ESUG, 2003–08–28, Bled.si

Extreme Late Binding

why syntax, semantics, and pragmatics should be 1st-class objects

Ian Piumarta
`ian.piumarta@inria.fr`
`http://www-sor.inria.fr/projects/vvm`

about this talk

dynamic architecture for *building* dynamic languages / systems . . .

autonomous systems

- embedded systems
- vertical evolutionary maintenance

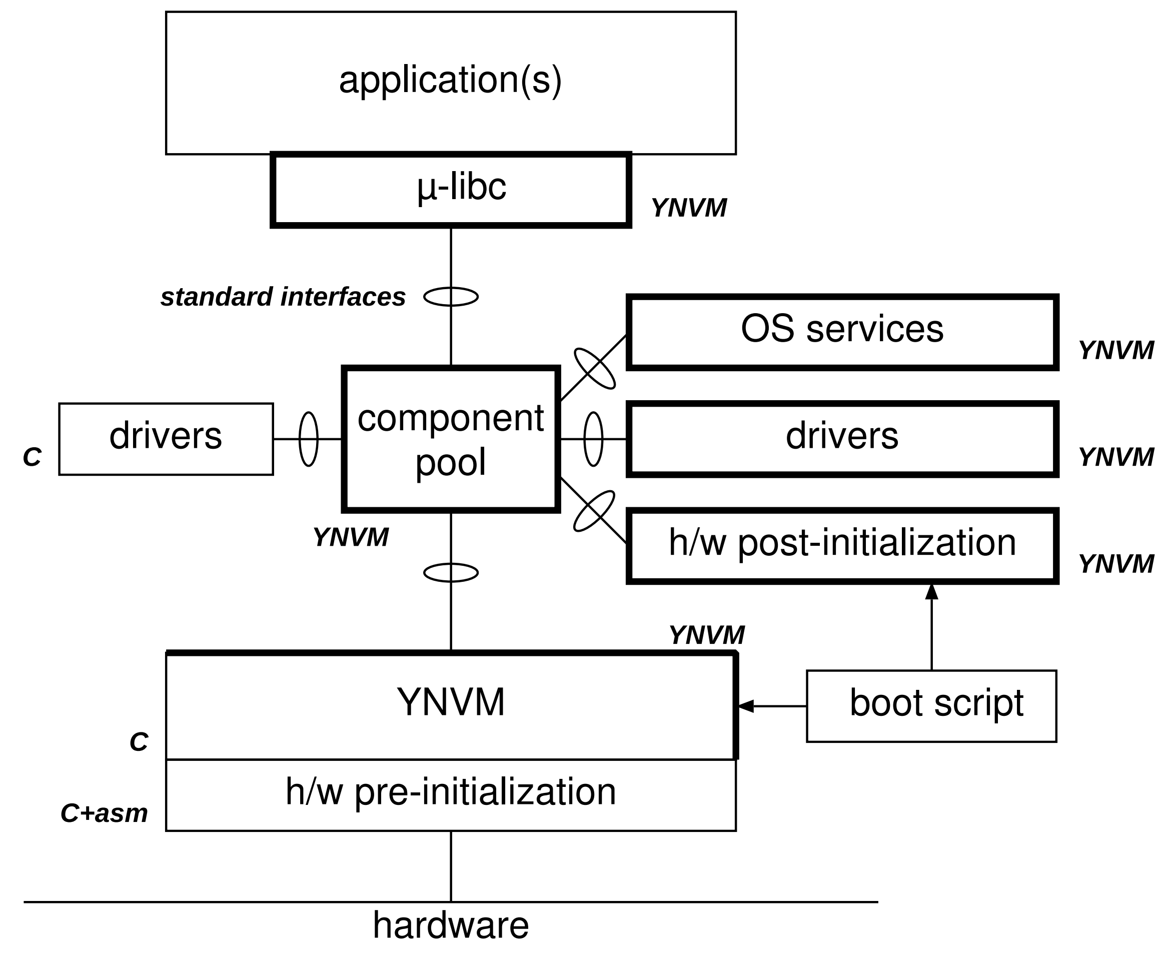
dedicated systems

- DSLs
- domain-specific semantics

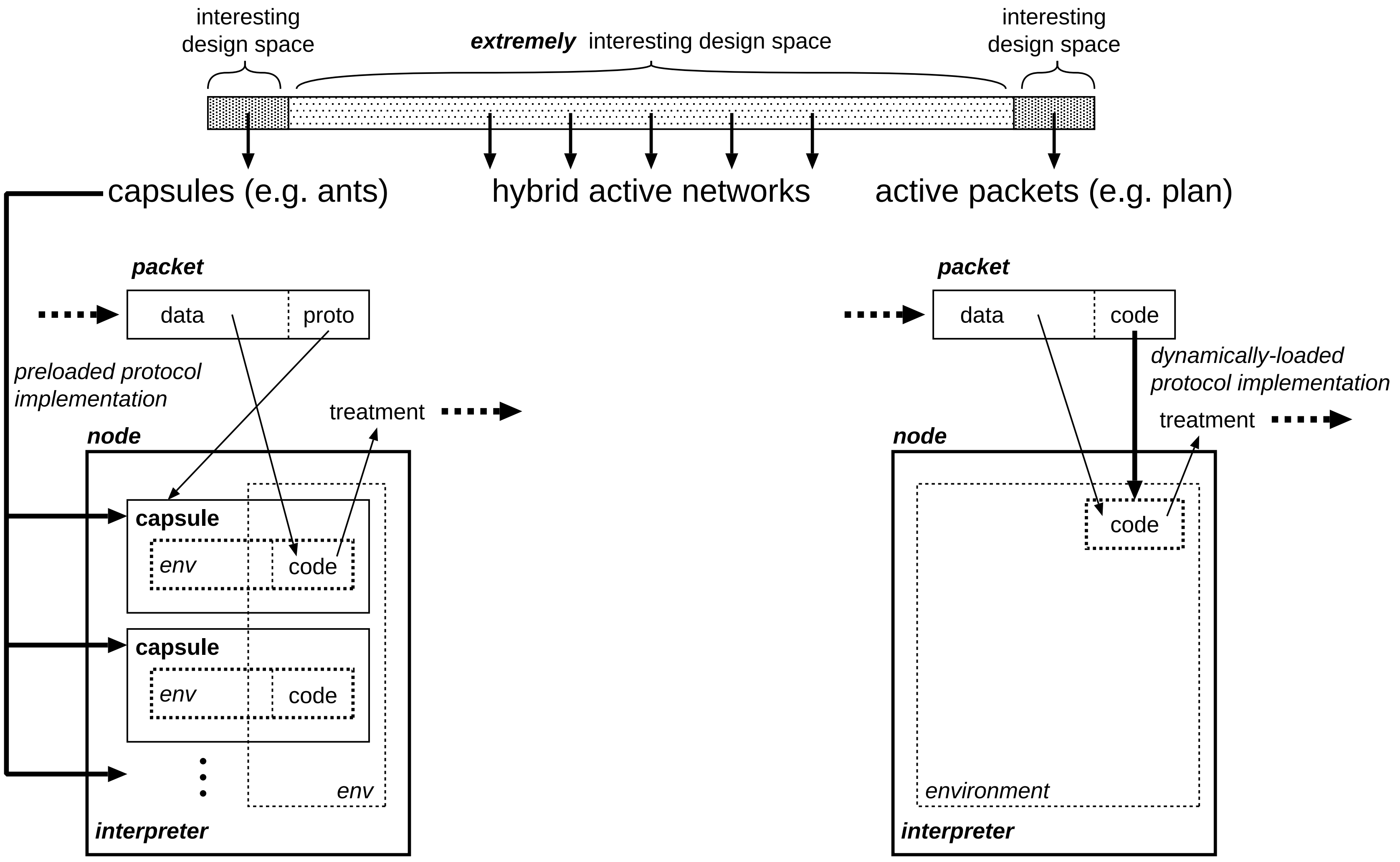
highly reactive &| dynamic systems

- active networks
- virtual virtual machines

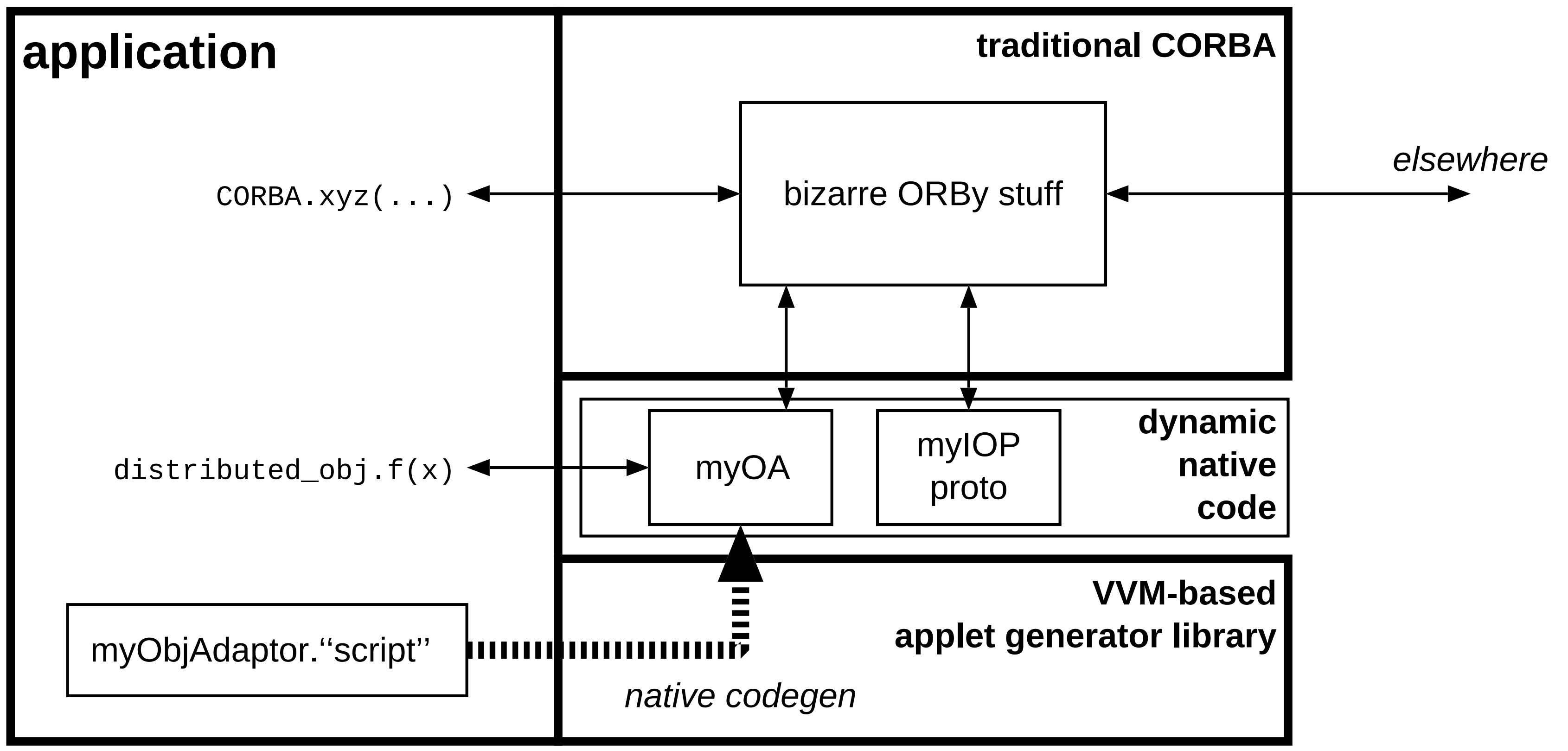
embedded systems



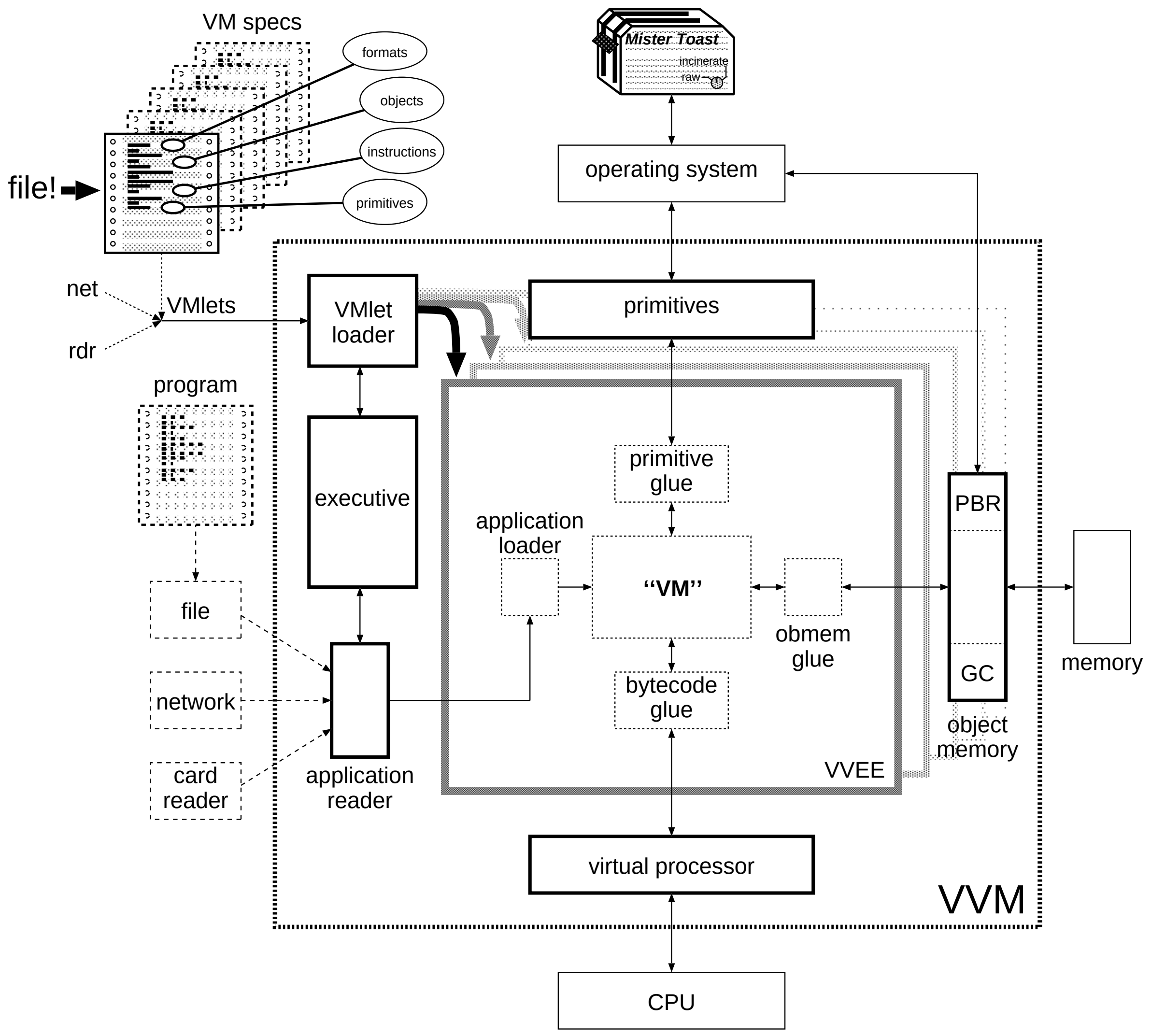
active networks



domain-specific components



virtual virtual machines



implementation challenges

object storage?

- ‘object’ has as many definitions as there are systems that claim to be ‘object-oriented’ (and several more besides)
  - GC? primitive (or new/unusual) types? persistence? . . . ?

  few ‘fundamental’ concepts

  (very) many interpretations/implementations of each

  $\Rightarrow$ need for genericity

implementation challenges (2)

primitives?

- semantics
  - synchronisation, saturated arithmetic, . . .
- pragmatics
  - import from execution environment (libc, . . .)
  - resource discovery? can we use the result? FFIs are *expensive*
- launch compiler, link `.so`
  - delays, resource comsumption, sandboxing
  - who says there's a compiler at all
  - compilers aren't created equal

implementation challenges (3)

basic execution machinery?

- app loading/initialisation
- invocation mechanisms (dynamic binding, . . .)
- new data types (and operations thereon)

  N 'basic' concepts
  M interpretations/implementations of each concept
  $M >> N$
  $\Rightarrow$ need for *genericity*

fancy execution machinery?

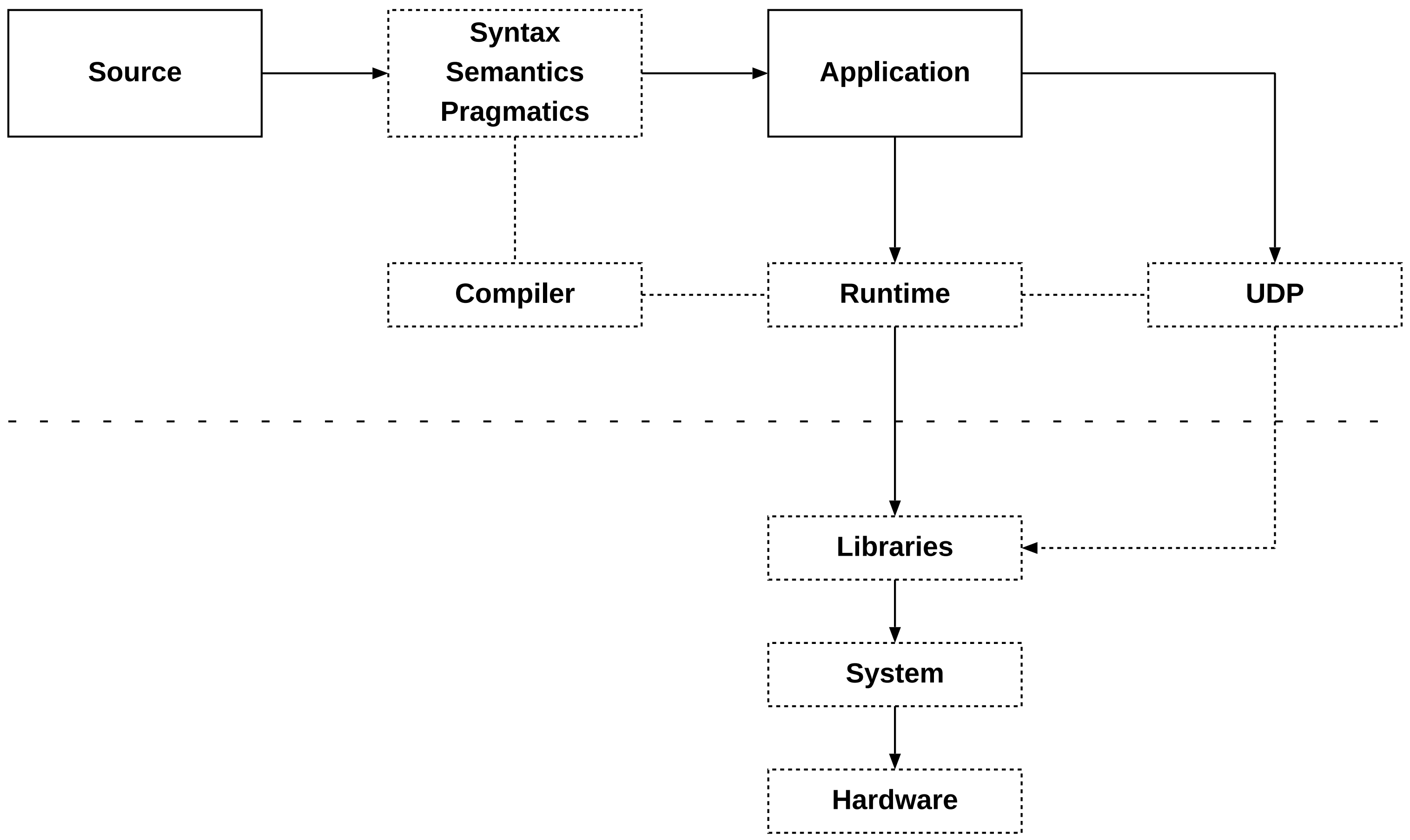
- reflection, introspection, intercession?

  (extremely hard to decouple from both object model *and* execution mechanisms simultaneously)

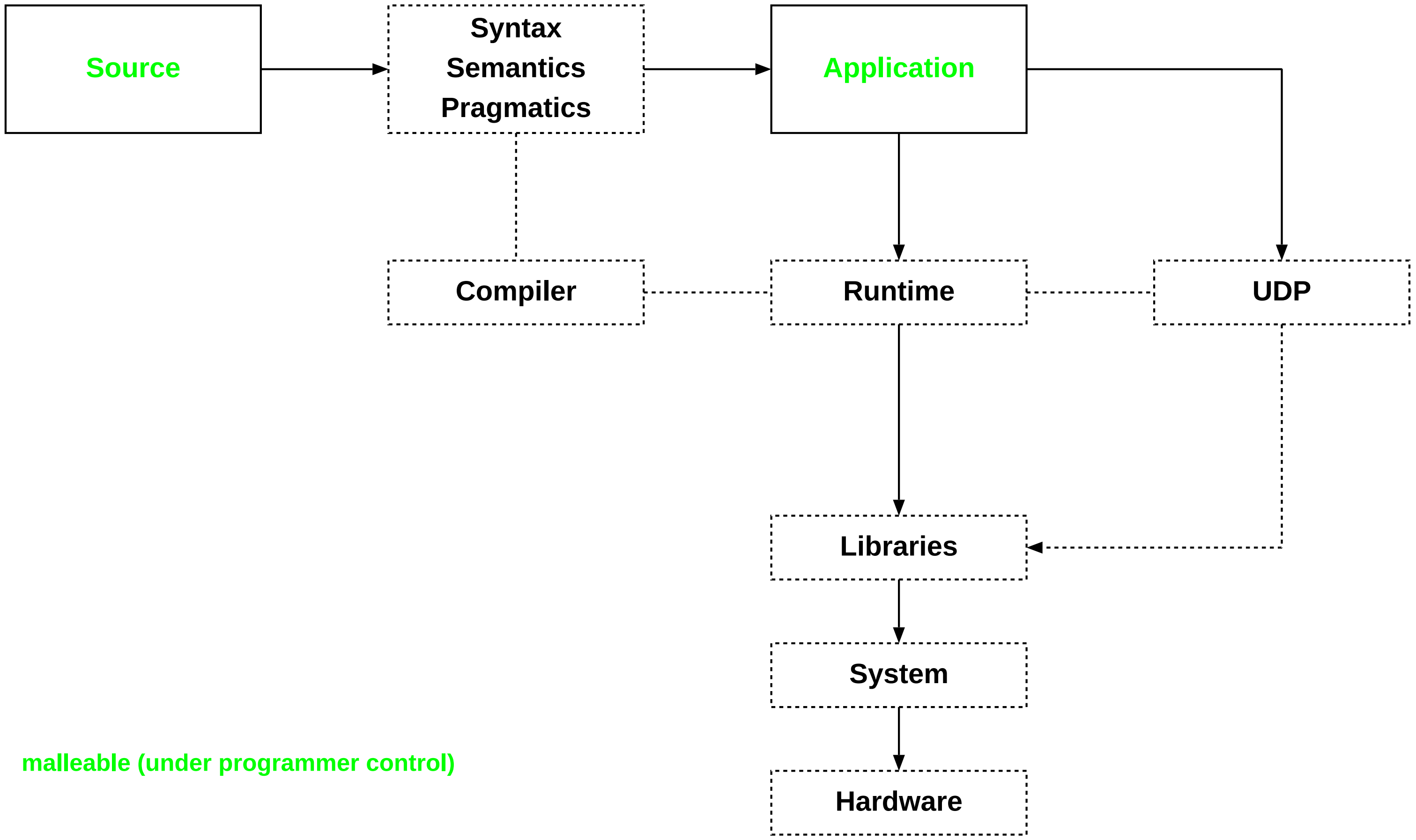
conventional programming languages

(with respect to *implementing* dynamic systems / languages)

what’s wrong with this picture?

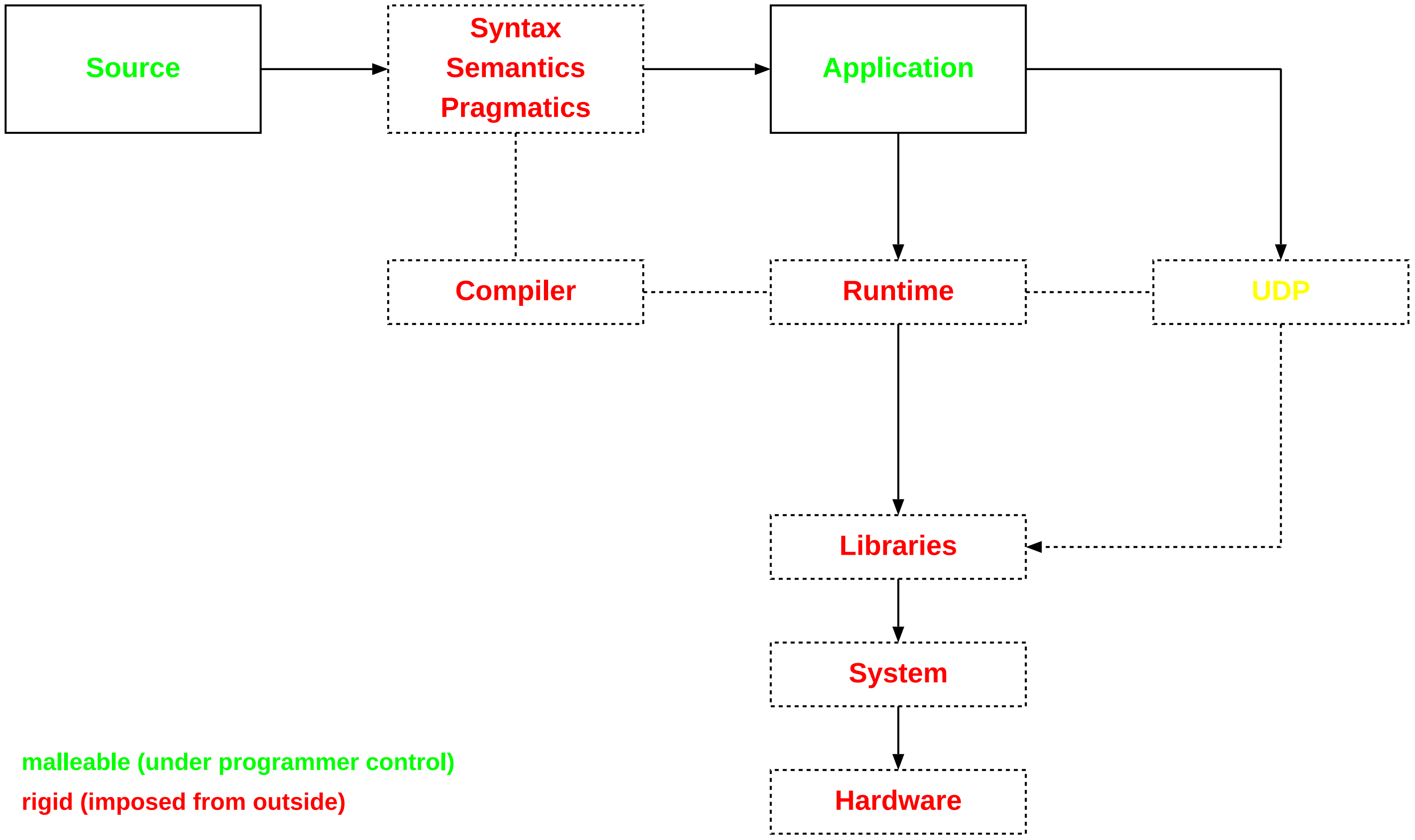


conventional programming languages



malleable (under programmer control)

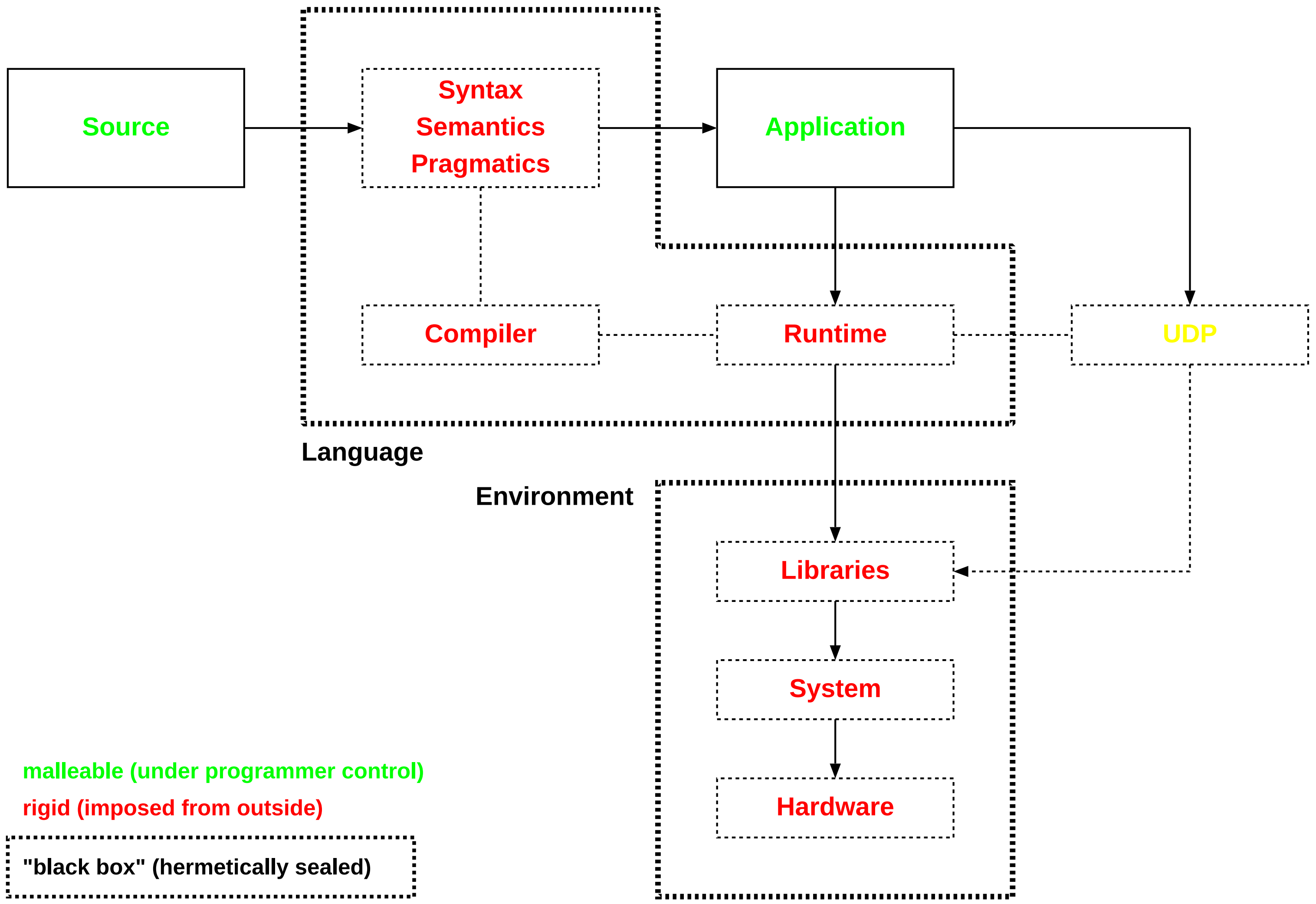
conventional programming languages



malleable (under programmer control)

rigid (imposed from outside)

conventional programming languages



conventional programming languages

two black boxes:

- language
- environment

rigidity

- syntax (DSLs?)
- semantics (atomic test-and-set?)
- pragmatics (primitives?)
- libraries (protocol evolution / in-field servicing?)
- system (drivers, scheduling, resource management, . . . ?)

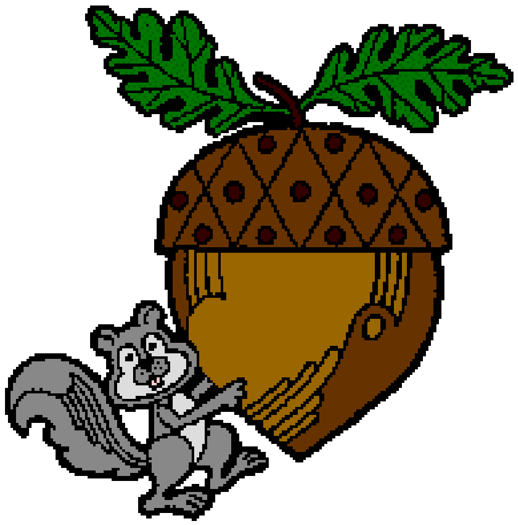
let's look at this another way. . .

“typical” programming languages

are:

collections of concrete syntax, semantics and pragmatics, (often) designed by a committee (that typical programmers never get to meet), then sealed inside an impermeable shell and presented as *faits accomplis* to. . .

“typical” programmers



who (all to frequently) spend *inordinate* amounts of time. . .

tackling “artificial barriers”

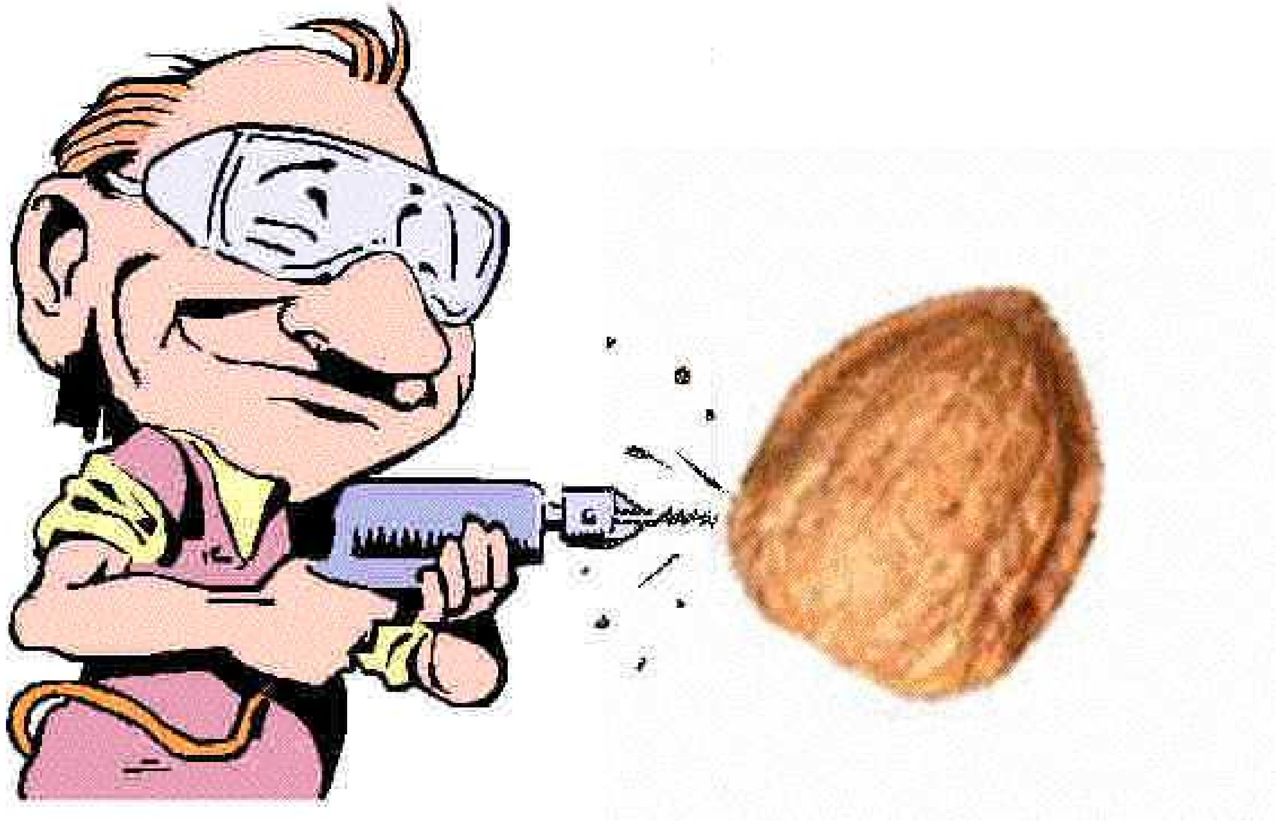
to *expressivity* and *creativity*, from the “outside” of their programming language’s implementation, wishing desperately that they could get at. . .

powerful, low-level language features

(and other juicy stuff) *locked away* on the “inside”.

for example: if you need new “primitive” functionality in Smalltalk, your only recourse is often to. . .

a “typical” virtual machine hacker



who will drill a new “primitive” hole (or three) through the *hermetic* “language shell” for you.

(but don’t forget we’re talking as much about C[++] as Smalltalk here)

some observations about conventional languages

without much justification (for lack of time)

- *disasterously* early-bound
- insufficient meta-data
- no reification of implementation
- artificially-closed access to internals

(characteristics which are *inherited* by their application “offspring”)

$\Rightarrow$

- late-bind *everything* (including language implementation)
- make *everything* 1st-class (including input [‘programs’] and output [native code])
- leave the entire compilation chain open by default (but make it trivial to close it off entirely, selectively, or according to artbitrary verification policies as needed)

for some suitable definition of “*everything*”

(traditionally: the *simplest possible one*)

the simplest possible definition of “everything”

lowest common (implementation) denominator:

- platform’s native code
- C ABI
- connection to `libc` / posix / whatever . . .

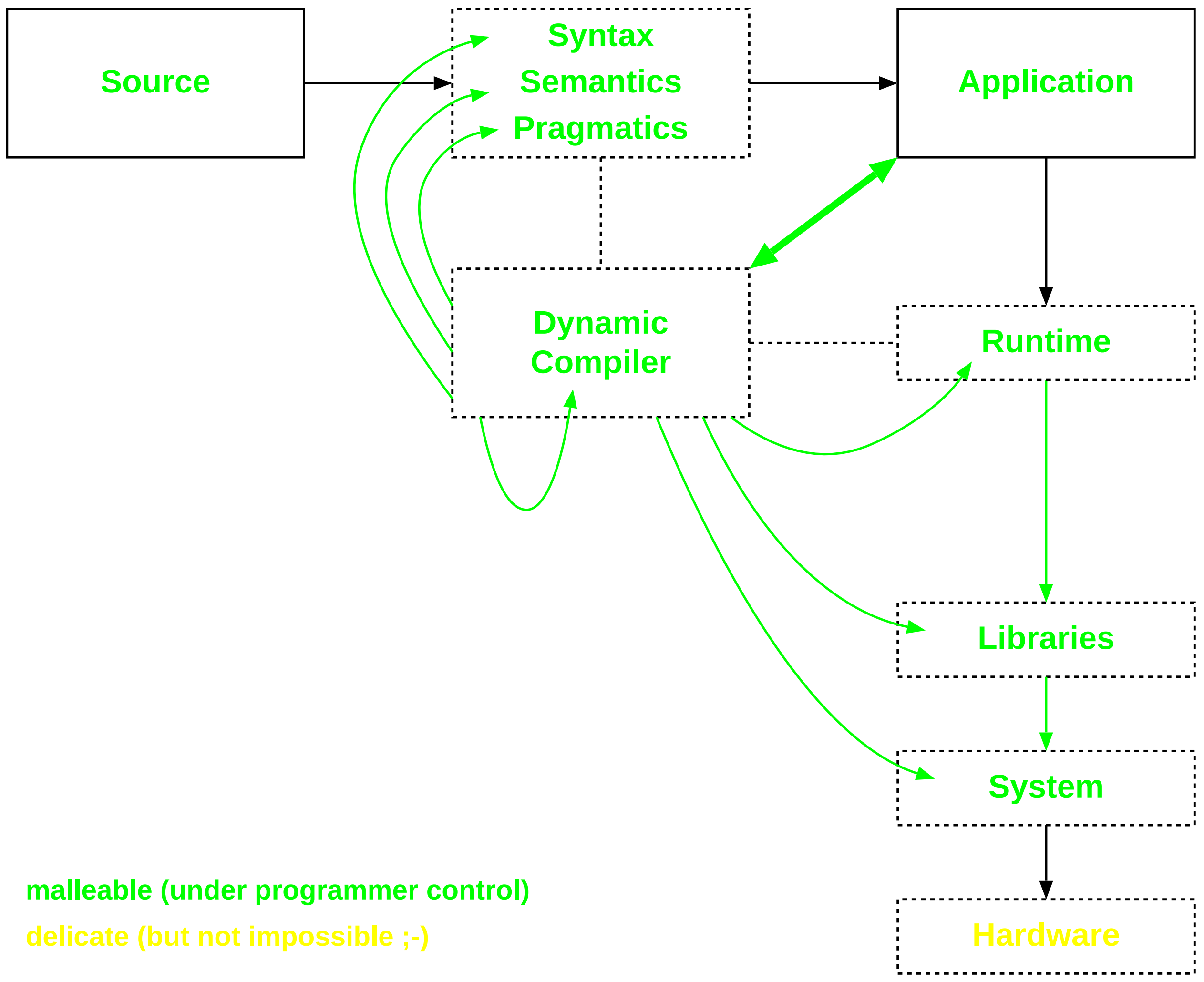
assume nothing about anything else

expose this “everything” to the application (or the app’s runtime)

- in the simplest, least-constrained form possible

then go have (lots of) fun building maximally-dynamic systems on top of it

unconventional programming languages



malleable (under programmer control)

delicate (but not impossible ;-)

a universal dynamic compiler: YNVM

it’s *not* a VM (in the *language* sense) but it sure smells like one

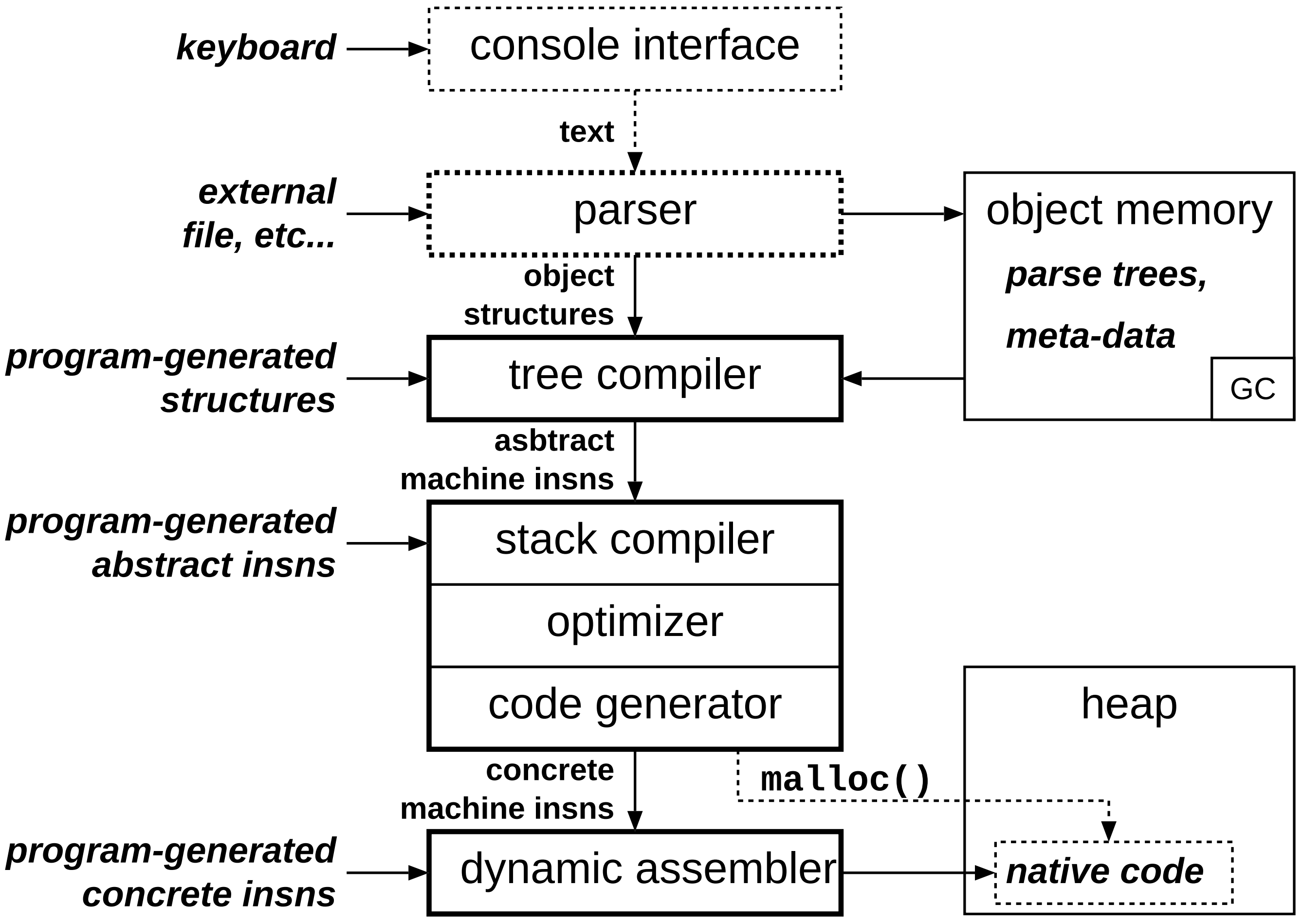
- dynamic compilation (native code, CABI)
- structured, language / host-independent input
- simple, persistent metadata (program structure, environment)

  $+$ visible bindings to metadata constructors / accessors
- customisable compiler semantics (application-defined transformations on input)
- access to all levels of compilation & code generation (for irrepressible hackers)
- access to the host system (`dlsym` or emulation)

applications inherit

- incremental development / debugging
- unlimited reflection & intercession
- live, in-field serviceability
- 100% (+/– 15%) performance of *fully optimised* C code

YNVM: time for a picture



the YNVM food chain

TL 3: the system implementor (end-user VM, language or app designer)

TL 2: dynamic C compiler (YNVM)

TL 1: platform-independent code generator (VPU)

TL 0: dynamic assembler (ccg)

bottom-up (with examples). . .

dynamic assembler

puts bytes in memory corresponding to (parameterised) machine instructions

- practical
  - instruction “templates” inline in source code
  - standard symbolic assembler notation
  - converted to emitters by a combination of preprocessor $+$ macro library
- fast ($<$ 3 insns executed / insn generated)
- horrendously non-portable

  (but widely ported: PowerPC, Sparc, ARM, IA32, IA64)

example?

“virtual” processor

abstracts over the actual hardware

- stack machine
- implements everything needed for C
  - arithmetic, logical operators
  - control structures
  - stack allocation (c.f., `alloca`)
  - labels, computed gotos
  - variadic functions
  - . . .
- performs *all* substantive tasks for “client” application:
  - code optimisation, register allocation
  - calling convention weirdness
- idempotent
- no externally-visible state (other than a “closure” over the vpu’s state)

example?

dynamic C compiler (YNVM)

one “primitive” function

- `dlsym` (or an emulated equivalent)
- C ABI compliance does the rest

examples?

a *truly* symbolic programming language over the vpu

- functionally equivalent to C (with some obvious improvements)
- input $=$ ASTs (S-exprs)
- arbitrary concrete syntax (but I *like* parentheses and prefix operators)
- top-level expressions compiled & executed *immediately*
  - modifications to compiler (syntax, semantics) felt *immediately*

example?

dynamic C compiler (YNVM)

tiny, internal object memory

- list, symbol, string, number
- input forms
- intermediate compilation state
- persistent metadata
- reifies *all* syntax and semantics implemented by compiler

data $=$ programs $=$ metadata $=$ programs . . .

- arbitray program-directed program transformations

example?

1st-class syntax and semantics

symbols arranged in *namespaces*

variables bound to symbols (obviously)

all compiler syntactic and semantic elements bound to symbols

- default behaviour supplied for a few
- redefined / extended arbitrarily
- effects can be localised within a namespace
  $\Rightarrow$ context-dependent syntax / semantics
- idem for “intrinsics” (e.g., __APPLY__)

$\Rightarrow$ (re)definition of *basic execution machinery*

  - vpu and assembler can be reified
  - new calling conventions, . . .

example?

example: inline cache detail

```
(define (syntax ->)
  (lambda (form)        ; (-> object (selector args))
    (syntax.match form (? ,recv ( ,[object.symbol? sel] ,@args ))
      (let ([posic (:posic.malloc)])
        ‘(let ([_recv ,recv])
           ((if (= (:class.class-of _recv)
                   (:posic.prev-class ,(object.integer posic)))
                (:posic.dest-meth ,(object.integer posic))
                (relink ,(object.integer posic)
                        _recv
                        ,(object.integer sel)))
             _recv
             ,@args)))])))
```

relevance to Smalltalk?

convert Slang to native code on-the-fly

```
SmallInteger>>+
  <primitive>
  | lhs rhs sum |
  lhs ← self stackValue: 1.
  rhs ← self stackValue: 0.
  ((self isIntegerObject: rhs)
    and: [sum ← (self integerValue: lhs) + (self integerValue: rhs).
          self isIntegerValue: sum])
      ifTrue:  [self pop2thenPush: (self integerObject: sum)]
      ifFalse: [self success: false]
```

- c.f., Ralph et al. ∼1990
- failure code?
- intercession?

reduced ‘Smalltalk’ systems:

- SqueakScript
- a “pragmatic” version of Slate

system-building methodology

c.f., Herbert Simon, *parable of the watchmakers*

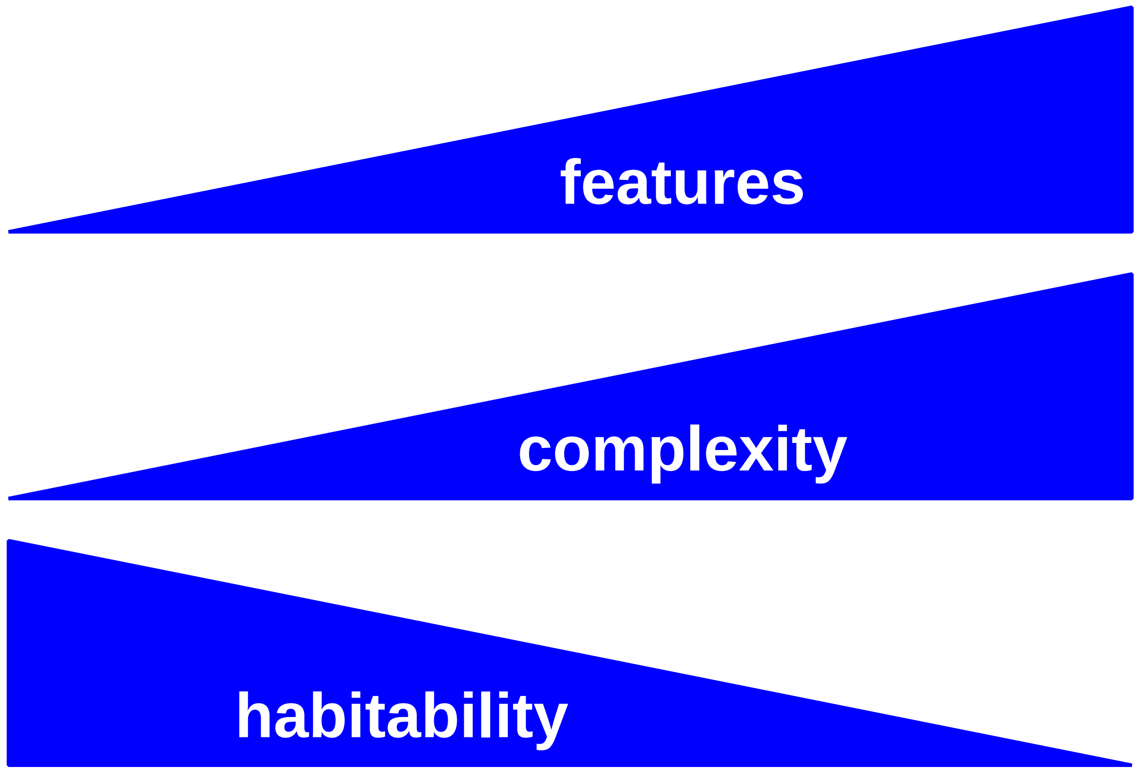
- complex systems will evolve from simple systems much more rapidly if there are stable intermediate forms than if there are not; the resulting complex systems in the former case will be hierarchic.

need for stable intermediate forms

- in “vertical” system / language design
- in deployment

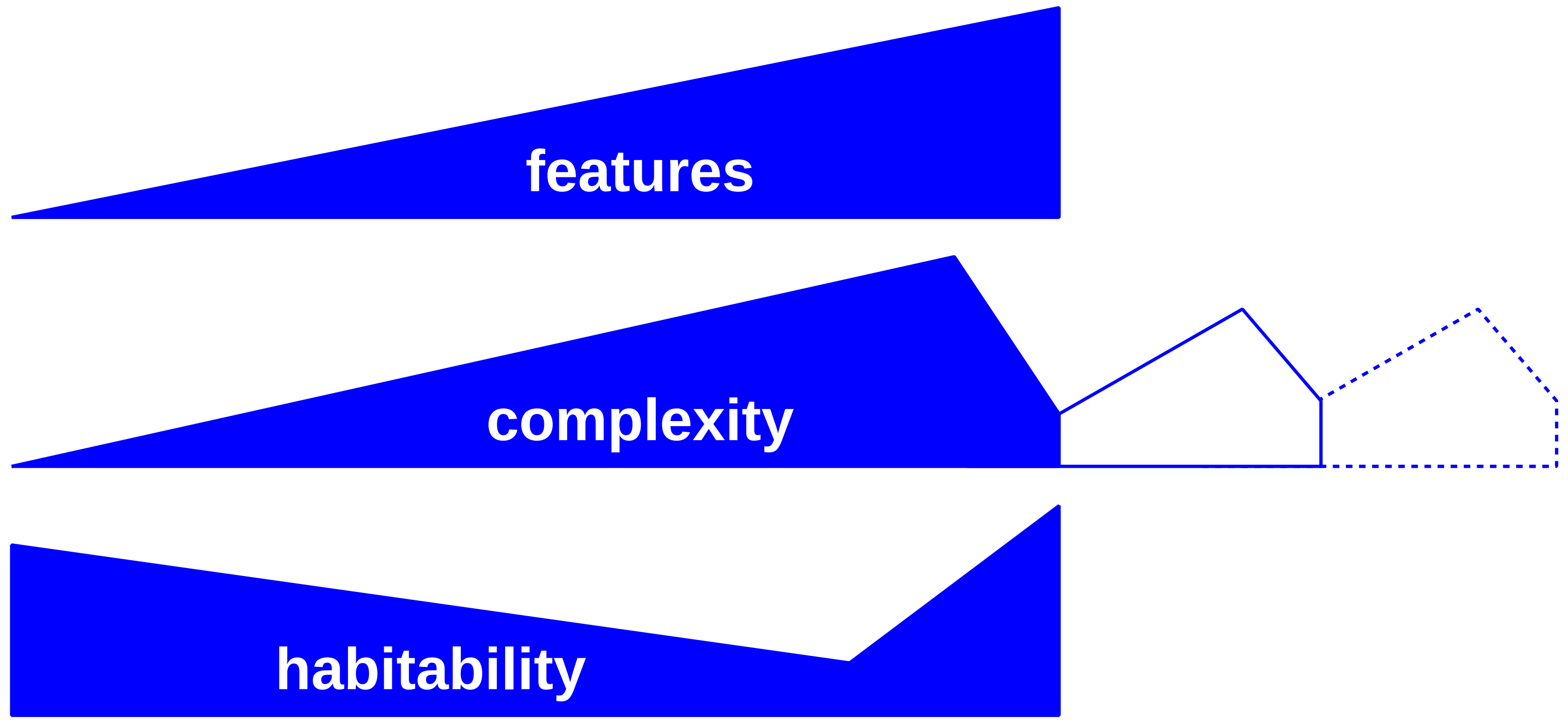
usability

user interfaces (toolsets) are environments for constructing applications



usability (2)

YNVM = “user interface” for constructing programming systems



conclusion

- choice of flexibility / complexity “compromise”

    5 (well-defined, in reality even more) levels of entry

- dynamic compilation overhead seen by applications

    $>=$ 1 Mb native code per second (on 3 year-old hardware)

- symbolic space affects semantic space

    domain-specific compiler customisation

    design & optimise your solution space, then use it

- agility

    programs are just meta-data (like the rest)

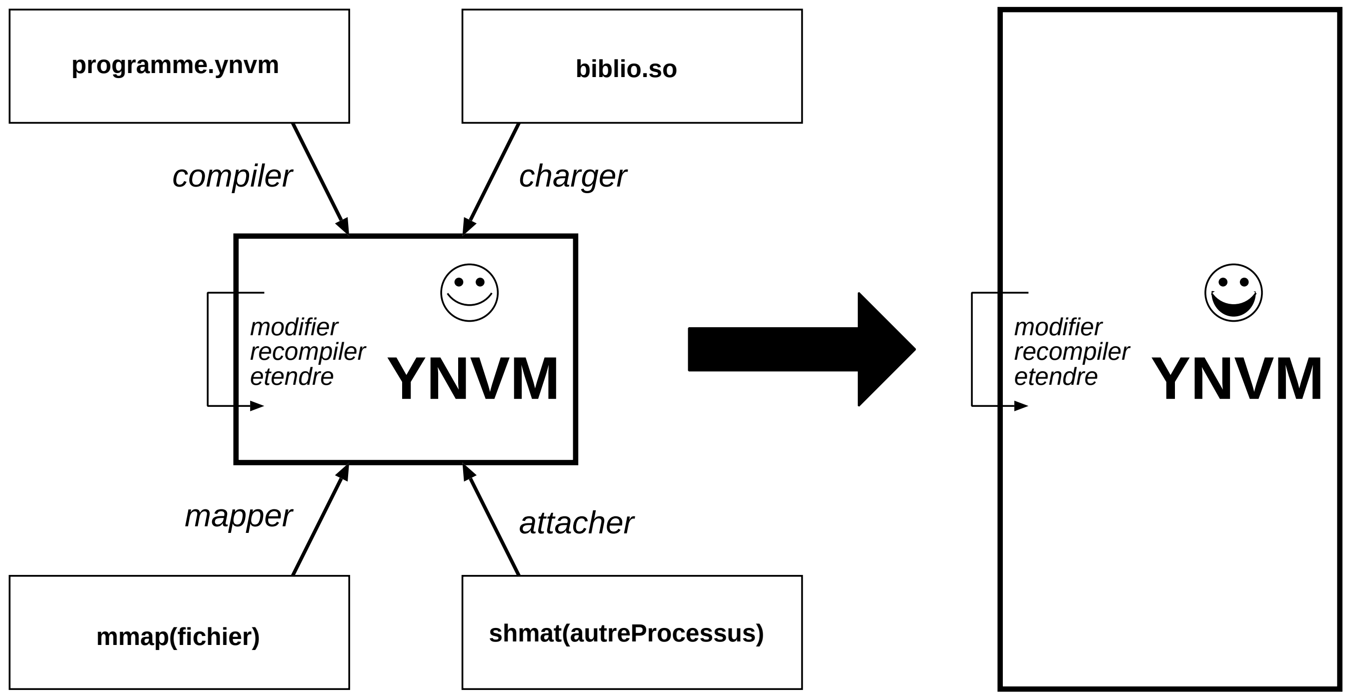
    semantics are dynamic (interpret or compile the same code)

    the compiler’s implementation is just part of the client’s namespace

    system state created incrementally, described dynamically, changes felt immediately

- simplicity: $\sim$ 10 KLOC, 250 KB code, for *everything*
- performance: 0.85 – 1.15 static (optimised) C programs

conclusion-in-a-picture



and finally:

how does building a language / system / environment / application on top of all this help our “typical” programmer?

an “atypical” programming language

$\Rrightarrow$

homogenous

- no artificial distinctions between language implementation, runtime and application
- no artificial barriers to expressivity or creativity

but can be *very* confusing for. . .

a “typical” user of an atypical language

to whom the extreme *generality* of the YNVM might pose problems, until they understand that

programming the YNVM

must commence by

- adding semantic/syntactic sugar (appropriate to the domain/application) over the homogenous base
- *then* building systems on top of it

however, compared to our traditional “walnut” languages. . .

digging beneath the “concrete” aspects

of any system build on the YNVM, to fiddle with or extend language / implementation features at an arbitrary depth

remains *entirely* possible. . .

. . . for anyone

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

typedef int (*pifi)(int);

pifi rpnCompile(char *expr);

int main()
{
  pifi c2f= rpnCompile("9*5/32+");
  pifi f2c= rpnCompile("32-5*9/");
  int i;
  printf("\nC:");
  for (i = 0; i <= 100; i+= 10) printf("%3d ", i);
  printf("\nF:");
  for (i = 0; i <= 100; i+= 10) printf("%3d ", c2f(i));
  printf("\n");
  printf("\nF:");
  for (i = 32; i <= 212; i+= 10) printf("%3d ", i);
  printf("\nC:");
  for (i = 32; i <= 212; i+= 10) printf("%3d ", f2c(i));
  printf("\n");
  return 0;
}

#cpu powerpc

pifi rpnCompile(char *expr)
{
  static insn *codePtr= 0;
  pifi fn;
  int top= 3;
  if (codePtr == 0) codePtr= (insn *)malloc(1024);
  #[
                                        .org    codePtr
  ]#;

  while (*expr)
    {
      char buf[32];
      int n;
      if (sscanf(expr, "%[0-9]%n", buf, &n))
        {
          ++top;
          #[                            lis     r(top), _HI(atoi(buf))
                                        ori     r(top), r(top), _LO(atoi(buf))  ]#;
          expr+= n - 1;
        }
      else if (*expr == '+')
        {
          --top;
          #[                            add     r(top), r(top), r(top+1)        ]#;
        }
      else if (*expr == '-')
        {
          --top;
          #[                            sub     r(top), r(top), r(top+1)        ]#;
        }
      else if (*expr == '*')
        {
          --top;
          #[                            mullw   r(top), r(top), r(top+1)        ]#;
        }
      else if (*expr == '/')
        {
          --top;
          #[                            divw    r(top), r(top), r(top+1)        ]#;
        }
      else
        {
          fprintf(stderr, "cannot compile: %s\n", expr);
          abort();
        }
      ++expr;
    }
  #[                                    blr                                     ]#;
  iflush(codePtr, asm_pc);
  fn= (pifi)codePtr;
  codePtr= asm_pc;
  return fn;
}
```

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

typedef int (*pifi)(int);

pifi rpnCompile(char *expr);

int main()
{
  pifi c2f= rpnCompile("9*5/32+");
  pifi f2c= rpnCompile("32-5*9/");
  int i;
  printf("\nC:");
  for (i = 0; i <= 100; i+= 10) printf("%3d ", i);
  printf("\nF:");
  for (i = 0; i <= 100; i+= 10) printf("%3d ", c2f(i));
  printf("\n");
  printf("\nF:");
  for (i = 32; i <= 212; i+= 10) printf("%3d ", i);
  printf("\nC:");
  for (i = 32; i <= 212; i+= 10) printf("%3d ", f2c(i));
  printf("\n");
  return 0;
}

#include <VPU.h>

pifi rpnCompile(char *expr)
{
  VPU *vpu= new VPU;
  vpu                                           ->Ienter()->Iarg()
                                                ->LdArg(0);
  while (*expr)
    {
      char buf[32];
      int n;
      if (sscanf(expr, "%[0-9]%n", buf, &n))
        {
          expr += n-1;                          vpu->Ld(atoi(buf));
        }
      else if (*expr == '+')                    vpu->Add();
      else if (*expr == '-')                    vpu->Sub();
      else if (*expr == '*')                    vpu->Mul();
      else if (*expr == '/')                    vpu->Div();
      else
        {
          fprintf(stderr, "cannot compile: %s\n", expr);
          abort();
        }
      ++expr;
    }

  void *entry= vpu                              ->Ret()
    ->compile(/*(1 << VPU::DumpAsm)*/);
  delete vpu;
  return (pifi)entry;
}
```

```
(define strtol (system.dlsym 0 "strtol"))

(define rpn-compile
  (lambda (expr)
    (let ([stack (object.list.copy '(i))]
          [mk-op (lambda (op stack)
                   (let ([expr (object.list 3)]
                         [rhs  (object.list.remove-last stack)]
                         [lhs  (object.list.remove-last stack)])
                     (object.list.append expr op)
                     (object.list.append expr lhs)
                     (object.list.append expr rhs)
                     (object.list.append stack expr)))])
      (while (byte expr)
        (let ([c (byte expr)])
          (cond
           [(= c $+) (mk-op '+ stack)]
           [(= c $-) (mk-op '- stack)]
           [(= c $*) (mk-op '* stack)]
           [(= c $/) (mk-op '/ stack)]
           [1
            (let ([end (system.malloc 4)]
                  [n   (strtol expr end 0)])
              (object.list.append stack (object.integer n))
              (set! expr (- (word end) 1))
              (system.free end))]))
        (set! expr (+ 1 expr)))
      (let ([func (object.list.copy '(lambda (i)))])
        (object.list.append func (object.list.remove-last stack))
        (object.println func)
        (object.eval func)))))


(let ([c2f (rpn-compile "9*5/32+")]
      [f2c (rpn-compile "32-5*9/")])
  (system.printf "\nC:")
  (do ([i 0 (+ i 10)]) ([<= i 100]) (system.printf "%3d " i))
  (system.printf "\nF:")
  (do ([i 0 (+ i 10)]) ([<= i 100]) (system.printf "%3d " (c2f i)))
  (system.printf "\n")
  (system.printf "\nF:")
  (do ([i 32 (+ i 10)]) ([<= i 212]) (system.printf "%3d " i))
  (system.printf "\nC:")
  (do ([i 32 (+ i 10)]) ([<= i 212]) (system.printf "%3d " (f2c i)))
  (system.printf "\n"))
```

```
(define (syntax rpn-compile)
  (lambda (form)
    (let ([stack `(i)])
      (loop [i 1 (object.list.size form)]
        (let ([atom (object.list.at form i)])
          (cond
           [(object.integer? atom)  (object.list.append stack atom)]
           [(object.symbol?  atom)  (let ([rhs  (object.list.remove-last stack)]
                                          [lhs  (object.list.remove-last stack)])
                                      (object.list.append stack `(,atom ,lhs ,rhs)))]
           [1                       (system.error "cannot parse: %s"
                                                  (object.print-string atom))])))
      `(lambda (i) ,@stack))))

(let ([c2f (rpn-compile  9 * 5 / 32 +)]
      [f2c (rpn-compile 32 - 5 *  9 /)])
  (system.printf "\nC:")
  (do ([i 0 (+ i 10)]) ([<= i 100]) (system.printf "%3d " i))
  (system.printf "\nF:")
  (do ([i 0 (+ i 10)]) ([<= i 100]) (system.printf "%3d " (c2f i)))
  (system.printf "\n")
  (system.printf "\nF:")
  (do ([i 32 (+ i 10)]) ([<= i 212]) (system.printf "%3d " i))
  (system.printf "\nC:")
  (do ([i 32 (+ i 10)]) ([<= i 212]) (system.printf "%3d " (f2c i)))
  (system.printf "\n"))
```

```
(define gtod  (system.dlsym 0 "gettimeofday"))
(define ctime (system.dlsym 0 "ctime"))
(define sleep (system.dlsym 0 "sleep"))
(define buf   (system.malloc 4))

(while 1
  (gtod buf 0)
  (system.printf "\033[1A%s" (ctime buf))
  (sleep 1))
```