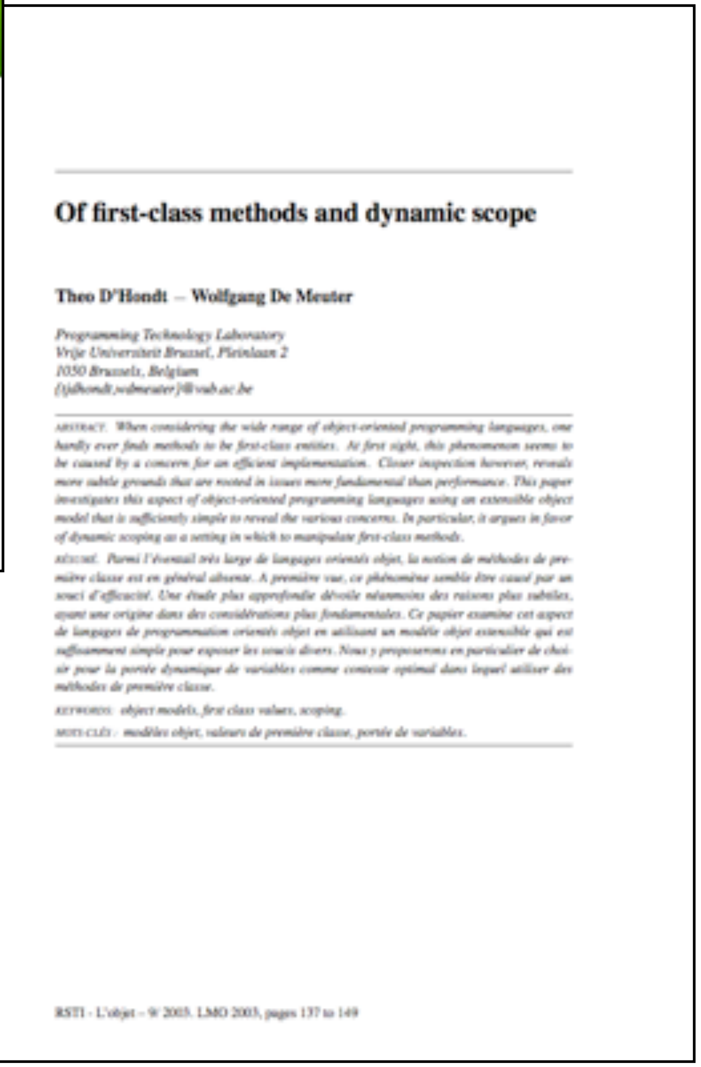
Ambient-Oriented Programming

ce: » What is AmbientTalk about?

Ambient-Oriented Programming is a paradigm for programming peer-to-peer interactions between mobile and/or embedded devices.

This page is also the home of AmbientTalk, our experimental programming language to develop applications for software running on mobile ad hoc networks. To get started, you can read the tutorial or the essence of AmbientTalk in 10 steps. To experiment with the language, you can download IdeAT, our Eclipse plug-in for AmbientTalk or simply download a stand-alone version of the interpreter and develop code using your favorite text editor (an Emacs mode and a TextMate bundle are available).

AmbientTalk is now open-sourced on Google Code under an MIT License!


Of first-class methods and dynamic scope

Theo D'Hondt — Wolfgang De Meuter

Programming Technology Laboratory
Vrije Universiteit Brussel, Pleinlaan 2
1050 Brussels, Belgium
{tjdhondt,wdmeuter}@vub.ac.be

ABSTRACT. *When considering the wide range of object-oriented programming languages, one hardly ever finds methods to be first-class entities. At first sight, this phenomenon seems to be caused by a concern for an efficient implementation. Closer inspection however, reveals more subtle grounds that are rooted in issues more fundamental than performance. This paper investigates this aspect of object-oriented programming languages using an extensible object model that is sufficiently simple to reveal the various concerns. In particular, it argues in favor of dynamic scoping as a setting in which to manipulate first-class methods.*

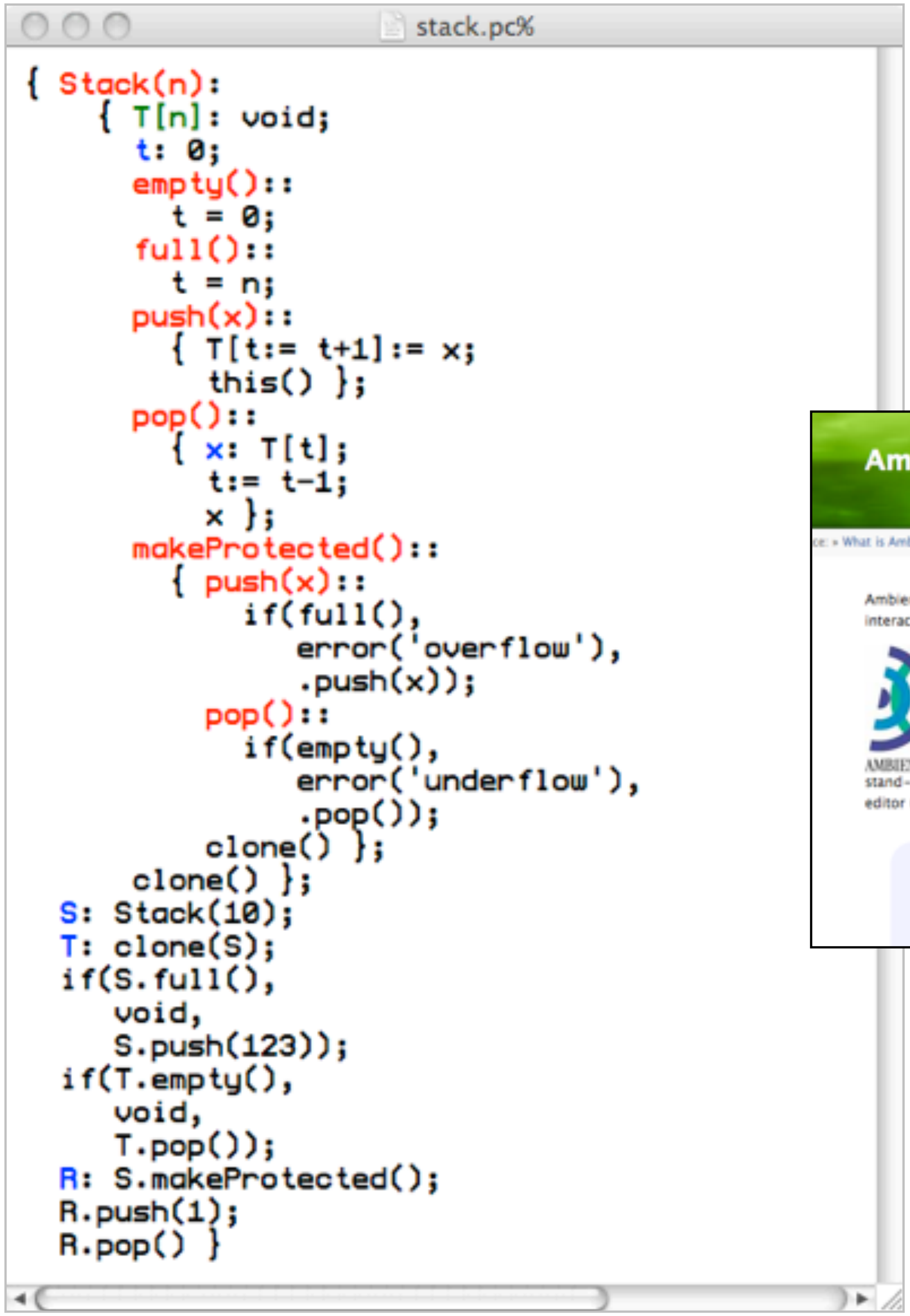
RÉSUMÉ. *Parmi l'éventail très large de langages orientés objet, la notion de méthodes de première classe est en général absente. A première vue, ce phénomène semble être causé par un souci d'efficacité. Une étude plus approfondie dévoile néanmoins des raisons plus subtiles, ayant une origine dans des considérations plus fondamentales. Ce papier examine cet aspect de langages de programmation orientés objet en utilisant un modèle objet extensible qui est suffisamment simple pour exposer les soucis divers. Nous y proposons en particulier de choisir pour la portée dynamique de variables comme contexte optimal dans lequel utiliser des méthodes de première classe.*

KEYWORDS: *object models, first class values, scoping.*

MOTS-CLÉS : *modèles objet, valeurs de première classe, portée de variables.*

RSTI - L'objet – 9/2003. LMO 2003, pages 137 to 149

History: Pic%



```
{ Stack(n):
  { T[n]: void;
   t: 0;
   empty()::
    t = 0;
   full()::
    t = n;
   push(x)::
    { T[t:= t+1]:= x;
     this() };
   pop()::
    { x: T[t];
     t:= t-1;
     x };
   makeProtected()::
    { push(x)::
      if(full(),
        error('overflow'),
        .push(x));
     pop()::
      if(empty(),
        error('underflow'),
        .pop());
     clone() };
   clone() };
 S: Stack(10);
 T: clone(S);
 if(S.full(),
   void,
   S.push(123));
 if(T.empty(),
   void,
   T.pop());
 R: S.makeProtected();
 R.push(1);
 R.pop() }
```

emoose
EMOOSE · Program · Admissions · Campus & Accommodation · Alumni · Partners · Contact

Program 2008-2009
is closed

The EMOOSE Program

Emoose brochure

Candidate Profile

This program is the right choice for candidates t
master object-oriented technology, either for ind
education purposes. Candidates will receive a c

Ambient-Oriented Programming
ce: » What is AmbientTalk about?
AMBIENTTALK

Ambient-Oriented Programming is a paradigm for programming peer-to-
interactions between mobile and/or embedded devices.

This page is also the home of AmbientTalk, our experim
programming language to develop applications for sof
running on mobile ad hoc networks. To get started, you
read the tutorial or the essence of AmbientTalk in 10 s
To experiment with the language, you can download lo
our Eclipse plug-in for AmbientTalk or simply downlo
stand-alone version of the interpreter and develop code using your favorite
editor (an Emacs mode and a TextMate bundle are available).

AmbientTalk is now open-sourced on Google Code under an MIT License!

Of first-class methods and dynamic scope

Theo D'Hondt — Wolfgang De Meuter

Programming Technology Laboratory
Vrije Universiteit Brussel, Pleinlaan 2
1050 Brussels, Belgium
{tjdhondt,wdmeuter}@vub.ac.be

ABSTRACT. *When considering the wide range of object-oriented programming languages, one hardly ever finds methods to be first-class entities. At first sight, this phenomenon seems to be caused by a concern for an efficient implementation. Closer inspection however, reveals more subtle grounds that are rooted in issues more fundamental than performance. This paper investigates this aspect of object-oriented programming languages using an extensible object model that is sufficiently simple to reveal the various concerns. In particular, it argues in favor of dynamic scoping as a setting in which to manipulate first-class methods.*

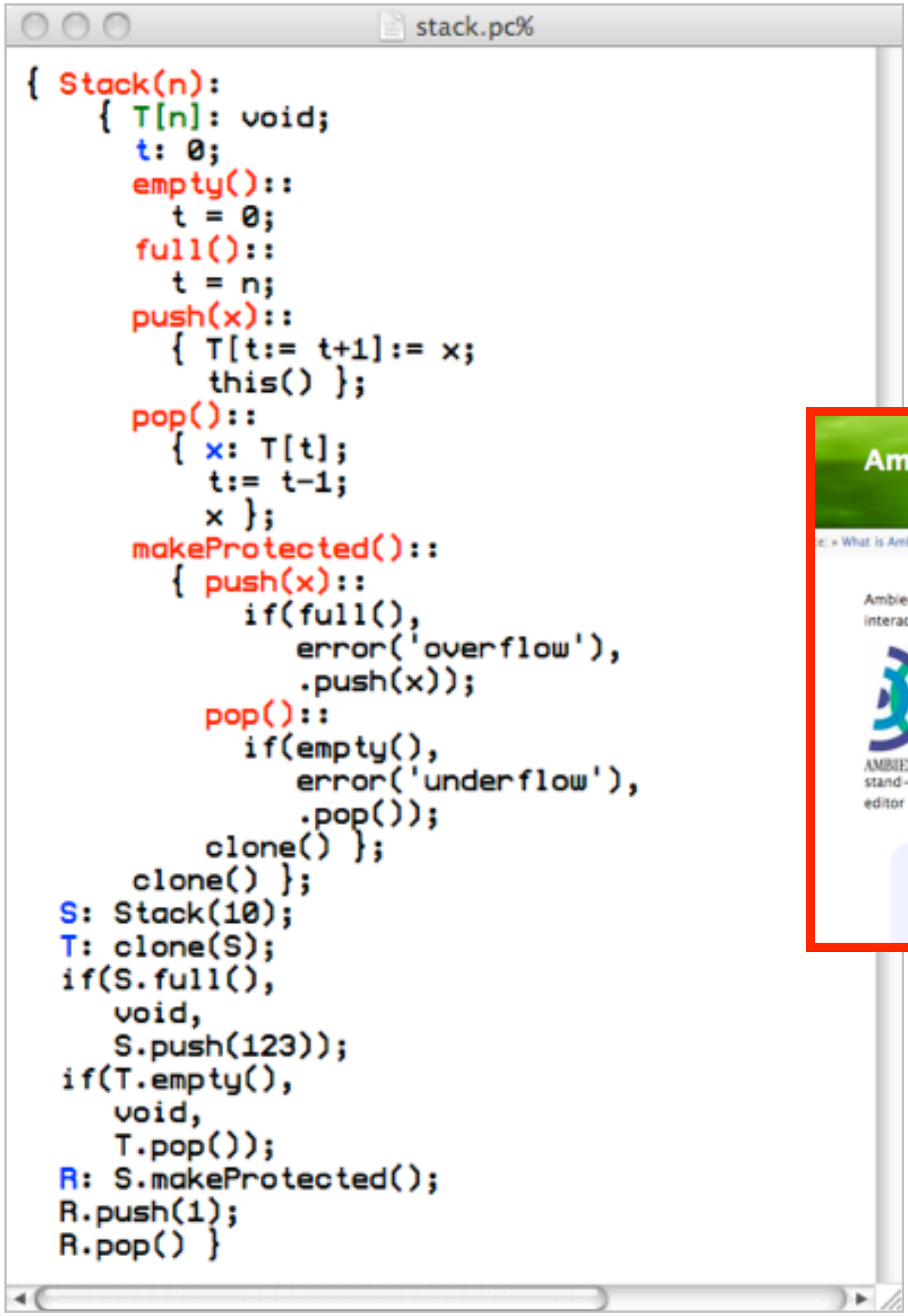
RÉSUMÉ. *Parmi l'éventail très large de langages orientés objet, la notion de méthodes de première classe est en général absente. A première vue, ce phénomène semble être causé par un souci d'efficacité. Une étude plus approfondie dévoile néanmoins des raisons plus subtiles, ayant une origine dans des considérations plus fondamentales. Ce papier examine cet aspect de langages de programmation orientés objet en utilisant un modèle objet extensible qui est suffisamment simple pour exposer les soucis divers. Nous y proposerons en particulier de choisir pour la portée dynamique de variables comme contexte optimal dans lequel utiliser des méthodes de première classe.*

KEYWORDS: *object models, first class values, scoping.*

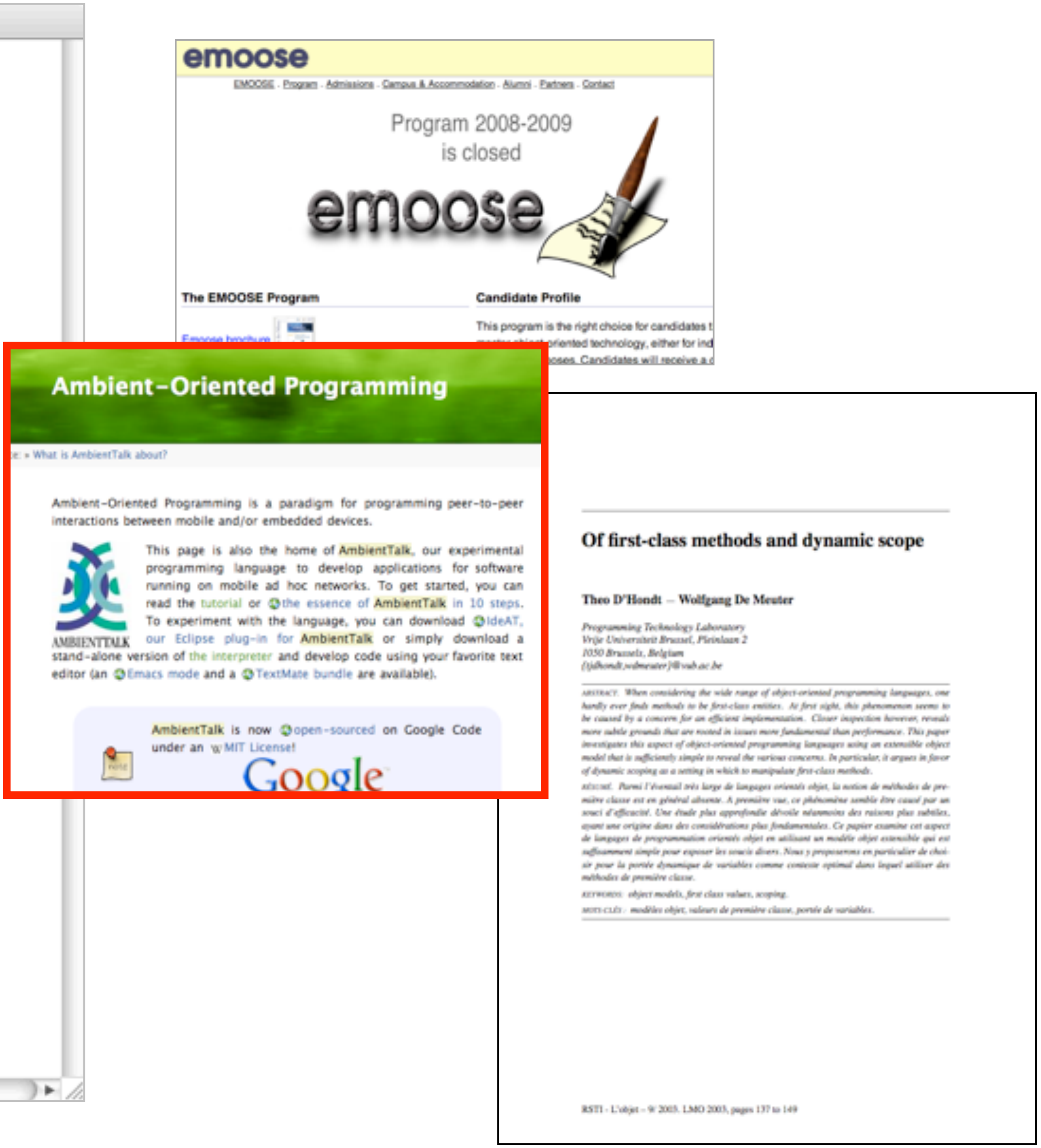
MOTS-CLÉS : *modèles objet, valeurs de première classe, portée de variables.*

RSTI - L'objet – 9/ 2003. LMO 2003, pages 137 to 149

History: Pic%



```
{ Stack(n):
  { T[n]: void;
   t: 0;
   empty()::
    t = 0;
   full()::
    t = n;
   push(x)::
    { T[t:= t+1]:= x;
     this() };
   pop()::
    { x: T[t];
     t:= t-1;
     x };
   makeProtected()::
    { push(x)::
      if(full(),
        error('overflow'),
        .push(x));
     pop()::
      if(empty(),
        error('underflow'),
        .pop());
     clone() };
   clone() };
 S: Stack(10);
 T: clone(S);
 if(S.full(),
   void,
   S.push(123));
 if(T.empty(),
   void,
   T.pop());
 R: S.makeProtected();
 R.push(1);
 R.pop() }
```



emoose

EMOOSE . Program . Admissions . Campus & Accommodation . Alumni . Partners . Contact

Program 2008-2009
is closed

emoose

The EMOOSE Program

Emoose brochure

Candidate Profile

This program is the right choice for candidates t
oriented technology, either for ind
oses. Candidates will receive a c

Ambient-Oriented Programming

e: » What is AmbientTalk about?

Ambient-Oriented Programming is a paradigm for programming peer-to-peer interactions between mobile and/or embedded devices.

AMBIENTTALK

This page is also the home of AmbientTalk, our experimental programming language to develop applications for software running on mobile ad hoc networks. To get started, you can read the tutorial or the essence of AmbientTalk in 10 steps. To experiment with the language, you can download IdeAT, our Eclipse plug-in for AmbientTalk or simply download a stand-alone version of the interpreter and develop code using your favorite text editor (an Emacs mode and a TextMate bundle are available).

AmbientTalk is now open-sourced on Google Code under an MIT License!

Google

Of first-class methods and dynamic scope

Theo D'Hondt — Wolfgang De Meuter

Programming Technology Laboratory
Vrije Universiteit Brussel, Pleinlaan 2
1050 Brussels, Belgium
{tjdhondt,wdmeuter}@vub.ac.be

ABSTRACT. *When considering the wide range of object-oriented programming languages, one hardly ever finds methods to be first-class entities. At first sight, this phenomenon seems to be caused by a concern for an efficient implementation. Closer inspection however, reveals more subtle grounds that are rooted in issues more fundamental than performance. This paper investigates this aspect of object-oriented programming languages using an extensible object model that is sufficiently simple to reveal the various concerns. In particular, it argues in favor of dynamic scoping as a setting in which to manipulate first-class methods.*

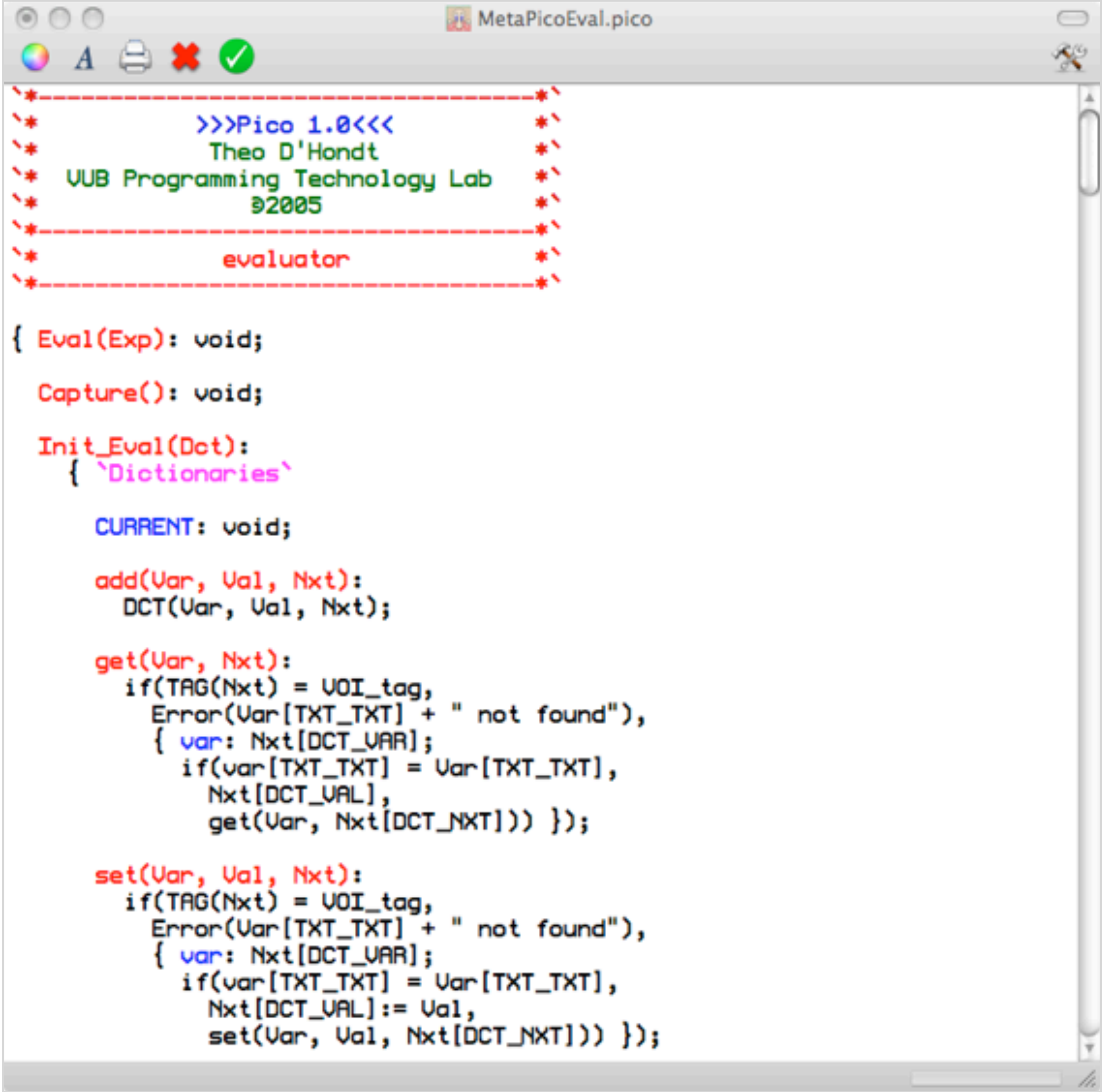
RÉSUMÉ. *Parmi l'éventail très large de langages orientés objet, la notion de méthodes de première classe est en général absente. A première vue, ce phénomène semble être causé par un souci d'efficacité. Une étude plus approfondie dévoile néanmoins des raisons plus subtiles, ayant une origine dans des considérations plus fondamentales. Ce papier examine cet aspect de langages de programmation orientés objet en utilisant un modèle objet extensible qui est suffisamment simple pour exposer les soucis divers. Nous y proposerons en particulier de choisir pour la portée dynamique de variables comme contexte optimal dans lequel utiliser des méthodes de première classe.*

KEYWORDS: *object models, first class values, scoping.*

MOTS-CLÉS : *modèles objet, valeurs de première classe, portée de variables.*

RSTI - L'objet – 9/2003. LMO 2003, pages 137 to 149

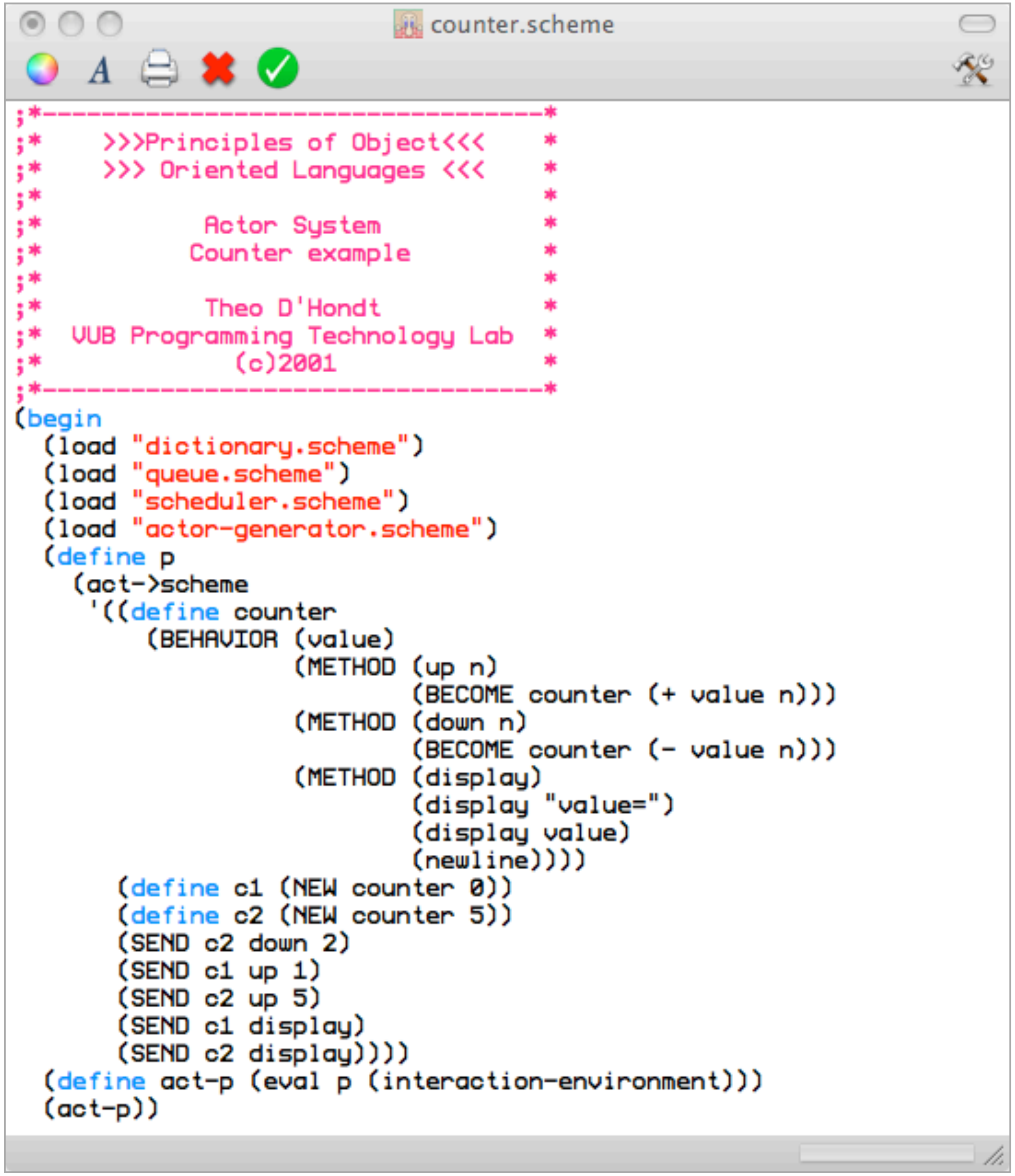
History: Pico 2



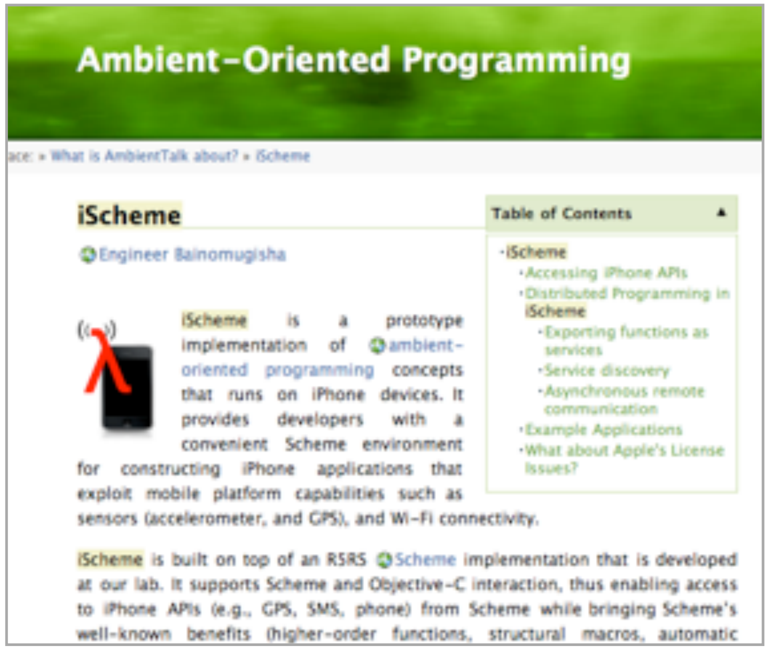
```
`*------------------------------------*`
`*             >>>Pico 1.0<<<          *`
`*             Theo D'Hondt            *`
`*  VUB Programming Technology Lab     *`
`*               ©2005                 *`
`*------------------------------------*`
`*              evaluator              *`
`*------------------------------------*`

{ Eval(Exp): void;

  Capture(): void;

  Init_Eval(Dct):
    { `Dictionaries`

      CURRENT: void;

      add(Var, Val, Nxt):
        DCT(Var, Val, Nxt);

      get(Var, Nxt):
        if(TAG(Nxt) = VOI_tag,
          Error(Var[TXT_TXT] + " not found"),
          { var: Nxt[DCT_VAR];
            if(var[TXT_TXT] = Var[TXT_TXT],
              Nxt[DCT_VAL],
              get(Var, Nxt[DCT_NXT])) });

      set(Var, Val, Nxt):
        if(TAG(Nxt) = VOI_tag,
          Error(Var[TXT_TXT] + " not found"),
          { var: Nxt[DCT_VAR];
            if(var[TXT_TXT] = Var[TXT_TXT],
              Nxt[DCT_VAL]:= Val,
              set(Var, Val, Nxt[DCT_NXT])) });
```

- ☑ 1st class ∀
- ☑ no compromises
- ☑ abstract grammars
- ☑ uniform memory
- ☑ continuations
- ☑ interpretation

History: \'skēm\



```
;*----------------------------------*
;*    >>>Principles of Object<<<    *
;*    >>> Oriented Languages <<<    *
;*                                  *
;*           Actor System           *
;*          Counter example         *
;*                                  *
;*           Theo D'Hondt           *
;*  VUB Programming Technology Lab  *
;*             (c)2001              *
;*----------------------------------*
(begin
  (load "dictionary.scheme")
  (load "queue.scheme")
  (load "scheduler.scheme")
  (load "actor-generator.scheme")
  (define p
    (act->scheme
     '((define counter
         (BEHAVIOR (value)
                   (METHOD (up n)
                           (BECOME counter (+ value n)))
                   (METHOD (down n)
                           (BECOME counter (- value n)))
                   (METHOD (display)
                           (display "value=")
                           (display value)
                           (newline))))
       (define c1 (NEW counter 0))
       (define c2 (NEW counter 5))
       (SEND c2 down 2)
       (SEND c1 up 1)
       (SEND c2 up 5)
       (SEND c1 display)
       (SEND c2 display))))
  (define act-p (eval p (interaction-environment)))
  (act-p))
```



Ambient-Oriented Programming

ace: » What is AmbientTalk about? » iScheme

iScheme

Engineer Bainomugisha

iScheme is a prototype implementation of ambient-oriented programming concepts that runs on iPhone devices. It provides developers with a convenient Scheme environment for constructing iPhone applications that exploit mobile platform capabilities such as sensors (accelerometer, and GPS), and Wi-Fi connectivity.

iScheme is built on top of an R5RS Scheme implementation that is developed at our lab. It supports Scheme and Objective-C interaction, thus enabling access to iPhone APIs (e.g., GPS, SMS, phone) from Scheme while bringing Scheme's well-known benefits (higher-order functions, structural macros, automatic

Table of Contents ▲

SLIP: design

☑ performance,performance,performance

☑ smallest possible footprint (loc, kb)

☑ abstract syntax everywhere

☑ no compromises or limitations

☑ minimal dynamic language

☑ main focus on interpretation

☑ clean code

☑ incremental implementation

SLIP: the language

```
(begin
 (define (counter count)
  (define (self message)
   (if (eq? message '+)
     (begin
      (set! count (+ count 1))
      self)
     (if (eq? message '-)
      (begin
       (set! count (- count 1))
       self)
     (if (eq? message '?)
      count
      'error))))
   self)
 (define c (counter 10))
 ((((c '+) '+) '-) '?))
```

SLIP: the language (cont'd)

```
(begin
  (define empty   0)
  (define full    1)
  (define push    2)
  (define pop     3)
  (define (Stack n)
    (define stack (make-vector n))
    (define top -1)
    (define (empty)
      (< top 0))
    (define (full)
      (>= top n))
    (define (push item)
      (set! top (+ top 1))
      (vector-set! stack top item)
      ())
    (define (pop)
      (define item (vector-ref stack top))
      (set! top (- top 1))
      item)
    (define (self message . arguments)
      (define methods (vector empty full push pop))
      (apply (vector-ref methods message) arguments))
    self)
```

```
(define S (Stack 10))
(define T (Stack 20))
(if (S full)
  (display 'Overflow)
  (S push 123))
(T push 456)
(if (S empty)
  (display 'Underflow)
  (S pop))
(display (T pop))
(newline)
(if (S empty)
  (display 'Underflow)
  (S pop)))
```

SLIP: the language (cont'd)

☑ `begin`, `define`, `if`, `lambda`, `set!`(,`while`)

☑ `define` and `set!` have a value

☑ `define` used anywhere

☑ `()` instead of `'()`

☑ local variables ≈ parameters

☑ no forward references

☑ natives inherited from metalevel

☑ no top-level sequences

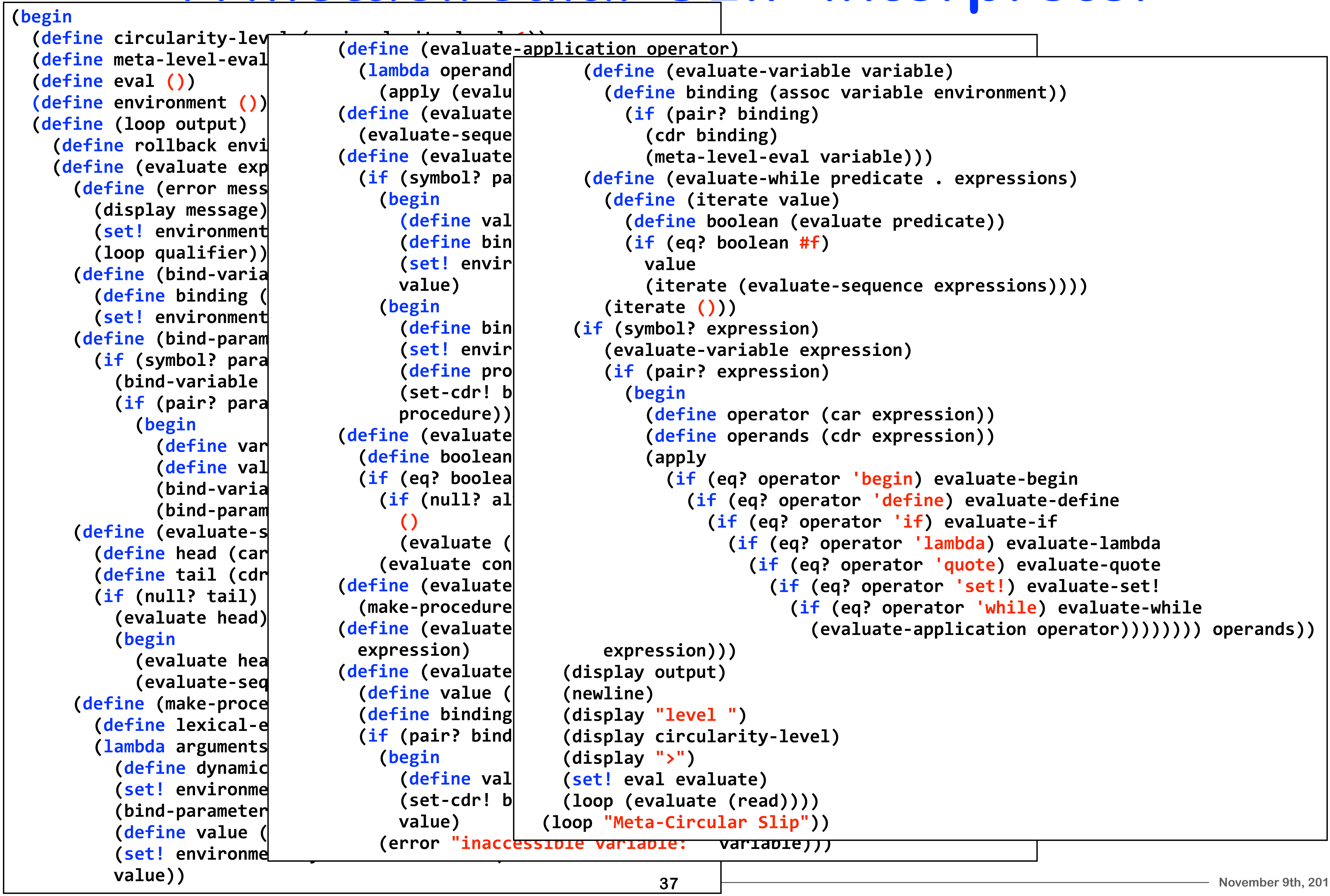
A Scheme interpreter for SLIP

```scheme
(begin
 (define environment '())
 (define (loop output)
  (define rollback environment)
  (define (error message qualifier)
   (display message)
   (set! environment rollback)
   (loop qualifier))
  (define (bind-variable variable value)
   (define binding (cons variable value))
   (set! environment (cons binding environment)))
  (define (bind-parameters parameters arguments)
   (for-each bind-variable parameters arguments))
  (define (evaluate-sequence expressions)
   (define head (car expressions))
   (define tail (cdr expressions))
   (if (null? tail)
     (evaluate head)
     (evaluate-sequence tail)))
  (define (make-procedure parameters expression)
   (define lexical-scope environment)
   (lambda arguments
    (define dynamic-scope environment)
    (set! environment lexical-scope)
    (bind-parameters parameters arguments)
    (let ((value (evaluate expression)))
     (set! environment dynamic-scope)
     value)))
  (define (evaluate-application operator)
   (lambda operands
    (apply (evaluate operator) (map evaluate operands))))
  (define (evaluate-begin . expressions)
   (evaluate-sequence expressions))
  (define (evaluate-define variable expression)
   (define binding (cons variable '()))
   (set! environment (cons binding environment))
   (let ((value (evaluate expression)))
    (set-cdr! binding value)
    value))
  (define (evaluate-if predicate consequent alternative)
   (define boolean (evaluate predicate))
   (if (eq? boolean #f)
    (evaluate alternative)
    (evaluate consequent)))
  (define (evaluate-lambda parameters expression)
   (make-procedure parameters expression))
  (define (evaluate-set! variable expression)
   (define binding (assoc variable environment))
   (if binding
    (let ((value (evaluate expression)))
     (set-cdr! binding value)
     value)
    (error "inaccessible variable: " variable)))
  (define (evaluate-variable variable)
   (define binding (assoc variable environment))
    (if binding
     (cdr binding)
     (eval variable (interaction-environment))))
  (define (evaluate expression)
   (cond
    ((symbol? expression)
     (evaluate-variable expression))
    ((pair? expression)
     (let ((operator (car expression))
           (operands (cdr expression)))
      (apply
       (case operator
        ((begin)  evaluate-begin )
        ((define) evaluate-define)
        ((if)     evaluate-if    )
        ((lambda) evaluate-lambda)
        ((set!)   evaluate-set!  )
        (else     (evaluate-application operator))) operands)))
    (else
     expression)));
  (display output)
  (newline)
  (display ">>>")
  (loop (evaluate (read))))
 (loop "Slip version 0"))
```

A Scheme interpreter for SLIP

```
(begin
  (define environment '())
  (define (loop output)
    (define rollback environment)
    (define (error message qualifier)
      (display message)
      (set! environment rollback)
      (loop qualifier))
    (define (bind-variable variable value)
      (define binding (cons variable value))
      (set! environment (cons binding environment)))
    (define (bind-parameters parameters arguments)
      (for-each bind-variable parameters arguments))
    (define (evaluate-sequence expressions)
      (define head (car expressions))
      (define tail (cdr expressions))
      (if (null? tail)
         (evaluate head)
         (evaluate-sequence tail)))
    (define (make-procedure parameters expression)
      (define lexical-scope environment)
      (lambda arguments
        (define dynamic-scope environment)
        (set! environment lexical-scope)
        (bind-parameters parameters arguments)
        (let ((value (evaluate expression)))
          (set! environment dynamic-scope)
          value)))
    (define (evaluate-application operator)
      (lambda operands
        (apply (evaluate operator) (map evaluate operands))))
    (define (evaluate-begin . expressions)
      (evaluate-sequence expressions))
    (define (evaluate-define variable expression)
      (define binding (cons variable '()))
      (set! environment (cons binding environment))
      (let ((value (evaluate expression)))
        (set-cdr! binding value)
        value))
    (define (evaluate-if predicate consequent alternative)
      (define boolean (evaluate predicate))
      (if (eq? boolean #f)
        (evaluate alternative)
        (evaluate consequent)))
    (define (evaluate-lambda parameters expression)
      (make-procedure parameters expression))
    (define (evaluate-set! variable expression)
      (define binding (assoc variable environment))
      (if binding
        (let ((value (evaluate expression)))
          (set-cdr! binding value)
          value)
        (error "inaccessible variable: " variable)))
    (define (evaluate-variable variable)
      (define binding (assoc variable environment))
        (if binding
          (cdr binding)
          (eval variable (interaction-environment))))
    (define (evaluate expression)
      (cond
        ((symbol? expression)
         (evaluate-variable expression))
        ((pair? expression)
         (let ((operator (car expression))
               (operands (cdr expression)))
           (apply
             (case operator
               ((begin)  evaluate-begin )
               ((define) evaluate-define)
               ((if)     evaluate-if    )
               ((lambda) evaluate-lambda)
               ((set!)   evaluate-set!  )
               (else     (evaluate-application operator))) operands)))
        (else
          expression)));
    (display output)
    (newline)
    (display ">>>")
    (loop (evaluate (read))))
  (loop "Slip version 0"))
```

A metacircular SLIP interpreter



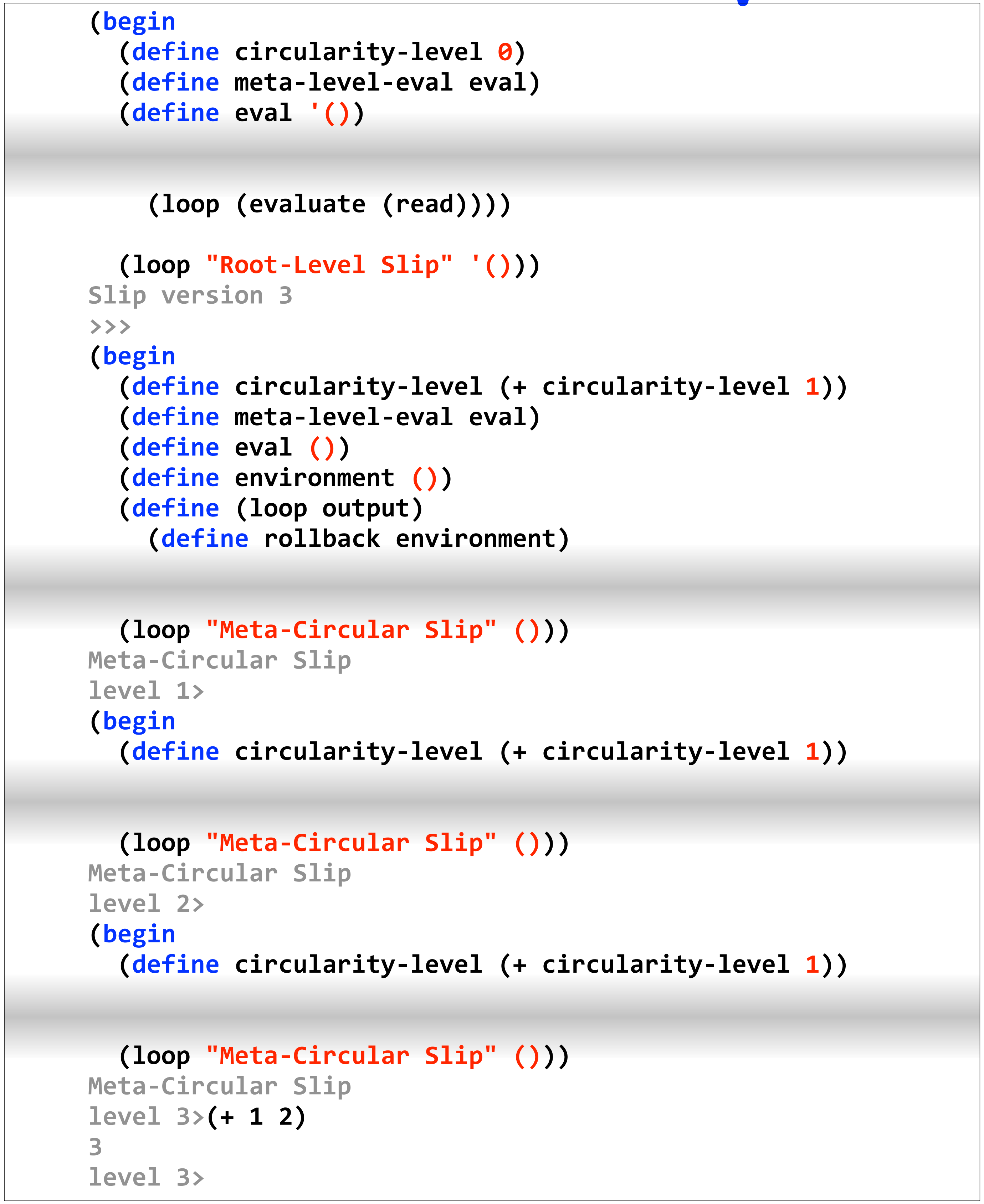
```
(begin
  (define circularity-lev
  (define meta-level-eval
  (define eval ())
  (define environment ())
  (define (loop output)
    (define rollback envi
    (define (evaluate exp
      (define (error mess
        (display message)
        (set! environment
        (loop qualifier))
      (define (bind-varia
        (define binding (
        (set! environment
      (define (bind-param
        (if (symbol? para
          (bind-variable
          (if (pair? para
            (begin
              (define var
              (define val
              (bind-varia
              (bind-param
      (define (evaluate-s
        (define head (car
        (define tail (cdr
        (if (null? tail)
          (evaluate head)
          (begin
            (evaluate hea
            (evaluate-seq
      (define (make-proce
        (define lexical-e
        (lambda arguments
          (define dynamic
          (set! environme
          (bind-parameter
          (define value (
          (set! environme
          value))
```

37

```
      (define (evaluate-application operator)
        (lambda operand
          (apply (evalu
      (define (evaluate
        (evaluate-seque
      (define (evaluate
        (if (symbol? pa
          (begin
            (define val
            (define bin
            (set! envir
            value)
          (begin
            (define bin
            (set! envir
            (define pro
            (set-cdr! b
            procedure))
      (define (evaluate
        (define boolean
        (if (eq? boolea
          (if (null? al
            ()
            (evaluate (
          (evaluate con
      (define (evaluate
        (make-procedure
      (define (evaluate
        expression)
      (define (evaluate
        (define value (
        (define binding
        (if (pair? bind
          (begin
            (define val
            (set-cdr! b
            value)
          (error "inaccessible variable:   variable)))
```

```
      (define (evaluate-variable variable)
        (define binding (assoc variable environment))
          (if (pair? binding)
            (cdr binding)
            (meta-level-eval variable)))
      (define (evaluate-while predicate . expressions)
        (define (iterate value)
          (define boolean (evaluate predicate))
          (if (eq? boolean #f)
            value
            (iterate (evaluate-sequence expressions))))
        (iterate ()))
     (if (symbol? expression)
        (evaluate-variable expression)
        (if (pair? expression)
          (begin
            (define operator (car expression))
            (define operands (cdr expression))
            (apply
              (if (eq? operator 'begin) evaluate-begin
                (if (eq? operator 'define) evaluate-define
                  (if (eq? operator 'if) evaluate-if
                    (if (eq? operator 'lambda) evaluate-lambda
                      (if (eq? operator 'quote) evaluate-quote
                        (if (eq? operator 'set!) evaluate-set!
                          (if (eq? operator 'while) evaluate-while
                            (evaluate-application operator))))))) operands))
        expression)))
    (display output)
    (newline)
    (display "level ")
    (display circularity-level)
    (display ">")
    (set! eval evaluate)
    (loop (evaluate (read))))
  (loop "Meta-Circular Slip"))
```

A metacircular SLIP interpreter (cont'd)



```
(begin
 (define circularity-level 0)
 (define meta-level-eval eval)
 (define eval '())


  (loop (evaluate (read))))

 (loop "Root-Level Slip" '()))
Slip version 3
>>>
(begin
 (define circularity-level (+ circularity-level 1))
 (define meta-level-eval eval)
 (define eval ())
 (define environment ())
 (define (loop output)
  (define rollback environment)


 (loop "Meta-Circular Slip" ()))
Meta-Circular Slip
level 1>
(begin
 (define circularity-level (+ circularity-level 1))


 (loop "Meta-Circular Slip" ()))
Meta-Circular Slip
level 2>
(begin
 (define circularity-level (+ circularity-level 1))


 (loop "Meta-Circular Slip" ()))
Meta-Circular Slip
level 3>(+ 1 2)
3
level 3>
```

A metacircular SLIP interpreter (cont'd)

```
(define environment ())
```

```
(define (evaluate-sequence expressions)
  (define head (car expressions))
  (define tail (cdr expressions))
  (if (null? tail)
    (evaluate head)
                                  e tail))))
```

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

```
                  ue value))
                  ing environment)))
                  eters arguments)

                  arguments)

      (begin
        (define variable (car parameters))
        (define value (car arguments ))
        (bind-variable variable value)
        (bind-parameters (cdr parameters)
                         (cdr arguments))))))
```

```
(define (evaluate-appli
  (lambda operands
    (apply (evaluate operator) (map evaluate operands))))
```

A metacircular SLIP interpreter (cont'd)

```
(define environment ())
```

```
(define (evaluate-sequence expressions)
  (define head (car expressions))
  (define tail (cdr expressions))
  (if (null? tail)
    (evaluate head)
    (begin
      (evaluate head)
      (evaluate-sequence tail))))
```

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

```
(define (bind-variable variable value)
  (define binding (cons variable value))
  (set! environment (cons binding environment)))
(define (bind-parameters parameters arguments)
  (if (symbol? parameters)
    (bind-variable parameters arguments)
    (if (pair? parameters)
      (begin
        (define variable (car parameters))
        (define value (car arguments ))
        (bind-variable variable value)
        (bind-parameters (cdr parameters)
                         (cdr arguments))))))
```

```
(define (evaluate-application operator)
  (lambda operands
    (apply (evaluate operator) (map evaluate operands))))
```

A metacircular SLIP interpreter (cont'd)

```
(define environment ())
```

```
(define (evaluate-sequence expressions)
  (define head (car expressions))
  (define tail (cdr expressions))
  (if (null? tail)
    (evaluate head)
                    e tail))))
```

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

```
e value)
le value))
ing environment)))
eters arguments)

arguments)

(begin
  (define variable (car parameters))
  (define value (car arguments ))
  (bind-variable variable value)
  (bind-parameters (cdr parameters)
                   (cdr arguments))))))
```

```
(define (evaluate-application operator)
  (lambda operands
    (apply (evaluate operator) (map evaluate operands))))
```

A metacircular SLIP interpreter (cont'd)

```
(define environment ())
```

```
(define (evaluate-sequence expressions)
  (define head (car expressions))
  (define tail (cdr expressions))
  (if (null? tail)
    (evaluate head)
    (begin
      (evaluate head)
      (evaluate-sequence tail))))
```

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

```
(define (bind-variable variable value)
  (define binding (cons variable value))
  (set! environment (cons binding environment)))
(define (bind-parameters parameters arguments)
  (if (symbol? parameters)
    (bind-variable parameters arguments)
    (if (pair? parameters)
      (begin
        (define variable (car parameters))
        (define value (car arguments ))
        (bind-variable variable value)
        (bind-parameters (cdr parameters)
                                 ents))))))
```

```
(define (evaluate-application operator)
  (lambda operands
    (apply (evaluate operator) (map evaluate operands))))
```

A metacircular SLIP interpreter (cont'd)

```
(define environment ())
```

```
(define (evaluate-sequence expressions)
  (define head (car expressions))
  (define tail (cdr expressions))
  (if (null? tail)
    (evaluate head)
    (begin
      (evaluate head)
```

```
(define (make-procedure parameters expressions)
  (define lexical-environment environment)
  (lambda arguments
    (define dynamic-
    (set! environmen
    (bind-parameters
    (define value (e
    (set! environmen
    value))
```

```
(define (bind-variable variable value)
  (define binding (cons variable value))
  (set! environment (cons binding environment)))
(define (bind-parameters parameters arguments)
  (if (symbol? parameters)
    (bind-variable parameters arguments)
    (if (pair? parameters)
      (begin
        (define variable (car parameters))
        (define value (car arguments ))
        (bind-variable variable value)
        (bind-parameters (cdr parameters)
                         (cdr arguments))))))
```

```
(define (evaluate-appl
  (lambda operands
    (apply (evaluate operator) (map evaluate operands))))
```

A metacircular SLIP interpreter (cont'd)

```
(define environment ())
```

```
(define (make-procedure para
  (define lexical-environme
  (lambda arguments
    (define dynamic-environment environment)
    (set! environment lexical-environment)
    (bind-parameters parameters arguments)
    (define value (evaluate-sequence expressions))
    (set! environment dynamic-environment)
    value))
```

```
(define (evaluate-sequence expressions)
  (define head (car expressions))
  (define tail (cdr expressions))
  (if (null? tail)
    (evaluate head)
    (begin
      (evaluate head)
      (evaluate-sequence tail))))
```

```
(define
  (define binding (cons variable value))
  (set! environment (cons binding environment)))
(define (bind-parameters parameters arguments)
  (if (symbol? parameters)
    (bind-variable parameters arguments)
    (if (pair? parameters)
      (begin
        (define variable (car parameters))
        (define value (car arguments ))
        (bind-variable variable value)
        (bind-parameters (cdr parameters)
                         (cdr arguments))))))
```

```
(define (evaluate-application operator)
  (lambda operands
    (apply (evaluate operator) (map evaluate operands))))
```

SLIP in cps

```
(define (evaluate expression continue)
  (cond
    ((symbol? expression)
     (evaluate-variable expression continue))
    ((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)))
    (else
      (continue expression))))
```

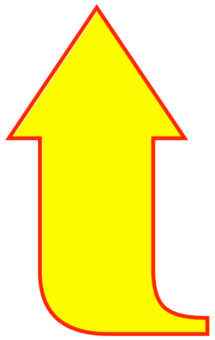
SLIP in cps

```
(define (evaluate expression continue)
  (cond
    ((symbol? expression)
     (evaluate-variable expression continue))
    ((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)))
    (else
      (continue expression))))
```

```
(define environment '())

(define (loop output)
  (define rollback environment)

  (define (error message qualifier)
    (set! environment rollback)
    (display message)
    (loop qualifier))

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

(loop "Meta-Circular Slip"))
```

SLIP in cps (cont'd)

```
(define (evaluate-set! variable expression)
  (lambda (continue)
    (define (continuation value)
      (define binding (assoc variable environment))
      (set-cdr! binding value)
      (continue value))
    (evaluate expression continuation)))
```



```
(let ((operator (car expression))
      (operands (cdr expression)))
```

```
(define (evaluate expression continue)
     ...
  ((apply evaluate-set! operands) continue)
     ... )
```

SLIP in cps (cont'd)

```
(define (evaluate-set! variable expression)
  (lambda (continue)
    (define (continuation value)
      (define binding (assoc variable environment))
      (set-cdr! binding value)
      (continue value))
    (evaluate expression continuation)))
```

currying

```
(let ((operator (car expression))
      (operands (cdr expression)))
```

```
(define (evaluate expression continue)
     ...
   ((apply evaluate-set! operands) continue)
     ... )
```

SLIP in cps (cont'd)

```
(define (evaluate-set! variable expression)
  (lambda (continue)
    (define (continuation value)
      (define binding (assoc variable environment))
      (set-cdr! binding value)
      (continue value))
    (evaluate expression continuation)))
```

continuation

```
(let ((operator (car expression))
      (operands (cdr expression)))
```

```
(define (evaluate expression continue)
     ...
  ((apply evaluate-set! operands) continue)
     ... )
```

SLIP in cps (cont'd)

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

price to pay ...

SLIP in cps (cont'd)

```
(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)))
```

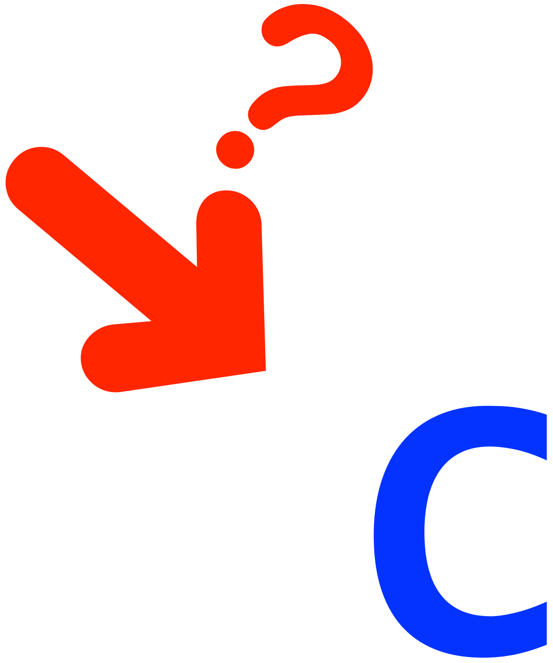
SLIP in cps (cont'd)

```
(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)))
```

SLIP in C: 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)))
```

```
(fibonacci 15 display)
```



C

SLIP in C: continuations (cont'd)

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

SLIP in C: continuations (cont'd)

```
(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))
```

```
          (continuatio
          (nested-clos
      (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 '()))
```

SLIP in C: continuations (cont'd)

```
(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))
```

```
          (continuatio
          (nested-clos
      (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 '()))
```

SLIP in C: continuations (cont'd)

```
(begin
  (define (factorial n)
    (if (> n 1)
      (
      1
  (fact
```

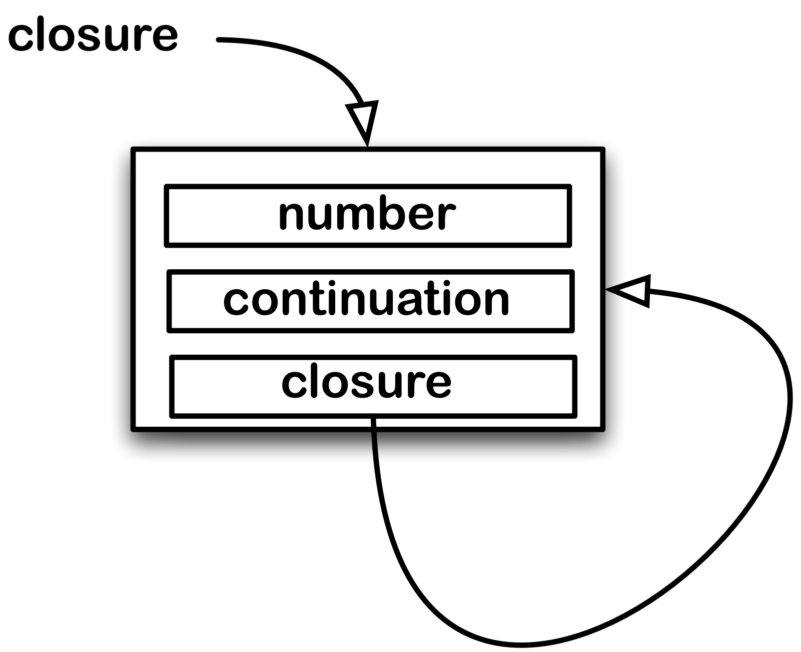
```
begin
 (define (factorial n continue)
   (define (continuation p)
                                          continuation)
```

```
(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 '()))
```

requires 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 (cont'd)

```
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 (cont'd)

```
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 (cont'd)

```
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; }
```

Incremental SLIP implementation in C

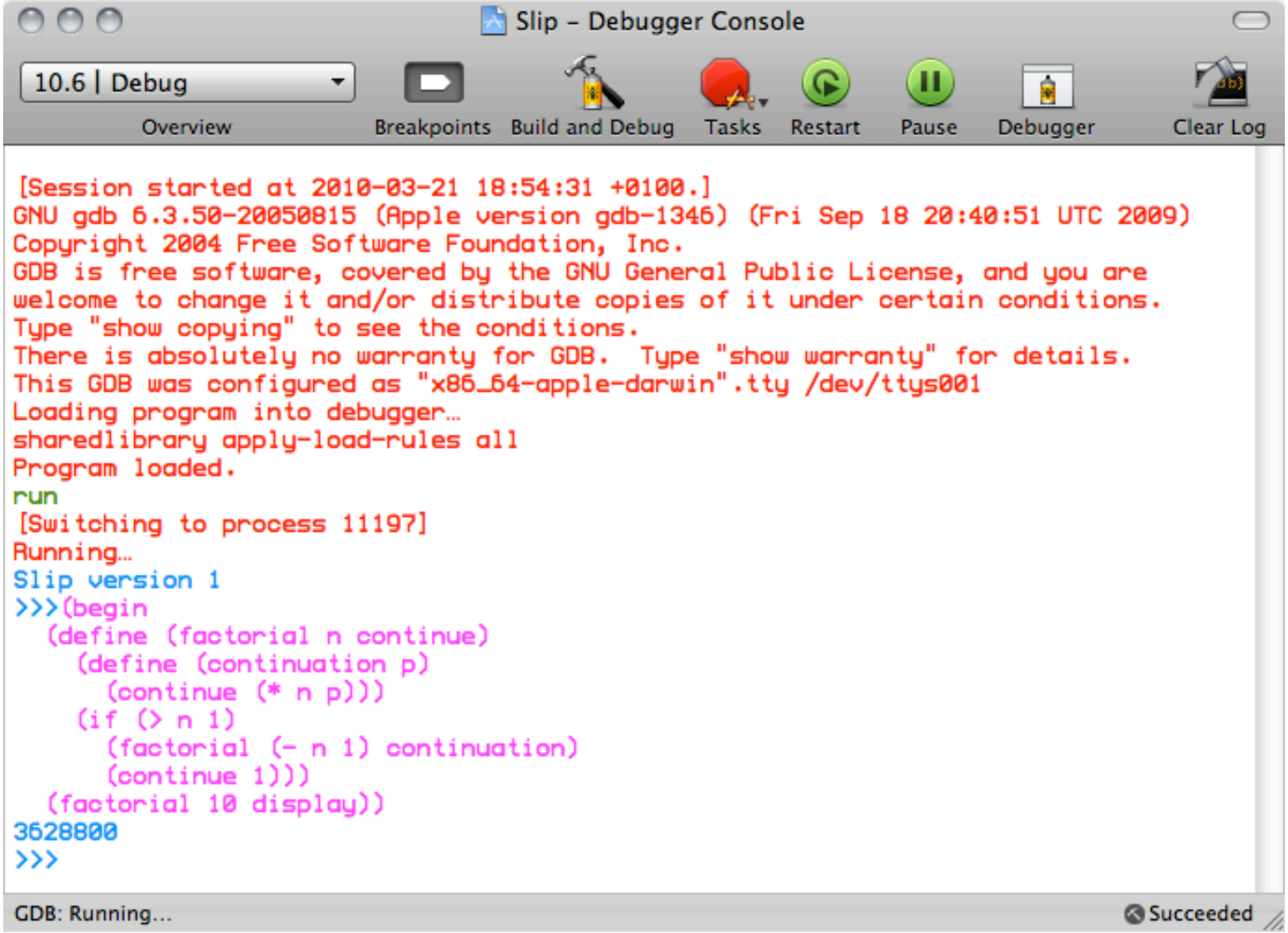
version 1:	straightforward code
version 2:	using a trampoline
version 3:	factored out environment
version 4:	threaded continuations
version 5:	functional continuations
version 6:	partial evaluation
version 7:	iterative constructs
version 8:	lexical addressing
version 9:	garbage collection
version 10:	proper tail recursion
version 11:	1st class continuations
version 12:	smart caching
version 13:	multicores

SLIP/C client interface

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

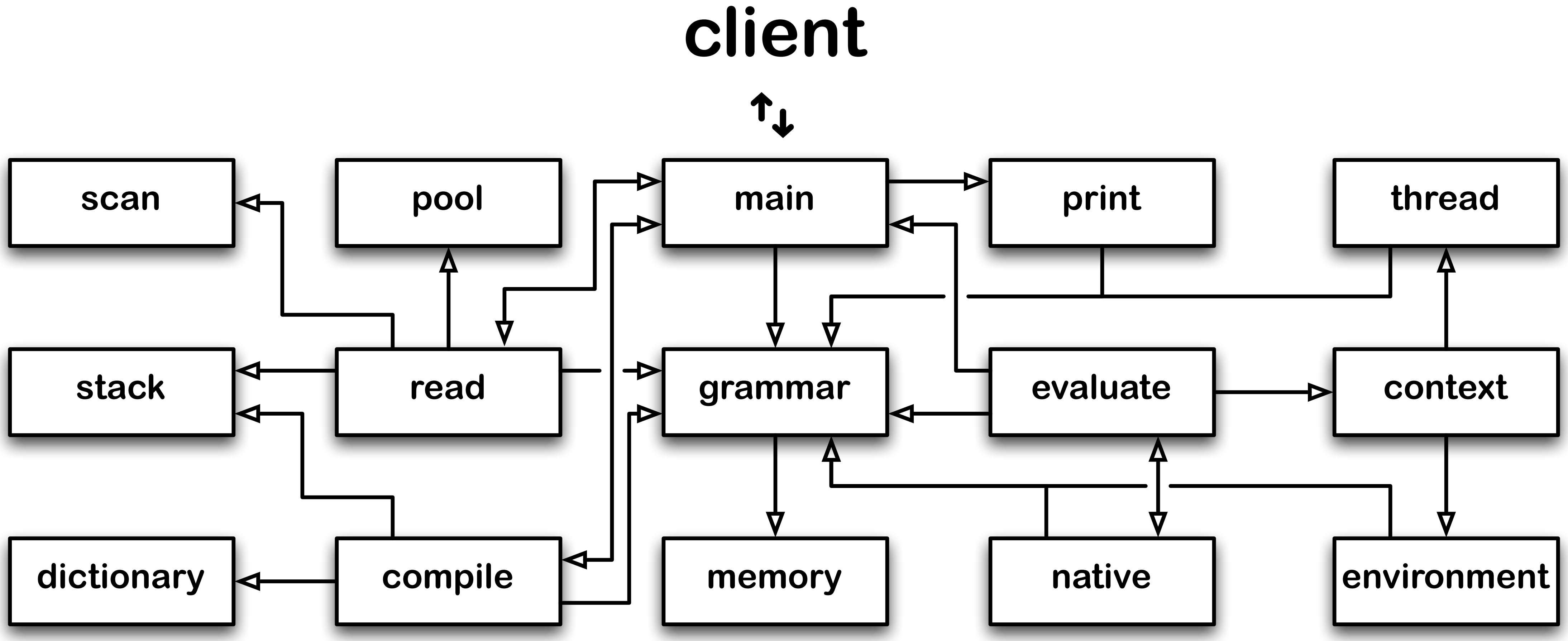
```
/*------------------------------ imported functions --------------------------*/

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

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

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

SLIP/C example

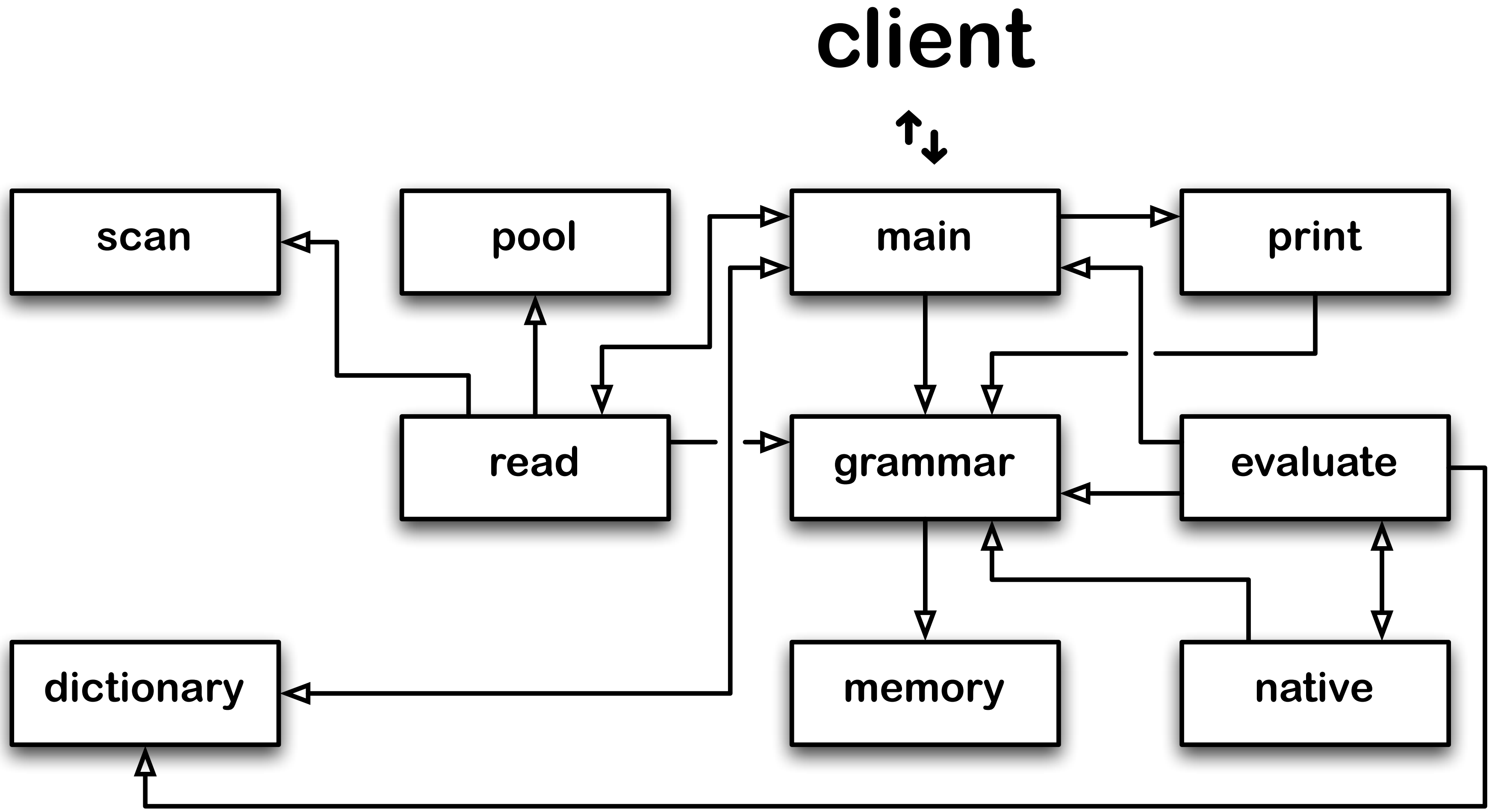


```
[Session started at 2010-03-21 18:54:31 +0100.]
GNU gdb 6.3.50-20050815 (Apple version gdb-1346) (Fri Sep 18 20:40:51 UTC 2009)
Copyright 2004 Free Software Foundation, Inc.
GDB is free software, covered by the GNU General Public License, and you are
welcome to change it and/or distribute copies 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
>>>
```

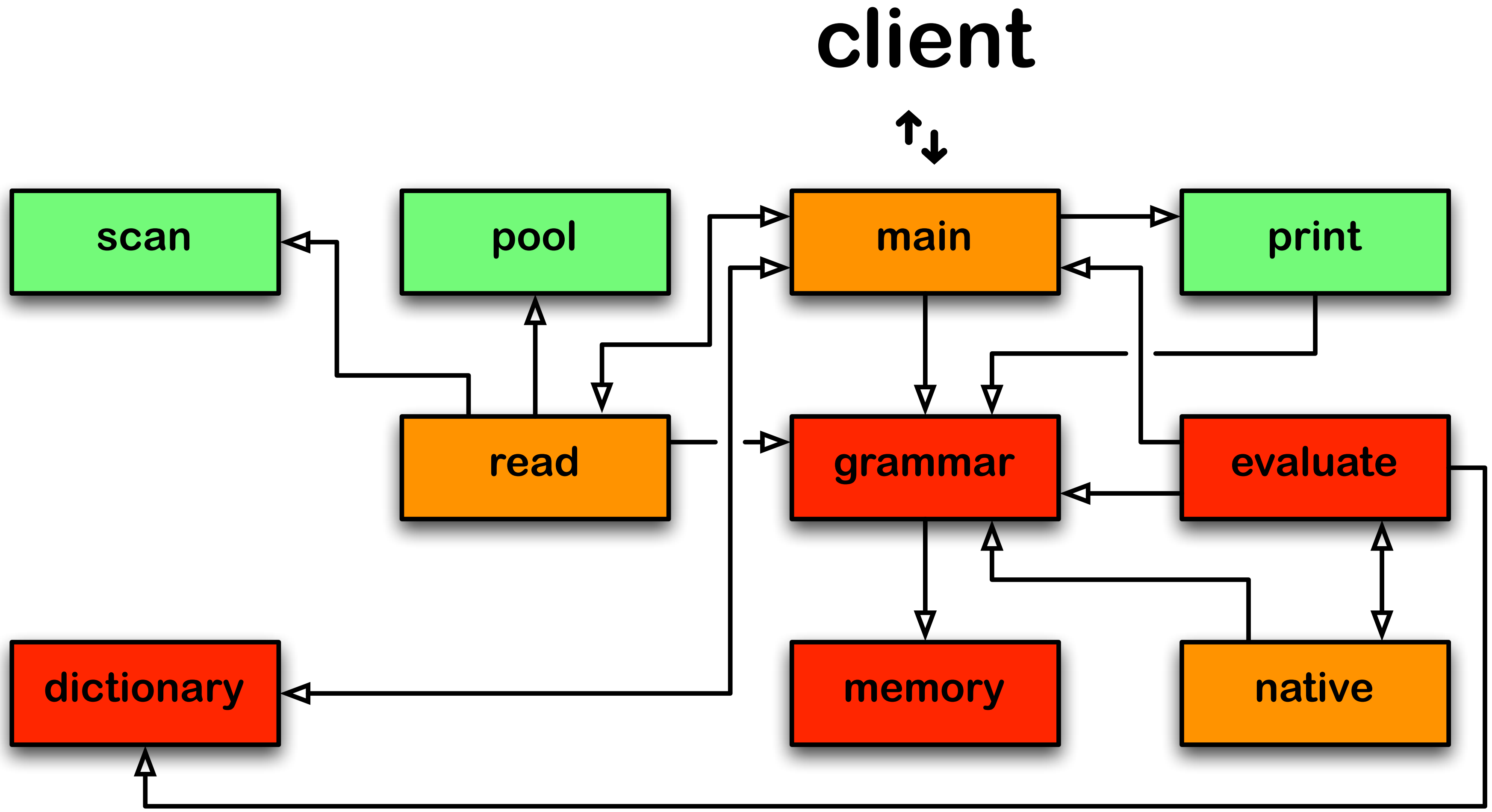
SLIP/C ultimate implementation



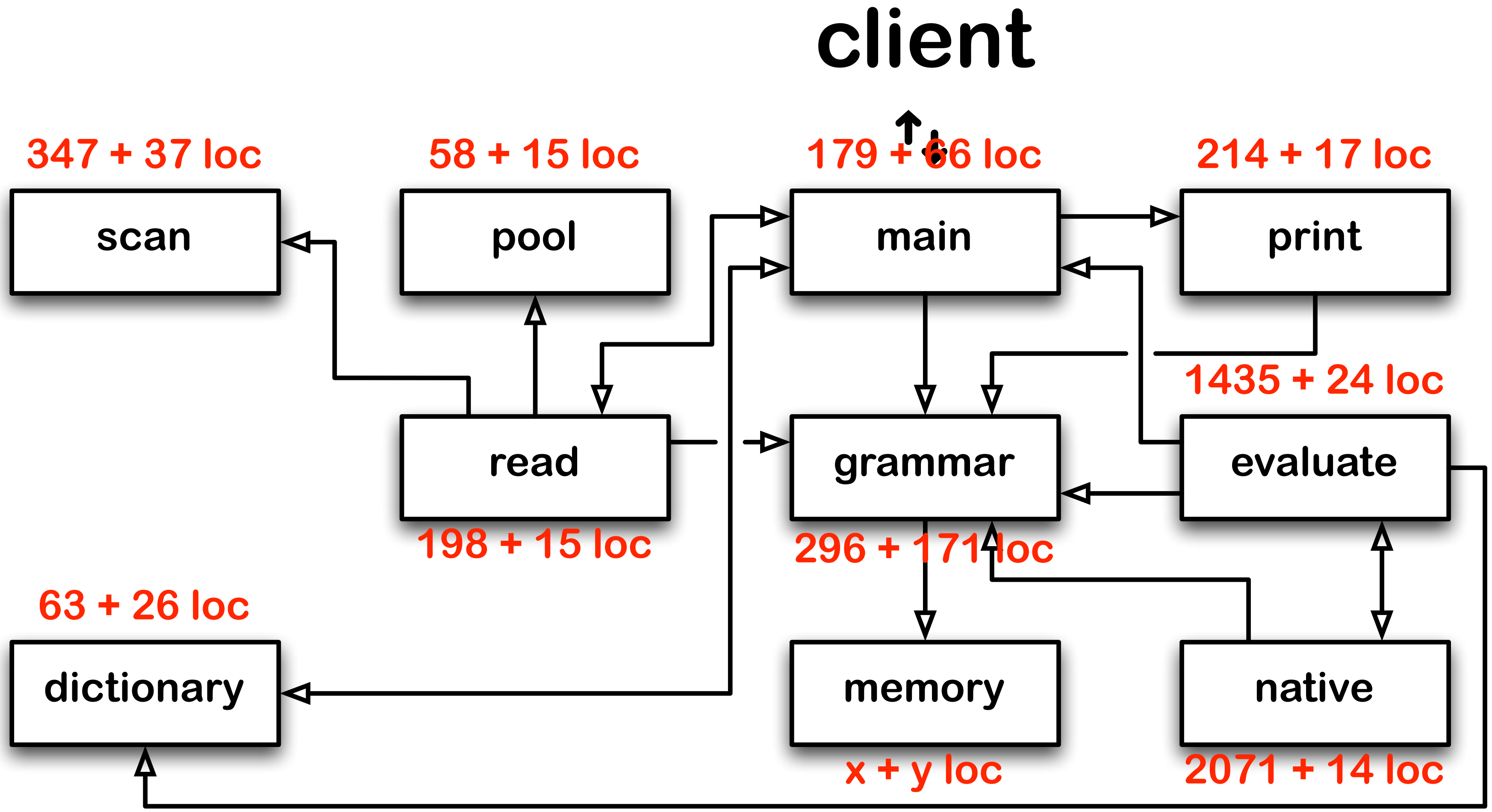
SLIP/C initial implementation



SLIP/C initial implementation (cont'd)

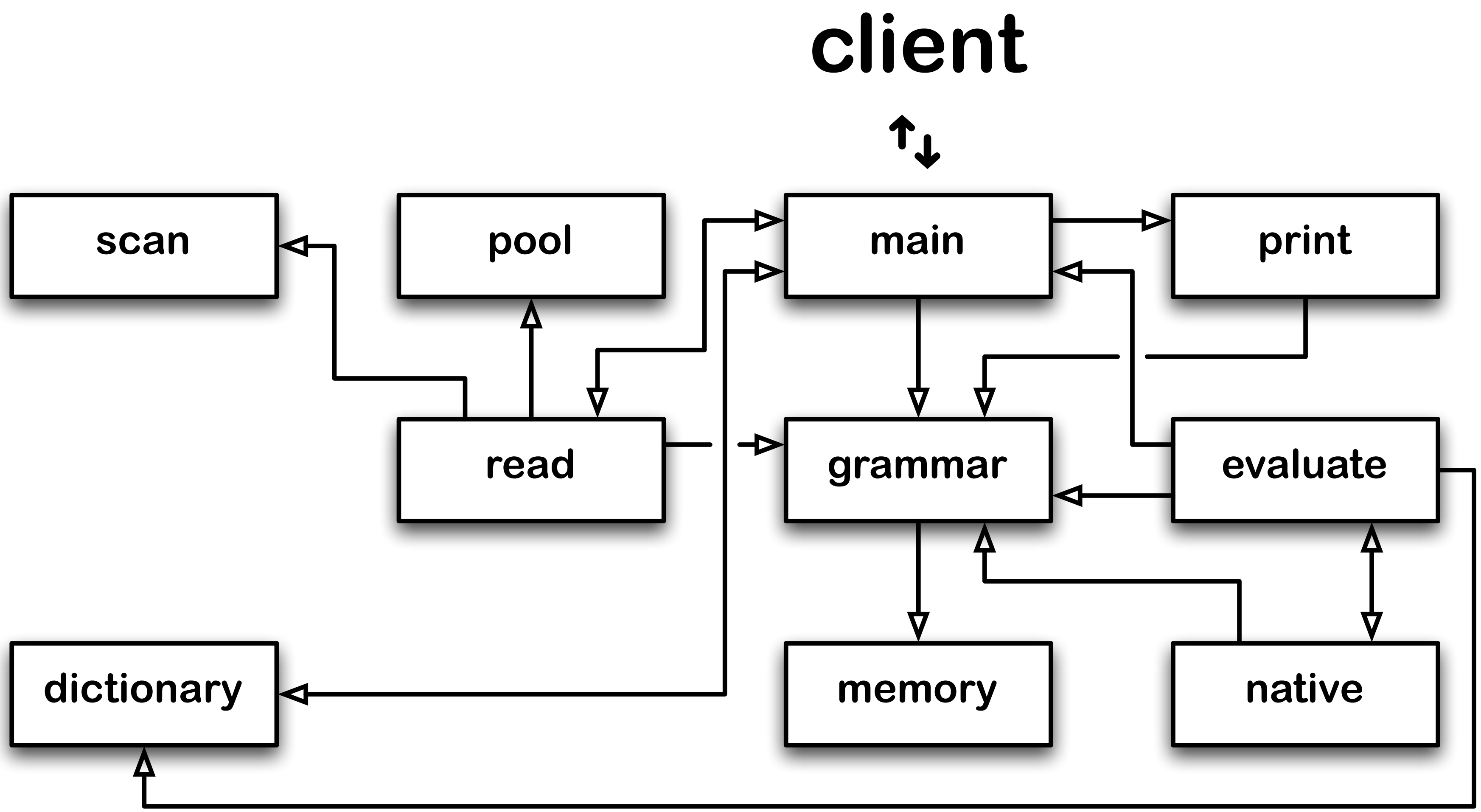


SLIP/C initial implementation (cont'd)



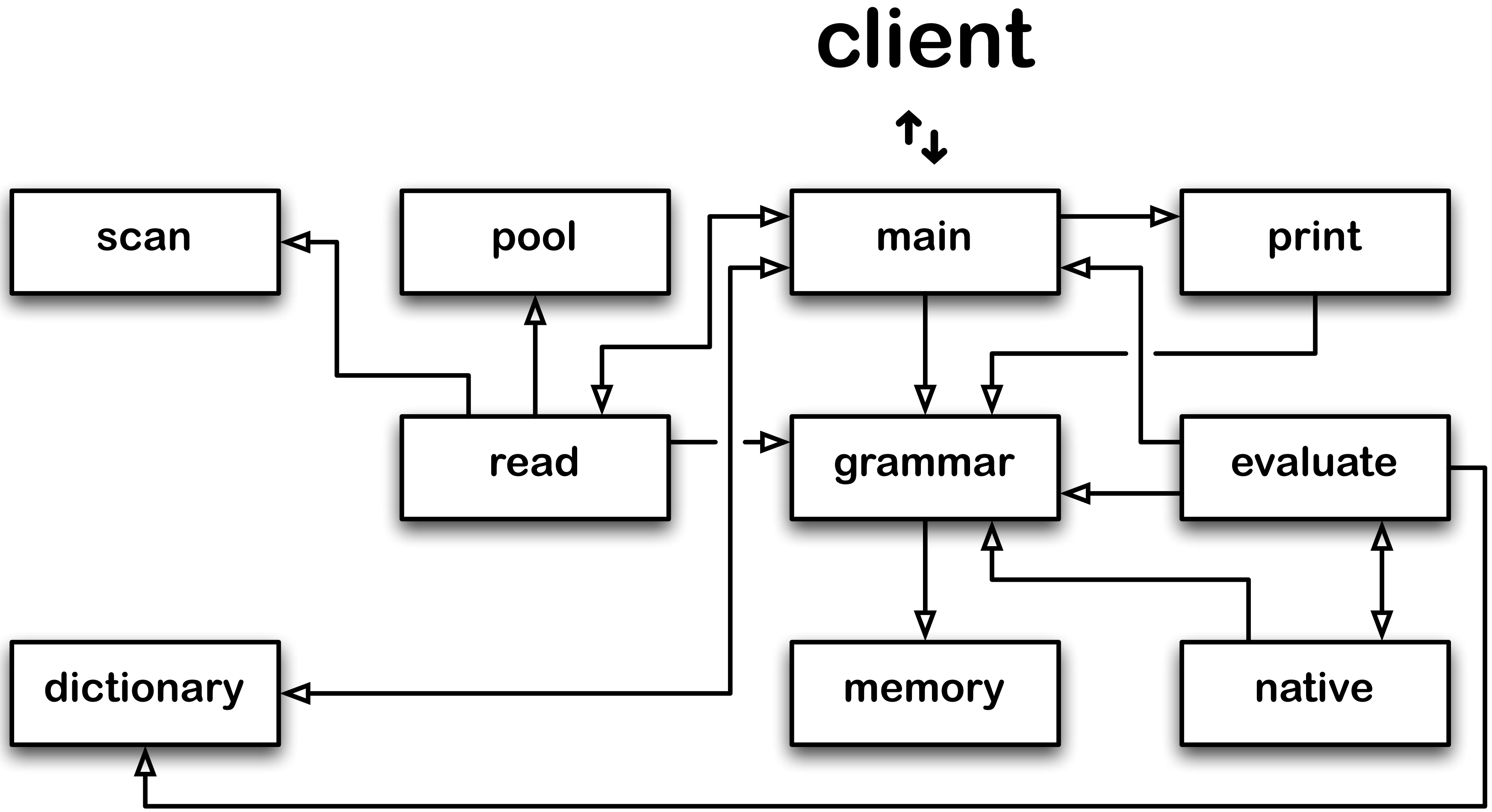
5347 loc

SLIP/C first stage



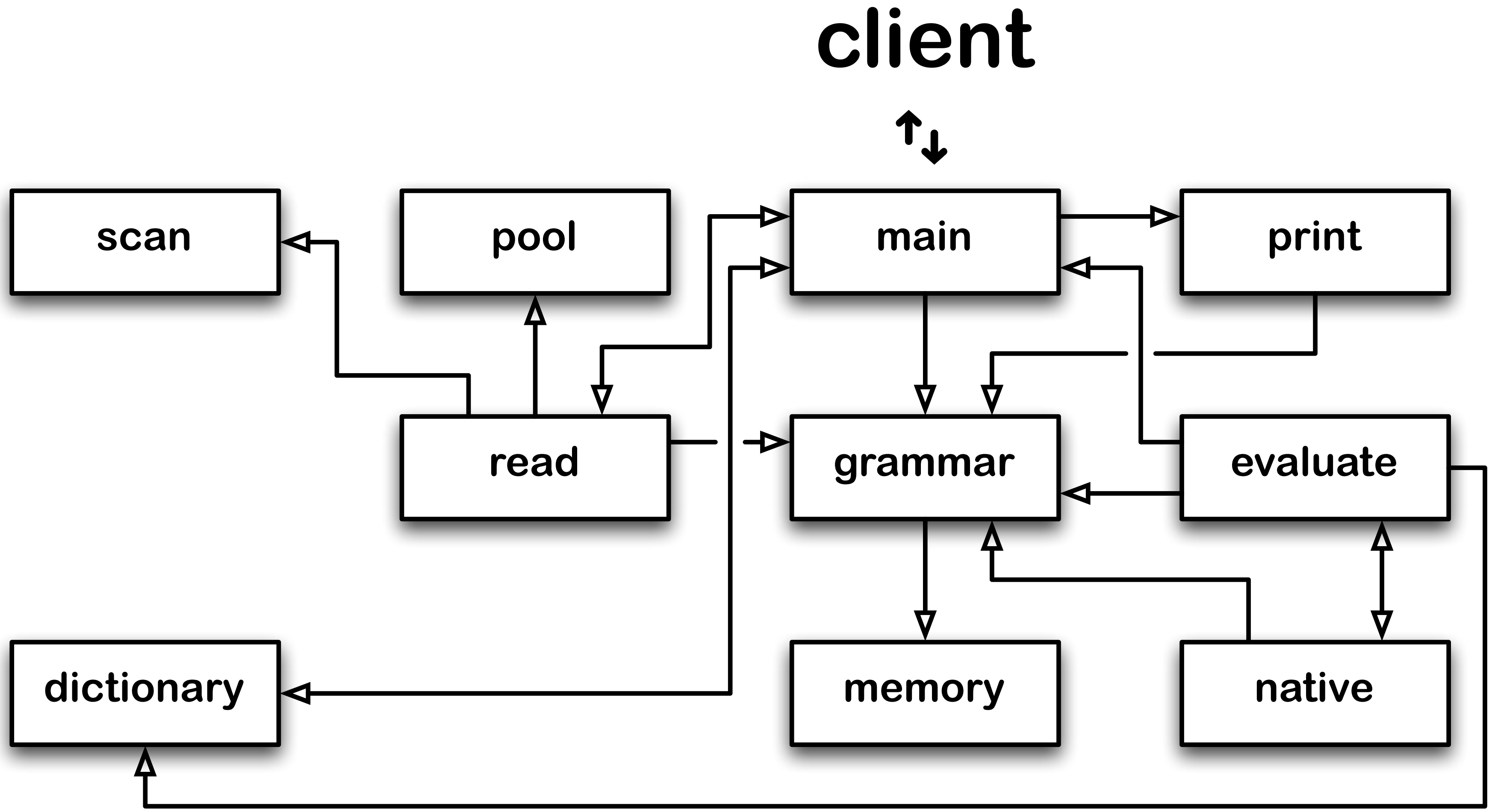
version 1: straightforward cps implementation

SLIP/C first stage (cont'd)



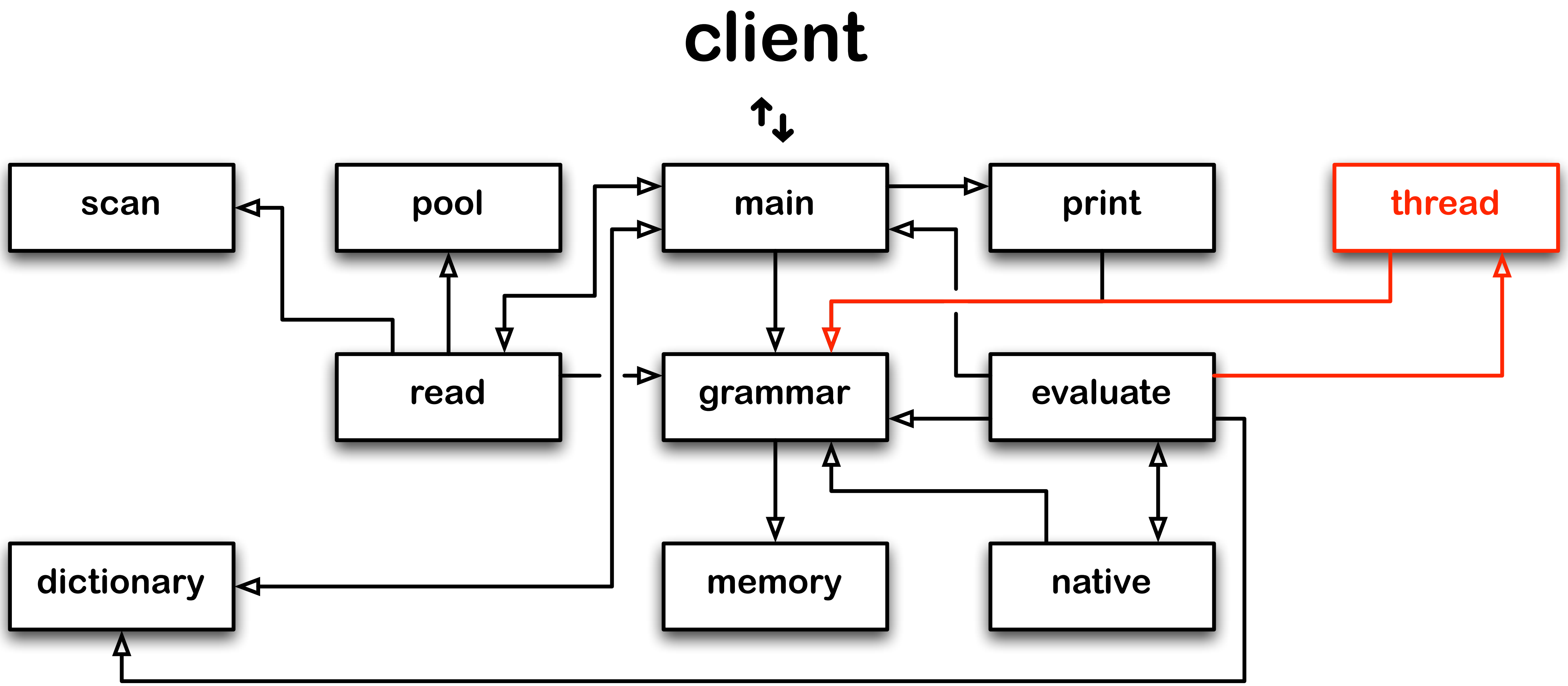
version 2: introducing a trampoline

SLIP/C first stage (cont'd)



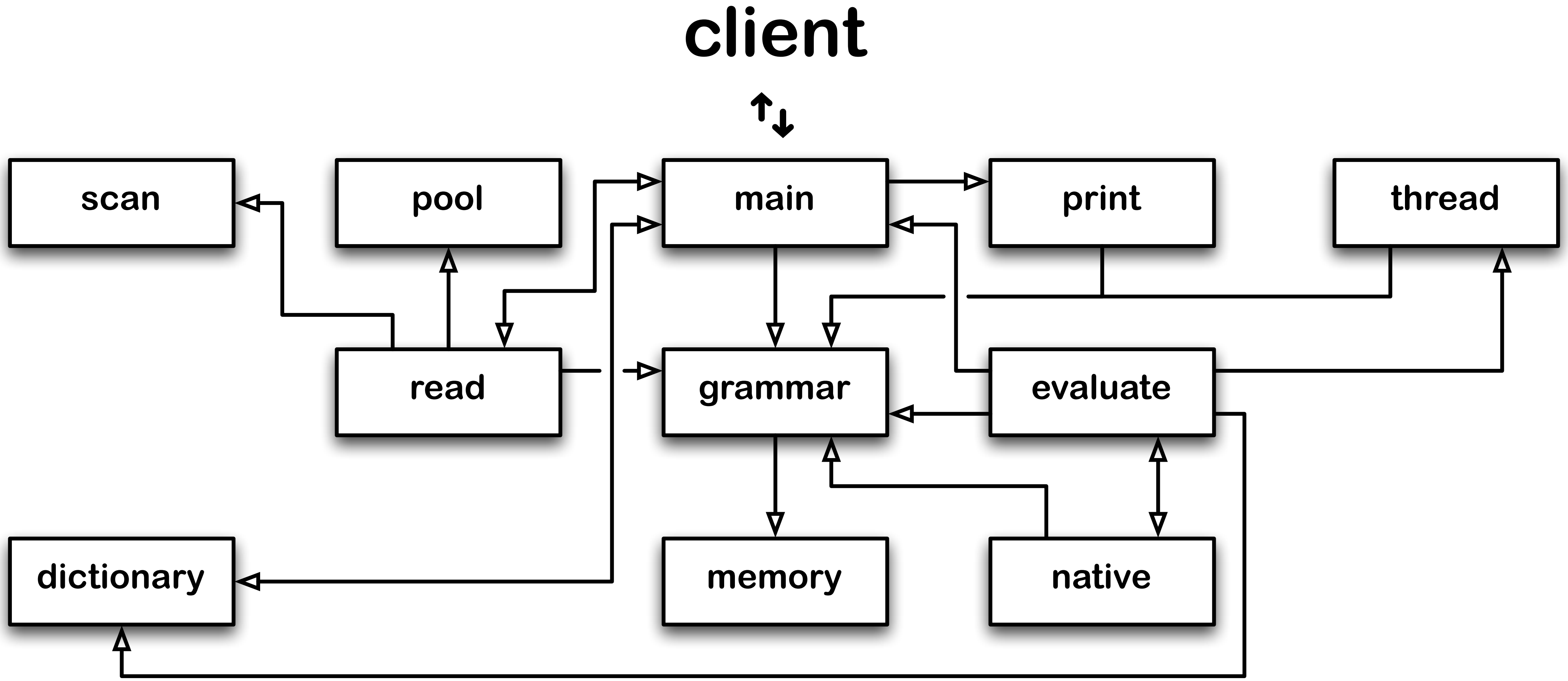
version 3: factored out environment

SLIP/C second stage



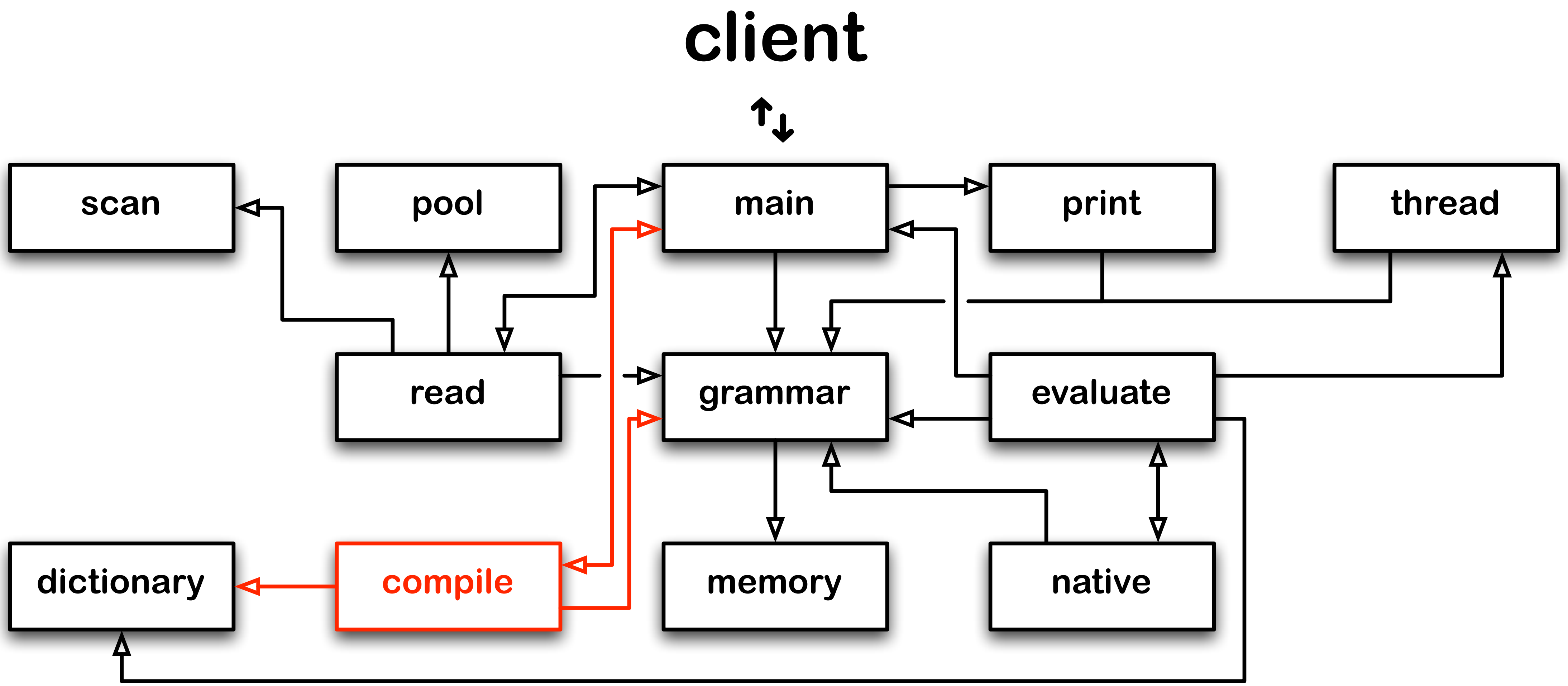
version 4: threaded continuations

SLIP/C second stage (cont'd)



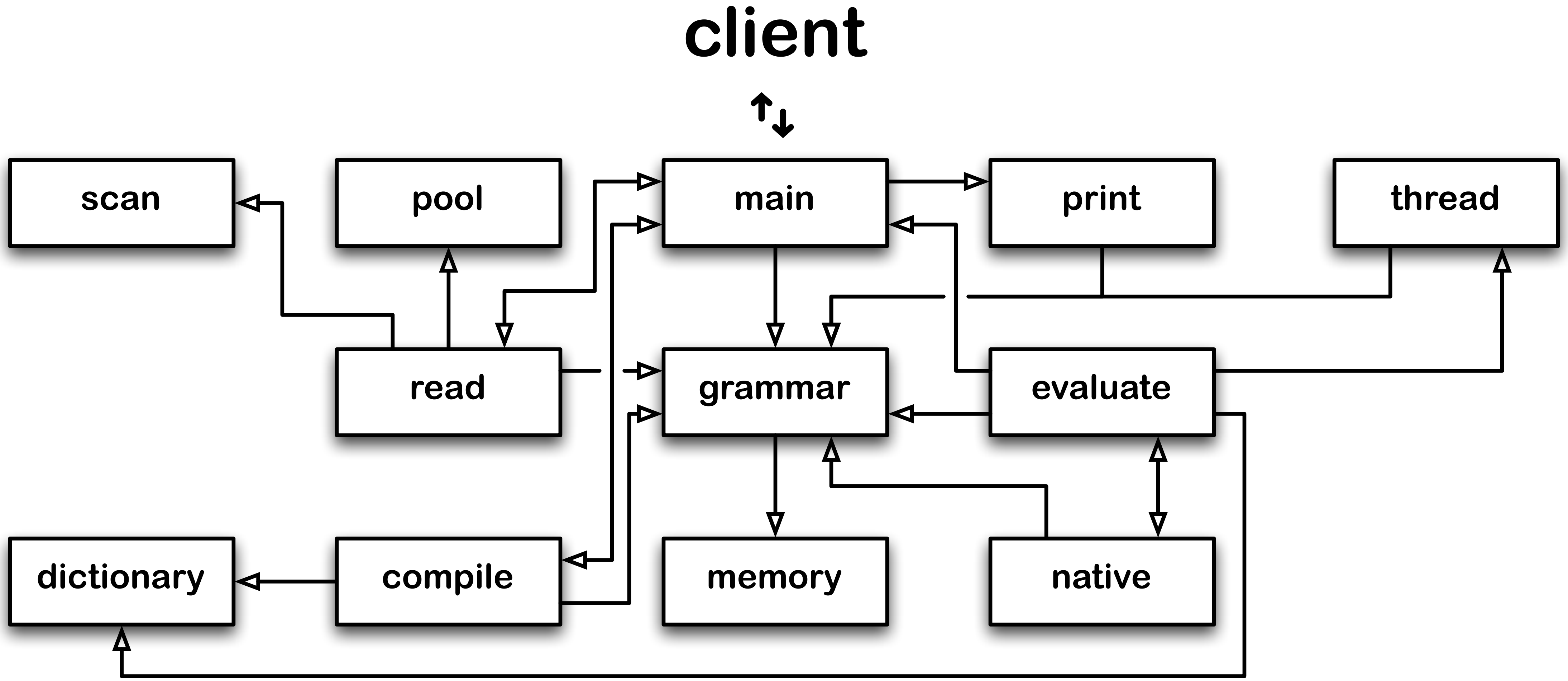
version 5: functional continuations

SLIP/C third stage



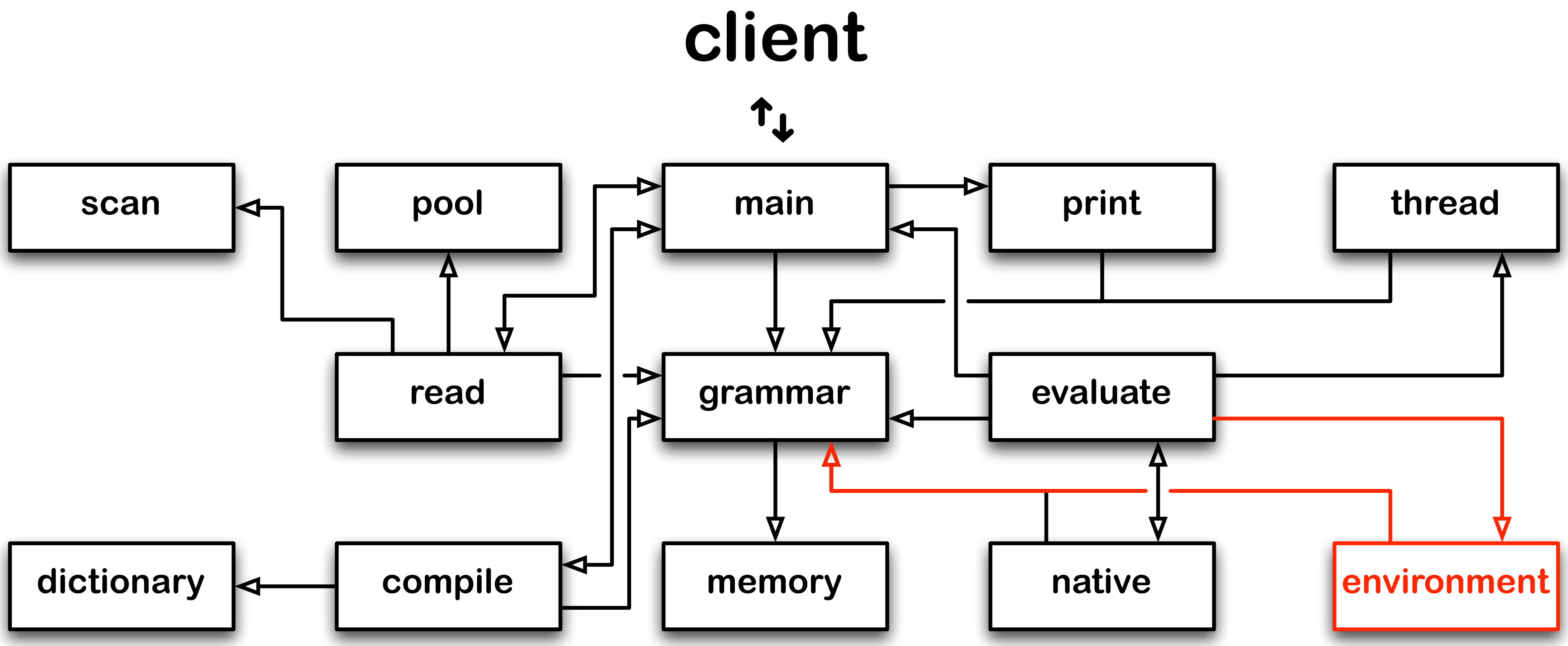
version 6: partial evaluation

SLIP/C third stage (cont'd)



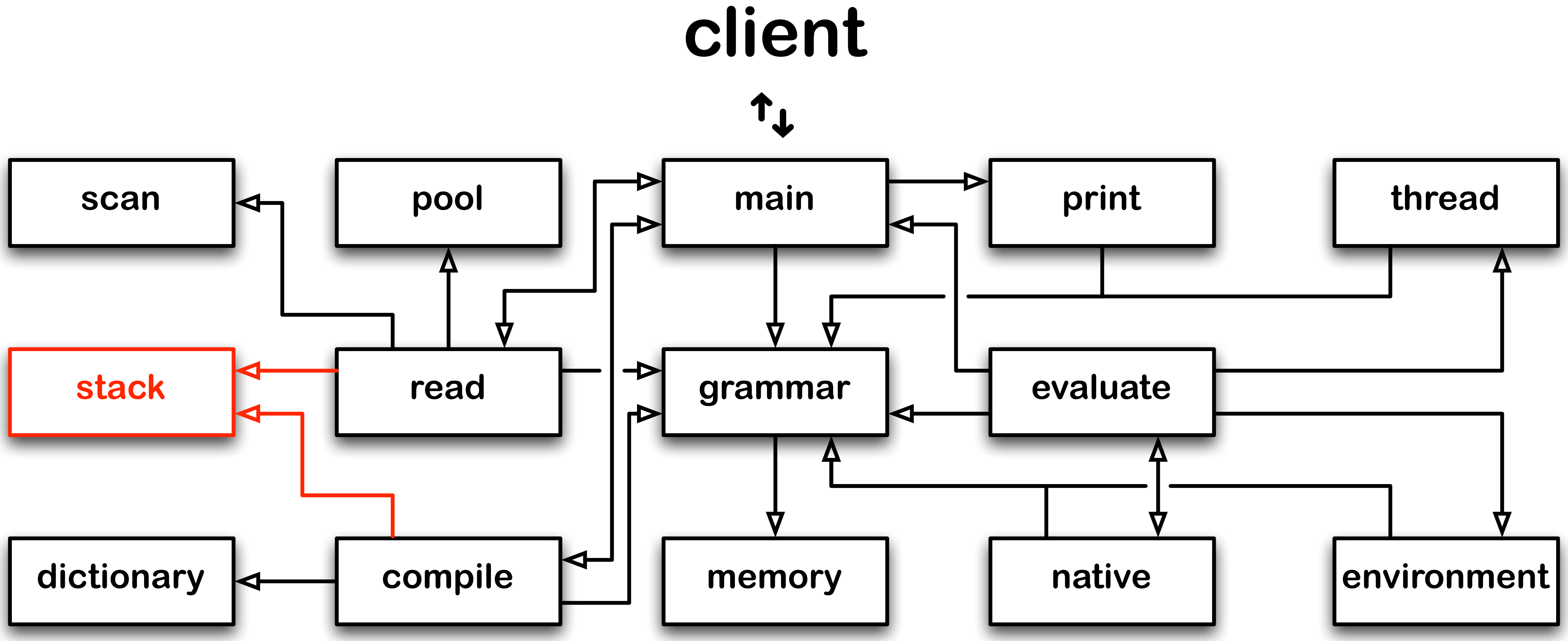
version 7: iterative constructs

SLIP/C fourth stage



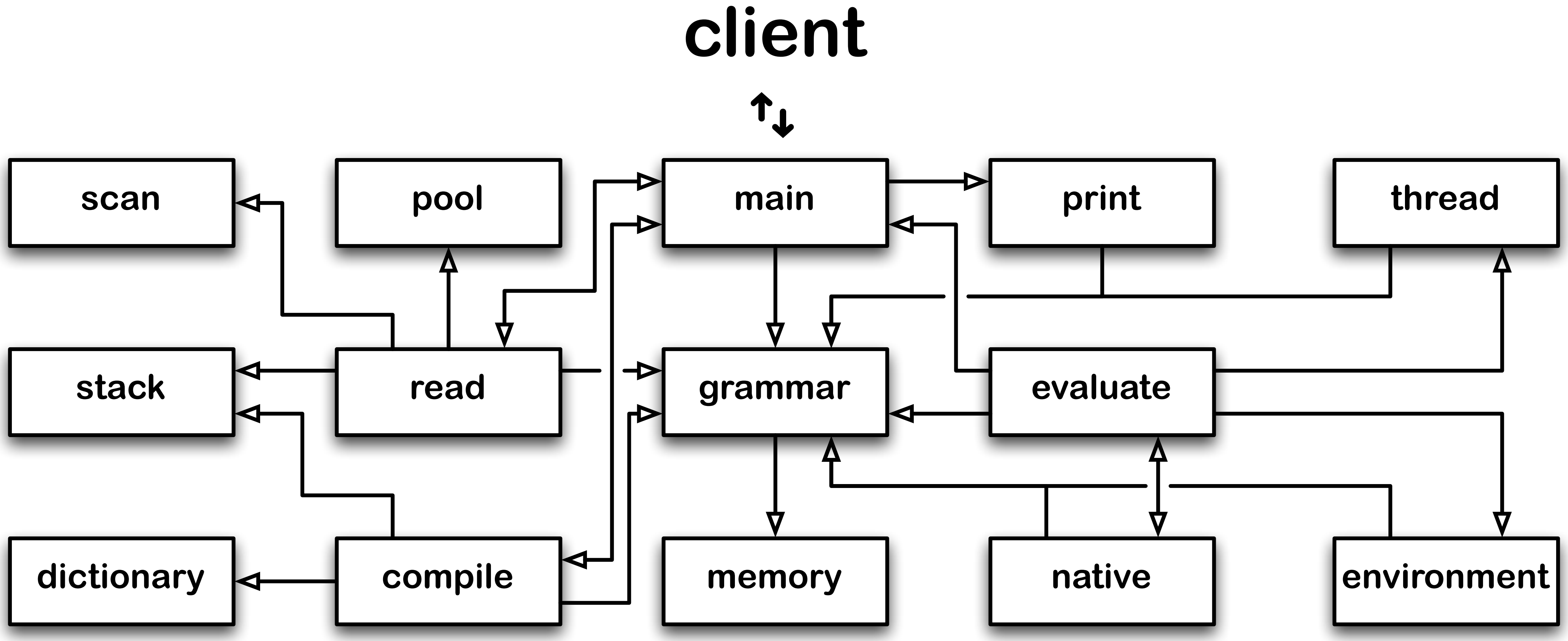
version 8: lexical addressing

SLIP/C fifth stage



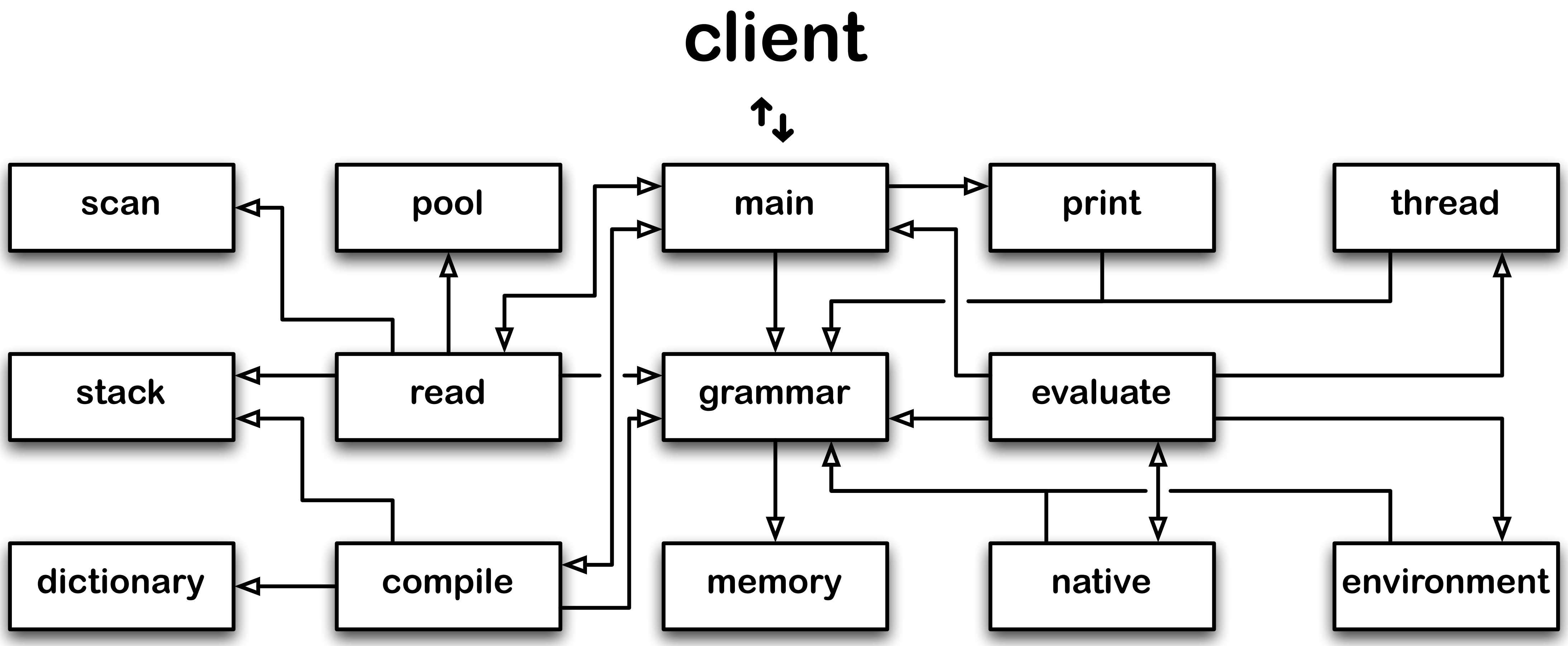
version 9: garbage collection

SLIP/C fifth stage (cont'd)



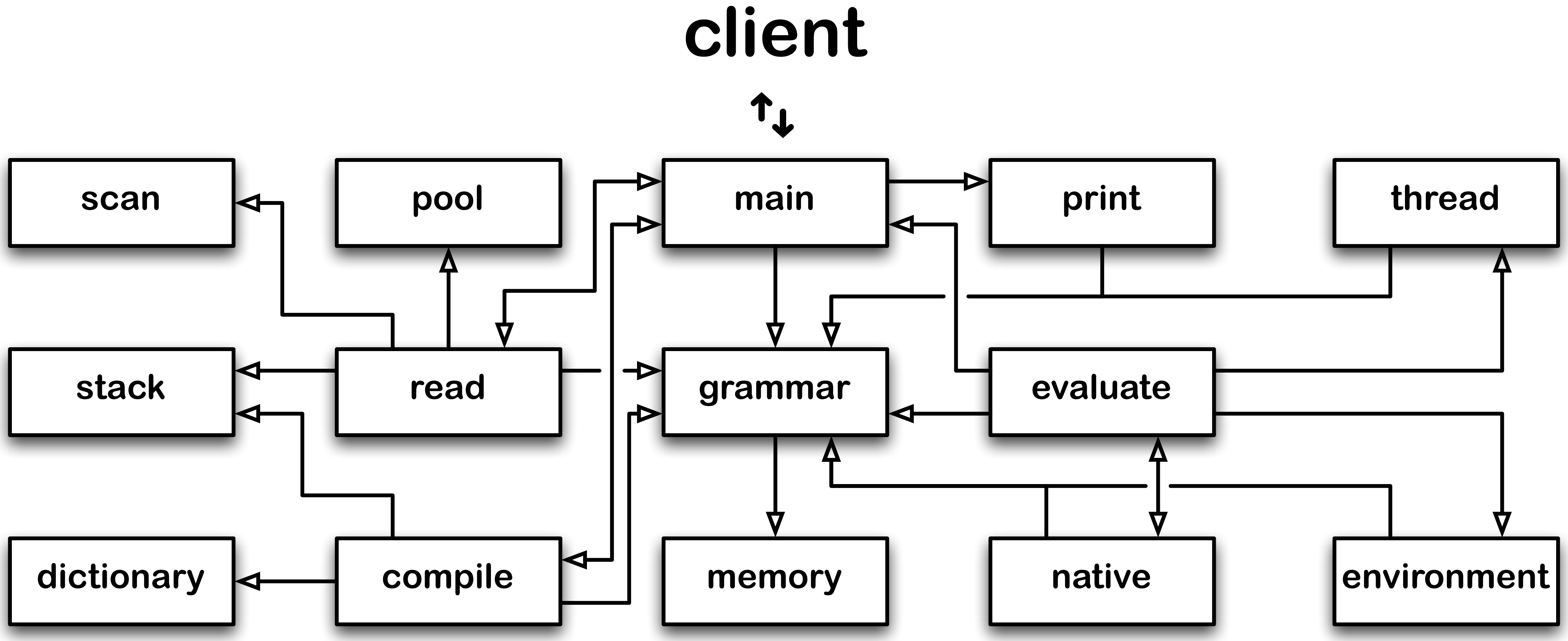
version 10: proper tail recursion

SLIP/C fifth stage (cont'd)



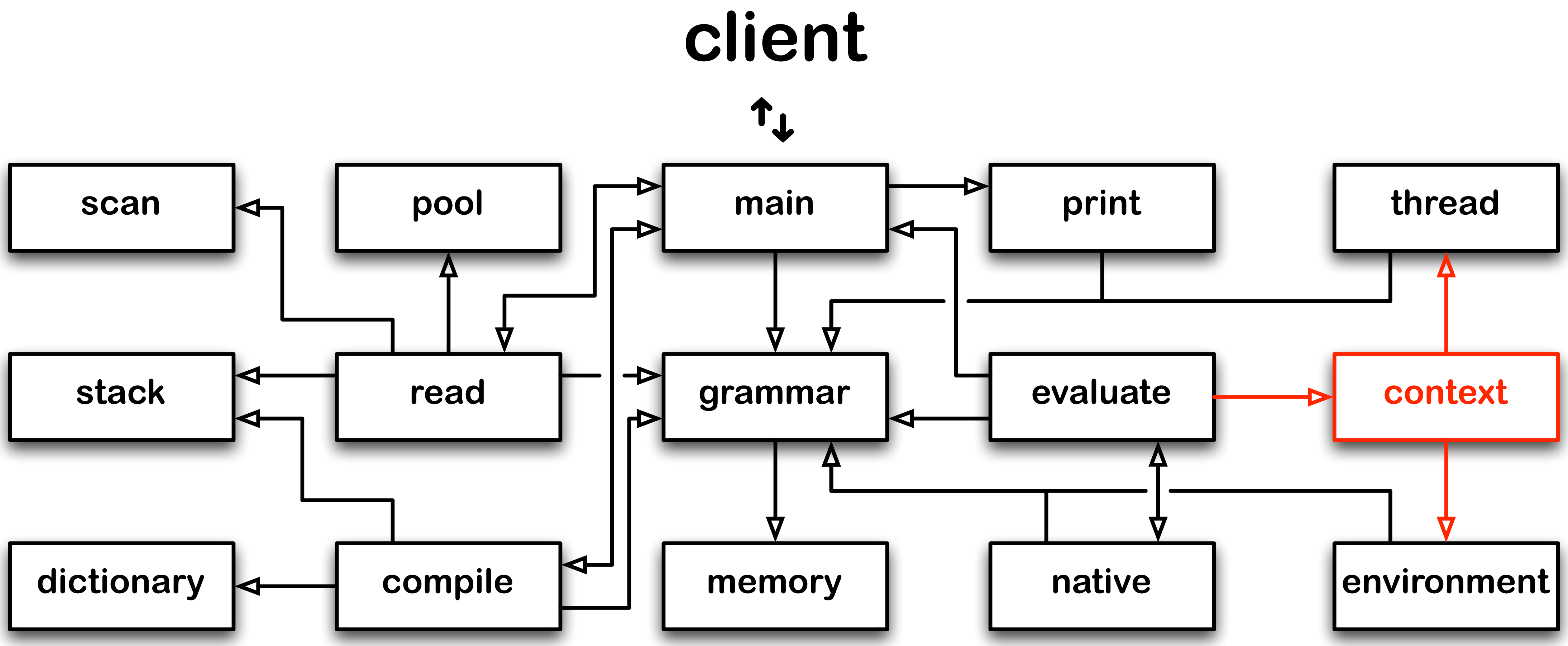
version 11: first class continuations

SLIP/C fifth stage (cont'd)



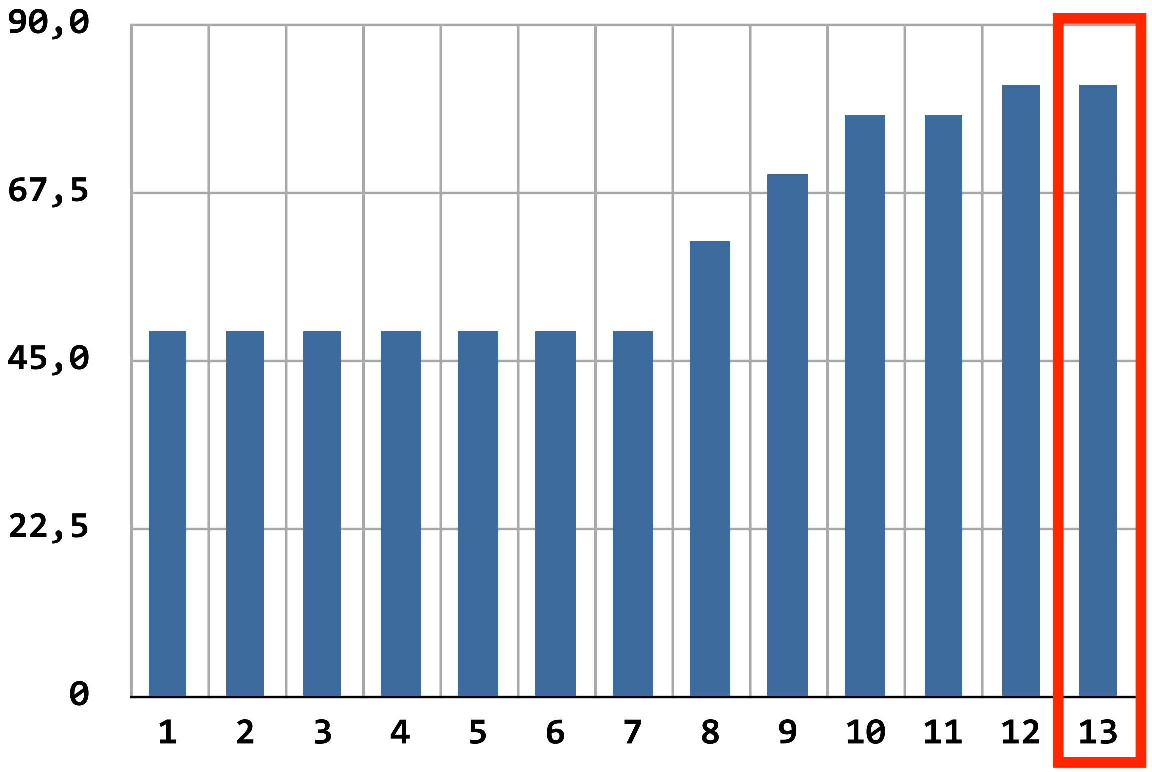
version 12: smart caches

SLIP/C new stage

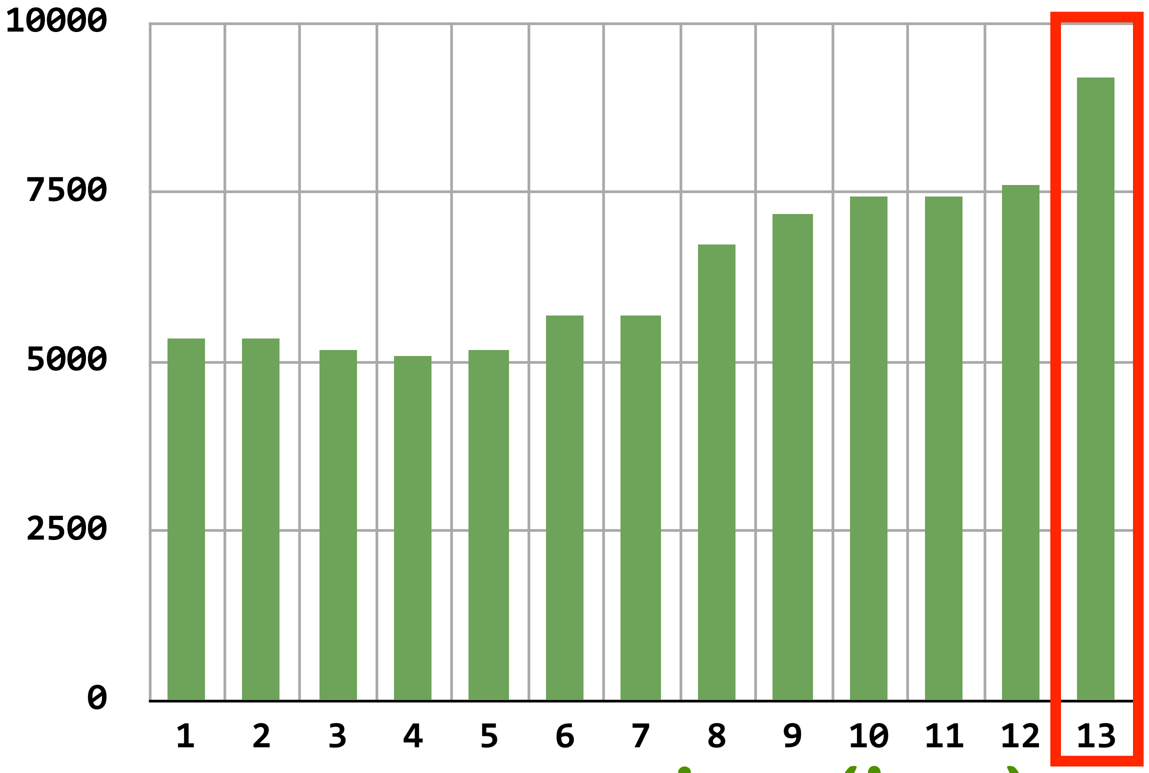


version 13: multicore support

SLIP/C implementation size



code size (kb)



source size (loc)

Multicore memory management

```
BYT_type Memory_Claim(UNS_type Claim)
  { BYT_type overflow;
    Slip_Spin_Lock(Memory_lock);
    Claim_size += Claim + 1;
    Claim_counter++;
    overflow = (Tail_pointer - Free_pointer <= Claim_size);
    if (overflow)
      { collect();
        if (Tail_pointer - Free_pointer <= Claim_size)
          Memory_Fail(); }
    Slip_Spin_Unlock(Memory_lock);
    return overflow; }
```

```
          emory_Make_Chunk(BYT_type Tag,
                           UNS_type Size)
  { PTR_type pointer;
    UNS_type size;
    size = Size + 1;
    Slip_Spin_Lock(Memory_lock);
    if (size > Claim_size)
      Memory_Fail();
    pointer = Free_pointer;
    Free_pointer += size;
    Slip_Spin_Unlock(Memory_lock);
    pointer->cel = make_header(Tag,
                               Size);
    return pointer; }
```

```
NIL_type Memory_Release(UNS_type (
  { Slip_Spin_Lock(Memory_lock);
    Claim_size -= Claim + 1;
    Claim_counter--;
    Slip_Spin_Unlock(Memory_lock); }
```

Multicore memory management

```
BYT_type Memory_Claim(UNS_type Claim)
  { BYT_type overflow;
    Slip_Spin_Lock(Memory_lock);
    Claim_size += Claim + 1;
    Claim_counter++;
    overflow = (Tail_pointer - Free_pointer <= Claim_size);
    if (overflow)
      { collect();
        if (Tail_pointer - Free_pointer <= Claim_size)
          Memory_Fail(); }
    Slip_Spin_Unlock(Memory_lock);
    return overflow; }
```

```
PTR_type Memory_Make_Chunk(BYT_type Tag,
                           UNS_type Size)
  { PTR_type pointer;
    UNS_type size;
    size = Size + 1;
    Slip_Spin_Lock(Memory_lock);
    if (size > Claim_size)
      Memory_Fail();
    pointer = Free_pointer;
    Free_pointer += size;
    Slip_Spin_Unlock(Memory_lock);
    pointer->cel = make_header(Tag,
                               Size);
    return pointer; }
```

```
NIL_type Memory_Release(UNS_type (
  { Slip_Spin_Lock(Memory_lock);
    Claim_size -= Claim + 1;
    Claim_counter--;
    Slip_Spin_Unlock(Memory_lock); }
```

Multicore memory management

```
BYT_type Memory_Claim(UNS_type Claim)
  { BYT_type overflow;
    Slip_Spin_Lock(Memory_lock);
    Claim_size += Claim + 1;
    Claim_counter++;
    overflow = (Tail_pointer - Free_pointer <= Claim_size);
    if (overflow)
      { collect();
        if (Tail_pointer - Free_pointer <= Claim_size)
          Memory_Fail(); }
    Slip_Spin_Unlock(Memory_lock);
    return overflow; }
```

```
emory_Make_Chunk(BYT_type Tag,
                 UNS_type Size)
{ PTR_type pointer;
  UNS_type size;
  size = Size + 1;
  Slip_Spin_Lock(Memory_lock);
  if (size > Claim_size)
    Memory_Fail();
  pointer = Free_pointer;
  Free_pointer += size;
  Slip_Spin_Unlock(Memory_lock);
  pointer->cel = make_header(Tag,
                             Size);
  return pointer; }
```

```
NIL_type Memory_Release(UNS_type Claim)
  { Slip_Spin_Lock(Memory_lock);
    Claim_size -= Claim + 1;
    Claim_counter--;
    Slip_Spin_Unlock(Memory_lock); }
```

Multicore memory management

```
BYT_type Memory_Claim(UNS_type Claim)
  { BYT_type overflow;
    Slip_Spin_Lock(Memory_lock);
    Claim_size += Claim + 1;
    Claim_counter++;
    overflow = (Tail_pointer - Free_pointer <= Claim_size);
    if (overflow)
      { collect();
        if (Tail_pointer - Free_pointer <= Claim_size)
          Memory_Fail(); }
    Slip_Spin_Unlock(Memory_lock);
    return overflow; }
```

```
PTR_type Memory_Make_Chunk(BYT_type Tag,
                           UNS_type Size)
  { PTR_type pointer;
    UNS_type size;
    size = Size + 1;
    Slip_Spin_Lock(Memory_lock);
    if (size > Claim_size)
      Memory_Fail();
    pointer = Free_pointer;
    Free_pointer += size;
    Slip_Spin_Unlock(Memory_lock);
    pointer->cel = make_header(Tag,
                               Size);
    turn pointer; }
```

```
NIL_type Memory_Release(UNS_type Claim)
  { Slip_Spin_Lock(Memory_lock);
    Claim_size -= Claim + 1;
    Claim_counter--;
    Slip_Spin_Unlock(Memory_lock); }
```

SLIP/C multicore quicksort

```
(define (Sort V Low High Recurse)
  (define Left Low)
  (define Right High)
  (define Pivot (vector-ref V (quotient (+ Left Right) 2)))
  (define Save 0)
  (while (< Left Right)
         (while (< (vector-ref V Left) Pivot)
                (set! Left (+ Left 1)))
         (while (> (vector-ref V Right) Pivot)
                (set! Right (- Right 1)))
         (if (<= Left Right)
             (begin
               (set! Save (vector-ref V Left))
               (vector-set! V Left (vector-ref V Right))
               (vector-set! V Right Save)
               (set! Left (+ Left 1))
               (set! Right (- Right 1)))))
  (Recurse Left Right))
```

```
(define (SingleCore-QuickSort V Low High)
  (define (SingleCore-Recurse Left Right)
    (if (< Low Right)
        (SingleCore-QuickSort V Low Right))
    (if (> High Left)
        (SingleCore-QuickSort V Left High)))
  (Sort V Low High SingleCore-Recurse))
```

```
                                      ow High)
  (define (MultiCore-Recurse Left Right)
    (if (> Depth 0)
        (begin
          (define promise
            (if (< Low Right)
                (spawn (MultiCore-QuickSort (- Depth 1) V Low Right))))
          (if (> High Left)
              (MultiCore-QuickSort (- Depth 1) V Left High))
          (sync promise))
        (SingleCore-QuickSort V Low High)))
  (Sort V Low High MultiCore-Recurse))
```

SLIP/C multicore quicksort (cont'd)

```
(define (Sort V Low High Recurse)
  (define Left Low)
  (define Right High)
  (define Pivot (vector-ref V (quotient (+ Left Right) 2)))
  (define Save 0)
  (while (< Left Right)
         (while (< (vector-ref V Left) Pivot)
                (set! Left (+ Left 1)))
         (while (> (vector-ref V Right) Pivot)
                (set! Right (- Right 1)))
         (if (<= Left Right)
             (begin
               (set! Save (vector-ref V Left))
               (vector-set! V Left (vector-ref V Right))
               (vector-set! V Right Save)
               (set! Left (+ Left 1))
               (set! Right (- Right 1)))))
  (Recurse Left Right))
```

```
SingleCore-QuickSort V Low High)
 (SingleCore-Recurse Left Right)
< Low Right)
SingleCore-QuickSort V Low Right))
> High Left)
SingleCore-QuickSort V Left High)))
 Low High SingleCore-Recurse))
```

```
                                        ow High)
                                        t)
      (if (> Depth 0)
          (begin
            (define promise
              (if (< Low Right)
                  (spawn (MultiCore-QuickSort (- Depth 1) V Low Right))))
            (if (> High Left)
                (MultiCore-QuickSort (- Depth 1) V Left High))
            (sync promise))
          (SingleCore-QuickSort V Low High)))
    (Sort V Low High MultiCore-Recurse))
```

SLIP/C multicore quicksort (cont'd)

```
(define (Sort V Low High Recurse)
  (define Left Low)
  (define Right High)
  (define Pivot (vector-ref V (quotient (+ Left Right) 2)))
  (define Save 0)
  (while (< Left Right)
         (while (< (vector-ref V Left) Pivot)
                (set! Left (+ Left 1)))
         (while (> (vector-ref V Right) Pivot)
                (set! Right (- Right 1)))
         (if (<= Left Right)
             (begin
               (set! Save (vector-ref V Left))
               (vector-set! V Left (vector-ref V Right))
               (vector-set! V Right Save)
               (set! Left (+ Left 1))
               (set! Right (- Right 1)))))
  (Recurse Left Right))
```

```
(define (SingleCore-QuickSort V Low High)
  (define (SingleCore-Recurse Left Right)
    (if (< Low Right)
        (SingleCore-QuickSort V Low Right))
    (if (> High Left)
        (SingleCore-QuickSort V Left High)))
  (Sort V Low High SingleCore-Recurse))
```

```
(define (MultiCore-QuickSort Depth V Low High)
  (define (MultiCore-Recurse Left Right)
    (if (> Depth 0)
        (begin
          (define promise
            (if (< Low Right)
                (spawn (MultiCore-QuickSort (- Depth 1) V Low Right))))
          (if (> High Left)
              (MultiCore-QuickSort (- Depth 1) V Left High))
          (sync promise))
        (SingleCore-QuickSort V Low High)))
  (Sort V Low High MultiCore-Recurse))
```

SLIP/C multicore quicksort (cont'd)

```
(define (Sort V Low High Recurse)
  (define Left Low)
  (define Right High)
  (define Pivot (vector-ref V (quotient (+ Left Right) 2)))
  (define Save 0)
  (while (< Left Right)
         (while (< (vector-ref V Left) Pivot)
                (set! Left (+ Left 1)))
         (while (> (vector-ref V Right) Pivot)
                (set! Right (- Right 1)))
         (if (<= Left Right)
             (begin
               (set! Save (vector-ref V Left))
               (vector-set! V Left (vector-ref V Right))
               (vector-set! V Right Save)
               (set! Left (+ Left 1))
               (set! Right (- Right 1)))))
  (Recurse Left Rig
```

```
(define (SingleCore-QuickSort V Low High)
  (define (SingleCore-Recurse Left Right)
    (if (< Low Right)
        (SingleCore-QuickSort V Low Right))
    (if (> High Left)
        (SingleCore-QuickSort V Left High)))
  (Sort V Low High SingleCore-Recurse))
```

```
(define (MultiCore-QuickSort Depth V Low High)
  (define (MultiCore-Recurse Left Right)
    (if (> Depth 0)
        (begin
          (define promise
            (if (< Low Right)
                (spawn (MultiCore-QuickSort (- Depth 1) V Low Right))))
          (if (> High Left)
              (MultiCore-QuickSort (- Depth 1) V Left High))
          (sync promise))
        (SingleCore-QuickSort V Low High)))
  (Sort V Low High MultiCore-Recurse))
```

SLIP/C multicore quicksort (cont'd)

```
(define (Sort V Low High Recurse)
  (define Left Low)
  (define Right High)
  (define Pivot (vector-ref V (quotient (+ Left Right) 2)))
  (define Save 0)
  (while (< Left Right)
         (while (< (vector-ref V Left) Pivot)
                (set! Left (+ Left 1)))
         (while (> (vector-ref V Right) Pivot)
                (set! Right (- Right 1)))
         (if (<= Left Right)
             (begin
               (set! Save (vector-ref V Left))
               (vector-set! V Left (vector-ref V Right))
               (vector-set! V Right Save)
               (set! Left (+ Left 1))
               (set! Right (- Right 1)))))
  (Recurse Left Rig
```

```
(define (SingleCore-QuickSort V Low High)
  (define (SingleCore-Recurse Left Right)
    (if (< Low Right)
        (SingleCore-QuickSort V Low Right))
    (if (> High Left)
        (SingleCore-QuickSort V Left High)))
  (Sort V Low High SingleCore-Recurse))
```

```
(define (MultiCore-QuickSort Depth V Low High)
  (define (MultiCore-Recurse Left Right)
    (if (> Depth 0)
        (begin
          (define promise
            (if (< Low Right)
                (spawn (MultiCore-QuickSort (- Depth 1) V Low Right))))
          (if (> High Left)
              (MultiCore-QuickSort (- Depth 1) V Left High))
          (sync promise))
        (SingleCore-QuickSort V Low High)))
  (Sort V Low High MultiCore-Recurse))
```

SLIP/C multicore quicksort (cont'd)

```
(define (Sort V Low High Recurse)
  (define Left Low)
  (define Right High)
  (define Pivot (vector-ref V (quotient (+ Left Right) 2)))
  (define Save 0)
  (while (< Left Right)
         (while (< (vector-ref V Left) Pivot)
                (set! Left (+ Left 1)))
         (while (> (vector-ref V Right) Pivot)
                (set! Right (- Right 1)))
         (if (<= Left Right)
             (begin
               (set! Save (vector-ref V Left))
               (vector-set! V Left (vector-ref V Right))
               (vector-set! V Right Save)
               (set! Left (+ Left 1))
               (set! Right (- Right 1)))))
  (Recurse Left Rig
```

```
(define (SingleCore-QuickSort V Low High)
  (define (SingleCore-Recurse Left Right)
    (if (< Low Right)
        (SingleCore-QuickSort V Low Right))
    (if (> High Left)
        (SingleCore-QuickSort V Left High)))
  (Sort V Low High SingleCore-Recurse))
```

```
(define (MultiCore-QuickSort Depth V Low High)
  (define (MultiCore-Recurse Left Right)
    (if (> Depth 0)
        (begin
          (define promise
            (if (< Low Right)
                (spawn (MultiCore-QuickSort (- Depth 1) V Low Right))))
          (if (> High Left)
              (MultiCore-QuickSort (- Depth 1) V Left High))
          (sync promise))
        (SingleCore-QuickSort V Low High)))
  (Sort V Low High MultiCore-Recurse))
```

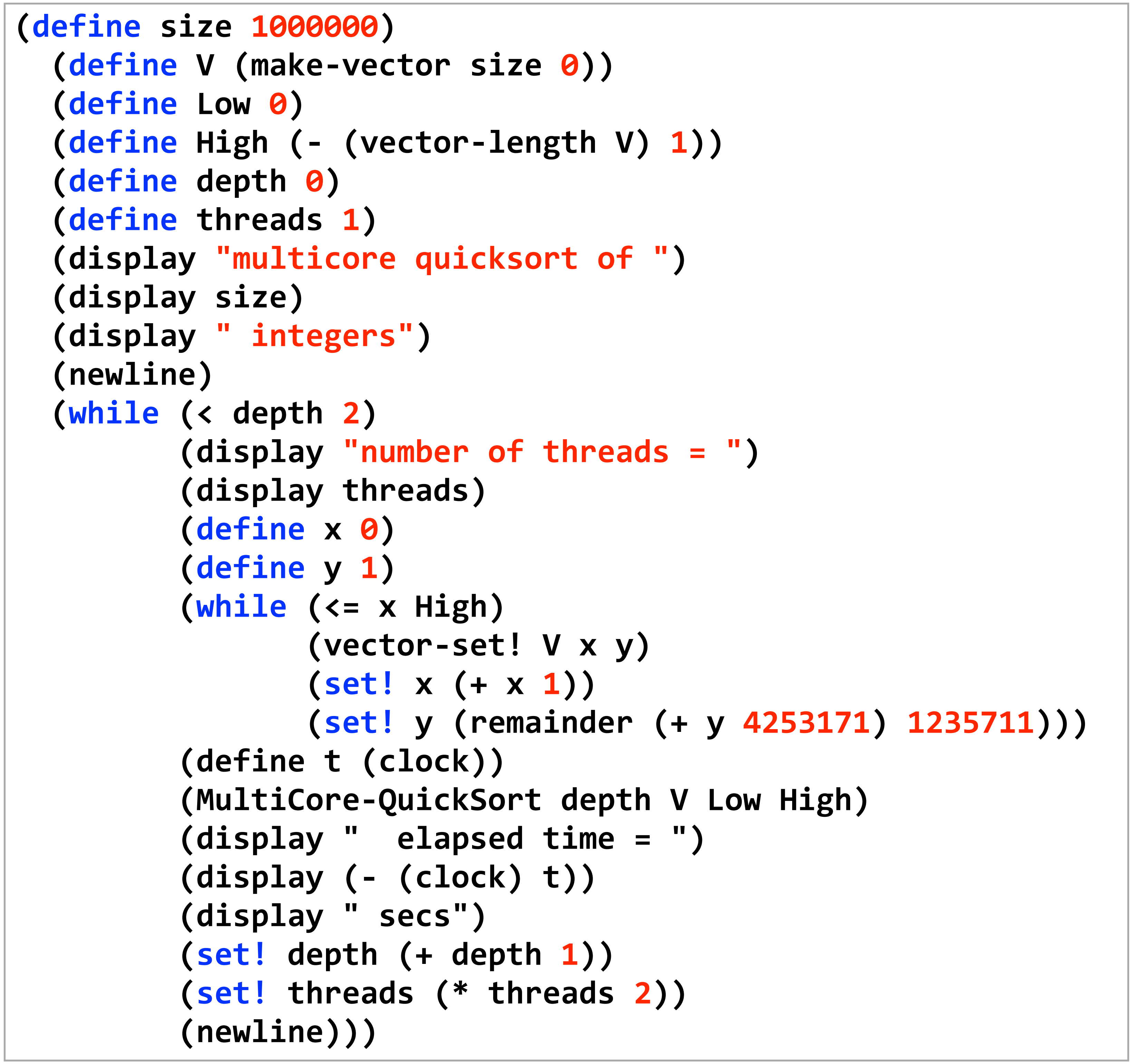
SLIP/C multicore quicksort (cont'd)

```
(define (Sort V Low High Recurse)
  (define Left Low)
  (define Right High)
  (define Pivot (vector-ref V (quotient (+ Left Right) 2)))
  (define Save 0)
  (while (< Left Right)
         (while (< (vector-ref V Left) Pivot)
                (set! Left (+ Left 1)))
         (while (> (vector-ref V Right) Pivot)
                (set! Right (- Right 1)))
         (if (<= Left Right)
             (begin
               (set! Save (vector-ref V Left))
               (vector-set! V Left (vector-ref V Right))
               (vector-set! V Right Save)
               (set! Left (+ Left 1))
               (set! Right (- Right 1)))))
  (Recurse Left Rig
```

```
(define (SingleCore-QuickSort V Low High)
  (define (SingleCore-Recurse Left Right)
    (if (< Low Right)
        (SingleCore-QuickSort V Low Right))
    (if (> High Left)
        (SingleCore-QuickSort V Left High)))
  (Sort V Low High SingleCore-Recurse))
```

```
(define (MultiCore-QuickSort Depth V Low High)
  (define (MultiCore-Recurse Left Right)
    (if (> Depth 0)
        (begin
          (define promise
            (if (< Low Right)
                (spawn (MultiCore-QuickSort (- Depth 1) V Low Right))))
          (if (> High Left)
              (MultiCore-QuickSort (- Depth 1) V Left High))
          (sync promise))
        (SingleCore-QuickSort V Low High)))
  (Sort V Low High MultiCore-Recurse))
```

SLIP/C multicore quicksort (cont'd)



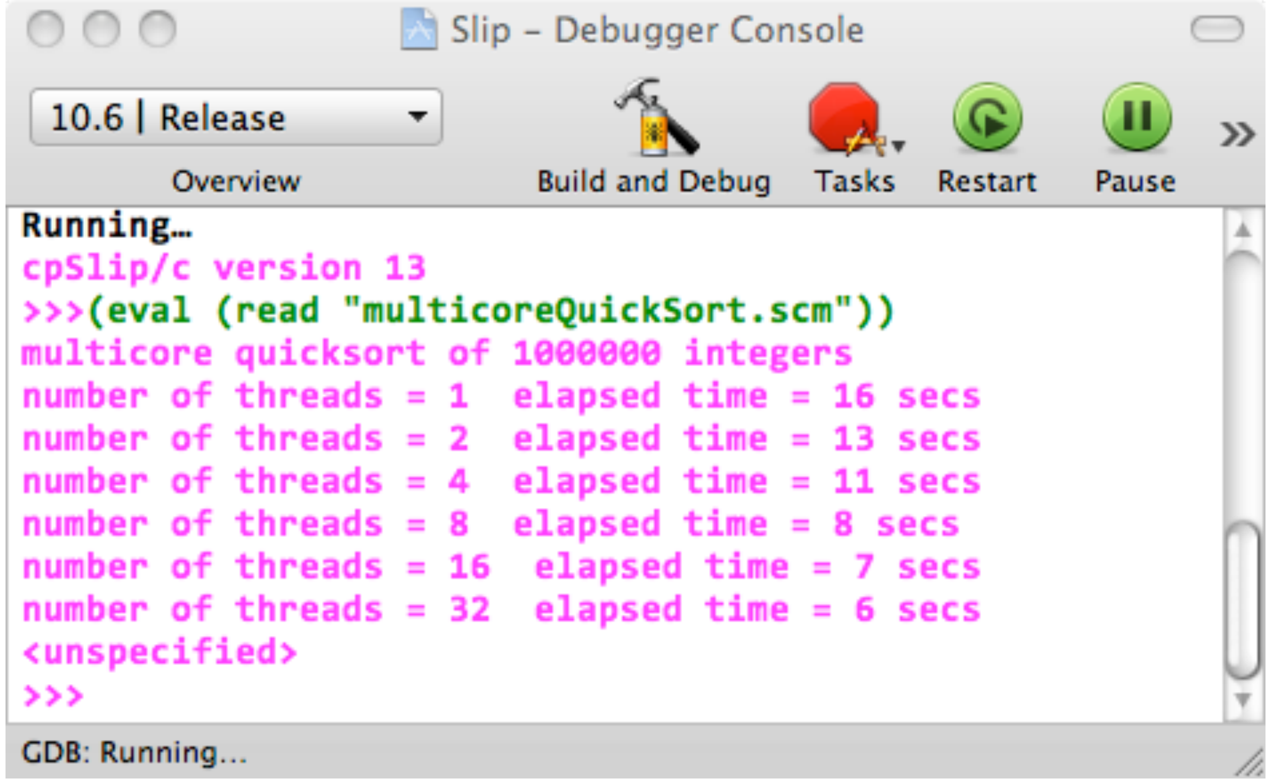
```
(define size 1000000)
  (define V (make-vector size 0))
  (define Low 0)
  (define High (- (vector-length V) 1))
  (define depth 0)
  (define threads 1)
  (display "multicore quicksort of ")
  (display size)
  (display " integers")
  (newline)
  (while (< depth 2)
         (display "number of threads = ")
         (display threads)
         (define x 0)
         (define y 1)
         (while (<= x High)
                (vector-set! V x y)
                (set! x (+ x 1))
                (set! y (remainder (+ y 4253171) 1235711)))
         (define t (clock))
         (MultiCore-QuickSort depth V Low High)
         (display "  elapsed time = ")
         (display (- (clock) t))
         (display " secs")
         (set! depth (+ depth 1))
         (set! threads (* threads 2))
         (newline)))
```

SLIP/C multicore quicksort (cont'd)

```
(define size 1000000)
  (define V (make-vector size 0))
  (define Low 0)
  (define High (- (vector-length V) 1))
  (define depth 0)
  (define threads 1)
  (display "multicore quicksort of ")
  (display size)
  (display " integers")
  (newline)
  (while (< depth 2)
         (display "number of threads = ")
         (display threads)
         (define x 0)
         (define y 1)
         (while (<= x High)
                (vector-set! V x y)
                (set! x (+ x 1))
                (set! y (remainder (+ y 4253171) 1235711)))
         (define t (clock))
         (MultiCore-QuickSort depth V Low High)
         (display "  elapsed time = ")
         (display (- (clock) t))
         (display " secs")
         (set! depth (+ depth 1))
         (set! threads (* threads 2))
         (newline)))
```

```
(define (report text c)
  (protect
   (display text)
   (display c)
   (display " ")
   (display " ... ")
   (display (- (clock) t))
   (display " secs")
   (newline)))
```

Multicore quicksort on a 4core

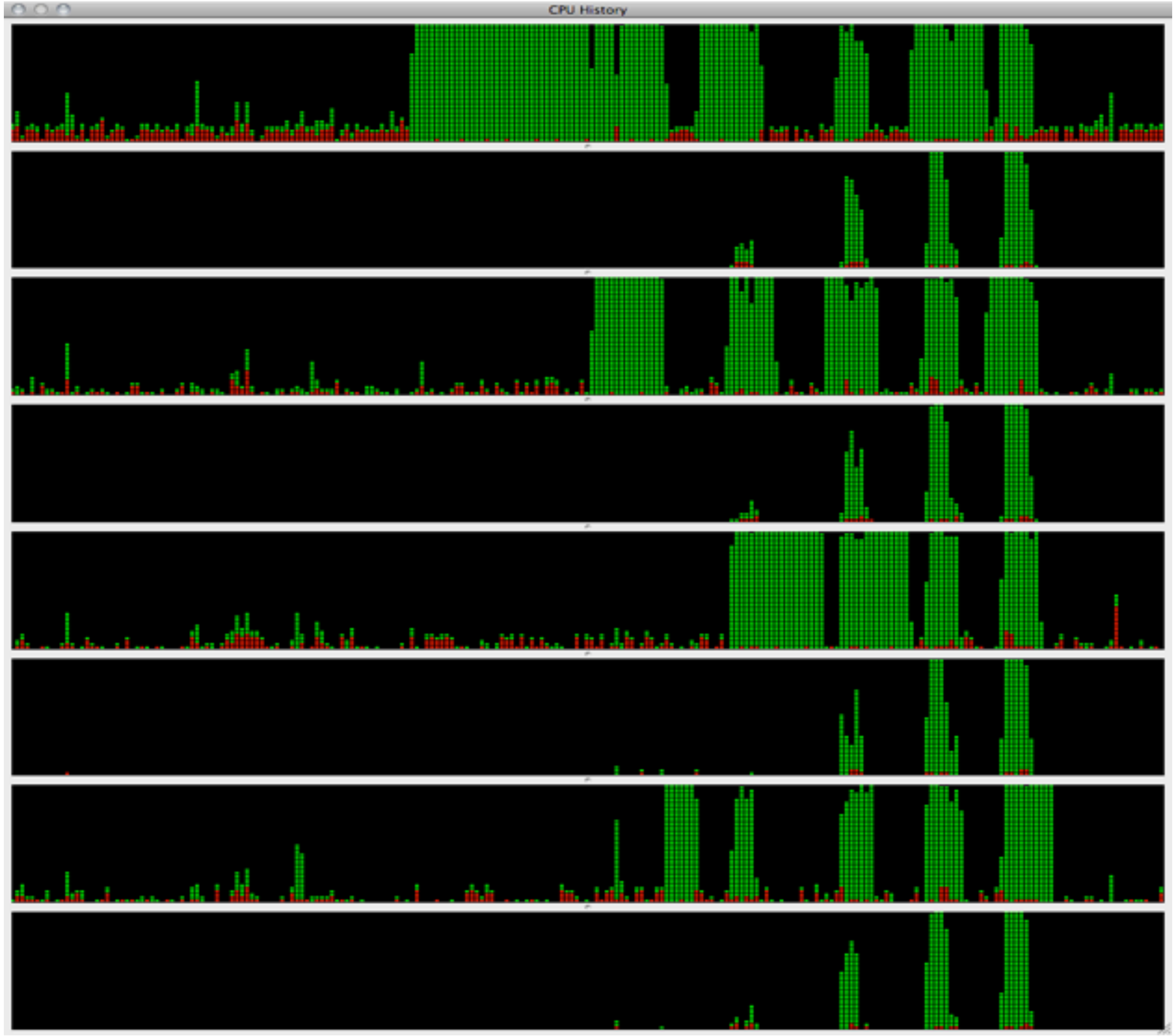


```
Running…
cpSlip/c version 13
>>>(eval (read "multicoreQuickSort.scm"))
multicore quicksort of 1000000 integers
number of threads = 1  elapsed time = 16 secs
number of threads = 2  elapsed time = 13 secs
number of threads = 4  elapsed time = 11 secs
number of threads = 8  elapsed time = 8 secs
number of threads = 16  elapsed time = 7 secs
number of threads = 32  elapsed time = 6 secs
<unspecified>
>>>
```

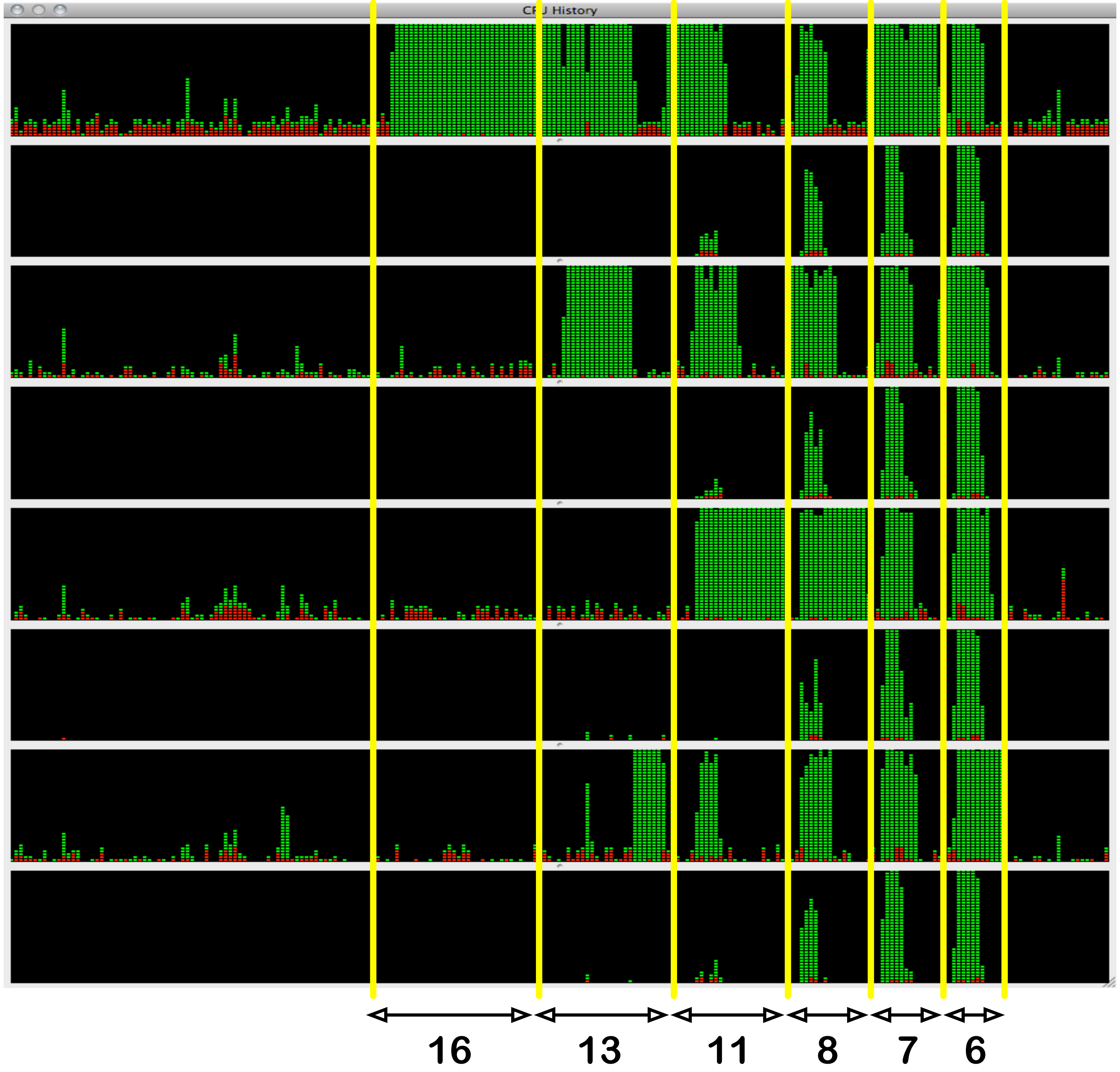
GDB: Running…

MacPro 4core

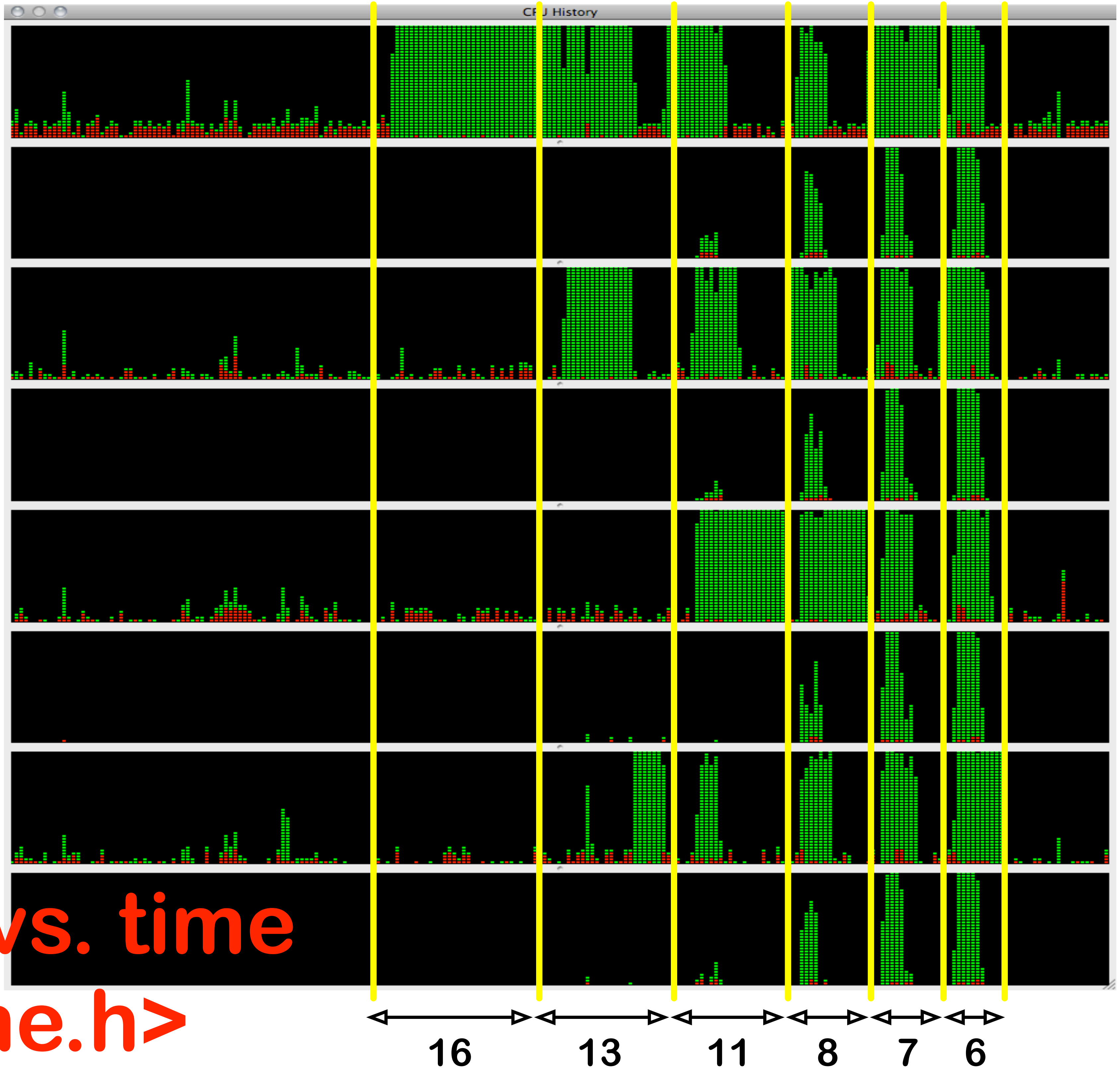
Multicore quicksort on a 4core (cont'd)



Multicore quicksort on a 4core (cont'd)

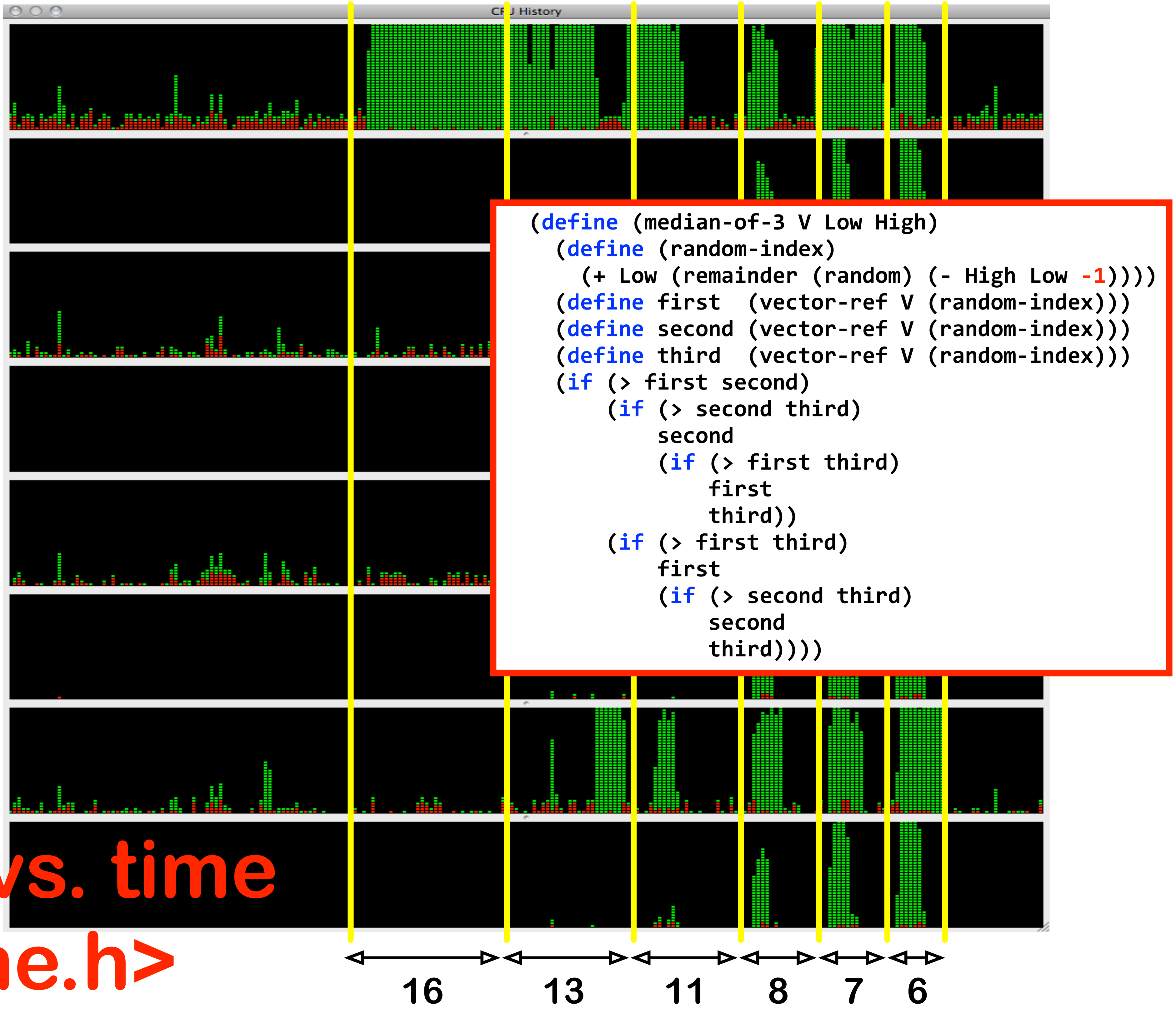


Multicore quicksort on a 4core (cont'd)



clock vs. time
in <time.h>

Multicore quicksort on a 4core (cont'd)



```
(define (median-of-3 V Low High)
  (define (random-index)
    (+ Low (remainder (random) (- High Low -1))))
  (define first  (vector-ref V (random-index)))
  (define second (vector-ref V (random-index)))
  (define third  (vector-ref V (random-index)))
  (if (> first second)
      (if (> second third)
          second
          (if (> first third)
              first
              third))
      (if (> first third)
          first
          (if (> second third)
              second
              third))))
```

clock vs. time in <time.h>

Multicore primitives

```
static NIL_type continue_spawn(CID_type Context_id)
  { EXP_type value;
    value = Context_Get_Expression(C
    Context_Thread_Zap(Context_id);
    Main_Stop_Thread(value); }
```

```
static EXP_type worker_procedure(ADR_type Address)
  { CID_type context_id;
    context_id = *(CID_type *)Address;
    Main_Claim_Default();
    Context_Thread_Push_M(context_id,
                          Continue_spawn,
                          Main_False,
                          sPN_size);
    Main_Release_Default();
    evaluate_context(context_id,
                     Main_False);
    for (;;)
      Context_Proceed(context_id);
    return Main_Unspecified; }
```

```
static NIL_type evaluate_sp

  { CID_type context_id;
    SPN_type spawn_express
    EXP_type expression;
    PRM_type promise;
    Main_Claim_Default();
    spawn_expression = Context_Get_Expression(Context_id);
    expression = spawn_expression->exp;
    context_id = Context_Clone_M(Context_id,
                                 expression);
    Main_Release_Default();
    promise = Main_Spawn_Thread_M(worker_procedure,
                                  &context_id);
    Context_Set_Expression(context_id,
                           promise); }
```

Multicore primitives

```
static NIL_type continue_spawn(CID_type Context_id)
  { EXP_type value;
    value = Context_Get_Expression(Context_id);
    Context_Thread_Zap(Context_id);
    Main_Stop_Thread(value); }
```

```
static EXP_type worker_procedure(ADR_type Address)
  { CID_type context_id;
    context_id = *(CID_type *)Address;
    Main_Claim_Default();
    Context_Thread_Push_M(context_id,
                         Continue_spawn,
                         Main_False,
                         sPN_size);
    Main_Release_Default();
                                                  ;
                                                  );
```

```
static NIL_type evaluate_spawn(CID_type Context_id,
                               EXP_type Tailposition)
  { CID_type context_id;
    SPN_type spawn_expression;
    EXP_type expression;
    PRM_type promise;
    Main_Claim_Default();
    spawn_expression = Context_Get_Expression(Context_id);
    expression = spawn_expression->exp;
    context_id = Context_Clone_M(Context_id,
                                 expression);
    promise = Main_Spawn_Thread_M(worker_procedure,
                                  &context_id);
    Main_Release_Default();
    Context_Set_Expression(context_id,
                           promise); }
```

Multicore primitives

```
static NIL_type continue_spawn(CID_type Context_id)
  { EXP_type value;
    value = Context_Get_Expression(Context_id);
    Context_Thread_Zap(Context_id);
    Main_Stop_Thread(value); }
```

```
static EXP_type worker_procedure(ADR_type Address)
  { CID_type context_id;
    context_id = *(CID_type *)Address;
    Main_Claim_Default();
    Context_Thread_Push_M(context_id,
                          Continue_spawn,
                          Main_False,
                          sPN_size);
    Main_Release_Default();
                                                     ;

                                                     );
```

```
static NIL_type evaluate_spawn(CID_type Context_id,
                               EXP_type Tailposition)
  { CID_type context_id;
    SPN_type spawn_expression;
    EXP_type expression;
    PRM_type promise;
    Main_Claim_Default();
    spawn_expression = Context_Get_Expression(Context_id);
    expression = spawn_expression->exp;
    context_id = Context_Clo

    promise = Main_Spawn_Thr

    Main_Release_Default();
    Context_Set_Expression(c
                          p
```

```
PRM_type Main_Spawn_Thread_M(WPR_type Worker_procedure,
                             ADR_type Address)
  { PRM_type promise;
    STH_type slip_thread;
    Slip_Create_Thread(slip_thread,
                       Worker_procedure,
                       Address);
    promise = make_PRM(slip_thread);
    return promise; }
```

Multicore primitives

```
static NIL_type continue_spawn(CID_type Context_id)
  { EXP_type value;
    value = Context_Get_Expression(Context_id);
    Context_Thread_Zap(Context_id);
    Main_Stop_Thread(value); }
```

```
static EXP_type worker_procedure(ADR_type Address)
  { CID_type context_id;
    context_id = *(CID_type *)Address;
    Main_Claim_Default();
    Context_Thread_Push_M(context_id,
                          Continue_spawn,
                          Main_False,
                          sPN_size);
    Main_Release_Default();
                                                  ;
```

```
static NIL_type evaluate_spawn(CID_type Context_id,
                               EXP_type Tailposition)

  { CID_type context_id;
    SPN_type spawn_expression;
    EXP_type expression;
    PRM_type promise;
    Main_Claim_Default();
    spawn_expression = Context_Get_Expression(Context_id);
    expression = spawn_expression->exp;
    context_id = Context_Clo

    promise = Main_Spawn_Thr

    Main_Release_Default();
    Context_Set_Expression(c
                           p
```

```
#define Slip_Create_Thread(Thread, Worker, Argument)
  pthread_create(&Thread, NULL, (void *(*)(void *))
                 Worker, (void*)Argument)
```

```
PRM_type Main_Spawn_Thread_M(WPR_type Worker_procedure,
                             ADR_type Address)
  { PRM_type promise;
    STH_type slip_thread;
    Slip_Create_Thread(slip_thread,
                       Worker_procedure,
                       Address);
    promise = make_PRM(slip_thread);
    return promise; }
```

Multicore primitives

```
static NIL_type continue_spawn(CID_type Context_id)
  { EXP_type value;
    value = Context_Get_Expression(C
    Context_Thread_Zap(Context_id);
    Main_Stop_Thread(value); }
```

```
static EXP_type worker_procedure(ADR_type Address)
  { CID_type context_id;
    context_id = *(CID_type *)Address;
    Main_Claim_Default();
    Context_Thread_Push_M(context_id,
                          Continue_spawn,
                          Main_False,
                          sPN_size);
    Main_Release_Default();
    evaluate_context(context_id,
                     Main_False);
    for (;;)
      Context_Proceed(context_id);
    return Main_Unspecified; }
```

```
static NIL_type evaluate_s

  { CID_type context_id;
    SPN_type spawn_express
    EXP_type expression;
    PRM_type promise;
    Main_Claim_Default();
    spawn_expression = Context_Get_Expression(Context_id);
    expression = spawn_expression->exp;
    context_id = Context_Clone_M(Context_id,
                                 expression);
    Main_Release_Default();
    promise = Main_Spawn_Thread_M(worker_procedure,
                                  &context_id);
    Context_Set_Expression(context_id,
                           promise); }
```

Multicore primitives

```
static NIL_type continue_spawn(CID_type Context_id)
  { EXP_type value;
    value = Context_Get_Expression(Context_id);
    Context_Thread_Zap(Context_id);
    Main_Stop_Thread(value); }
```

```
 worker_procedure(ADR_type Address)
ntext_id;
= *(CID_type *)Address;
Default();
    Context_Thread_Push_M(context_id,
                          Continue_spawn,
                          Main_False,
                          sPN_size);
    Main_Release_Default();
    evaluate_Context(context_id,
                     Main_False);
    for (;;)
      Context_Proceed(context_id);
    return Main_Unspecified; }
```

```
static NIL_type evaluate_spawn(CID_type Context_id,
                               EXP_type Tailposition)
  { CID_type context_id;
    SPN_type spawn_expression;
    EXP_type expression;
    PRM_type promise;
    Main_Claim_Default();
    spawn_expression = Context_Get_Expression(Context_id);
    expression = spawn_expression->exp;
    context_id = Context_Clone_M(Context_id,
                                 expression);
    Main_Release_Default();
    promise = Main_Spawn_Thread_M(worker_procedure,
                                  &context_id);
    Context_Set_Expression(context_id,
                           promise); }
```

Multicore primitives

```
static NIL_type continue_spawn(CID_type Context_id)
  { EXP_type value;
    value = Context_Get_Expression(Context_id);
    Context_Thread_Zap(Context_id);
    Main_Stop_Thread(value); }
```

```
                  worker_procedure(ADR_type Address)
                 ntext_id;
                 = *(CID_type *)Address;
                 Default();
    Context_Thread_Push_M(context_id,
                          continue_spawn,
                          in_False,
                          N_size);
    Main_Release_Default();
    evaluate_context(context_id,
                     Main_False);
    for (;;)
      Context_Proceed(context_id);
    return Main_Unspecified; }
```

```
NIL_type Main_Stop_Thread(EXP_type Value)
  { Slip_Destroy_Thread(Value); }
```

```
static NIL_type evaluate_spawn(
                               EXP_type Tailposition)
  { CID_type context_id;
    SPN_type spawn_expression;
    EXP_type expression;
    PRM_type promise;
    Main_Claim_Default();
    spawn_expression = Context_Get_Expression(Context_id);
    expression = spawn_expression->exp;
    context_id = Context_Clone_M(Context_id,
                                 expression);
    Main_Release_Default();
    promise = Main_Spawn_Thread_M(worker_procedure,
                                  &context_id);
    Context_Set_Expression(context_id,
                           promise); }
```

Multicore primitives

```
static NIL_type continue_spawn(CID_type Context_id)
  { EXP_type value;
    value = Context_Get_Expression(Context_id);
    Context_Thread_Zap(Context_id);
    Main_Stop_Thread(value); }
```

```
 worker_procedure(ADR_type Address)
ntext_id;
= *(CID_type *)Address;
Default();
Context_Thread_Push_M(context_id,
                      Continue_spawn,
                      in_False,
                      N_size);
```

```
NIL_type Main_Stop_Thread(EXP_type Value)
  { Slip_Destroy_Thread(Value); }
```

```
Main_Release_Default();
evaluate_Context(context_id,
                 Main_False);
for (;;)
  Context_Proceed(context_id);
return Main_Unspecified; }
```

```
static NIL_type evaluate_spawn(CID_type Context_id,
                               EXP_type Tailposition)
```

```
#define Slip_Destroy_Thread(Value)
  pthread_exit(Value);
```

```
    EXP_type expression;
    PRM_type promise;
    Main_Claim_Default();
    spawn_expression = Context_Get_Expression(Context_id);
    expression = spawn_expression->exp;
    context_id = Context_Clone_M(Context_id,
                                 expression);
    Main_Release_Default();
    promise = Main_Spawn_Thread_M(worker_procedure,
                                  &context_id);
    Context_Set_Expression(context_id,
                           promise); }
```

Status of version 13

- ☑ should be version 14
- ☑ persistent bug in standard GC
- ☑ untested multicore GC

grumble!

Some numbers

sorting 1000000 numbers

☑ SLIP/C version 9:	19 sec
☑ SLIP/C version 12:	14 sec
☑ SLIP/C version 13:	24 sec
☑ PLT Scheme:	9 sec
☑ \ 'skēm\:	16 sec

PLT Scheme: no JIT, no debug info

Bare metal debugging

☑ extremely hard
☑ assertions
☑ code reviewing

Bare metal debugging

- ☑ extremely hard
- ☑ assertions
- ☑ code reviewing

particularly hard for interpreters

Bare metal debugging

- ☑ extremely hard
- ☑ assertions
- ☑ code reviewing

only one solution: coding discipline